

Building Code That is Easy to DeBug and Maintain
Jimmy DeFoor
General Dynamics - Fort Worth

Essential Elements of Easy-To-Maintain Code

Conforms to the flow of the SAS Data Step
Reuses proven code
Uses only tested code
Each line has a clear and simple purpose
Sections are relatable to a known process
Approach is consistent from Step to Step
Covers all the possible conditions of the data
Is properly documented
Structured for easy modification by a full-screen editor

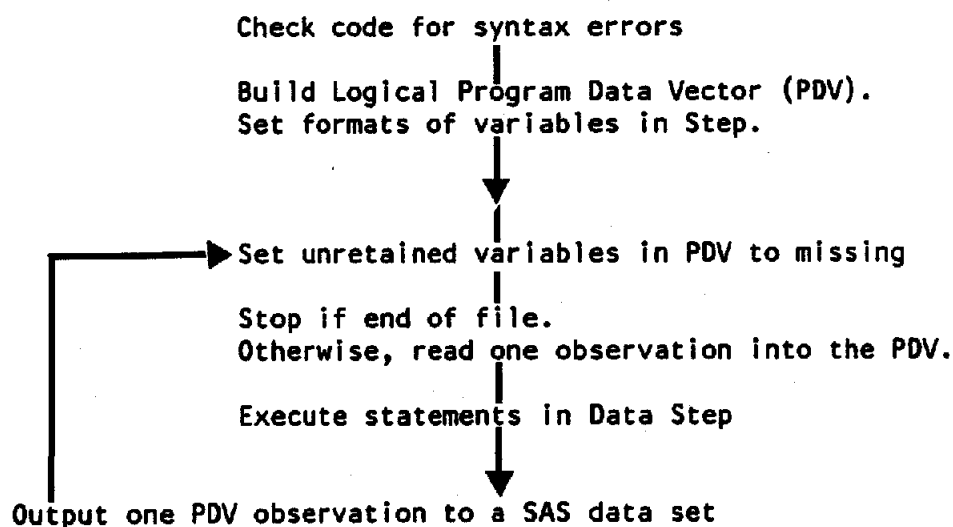
This paper discusses the essential elements of easy-to-maintain code and describes how a full-screen editor can be best used to modify code.

Conforms to the flow of the SAS Data Step

Code that conforms to the processing flow of the SAS Data Step is

Easier to debug
More predictable (less likely to break)

The performance process of the normal SAS Data Step under Version 6.06 can be described as:



The Logical Program Data Vector (PDV) is the work area into which each line of data, one at a time, is read and then manipulated. It is composed of five physical areas. The contents of the Logical PDV is copied to the output data set when an OUTPUT statement is encountered.

The above elements of the SAS Data Step occur at only certain locations in the SAS code:

```
DATA WORK2;                                {Check syntax of code in step.}
                                           {Establish Program Data Vector (PDV).}
                                           {Format variables used in step.}

                                           {Start processing of data.}
                                           {Set unretained variables to missing.}

    SET WORK1;                             {Stop if End of File.}
    (INPUT @ 1 RATE, ETC.)                 {Otherwise read data into PDV.}

    PROFIT = RATE * COST;                  {Execute programming statements.}
    OUTPUT;                               {Output PDV to SAS data set.}
RUN;
```

Knowing where data is read and output indicates the best locations for debugging statements:

- * Just before a line of data is read into the PDV.
- * Just after a line of data is read into the PDV.
- * Just after the PDV is output.

A good debugging statement is one that describes where the statement was encountered and shows the value of all relevant variables at that point.

```
DATA WORK2;
    PUT '1 ' _ALL_ ;
    SET WORK1;
    PUT '2 ' _ALL_ ;
    PROFIT = RATE * COST;
    OUTPUT;
    PUT '3 ' _ALL_ ;
RUN;
```

The ALL variable is a special variable that tells SAS to print the contents of all variables in the PDV, including the N variable that states which line of the input data set is being manipulated.

The three PUT ALL statements will generate this listing in the SAS LOG when reading a SAS data set of two observations.

```
1 RATE=.   COST=.   PROFIT=.   _ERROR_=0 _N_=1
2 RATE=.18 COST=1000 PROFIT=.   _ERROR_=0 _N_=1
3 RATE=.18 COST=1000 PROFIT=180 _ERROR_=0 _N_=1
1 RATE=.18 COST=1000 PROFIT=.   _ERROR_=0 _N_=2
2 RATE=.15 COST=2000 PROFIT=.   _ERROR_=0 _N_=2
3 RATE=.15 COST=2000 PROFIT=380 _ERROR_=0 _N_=2
1 RATE=.15 COST=2000 PROFIT=.   _ERROR_=0 _N_=3
```

Look particularly at the line printed by the first (1) PUT ALL statement, which shows the content of the PDV before the next line of data is read from the SAS data set.

All variables in the PDV, including those from the SAS data set, are missing (or blank) before the first line of the data set is read. When the next line of data is to be read, however, only the the Step-created variables are set to missing. The values of the SAS data set variables are retained until overwritten by the next line of data.

Finally, notice that the last debugging line issued by the Step is also the first (1) PUT ALL statement. Its N value of 3 indicates that the PDV was emptied before the Step ended.

The SAS LOG will show these results when reading an external file of two observations:

```

1 RATE=.    COST=.    PROFIT=.    _ERROR_=0  _N_=1
2 RATE=.18  COST=1000  PROFIT=.    _ERROR_=0  _N_=1
3 RATE=.18  COST=1000  PROFIT=180   _ERROR_=0  _N_=1
1 RATE=.    COST=.    PROFIT=.    _ERROR_=0  _N_=2
2 RATE=.15  COST=2000  PROFIT=.    _ERROR_=0  _N_=2
1 RATE=.    COST=.    PROFIT=.    _ERROR_=0  _N_=3

```

Compare this print output with that from the SAS data set. See that the only difference is that the variables from the external file are also set to missing just before the next line of data is read, which occurs just before the INPUT statement. The location of the INFILE statement is unimportant to the values of the PDV.

Besides showing where debugging statements would be best located, knowing when and what is done during the SAS Data Step helps you write your code so that it produces the desired results.

For example, assignment statements placed before the data is read into the PDV will generate unexpected values. In the example below, a missing value for PROFIT will be created for the first line of output data. The last line of output will have erroneous values since the PROFIT value will not have been generated from those values of COST and RATE.

```

DATA WORK2;
  PROFIT = RATE * COST;
  PUT '1 ' _ALL_;
  SET WORK1;
  PUT '2 ' _ALL_; /* OUTPUT LINE */
RUN;

```

```

1 PROFIT=.    RATE=.    COST=.    _ERROR_=0  _N_=1
2 PROFIT=.    RATE=0.18  COST=1000  _ERROR_=0  _N_=1
1 PROFIT=180  RATE=0.18  COST=1000  _ERROR_=0  _N_=2
2 PROFIT=180  RATE=0.15  COST=2000  _ERROR_=0  _N_=2
1 PROFIT=300  RATE=0.15  COST=2000  _ERROR_=0  _N_=3

```

Reuses proven code

Reusing proven code, of course, ensures that code will work when it is unchanged. However, it also reduces coding time even when the code is altered. Less knowledge is required to modify code than to start from scratch.

But to get the most advantage out of this approach, your code must be kept short and simple - and it must be stored somewhere where it is easily retrievable by everyone who would use it.

An example would be a partitioned data set (PDS) in which there is a one member that explains the function of each member of the PDS and which all members have descriptive names.

```
EDIT --- FWUG.SAS.CODE-----
@README    {Your Table of Contents}
ARRAYS
COMPUTE
DOLOOP
DOUNTIL
INPUT
MATH
PRINT
REPORT
ROWADD
SELECT
SETPOINT
```

If this approach is fully implemented, you will have many utility (common application) steps that you can mix and match to build your application programs. For example, you might be able to build a new program with the code from these members.

```
INPUT
SELECT
MATH
ROWADD
PRINT
```

Uses only tested code

No code should be placed in a program until you know exactly how it will perform. Proven code meets that objective, but new code must be tested.

The simplest way to test your new code is to enter SAS DMS, perhaps from the command line of your full-screen edit facility, and load the code into the DMS Program Edit window using the INCLUDE command. Just remember to TSO ALLOCATE your test program before you attempt to INCLUDE it.

```
COMMAND ==> TSO ALLOC F(PRGM) DA(FWUG.SAS.CODE)      Program Editor
```

```
COMMAND ==> INCLUDE PRGM(TEST1)                      Program Editor
```

```
01 DATA _NULL_;          /* CODE TO TEST PUT FUNCTION */
02   LINENUM = 1000;
03   LINECHAR = PUT(LINENUM,4.);
04   IF LINECHAR = '1000' THEN;
05   ELSE
06     PUT LINECHAR= ;
07   RUN;
```

If you do change your code, you can replace the code in your test member using the SAVE command with the REP option.

```
COMMAND ==> SAVE PRGM(TEST1) REP                    Program Editor
```

Or, you can save it to another member that does not yet exist. SAS builds the new member for you.

```
COMMAND ==> SAVE PRGM(TEST2)                        Program Editor
```

If you are uncertain about the exact syntax of the command, or even if you are unclear about its specific function, use the HELP command from inside the Display Manager (DMS). Examples: HELP PRINT, HELP FORMAT.

Using DMS as Reference source is often faster than wrestling with a SAS manual.

Each line has a clear and simple purpose

Code should clearly describe each action it is performing and which data it is creating and using.

```
A = B * C;
```

does not communicate nearly as much as

```
EARNINGS = RATE * COST;
```

These variable names and their relationship have an inherent meaning to the reader.

Sections are relatable to a known process

Whenever possible, a section of code should reflect a recognized process of computing or calculating an understood value. This code follows the usual process of determining after-tax rental income.

```
BEFORETX = RENT - MORTGAGE;      /* BEFORE TAX EARNINGS */  
  
TXIMPACT = TAXRATE * (RECEIPTS - DEPRCATN); /* TAX IMPACT */  
  
AFTERTAX = BEFORETX - TXIMPACT; /* AFTER TAX EARNINGS */
```

This code, though mathematically correct, is certainly not easily relatable to the usual process of calculating rental earnings after taxes.

```
AFTERTAX = (RENT - MORTGAGE - (TAXRATE * (RENT - MORTGAGE - DEPRCATN)));
```

Consistent approach from Step to Step

Obviously, a reader of your code can anticipate the actions your code is performing much easier if you always do the same thing the same way. More than that, however, it can tell the reader of the code where to look for a particular problem in your code.

For example, imagine the SAS LOG of your program says 'Variable Not Defined' for a variable used in the calculation of another variable. Probably the error is occurring because the variable is not being kept from a previous Data Step.

If you always code your KEEP or DROP statements in a particular location, the reader of your code will know quickly where to go to check whether that is actually the problem. Otherwise, he or she will have to search each Step for both DROP and KEEP statements, since he or she will not know which (or both) were used.

Futhermore, if you never use DROP statements to exclude variables being passed to and from SAS data sets - and if you use KEEP= statements on both your input and output data sets - the reader of your code, and you, will always know what data came into and left that Step.

```
DATA WORK2(KEEP=DEPT YEAR COST BUDGET);  
  SET WORK1(KEEP=DEPT YEAR COST BUDGET VARIANCE);
```

Using this approach also provides good clues to the function of the Data Step. This Step must be calculating cost variance.

Covers all the possible conditions of the data

Make certain that your code doesn't have any loose ends. Cover all the bases.

For one, never assume that a numeric value may not be missing or zero. Code division statements like this

```
IF COST = 0 THEN
    RATE = 0;
ELSE
    RATE = PROFIT / COST;
```

Also, sum across observations with the SUM(OF function.

```
ROWTOTAL = SUM(OF YEAR1, YEAR2, YEAR3, YEAR4);
```

The SUM(OF function doesn't produce missing values in ROWTOTAL when any of the Year variables are missing, as will

```
ROWTOTAL = YEAR1 + YEAR2 + YEAR3 + YEAR4;
```

For another, don't treat data as if it has a limited universe of possibilities. Expect anything. Code IF statements so that they assign something to every possibility and/or print a message that a good value was not assigned.

```
IF      SEX = 'MALE'   THEN
    FACE = 'ROUGH';
ELSE IF SEX = 'FEMALE' THEN
    FACE = 'SMOOTH';
ELSE
    DO;
        PUT 'BAD CLASSIFICATION FOR ' SEX= ;
        FACE = 'ROUGHLY SMOOTH';
    END;
```

Is properly documented

Program documentation (not system documentation) provides the information needed to effectively maintain or modify the program. Five types of information are required.

- Why does the program exist?
- What is the program doing?
- How is it doing it?
- What data is it reading?
- What data is it creating?

A program is properly documented if the cost of the time spent documenting it is less than sum of the cost of

the additional time necessary to debug and understand it,

the loss of the productivity of the processes depending upon it.

As discussed earlier, code that is clear, consistent, and relatable to a known process is far easier to understand and predict. Also, clear, consistent, and relatable code is just as easy to develop as code that isn't - probably easier. Therefore, such code is really no-cost documentation.

Clear, consistent, and relatable code provides the information needed to answer the last four functions of program documentation. It also never gets out-of-sync with the actual actions of the program (as can line or Step documentation).

With the exception of the comments needed to clarify what the code itself cannot communicate, a statement of the purpose of the program is the only true documentation a properly-written program requires.

Structured for easy modification by a full-screen editor

Full-screen editors such as ISPF or XEDIT are very powerful code construction and modification tools, but only if you structure your code with their capabilities in mind.

This code structure is easy to modify and enhance, besides being easy to understand.

```
DATA WORK2(KEEP=TASKCNTR YEAR COST BUDGET);
  SET WORK1(KEEP=TASKCNTR YEAR COST BUDGET
            DEPT CLASS EARNINGS VARIANCE);
  LENGTH CLASS    $10
        VARIANCE  8
        ;
  VARIANCE = COST - BUDGET;
  IF      VARIANCE LT  0 THEN;
    /* NO ASSIGNMENT */
  IF      VARIANCE LE 100 THEN
    CLASS = 'SMALL';
  ELSE IF VARIANCE LE 200 THEN
    CLASS = 'MEDIUM';
  ELSE IF VARIANCE LE 300 THEN
    CLASS = 'LARGE';
  ELSE
    DO;
      CLASS = 'TOO LARGE';
      PUT 'VALUE IS OUT OF BOUNDS FOR ' _ALL_ ;
    END;
  RUN;
```

Notice that line continuations and nested statements are indented. Also, notice that assignment statements are separated from their evaluation (IF) statements.

The advantages of this structure are:

The KEEP= variables of the input data set can be used as a model for the KEEP= variables of an output data set.

Additional lines can be easily inserted without other lines having to be modified.

Null IF statements can be easily placed into IF THEN/ELSE structures.

The completeness of the linkage of an IF THEN/ELSE structure can be easily verified.

The alternative values assigned for a particular variable by an IF THEN/ELSE structure can be easily located. CLASS is either blank, SMALL, MEDIUM, LARGE, or TOO LARGE

More evaluations and assignments can be added by just replicating the existing evaluations and assignments and then changing them.

The nature and limits of nested DO groups are seen very quickly.

Additional assignments or evaluations can be added inside a DO/END group without disturbing the DO/END group itself.

Nesting DO/END groups in this manner also makes it very easy to check whether all DO/END groups are complete and accurately nested.

```
ELSE IF '20050' LE LINE LT '71200' THEN
DO;
  TYPE = '2'; /* DOLLARS */
  IF '20050' LE LINE LT '69999' THEN
  DO;
    IF '20050' LE LINE LT '29999' THEN /* LABOR DOLLARS */
    DO;
      IF LINE LT '20999' THEN
      DO;
        GROUP = '05';
        OUTPUT;
        LINE = '20999'; /* ENGINEERING SUBTOTAL */
        OUTPUT;
      END;
    END;
  END;
END;
END;
```

When the code is short, visual verification of the DO/END groups is easy. But when the code spans several screens, special techniques are necessary to verify the completeness of the DO/END groups.

```

. . . . .
EDIT ---- FW2L.SASLIB.CODE(TESTPRGM) - 01.99 ---- COLUMNS 001 072
COMMAND ==> SCROLL ==> CSR
000195     ELSE IF '20050' LE LINE LT '71200' THEN
000196     DO;
- - - - - 2 LINE(S) NOT DISPLAYED
000199     DO;
- - - - - 1 LINE(S) NOT DISPLAYED
000201     DO;
- - - - - 1 LINE(S) NOT DISPLAYED
000203     DO;
- - - - - 4 LINE(S) NOT DISPLAYED
000208     END;
- - - - - 1 LINE(S) NOT DISPLAYED
000210     DO;
- - - - - 4 LINE(S) NOT DISPLAYED
000215     END;
- - - - - 1 LINE(S) NOT DISPLAYED
000217     DO;
- - - - - 4 LINE(S) NOT DISPLAYED
000222     END;
- - - - - 1 LINE(S) NOT DISPLAYED
000224     DO;
- - - - - 4 LINE(S) NOT DISPLAYED
000229     END;
- - - - - 3 LINE(S) NOT DISPLAYED
000233     END;
- - - - - 96 LINE(S) NOT DISPLAYED
000330     END;
- - - - - 30 LINE(S) NOT DISPLAYED
000361     END;

```

To produce a compressed listing of DOs and ENDS, employ the Exclude line and Find word capabilities of your full-screen editor. In ISPF edit, the X ALL command can be used to mask all lines. Then a FIND DO WORD ALL command would locate all DO lines, while a FIND END WORD ALL command would locate all END lines. (XEDIT has similar capabilities.)

```

EDIT ---- FW2L.SASLIB.CODE(TESTPRGM) - 01.99 ---- COLUMNS 001 072
COMMAND ==> X ALL SCROLL ==> CSR

```

```

EDIT ---- FW2L.SASLIB.CODE(TESTPRGM) - 01.99 ---- COLUMNS 001 072
COMMAND ==> FIND DO WORD ALL SCROLL ==> CSR

```

```

EDIT ---- FW2L.SASLIB.CODE(TESTPRGM) - 01.99 ---- COLUMNS 001 072
COMMAND ==> FIND END WORD ALL SCROLL ==> CSR

```

The DO/END groups can then be verified from the inside/out. Place an XX on the first line of a set DO/END groups and another XX on the last line of that set of DO/END groups.

```

EDIT ---- FW2L.SASLIB.CODE(TESTPRGM) - 01.99 ---- COLUMNS 001 072
COMMAND ==>                                SCROLL ==> CSR
000195     ELSE IF '20050' LE LINE LT '71200' THEN
000196         DO;
-- -- -- -- -- 2 LINE(S) NOT DISPLAYED
000199         DO;
-- -- -- -- -- 1 LINE(S) NOT DISPLAYED
000201         DO;
-- -- -- -- -- 1 LINE(S) NOT DISPLAYED
XX          DO;
-- -- -- -- -- 4 LINE(S) NOT DISPLAYED
000208         END;
-- -- -- -- -- 15 LINE(S) NOT DISPLAYED
000224         DO;
-- -- -- -- -- 4 LINE(S) NOT DISPLAYED
XX          END;
-- -- -- -- -- 3 LINE(S) NOT DISPLAYED
000233         END;
-- -- -- -- -- 96 LINE(S) NOT DISPLAYED
000330         END;
-- -- -- -- -- 30 LINE(S) NOT DISPLAYED
000361     END;

```

After striking ENTER, those verified lines will be excluded from view. (The 15 LINE(S) NOT DISPLAYED has changed to 31 LINE(S).) The next level of DO/END groups can then be verified using the same technique.

```

EDIT ---- FW2L.SASLIB.CODE(TESTPRGM) - 01.99 ---- COLUMNS 001 072
COMMAND ==>                                SCROLL ==> CSR
000195     ELSE IF '20050' LE LINE LT '71200' THEN
000196         DO;
-- -- -- -- -- 2 LINE(S) NOT DISPLAYED
000199         DO;
-- -- -- -- -- 1 LINE(S) NOT DISPLAYED
000201         DO;
-- -- -- -- -- 31 LINE(S) NOT DISPLAYED
000233         END;
-- -- -- -- -- 96 LINE(S) NOT DISPLAYED
000330         END;
-- -- -- -- -- 30 LINE(S) NOT DISPLAYED
000361     END;

```

Other uses of the Exclude capabilities of your full-screen editor include checking the data flowing from Step to Step, provided KEEP= statements on DATA and SET statements are used.

```

EDIT ---- FW2L.SASLIB.CODE(TESTPRGM) - 01.99 ---- COLUMNS 001 072
COMMAND ==>                                     SCROLL ==> CSR
000195    DATA WORK2(KEEP=DEPT YEAR COST BUDGET VARIANCE);
000196        SET WORK1(KEEP=DEPT YEAR COST BUDGET);
- - - - - 40 LINE(S) NOT DISPLAYED
000236    DATA WORK3(KEEP=DEPT YEAR COST BUDGET VARIANCE CLASS);
000237        SET WORK2(KEEP=DEPT YEAR COST BUDGET VARIANCE);

```

Placing DO/END groups in the proper relative nested positions can often require that some be moved in mass to new column locations. The column shifting line commands '((or))' can be very helpful. Specify the number of columns to be shifted next to the ((or)) command.

```

EDIT ---- FW2L.SASLIB.CODE(TESTPRGM) - 01.99 ---- COLUMNS 001 072
COMMAND ==>                                     SCROLL ==> CSR
000195    ELSE IF '20050' LE LINE LT '71200' THEN
000196        DO;
- - - - - 2 LINE(S) NOT DISPLAYED
000199        DO;
- - - - - 1 LINE(S) NOT DISPLAYED
))4        DO;
- - - - - 31 LINE(S) NOT DISPLAYED
))        END;
- - - - - 96 LINE(S) NOT DISPLAYED
000330        END;
- - - - - 30 LINE(S) NOT DISPLAYED
000361    END;

```

Entering a))4 on the first line to be shifted and a)) on the last line to be shifted will move all characters right 4 columns. Even the excluded lines will be shifted.

```

EDIT ---- FW2L.SASLIB.CODE(TESTPRGM) - 01.99 ---- COLUMNS 001 072
COMMAND ==>                                     SCROLL ==> CSR
000195    ELSE IF '20050' LE LINE LT '71200' THEN
000196        DO;
- - - - - 2 LINE(S) NOT DISPLAYED
000199        DO;
- - - - - 1 LINE(S) NOT DISPLAYED
000201        DO;
- - - - - 31 LINE(S) NOT DISPLAYED
000233        END;
- - - - - 96 LINE(S) NOT DISPLAYED
000330        END;
- - - - - 30 LINE(S) NOT DISPLAYED
000361    END;

```

Using ((to bound the group of lines will shift all characters of each line to the left.

The shifting column line command when combined with the Move/Overlay line commands can also be quite useful in creating assignment statements or Proc Format lines.

Imagine that the following data lines exist in another file.

```
44AN5501
44AS6001
44AS6501
44A11795
44A17501
44A18001
44A18501
44A19501
44A2B594
44A21001
```

First, type a SELECT statement, followed by a WHEN statement model.

```
EDIT ---- FW2L.SASLIB.CODE(TESTPRGM) - 01.99 ---- COLUMNS 001 072
COMMAND ==>                                SCROLL ==> CSR
```

```
000200      SELECT;
000300      WHEN ( '          ' ) PART = 'UNAVAILABLE';
```

Then, using the COPY file command, copy the data lines into the program file. The lines can be copied from another member of the same PDS, as shown here, or from another data set.

```
EDIT ---- FW2L.SASLIB.CODE(TESTPRGM) - 01.99 ---- COLUMNS 001 072
COMMAND ==> copy testdata                                SCROLL ==> CSR
```

```
000200      SELECT;
A          WHEN ( '          ' ) PART = 'UNAVAILABLE';
```

Use the column shifting line command to move the copied data directly beneath the quoted area of the WHEN statement.

```
EDIT ---- FW2L.SASLIB.CODE(TESTPRGM) - 01.99 ---- COLUMNS 001 072
COMMAND ==>                                SCROLL ==> CSR
```

```
000200      SELECT;
000300      WHEN ( '          ' ) PART = 'UNAVAILABLE';
000400          44AN5501
000500          44AS6001
000600          44AS6501
000700          44A11795
000800          44A17501
000900          44A18001
001000          44A18501
001100          44A19501
001200          44A2B594
001300          44A21001
```

Now, place a M (Move) line command on the WHEN line. Then, before striking ENTER, type OO (Overlay) line commands on the first and last line of the group of lines to be copied.

```
EDIT ---- FW2L.SASLIB.CODE(TESTPRGM) - 01.99 ---- COLUMNS 001 072
COMMAND ==>                                SCROLL ==> CSR
```

```
000200      SELECT;
M           WHEN ( '          ' ) PART = 'UNAVAILABLE';
OO           44AN5501
000500           44AS6001
000600           44AS6501
000700           44A11795
000800           44A17501
000900           44A18001
001000           44A18501
001100           44A19501
001200           44A2B594
OO           44A21001
```

The result of the two line commands is set of assignment code whose values were not hand-entered and, therefore, should have no errors due to incompleteness or miskeying.

```
EDIT ---- FW2L.SASLIB.CODE(TESTPRGM) - 01.99 ---- COLUMNS 001 072
COMMAND ==>                                SCROLL ==> CSR
```

```
000300      SELECT;
000400      WHEN ('44AN5501') PART = 'UNAVAILABLE';
000500      WHEN ('44AS6001') PART = 'UNAVAILABLE';
000600      WHEN ('44AS6501') PART = 'UNAVAILABLE';
000700      WHEN ('44A11795') PART = 'UNAVAILABLE';
000800      WHEN ('44A17501') PART = 'UNAVAILABLE';
000900      WHEN ('44A18001') PART = 'UNAVAILABLE';
001000      WHEN ('44A18501') PART = 'UNAVAILABLE';
001100      WHEN ('44A19501') PART = 'UNAVAILABLE';
001200      WHEN ('44A2B594') PART = 'UNAVAILABLE';
001300      WHEN ('44A21001') PART = 'UNAVAILABLE';
```

Two related data sets can also be transformed into a Proc Format in the same manner.

```
10000
11000
11100
```

```
AIR VEHICLE
AIRFRAME
AIR VEHICLE I&A
```

After placing the tables in the same file through a COPY file process, move the descriptive strings to the right using the)) shift line command.

```
10000
11000
11100
```

```
AIR VEHICLE
AIRFRAME
AIR VEHICLE I&A
```

Then, using the Move/Overlay line command, move the descriptive strings next to the numeric characters.

```
10000      AIR VEHICLE
11000      AIRFRAME
11100      AIR VEHICLE I&A
```

Now, enter a model line of quotes and equal signs just above them.

```
'      ' = '      '
```

Finally, use the Move/Overlay command again to create the following Proc Format input.

```
'10000'    = 'AIR VEHICLE      '
'11000'    = 'AIRFRAME        '
'11100'    = 'AIR VEHICLE I&A  '
```

The line command and primary command capabilities of ISPF are fully detailed in the on-line edit tutorial of ISPF. The X ALL command is found under the EXCLUDE option of the FIND/CHANGE primary command. Overlay (O) is a Move/Copy line command.

TUTORIAL ----- EDIT - LINE COMMANDS ----- TUTORIAL
 OPTION ==>

```

-----
|               EDIT               |
|             LINE COMMANDS       |
|               |                 |
-----
  
```

The following topics are presented in sequence, or may be selected by number. Individual commands may be selected by entering the command name.

- 0 - Line commands - general information
- 1 - Basic commands
 - I (insert), D (delete), R (repeat)
- 2 - Move/Copy commands
 - C (copy), M (move), A (after), B (before), O (overlay)
- 3 - Shifting commands
 -) (cols right), ((cols left), > (data right), < (data left)
- 4 - Exclude/Show commands
 - X (exclude), F (first), L (last), S (show)
- 5 - Text handling commands
 - TS (text split), TF (text flow), TE (text enter)
 - LC (lowercase), UC (uppercase)
- 6 - Miscellaneous commands
 - BNDS (bounds), COLS (columns), MASK (mask), TABS (tabs)

TUTORIAL ----- EDIT PRIMARY COMMANDS ----- TUTORIAL
 OPTION ==>

```

-----
|   EDIT PRIMARY COMMANDS   |
|                           |
-----
  
```

The following topics are presented in sequence, or may be selected by number. Individual commands may be selected by entering the command name.

- 0 - Primary commands (general information)
- 1 - Miscellaneous commands - - - - LOCATE RESET CANCEL SAVE
 - PROFILE RECOVERY SUBMIT
- 2 - FIND/CHANGE commands - - - - FIND CHANGE EXCLUDE
- 3 - Number control commands - - - - NUMBER RENUM UNNUM
- 4 - Display mode control commands - CAPS HEX NULLS TABS
 - NOTE BOUNDS
- 5 - Termination control commands - AUTONUM AUTOLIST STATS PACK
 - AUTOSAVE VERSION LEVEL
- 6 - External data commands - - - - CREATE REPLACE COPY MOVE
 - MODEL
- 7 - Macro control commands - - - - DEFINE IMACRO RMACRO BUILTIN
- 8 - Data editing commands - - - - DELETE SORT EDIT

REFERENCES

Henderson, D.J. (1991), M.G. Rabb, and J.A. Polzin, 'The SAS Supervisor - A Version 6 Update', SAS Users Group International Conference Proceedings: SUGI 16, Cary, NC: SAS Institute Inc.

DeFoor, J.A. (1991), 'A Comparison of Data Step Performances in Releases 6.06 and 5.18', SAS Users Group International Conference Proceedings: SUGI 16, Cary, NC: SAS Institute Inc.

Caron, Amy (1988), 'Elements of Good Programming Style', SAS Users Group International Conference Proceedings: SUGI 13, Cary, NC: SAS Institute Inc.

ISPF On-line Edit Tutorial, ISPF Program Development Facility Tutorial, International Business Machines (IBM).

SAS Language: Reference, Version 6, First Edition, Cary, NC: SAS Institute Inc., 1990, 1042 pp.

SAS Programming Tips: A Guide to Efficient SAS Processing, Version 6, First Edition, Cary, NC: SAS Institute Inc., 1990, 155 pp.