

Mini(mizing)-Tips for Maxi(mizing)-Programmers
Jim DeFoor
Lockheed Fort Worth Company

Maxi-Programmers want to minimize their application development time, their mods and maintenance time, and the processing time of their programs. They realize they must understand the functions performed by the SAS Data Step if they are to develop dependable and efficient code. They know their code will sometimes have to use the more demanding capabilities of the SAS Data Step if it is to be really efficient or effective. They understand they must build utilities applicable to many programs if they are to reduce their redundant programming tasks.

This paper explains the basic structure of the SAS Data Step. It also describes and differentiates the functions of the SET Statement and the INPUT and INFILE statements.

In addition, this paper shows several minimizing program techniques:

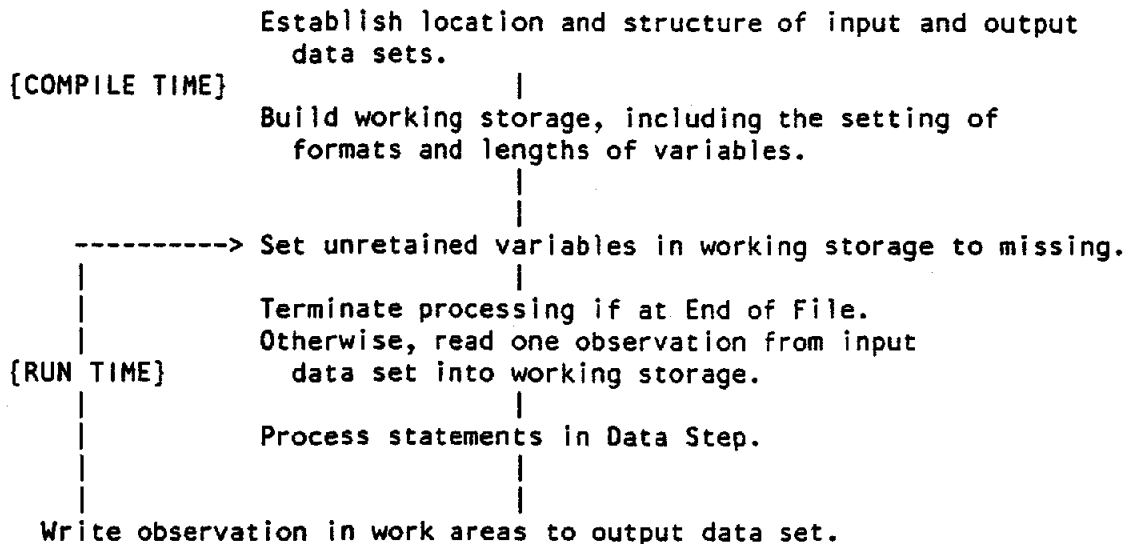
It describes the SET POINT technique of accessing a SAS data set directly instead of sequentially. It shows how the technique can reduce the cost of accessing data sets with a large number of variables. It also reveals why the technique is the only way to generate fully-customized reports which print multiple physical pages for each logical page.

It explains that the basic match-merge process is easy to code, is more efficient than the SAS Merge statement, and can be used on external data sets in addition to SAS data sets.

It shows when the PUT function and a user-built format assigns values more efficiently than IF-THEN-ELSE statements or SELECT statements. It also explains how to write SAS code which creates user-built formats from external files and how to include those dynamically created formats in the program assigning values to a variable.

It describes a utility Macro %COMPUTE which will perform single or multi-level summaries of source data. The Macro is explained and provided.

The usual flow of actions in SAS Data Step can be described as follows:



The SAS Data Step often has this structure:

<pre> DATA WORK; INFILE PARTDATA END=EOF; FILE ERROR; LENGTH PART \$15; INPUT @ 1 PART \$12.0 @20 ORDER \$12.0 @35 CONTRACT \$10.0 @50 COST 10.2 ; RUN; </pre>	<pre> DATA WORK; FILE ERROR; LENGTH PART \$15; SET PARTDATA END=EOF; RUN; </pre>
--	---

Because this form of the SAS Data Step does not have the COBOL or PASCAL-type divisions for Environment, Data, and Procedure, it obscures the three separate functions being performed by the SET Statement and the INPUT and INFILE Statements.

ENVIRONMENT:	Defining Input File Location and Structure
DATA:	Defining Working Storage
PROCEDURE:	Reading Input Data into Working Storage

The first two functions are defined during compile; the second performed during run-time.

The INFILE Statement defines the location and structure of the input file. The INPUT Statement defines working storage and reads the data into working storage. The SET Statement performs all three functions.

As said above, working storage is defined during compile, as is input file definition. Since the INPUT Statement and the SET Statement define working storage, any other definitions of working storage must precede those statements to be effective. That is why the LENGTH Statement must be coded before the SET or INPUT Statement.

The SET Statement's read function is somewhat different than the INPUT statement. A SET Statement defines a different input buffer and a different read process each time it occurs in a Data Step - even for the same input data set.

The INPUT Statement will also set up separate input buffers and read processes, but only when different INFILE Statements are used for the same input data set. Multiple INPUT Statements for the same INFILE Statement will continue the same read process for an external file as the first INPUT Statement for that INFILE.

To see this difference, run this simple test. The read process of SET WORK begins at the first observation in both of its read loops, whereas the read process of INFILE TEMP continues to sixth observation when beginning its second read loop.

```
DATA WORK;
  DO J = 1 TO 10;
    X + 1;
    OUTPUT;
  END;
RUN;
```

```
DATA NULL ;
  FILE TEMP;
  DO J = 1 TO 10;
    X + 1;
    PUT @ 1 X 2.0;
  END;
RUN;
```

```
DATA NULL ;
  DO K = 1 TO 5;
    SET WORK;
    PUT '1 ' K= J= X= ;
  END;
  DO K = 6 TO 10;
    SET WORK;
    PUT '2 ' K= J= X= ;
  END;
  STOP;
RUN;
```

```
DATA NULL ;
  DO K = 1 TO 5;
    INFILE TEMP;
    INPUT @ 1 2.0;
    PUT '1 ' K= J= X= ;
  END;
  DO K = 6 TO 10;
    INFILE TEMP;
    INPUT @ 1 2.0;
    PUT '2 ' K= J= X= ;
  END;
  STOP;
RUN;
```

The ability of the SET Statement to set up two entirely different read processes for the same SAS Data Set can provide special opportunities for processing data. One of these opportunities will be described during the discussion of Random-Access processing.

RANDOM (DIRECT) - ACCESS OF SAS DATA SETS

The name Random-Access is really a misnomer. The access technique is direct: only the observations wanted are moved into working storage (the PDV). Normal access is sequential: each observation is moved into working storage until the program is terminated.

The issue when using Random-Access is always the same: how does the program learn which observations are wanted. If this information is known from an Index or other external source, then the program can get the information from the external source directly or through the passage of a Macro variable. Another way, useful primarily when the Data Set has many variables, is to get the information from the Data Set itself.

```
DATA WORK2;
  LINE = 0;
  DO UNTIL(LINE=LINE_END);
    /* ONLY W1 PARTS AND ORDER ARE READ INTO WORKING STORAGE */
    SET WORK1(KEEP=W1_PARTS ORDER);
    LINE + 1;
    IF W1_PARTS = 'C247-8-16' THEN
      DO;
        IF ORDER = 'P501221' THEN
          DO;
            /* THE ENTIRE RECORD IS READ INTO WORKING STORAGE */
            SET WORK1 POINT=LINE NOBS=LINE_END;
            OUTPUT;
            STOP;
          END;
        END;
      END;
    /* THE STEP IS TERMINATED WHEN NO MATCHES ARE POSSIBLE */
    IF W1_PARTS GT 'C247-8-16' THEN
      STOP;
  END;
  STOP;
RUN;
```

This Step reads the SAS Data Set WORK1 sequentially to find the right record and then uses the SET POINT NOBS option to directly access WORK1 again.

The (KEEP=W1_PARTS ORDER) option on the SET WORK1 causes the first SET Statement to move only those two variables into working storage. All other variables on that observation are left in the buffer of the first SET Statement.

After reading Part Number and Order, the Data Step records the line number of the record and checks to see whether the Part Number and Order match the values wanted. If a match is found, the Data Step opens the WORK2 Data Set again, moves data into a different buffer until the line number matches the value of Line, copies the observation into working storage, and moves it to the output data set. Processing is then terminated.

Obviously, this approach is useful only when the number of variables in the SAS Data Set is quite large. Otherwise, the added cost of buffering through the same data set twice exceeds the cost foregone, which are the moves from the buffer to working storage that were avoided.

Also, with the addition of the Where Statement, SAS has made this approach redundant except when nested evaluations are required.

There is one situation, however, in which Random-Access is vital: the writing of customized reports that have multiple physical pages for each logical page.

Imagine a data set with 12 months of numeric data, but only a maximum of five months can fit on one page. The Data Null Step needs to print three physical pages (5, 5, and 2 months) of data for each logical page (12 months) of data. Only Random-Access can create such a report in one Data Step, from one SAS Data Set. Here is such a Data Step:

```
DATA NULL ;
FILE PRINT HEADER = HEAD;
/* INITIALIZE CONTROL VARIABLES */
DATA_END = 0; /* INDICATOR OF END OF DATA */
DATA_CNT = 1; /* CURRENT LINE OF INPUT DATA */
FIRST = 1; /* LINE LOCATION OF START OF LOGICAL PAGE */
LINE_CNT = 1; /* COUNT OF PHYSICAL PAGE LINES PRINTED */
PAGE_CNT = 1; /* COUNT OF PHYSICAL PAGES PRINTED PER LOGICAL PAGE */
/* TERMINATE PROCESSING AT END OF DATA */
DO UNTIL(DATA_END);
/* A CONTROL TO PREVENT ENDLESS LOOPING */
IF DATA_CNT GT 5000 THEN
  STOP;
SET WORK2 POINT=DATA_CNT NOBS=TOT_OBS;
```

```

/* BYPASS CITY-CHECKING IF AT BEGINNING OF DATA */
IF DATA_CNT EQ 1 THEN;
/* ELSE START NEW LOGICAL PAGE IF CITY CHANGES */
ELSE IF CITYSTRT EQ 1 THEN
DO;
    PAGE_CNT = 1; /* SET PHYSICAL PAGE COUNT TO 1 */
    LINE_CNT = 1; /* SET PHYSICAL PAGE LINE COUNT TO 1 */
    FIRST = DATA_CNT; /* MARK BEGINNING OF LOGICAL PAGE */
    PUT _PAGE_; /* LINK TO HEADER SECTION AND RETURN */
END;
/* ELSE START NEW PHYSICAL PAGE IF PAGE LINE COUNT IS 1 */
ELSE IF LINE_CNT EQ 1 THEN
    PUT _PAGE_; /* LINK TO HEADER SECTION AND RETURN */
PUT @ 1 DEPT $20.
@ ;
IF PAGE_CNT = 1 THEN
    PUT @ 25 M1 - M5 (COMMA10.2 +5);
ELSE IF PAGE_CNT = 2 THEN
    PUT @ 25 M6 - M10 (COMMA10.2 +5);
ELSE
    PUT @ 25 M11 - M12 (COMMA10.2 +5);
DATA_CNT + 1; /* ADD 1 TO DATA COUNT */
LINE_CNT + 1; /* ADD 1 TO PAGE LINE COUNT */
/* TERMINATE LOOP IF AT END OF DATA SET */
IF DATA_CNT GT TOT_OBS THEN
    DATA_END = 1;
/* ELSE START NEW PAGE IF AT END OF CURRENT PHYSICAL PAGE */
ELSE IF LINE_CNT GT 55 THEN
DO;
    /* START NEW LOGICAL PAGE IF AT END OF LAST PHYSICAL PAGE */
    IF PAGE_CNT = 3 THEN
    DO;
        PAGE_CNT = 1;
        FIRST = DATA_CNT;
    END;
    /* ELSE START NEW PHYSICAL PAGE */
    ELSE
    DO;
        /* SET DATA LINE TO START OF LOGICAL PAGE */
        DATA_CNT = FIRST;
        PAGE_CNT + 1;
    END;
    LINE_CNT = 1;
END;
END;
STOP; /* TERMINATE PROCESSING */
/* PAGE HEADING */
HEAD:
    PUT @30 'DEPARTMENT MONTHLY EXPENSE REPORT' /
    @15 CITY $20. // ;
RETURN;
RUN;

```

When using Random-Access in printing multiple physical pages per logical page, the Data Step must keep track of the starting line position for each logical page so that it can return to that data line when printing the second, third, and other physical pages of that logical page. It also must know which physical page it is printing so that it knows which group of columns to print on that page.

Furthermore, it must know whether it has completed printing a physical page so that it can determine whether to print a new physical page or a new logical page. These requirements are satisfied in the above Data Step by the variables FIRST, LINE_CNT, and PAGE_CNT.

Random-Access is very useful, but restrictive. The programmer must use a STOP Statement or the Step will never end. Also, a BY Statement cannot be used. If the programmer wishes to employ the capabilities of the FIRST. and LAST. variables, he must create variables in the data set that contain their 1 or 0 values. One method involves creating them in an earlier Data Step using the FIRST. and LAST. variables themselves. Below is an example:

```
DATA WORK2;
  DO UNTIL(EOF);
    SET WORK1 END=EOF;
    BY CITY DEPT;
    IF FIRST.CITY THEN
      CITYSTRT = 1;
    ELSE CITYSTRT = 0;
    IF FIRST.DEPT THEN
      DEPTSTRT = 1;
    ELSE DEPTSTRT = 0;
    OUTPUT;
  END;
  STOP;
RUN;
```

MATCH-MERGING WITHOUT USING THE MERGE STATEMENT

The SAS MERGE Statement is a very effective method of match-merging data when combined with a BY Statement. It not only match-merges two data sets, it also keeps track of the non-matches in those data sets and from which data set the non-matches originated.

However, if the non-matches are unimportant, the MERGE statement expends processing which is unnecessary and can be easily avoided. Here is a Data Step which merges two SAS Data Sets without using the MERGE Statement.

```
DATA WORK3; /* MATCH-MERGING OF TWO SAS DATA SETS */
  LENGTH WS_PARTS $15;
  W2_PARTS = ' ';
  DO UNTIL(W1_EOF);
    /* READ INQUIRY FILE */
    SET WORK1 END = W1_EOF;
    IF W1_PARTS EQ W2_PARTS THEN
      OUTPUT;
    /* READ MASTER FILE */
    DO WHILE(W2_PARTS LE W1_PARTS AND
      NOT W2_EOF);
      SET WORK2 END=W2_EOF;
      IF W1_PARTS EQ W2_PARTS THEN
        OUTPUT;
      END;
    IF W2_EOF THEN
      STOP;
  END;
  STOP;
RUN;
```

The MERGE Statement cannot match-merge external files, but this Data Step technique can. Such a Data Step avoids building unnecessary intermediate SAS Data Sets and can avoid moving unwanted external data into working storage.

Here is an example:

```
DATA WORK3; /* MATCH MERGING OF TWO EXTERNAL FILES */
  LENGTH WS_PARTS $15;
  W2_PARTS = ' ';
  DO UNTIL(W1_EOF);
    /* READ INQUIRY FILE */
    INFILE WORK1 END=W1_EOF;
    INPUT @ 1 W1_PARTS $15.0;
    IF W1_PARTS EQ W2_PARTS THEN
      OUTPUT;
```

```

/* READ MASTER FILE */
DO WHILE(W2_PARTS LE W1_PARTS AND
        NOT W2_EOF);
/* READ ONLY THE W2_PARTS FIELD */
INFILE WORK2 END=W2_EOF:
INPUT @ 1 W2_PARTS $15.0
      @ ;
/* IF THE PARTS MATCH, READ THE */
/* REST OF THE RECORD AND OUTPUT */
IF W1_PARTS EQ W2_PARTS THEN
DO;
    INPUT @20 TRANSCTN $ 4.0
          @25 PLANT    $ 4.0
          @30 DEPT     $ 4.0
          @35 COST     10.2
          ;
    OUTPUT;
END;
/* ELSE FLUSH THE RECORD */
ELSE
    INPUT;
END;
IF W2_EOF THEN
    STOP;
END;
STOP;
RUN;

```

The essence of the two Data Steps is the same:

```

Read first data set.
If a match occurs with the part in the second data set, output.
If the part in the second data set is less than or equal to the
part in the first data set, then
    Read second data set
    Output if a match occurs
    Stop reading when the part in the second data set becomes
    greater than the part in the first data set.
Return to top of loop.

```

Notice the use of the training @ sign on the INPUT Statement of the second Data Step. It holds the input pointer on that line until the existence of a match is determined. If a match exists, the rest of the data on that line is moved into working storage. Otherwise, a null INPUT Statement is issued so that the pointer is released, which allows the next INPUT Statement to read a new line of data.

It is the use of the two INPUT Statements, the first with a trailing @ sign, which keeps unneeded data from being moved into working storage.

ASSIGNING AND SELECTING OBSERVATIONS USING THE PUT FUNCTION

Many programmers use the IF-THEN-ELSE statement or the SELECT Statement to assign and select observations based upon value of another variable, particularly when nested evaluations are required, such as

```
IF      STATE = 'ARK' THEN
DO;
    IF      COUNTY = 'GARLAND'      THEN
        RANK = '05';
    ELSE IF COUNTY = 'HOWARD'      THEN
        RANK = '02';
END;
ELSE IF STATE = 'TEX' THEN
DO;
    IF      COUNTY = 'TARRANT'      THEN
        RANK = '10';
    ELSE IF COUNTY = 'DALLAS'      THEN
        RANK = '01';
END;
IF RANK = '10' THEN
    OUTPUT;
```

The PUT function could have been used here instead. A PROC FORMAT could have been written for RANK which combined the State and County values:

```
PROC FORMAT;
    VALUE $ RANK
        'ARKGARLAND' = '05'
        'ARKHOWARD'  = '02'
        'TEXTARRANT' = '10'
        'TEXDALLAS'  = '01'
    ;
RUN;
```

Then, a Data Step could have concatenated the two fields and used the PUT Function and \$RANK to assign values for RANK.

```
DATA WORK2;
    LENGTH RANK $10;
    STATECTY = RIGHT(STATE)||LEFT(COUNTY);
    RANK = PUT(STATECTY,$RANK02.0);
    IF RANK = 10 THEN
        OUTPUT;
RUN;
```

This approach has a major advantage over the IF-THEN-ELSE or SELECT methods when a large amount of data is to be assigned values. It reduces the number of evaluations per data line.

The IF-THEN-ELSE or SELECT approach passes a line of data through every primary (outer loop) evaluation until a match occurs there. It then passes that line through all evaluations within that loop until match occurs.

If the IF-THEN-ELSE or SELECT approach included all 50 states and an average of 100 counties per state, the average number of evaluations per line would be 25 (primary) + 50 (secondary) or 75. If there were one million lines in the input data set, the total number of evaluations would be 75 million, quite intensive. But with the PUT Function, the average number of evaluations per line would be 12 ($\log(50 \times 100) / \log(2)$) because of the binary search technique SAS uses to find matches for FORMAT values. That would total to 12 million evaluations, 63 million less than the IF-THEN-ELSE or SELECT approach.

Setting up a FORMAT for \$RANK would create a table with 5000 entries, an additional processing task, but one that is far less than the processing of 63 million evaluations. Still, if the criteria is fairly constant, the format can be saved and the format-building expense spread over several runs.

Finally, using SAS code to create a PROC FORMAT from external files is a fairly easy task, avoids boring and time-consuming typing, and can be used even when the external file has numeric ranges.

The following code creates a PROC FORMAT from an external file with this structure:

```
ARK    1    5  03
ARK    6   10  04
ARK   11   15  02
ARK   16   20  03
```

```

DATA NULL ;
  /* FILE TO CONTAIN PROC FORMAT */
  FILE TEMP;
  INFILE RANKING END=EOF;
  LENGTH STATECTY $ 5;
  PUT 'PROC FORMAT;';
  PUT ' VALUE $RANK ';
  DO UNTIL(EOF);
    INPUT @ 1 STATE      $ 3.0
          @ 5 COUNTY1    3.0
          @10 COUNTY2    3.0
          @15 COUNTY2    $ 2.0
          ;
    DO J = COUNTY1 TO COUNTY2;
      STATECTY = RIGHT(STATE)||LEFT(PUT(J,3.0));
      PUT "" STATECTY "" ' = ' "" RANK "";
    END;
  END;
  PUT ' ';
  PUT 'RUN;';
  STOP;
RUN;

```

This code generates an external file of PROC FORMAT code that looks like this:

```

PROC FORMAT;
  VALUE $RANK
    ARK1 = '03'
    ARK2 = '03'
    ARK3 = '03'
    ARK4 = '03'
    ARK5 = '03'
    ARK6 = '04'
    ARK7 = '04'
  ;
RUN;

```

The PROC FORMAT is passed back into the SAS program using a %INCLUDE Statement attached to the DD name of the file containing the PROC FORMAT.

```

%INC TEMP;

```

A MACRO TO SORT AND SUM DATA

```
%MACRO COMPUTE(VAR1=, VAR2=, VAR3=, VAR4=, VAR5=,  
                VAR6=, VAR7=, VAR8=, VAR9=, VAR10=,  
                KEEPVARS=,GRANDSUM=,IN=,OUT=,ADDVARS=,VARNUM=,DEPTH=);
```

Many programs sort and sum data and then write a report for that summed data. The sums are often multi-level totals of the same basic data. For example, a report on expenses by major cost elements of a company would probably begin with a summary of the costs of those elements for the entire company.

Wilson Box Company
17 July 1993
Cost Report

Research	10,000,000.00
Materials	100,000,000.00
Manufacturing	250,000,000.00
Shipping	25,000,000.00
TOTAL	<u>385,000,000.00</u>

The next summary might be by CITY.

Wilson Box Company
17 July 1993
Cost Report
Houston

Research	5,000,000.00
Materials	50,000,000.00
Manufacturing	125,000,000.00
Shipping	12,000,000.00
TOTAL	<u>172,000,000.00</u>

The next might be by DIVISION, within CITY.

Wilson Box Company
17 July 1993
Cost Report
Houston
Crates for Freights

Research	1,000,000.00
Materials	10,000,000.00
Manufacturing	25,000,000.00
Shipping	1,000,000.00
TOTAL	<u>37,000,000.00</u>

The data for all of these reports can be generated from one sort and four Data Steps:

```
PROC SORT DATA = WORK1
      OUT = WORK2;
  BY ELEMENT CITY DIVISION;
RUN;

DATA WORK3(KEEP=ELEMENT CITY DIVISION COST);
  SET WORK2;
  BY ELEMENT CITY DIVISION; /* CITY AND DIVISION TOTALS */
  TOTAL + COST;
  IF LAST.DIVISION THEN
    DO;
      COST = TOTAL;
      TOTAL = 0;
      OUTPUT;
    END
RUN;

DATA WORK4(KEEP=ELEMENT CITY COST);
  SET WORK2;
  BY ELEMENT CITY;          /* CITY TOTALS */
  TOTAL + COST;
  IF LAST.CITY THEN
    DO;
      COST = TOTAL;
      TOTAL = 0;
      OUTPUT;
    END
RUN;

DATA WORK4(KEEP=ELEMENT COST);
  SET WORK2;
  BY ELEMENT;
  TOTAL + COST;
  IF LAST.ELEMENT THEN /* TOTALS FOR ALL CITIES */
    DO;
      COST = TOTAL;
      TOTAL = 0;
      OUTPUT;
    END
RUN;

DATA ALL;
  SET WORK4 /* TOTALS FOR ALL CITIES */
      WORK3 /* CITY TOTALS */
      WORK2; /* DIVISION AND CITY TOTALS */
  OUTPUT;
RUN;
```

Each sum has one common variable (ELEMENT) and shares the same sort order (ELEMENT CITY DEPARTMENT). Because of that, one sort provides the basic order and then each higher-level sum can be derived from the sum just below it.

Dropping each lower-level sort variable when it is no longer needed causes it to have blank values in the higher-level summaries when the data sets are concatenated. Thus, DIVISION has blank values in the lines that represent the totals by ELEMENT (all Cities) and the totals by ELEMENT and CITY, while CITY has blank values in the totals by Element.

For a top-level-to-detail report, then, the WORK2 data can be sorted by CITY, DIVISION, and ELEMENT and the summaries will stack as desired because blanks sort above non-blanks. (If blanks exist naturally in your data, your summing approach will have to assign a special non-blank character to each variable that represents a total, one that sorts above blank in the EBCDIC sort sequence.)

Because the process is repetitive, it is ideal for a Macro. Build your own Macro from scratch or use this %COMPUTE Macro as a guide.

These keyword entries in %COMPUTE would have generated the above summaries of CITY, DIVISION, and ELEMENT.

```
%COMPUTE(IN=WORK1,OUT=WORK2,
          VAR1=ELEMENT,VAR2=CITY,VAR3=DIVISION,
          ADDVARS=COST,VARNUM=1,DEPTH=3,GRANDSUM=NO)
```

The IN, OUT, ADDVARS, and VARNUM keywords are required. DEPTH defaults to the number of sort variables specified. GRANDSUM defaults to NO. At least one VAR keyword must be entered unless GRANDSUM is YES.

A single-level summary would have required a DEPTH of one (1).

```
%COMPUTE(IN=WORK1,OUT=WORK2,
          VAR1=ELEMENT,VAR2=CITY,VAR3=DIVISION,
          ADDVARS=COST,VARNUM=1,DEPTH=1,GRANDSUM=NO)
```

Summaries of more than one variable would have required different ADDVARS and VARNUM keywords.

```
%COMPUTE(IN=WORK1,OUT=WORK2,
          VAR1=ELEMENT,VAR2=CITY,VAR3=DIVISION,
          ADDVARS=COST REVENUES,VARNUM=2,DEPTH=1,GRANDSUM=NO)
```

The %COMPUTE Macro is shown below. Read the comments to understand how to enter the keyword parameter values.

```

/*****
/*
/* THE COMPUTE MACRO CREATES PROGRESSIVE (MULTI-LEVEL)
/* SUMMARIES OF A SAS DATA SET, SIMPLE SUMMARIES OF A SAS DATA
/* SET, AND/OR GRAND TOTALS OF A SAS DATA SET. A YES ENTRY INTO
/* THE GRANDSUM KEYWORD CREATES A GRAND TOTAL REGARDLESS OF ANY
/* OTHER ENTRY. THE ORDER OF ENTRY OF VARIABLE NAMES INTO THE
/* VAR1 - VAR20 KEYWORDS ESTABLISHES THE SORT STRUCTURE OF THE
/* SUMMARIZATION.
/* THE INPUT DATA SET FOR THE MACRO IS ENTERED INTO THE IN
/* KEYWORD. THE OUTPUT DATA SET IS ENTERED INTO THE OUT KEYWORD.
/* THE DATA IN THE OUTPUT DATA SET IS SORTED WITH THE HIGHEST
/* LEVEL SUMMARY DATA AT THE TOP AND THE LOWEST LEVEL SUMMARY at
/* THE BOTTOM.
/* IF THE DEPTH KEYWORD IS LEFT BLANK, THE MACRO WILL GENERATE
/* A PROGRESSIVE SUMMARY EQUAL TO THE NUMBER OF ENTRIES MADE INTO
/* THE VAR1 - VAR20 KEYWORDS; E.G., FOUR ENTRIES, A PROGRESSIVE
/* SUMMARY OF FOUR. OTHERWISE, THE PROGRESSIVE SUMMARY WILL BE
/* EQUAL TO THE NUMBER ENTERED INTO THE DEPTH KEYWORD. FOR
/* EXAMPLE, AN ENTRY OF 1 (ONE) WILL CREATE A SIMPLE SUMMARY.
/* TWO (2), A TWO-LEVEL PROGRESSIVE SUMMARY.
/* THE NAMES AND NUMBER OF NUMERIC VARIABLES TO BE ADDED ARE
/* SPECIFIED IN THE ADDVARS AND VARNUM KEYWORDS. IF NO VARIABLES
/* ARE ENTERED INTO THE KEEPVARS KEYOWRD, ONLY THE SORT
/* VARIABLES AND ADD VARIABLES ARE KEPT FROM THE INPUT DATA SET.
/* TO REDUCE THE LIKELIHOOD OF FAILURE DUE TO LACK OF SPACE,
/* THIS MACRO USES A SEPARATE SEQUENTIAL WORK AREA TO HOLD
/* SORTED INPUT DATA SET OF THE MACRO. THIS AREA IS IDENTIFIED
/* BY A DDNAME OF TAPE2 IN THE JCL.
*****/
%MACRO COMPUTE(VAR1=, VAR2=, VAR3=, VAR4=, VAR5=,
               VAR6=, VAR7=, VAR8=, VAR9=, VAR10=,
               VAR11=,VAR12=,VAR13=,VAR14=,VAR15=,
               VAR16=,VAR17=,VAR18=,VAR19=,VAR20=,
               KEEPVARS=,GRANDSUM=,IN=,OUT=,ADDVARS=,VARNUM=,DEPTH=);
  %LET NUMVAR = 0;
  %LET J = 1;
  %DO %WHILE(&&VAR&J NE );
    /* ISSUE A STRING SHOWING VARIABLES SELECTED */
    %PUT ZZZZZ &J &&VAR&J;
    %IF &&VAR&J NE %THEN
      %DO;
        %LET NUMVAR = &J;
        %LET Z      = &J;
      %END;
    %LET J = %EVAL(&J+1);
  %END;

```

```

/*****
/*  IF GRANDSUM IS NOT MARKED YES AND THE NUMBER OF SORT */
/*  VARIABLES ARE ZERO (0), A MESSAGE IS GENERATED AND   */
/*  THE PROGRAM IS ABORTED.                               */
*****/
%IF &GRANDSUM NE YES AND
    &NUMVAR EQ 0 %THEN
    %DO;
        DATA NULL ;
        FILE PRINT;
        PUT "COMPUTE MACRO WAS CALLED WITHOUT SORT      " /
            "VARIABLES OR GRANDSUM BEING SPECIFIED.      " /
            "PROGRAM WAS ABORTED."                      ;
        ABORT;

        RUN;
    %END;

/*****
/*  WHEN BLANK, THE VALUE OF DEPTH IS CALCULATED FROM THE */
/*  THE NUMBER OF SORT VARIABLES SPECIFIED.               */
*****/
%IF &DEPTH = %THEN
    %LET DEPTH = &VARNUM;

/*****
/*  IF THE DEPTH OF SUMMING EXCEEDS THE NUMBER OF SORT   */
/*  VARIABLES SPECIFIED, AN ERROR MESSAGE IS GENERATED AND */
/*  THE PROGRAM IS ABORTED.                               */
*****/
%ELSE %IF &DEPTH GT &VARNUM %THEN
    %DO;
        DATA NULL ;
        FILE PRINT;
        PUT "DEPTH OF SUMMARIZATION REQUESTED (&DEPTH) " /
            "EXCEEDED THE NUMBER OF SUM VARIABLES (&VARNUM)." /
            "REDUCE DEPTH TO NO MORE THAN &X."          /
            "PROGRAM WAS ABORTED."                      ;
        ABORT;

        RUN;
    %END;
%ELSE %IF &DEPTH LT &X %THEN
    %DO;
        %PUT SUMMARIZATION DEPTH OF &DEPTH WAS USED,;
        %PUT AS REQUESTED EXPLICITLY IN COMPUTE MACRO. ;
    %END;

```

```

/*****
/*  WHEN THE DEPTH EXCEEDS 1, THE INCOMING DATA SET IS  */
/*  READ INTO A DATA SET ESTABLISHED WITHIN THE COMPUTE  */
/*  MACRO AND SORTED BY THE VARIABLES ENTERED IN THE     */
/*  VAR1 - VAR20 PARAMETERS OF THE COMPUTE MACRO.        */
*****/
%IF &DEPTH GE 1 %THEN
  %DO;
    PROC SORT DATA = &IN(KEEP=&VAR1  &VAR2  &VAR3  &VAR4
                          &VAR5  &VAR6  &VAR7  &VAR8
                          &VAR9  &VAR10 &VAR11 &VAR12
                          &VAR13 &VAR14 &VAR15 &VAR16
                          &VAR17 &VAR18 &VAR19 &VAR20
                          &KEEPVARS  &ADDVARS)
      /* NOTICE THE SEPARATE WORK AREA FOR THE */
      /* OUTPUT SORT DATA SET.                */
      OUT = TAPE2.A1;
      BY &VAR1  &VAR2  &VAR3  &VAR4  &VAR5
         &VAR6  &VAR7  &VAR8  &VAR9  &VAR10
         &VAR11 &VAR12 &VAR13 &VAR14 &VAR15
         &VAR16 &VAR17 &VAR18 &VAR19 &VAR20;

    RUN;
  %END;
/*****
/*  A DEPTH OF 1 MEANS THAT ONLY ONE LEVEL OF SUMMING  */
/*  WILL BE DONE. IN THAT CASE, UNLESS A GRAND SUM HAS  */
/*  BEEN REQUESTED, THIS PROGRAM CONSERVES RESOURCES BY  */
/*  NOT CREATING ANY INTERMEDIATE SUMMED DATA SETS. IT  */
/*  BUILDS ONLY THE DATA SET TO BE OUTPUT BY THE MACRO. */
*****/
%IF &DEPTH = 1 %THEN
  %DO;
    %IF &GRANDSUM = YES %THEN
      %DO;
        %LET A = A;
        %LET C = A;
        %LET ALLDATA = B A;
      %END;
    %ELSE
      %DO;
        %LET A = &OUT;
      %END;
    %LET SORTDATA = &A;
  %END;

```

```

/*****
/*  THIS STEP SUMS THE SORTED INCOMING DATA AND      */
/*  MOVES IT TO THE OUTPUT DATA SET SPECIFIED IN THE */
/*  OUT PARAMETER OF THE COMPUTE MACRO.                */
*****/
DATA &OUTDATA(DROP=I A1-A&VARNUM);
  RETAIN A1-A&VARNUM 0;
  ARRAY A(&VARNUM) A1-A&VARNUM;
  ARRAY B(&VARNUM) &ADDVARS;
  DO UNTIL(EOF);
    SET TAPE2.A1(KEEP=&VAR1  &VAR2  &VAR3  &VAR4  &VAR5
                  &VAR6  &VAR7  &VAR8  &VAR9  &VAR10
                  &VAR11 &VAR12 &VAR13 &VAR14 &VAR15
                  &VAR16 &VAR17 &VAR18 &VAR19 &VAR20
                  &KEEPVARS  &ADDVARS)

    END = EOF;
  BY &VAR1  &VAR2  &VAR3  &VAR4  &VAR5
    &VAR6  &VAR7  &VAR8  &VAR9  &VAR10
    &VAR11 &VAR12 &VAR13 &VAR14 &VAR15
    &VAR16 &VAR17 &VAR18 &VAR19 &VAR20;
  DO J = 1 TO &VARNUM;
    A(J) + B(J);
    IF LAST.&&VAR&Z THEN
      DO;
        B(J) = A(J);
        A(J) = 0;
      END;
    END;
  IF LAST.&&VAR&Z THEN
    OUTPUT;
  END;
  STOP;
  RUN;
ZEND;

```

```

/*****
/*  WHEN THE DEPTH OF SUMMARIZATION IS GREATER THAN 1, THE */
/*  SUM STEP IS REPEATED ONCE FOR EACH LEVEL OF DEPTH    */
/*  REQUESTED. THE SUMMING STARTS WITH ALL SORT VARIABLES */
/*  SPECIFIED IN THE VAR1 - VAR20 PARAMETERS AND DROPS ONE */
/*  SORT VARIABLE IN EACH SUBSEQUENT STEP. THE VARIABLES ARE */
/*  DROPPED FROM THE RIGHT; I.E., FROM THE HIGHEST NUMBERED */
/*  SORT PARAMETER SPECIFIED.                               */
/*  THIS PROGRESSIVE DROPPING OF VARIABLES DURING THE     */
/*  SUMMARIZATION PROCESS CAUSES THE DROPPED VARIABLES TO  */
/*  HAVE BLANK VALUES IN THE TOP-LEVEL SUMMARY DATA WHEN */
/*  ALL OF THE SUMMARY DATA SETS ARE CONCATENATED TOGETHER */
/*  AT THE END OF THE MACRO.                                */
*****/
%ELSE %IF &DEPTH GT 1 %THEN
  %DO;
    %LET INDATA = %STR(TAPE2.);
    %DO J = 1 %TO &DEPTH;
      %LET K = %EVAL(&J+1);
      DATA A&K(DROP=I A1-A&VARNUM);
        RETAIN A1-A&VARNUM 0;
        ARRAY A(&VARNUM) A1-A&VARNUM;
        ARRAY B(&VARNUM) &ADDVARS;
        DO UNTIL(EOF);
          SET &INDATA.A&J(KEEP=&VAR1 &VAR2 &VAR3 &VAR4
                                &VAR5 &VAR6 &VAR7 &VAR8
                                &VAR9 &VAR10 &VAR11 &VAR12
                                &VAR13 &VAR14 &VAR15 &VAR16
                                &VAR17 &VAR18 &VAR19 &VAR20
                                &KEEPVARS &ADDVARS)
          END = EOF;
          BY &VAR1 &VAR2 &VAR3 &VAR4 &VAR5
             &VAR6 &VAR7 &VAR8 &VAR9 &VAR10
             &VAR11 &VAR12 &VAR13 &VAR14 &VAR15
             &VAR16 &VAR17 &VAR18 &VAR19 &VAR20;
          DO J = 1 TO &VARNUM;
            A(J) + B(J);
            IF LAST.&&VAR&Z THEN
              DO;
                B(J) = A(J);
                A(J) = 0;
              END;
            END;
          IF LAST.&&VAR&Z THEN
            OUTPUT;
          END;
        STOP;
        RUN;
        %LET VAR&Z = ;
        %LET Z = %EVAL(&Z-1);

```

```

/*****/
/* AFTER EACH SUMMED DATA SET IS CREATED, THE */
/* NAME OF THE SUMMED DATA SET IS ADDED TO THE */
/* MACRO VARIABLE ALLDATA. AFTER ALL SUMMED DATA */
/* SETS HAVE BEEN CREATED, THEY ARE CONCATENATED */
/* INTO THE OUTPUT DATA SET IN OPPOSITE ORDER OF */
/* THEIR CREATION. THIS CAUSES THE DATA IN THE */
/* OUTPUT DATA SET TO BE SORTED FROM LEAST DETAIL */
/* TO MOST DETAIL. IT ALSO ENSURES THAT THE OUTPUT */
/* DATA SET WILL CONTAIN ALL VARIABLES IN THE */
/* INITIAL SORT DATA SET. */
/*****/
%IF &J = 1 %THEN
    %LET ALLDATA = A&K;
%ELSE %LET ALLDATA = A&K &ALLDATA;
%LET INDATA = ;
%END;
%IF &GRANDSUM = YES %THEN
    %DO;
        %LET ALLDATA = B &ALLDATA;
        %LET C = A&K;
    %END;
    %LET SORTDATA = A&K;
%END;
/*****/
/* IF GRANDSUM WAS SPECIFIED IN THE COMPUTE MACRO CALL, */
/* THE LAST SUMMARY DATA SET BUILT BY THE MACRO WILL BE */
/* USED TO CREATE A GRAND TOTAL LINE. IN THE SITUATION */
/* WHERE ONLY GRANDSUM WAS REQUESTED, NO SUMMARY DATA SETS */
/* WILL EXIST AND THE INPUT DATA SET WILL BE USED TO CREATE */
/* GRAND TOTAL. */
/*****/
%IF &GRANDSUM = YES %THEN
    %DO;
        %IF &DEPTH = %THEN
            %DO;
                %LET B = &OUT;
                %LET C = &IN;
            %END;
        %ELSE
            %LET B = B;
        DATA &B(DROP=I A1-A&VARNUM);
        RETAIN A1-A&VARNUM 0;
        ARRAY A(I) A1-A&VARNUM;
        ARRAY B(I) &ADDVARS;
    %END;

```

```

DO UNTIL(EOF);
  SET &C(KEEP=&ADDVARS)
  END=EOF;
DO J = 1 %TO &ADDVARS;
  A(&ADDVARS) + B(&ADDVARS);
  IF EOF THEN
    DO;
      B(&ADDVARS) = A(&ADDVARS);
      A(&ADDVARS) = 0;
    END;
  END;
  IF EOF THEN
    OUTPUT;
  END;
  STOP;
  RUN;
%END;
/*****
/* THE LAST ACTION OF THE MACRO IS TO CREATE A DATA STEP */
/* THAT CONCATENATES ALL OF THE INTERMEDIATE DATA SETS */
/* CREATED BY THE MACRO.
*****/
%IF (&DEPTH = 1 AND &GRANDSUM = YES) OR
  &DEPTH GT 1 %THEN
  %DO;
    DATA &OUT;
    DO UNTIL(EOF);
      SET &ALLDATA END=EOF;
      OUTPUT;
    END;
    STOP;
    RUN;
  %END;
%END COMPUTE;

```