

THREE WAYS YOU CAN GET SPECIAL DATA INTO THE SAS® DATA STEP.

by William S. Calvert, Service Research, Inc.*
and Eduardo Farias, NICHD/NIH, Bethesda, Md.

ABSTRACT

Three techniques are presented for moving special data into the DATA step. In the first example, a binary database file of obscure origin is read to recapture and salvage the data. In the second case, a 'table look-up' is implemented to assign individual mortality probabilities from a rate table. Finally a mechanism is shown for passing global parameters to production code modules which have been 'compiled' using the SAS stored program facility. All three techniques are explained and presented in the context of the 'real world' programs and systems in which they are used. While the examples used come from the PC SAS (Version 6.04) environment, the concepts should be applicable to other SAS platforms.

INTRODUCTION

The first technique uses the INFILE / INPUT combination of statements to examine a file byte by byte. The remaining two of the three techniques presented use the SET statement as modified by the use of different parameters and context. The essential statements and syntax are shown and explained in the context of working program segments.

The code segments shown were selected from cases encountered in providing technical support in research and financial settings. The context in which the code was written is presented to provide broader understanding of the concept and the motivation for using it. Tips and comments regarding style and efficiency are mentioned where applicable.

READ A BINARY FILE

We were presented with a file that contained useful information but which could not be read by ordinary means. The system which had created the file was broken. By examination we found that the data was stored in ASCII with binary data interspersed. The record length was variable. The file structure indicated

that it probably had been an indexed file. Since the file was about 7 megabytes, it was too big to fix manually and the binary data choked editors we tried to use. We decided to use SAS to sort it out byte by byte.

To read in the data we used a DATA step like this:

```
DATA _NULL_;  
  INFILE 'binary file' RECFM=N END=eof;  
  FILE 'output file' NOPRINT;  
  DO UNTIL eof;  
    ***
```

The key to reading the file was to use the INFILE statement with the RECFM=N option. This in effect makes the file into a single record. We chose to use this file as a string of bytes. The INPUT statement read one byte at a time with control logic that calculated new values for the offset. The FILE statement defines where the bytes that are recovered are to be written. By trial and error we developed a logic for examining the bytes and converting the data into useful chunks to be written out. Note the END= eof option on the INFILE statement. It is essential to any control logic you might use in reading the file byte by byte.

See Example 1.a for a segment showing how we coded it. Example 1.b shows a page of the output text.

After some study we found that the data was delimited by specific bytes for records and fields. The program logic determines if we are building a record or field. As long as the bytes are 'printable' they are written out. Non-printable bytes are ignored unless they are field or record delimiters. The resulting file has one line per field and a page break between records. This was done for convenience so we could check with the client to define how the output should be structured. When we got the logic right and the field structure defined, we made a second pass and used SAS to convert to the final file structure using PROC DBF. The DBF file was then moved into a citation data base manager.

This process is slow, it is best to work on a smaller sample until you get it right. The code didn't need to

Note that we create a data set to contain the parameters. The parameter data set has only one observation. The specific parameters needed for this process are brought in with the `_N_=1` logic when the first observation is read and appended to the end of each record being processed. To change the programmed process now you need only update the parameter data set/observation. In a large system of 40 parameters or so on a PC, this approach can keep you going when a macro driven system runs out of memory.

See Example 3. for a sample of our code which uses this idea.

DISCUSSION

Three ways have been show to get special data into the SAS DATA step. We showed a way to read binary data from a broken database and salvage the data. We showed a way to use the SET statement with the POINT option to implement a table look-up. And finally, we have shown an innovative way to pass parameters to a data step in the Stored Program Facility.

We are aware that there are other ways to do some of these things. In Example 3. for instance, our way is just one way that may be effective for those using limited systems. Macros take up memory for the duration of the SAS job, the parameter file we used is stored on disk and only occupies memory when in use. We like the macro language, but find it is easier to pay the memory resource price in a mainframe environment.

While the context in which these ideas were developed is complex, the ideas themselves are relatively easy to grasp. We had to struggle to some extent to find solutions to our requirements even though the solution seems simple in the end. We hope our description saves others some trouble.

•♦•

SAS is a registered trademark of the SAS Institute, Inc. Cary, North Carolina, USA.

* Service Research, Inc. is located at 401 Hull Place, Rockville, MD 20852 FAX (301) 738-1041

EXAMPLE 1.a

```
DATA _NULL_ ;
  INFILE 'd:regroup1' RECFM=N END=eof;
  FILE 'd:\asciout' NOPRINT;
  BEFINREC = 'false' ;
  LENGTH BYTE $1. ;
  FORMFEED = '0C'X ; CRRETRUN = '0D'X ;
  COUNT = 00 ;      OFFSET = 00 ;
  LINK READ ;
  READ:
    OFFSET = OFFSET + 1 ;
    INPUT @OFFSET BYTE CHAR1. ;
    RETURN ;
  DO UNTIL (eof) ;
    IF BEGINREC = 'false' THEN LINK
  FNDBGREC;
  ELSE ;
    IF BYTE > '1F'X AND BYTE < '7F'X
      THEN LINK PRNTBYTE ;
    ELSE IF BYTE = '0D'X THEN DO ;
      LINK READ ;
      LINK FNDBGFLD ;
    END ;    *==> much more code ==> ;
```

EXAMPLE 1.b

```
1 = 12285
2 = AIDS
3 = YES
4 = NICHOLSON, J
5 = SPIRA T
6 = ALOISIO C
7 = JONES, B
8 = KENNEDY M
9 = HOLMAN, R
10= MCDUGAL, J
11=
12= SERIAL DETERMINATIONS OF HIV-1 TITERS IN
    HIV-INFECTED HOMO-
13= SEXUAL MEN: ASSOCIATION OF RISING TITERS WITH
    CD4 T CELL
14= DEPLETION AND PROGRESSION TO AIDS
15= AIDS RESEARCH AND HUMAN RETROVIRUSES VOL 5, NO 2
    1989
16=
17= AIDS
18=
19=
20=
```

be efficient, we just had to make it work right this one time. We successfully recovered over 99% of the data.

READ DATA FROM A TABLE

In this example we were assigning the probability of death and the probability of sickness to different segments of a population cohort in a population dynamics simulator. In the simplest form you might start with a group of nominally 65 year old females in good health; at each time interval you need to know how many have gotten sick and how many have died so you can update the book keeping in the simulator. It is necessary to use different mortality and morbidity rates for each subgroup of the population as the time ticks on. The easiest way we found to do this was to use a table look-up. This minimizes the number of values retained in memory at any one time and allows for changes and modifications to the tables. The process stays the same but the rate data change.

The required logic to implement this is not difficult:

```
DATA ***
  *** calculate value of I first;
  SET table(DROP=extra data) POINT=I ;
  IF _ERROR_ =1 THEN ABORT;
  IF eof THEN STOP ;
```

There are several important points to keep in mind. You must calculate the value of the point variable outside of the SET statement. This may change in later versions. Be sure to build in some protection against out of range point values. The ABORT and STOP statements are good choices, but don't overlook the use of some IF logic to catch conditions you know about. Finally be careful with variable names in the data sets you are using. In our case we had AGE in both data sets; when we explicitly dropped the one we didn't need, our problems went away.

See Example 2. for a segment from our application.

This code is a segment from a very large DATA step which does the main work of the population dynamics simulation system. Note that the look-up is accomplished within a loop restricting the action to integral years. The table we use is sorted by age and the point value is calculated as an offset from the age value. To get the value for age 60 :

```
POINT= 60-59= 1;
```

which gives the first entry in the table. The highest age in the table is 115. We have put in logic to prevent exceeding this value. Values are being assigned into male and female groups with two levels of 'health'. Life expectancy values are selected from the tables as well. Beyond this code segment the program is tracking dozens of cohorts each with hundreds of sub groups. I would not have been possible to do this without a table look-up.

PASS PARAMETERS TO STORED PROGRAMS

SAS has a facility which is not so widely acclaimed as it might be; ' the Stored Program Facility'. It is easy to use.

```
DATA mine;
  * do some processing;
  RUN PGM=sas.name;
```

The option after the run statement causes the 'compiled' segment to be stored under 'sas.name'. You then can execute this program with one line:

```
DATA PGM=sas.name;
  RUN;
```

It is faster. The problem is that the data set names must always be the same. Only the data that is processed can be changed. This sometimes causes people to give up using the facility. Why run the program again if you can't change things! Macros don't solve the problem. The resolved value of the macro at time of compilation is 'frozen' in the segment.

We had several complex steps in our simulation which were best seen as parameter driven processes. In the simplest case we were constantly adjusting tables such as were used in the previous example. In each simulation run we modified the table with a parameter driven algorithm. To make the process more efficient we developed a way to pass special data to the compiled segments. The essential idea works like this:

```
DATA ...
  IF _N_=1 THEN
    SET parms(KEEP=the right stuff);
  *** more processing;
  RUN PGM=sas.name;
```

EXAMPLE 2.

```

...
IF AGE_X =INT(AGE_X) THEN DO ;
* THE OFFSET FOR AGE IS 59.;
IF SDAGE_X GT 115 THEN I=115 -59 ;
ELSE I=SDAGE_X - 59 ;
    LINK LOOKER;
    FMT_SD_X=FEMAL*MORMOD;
    SFEX_X =FEMALEX ;
    MMT_SD_X=MAL*MORMOD ;
    SMEX_X =MALEX ;

IF AAGE_X GT 115 THEN I=115 -59 ;
ELSE I= AAGE_X - 59 ;
    LINK LOOKER;
    FMT_D_X=FEMAL*MORMOD;
    AFEX_X =FEMALEX ;
    MMT_D_X=MAL*MORMOD ;
    AMEX_X =MALEX ;
...

IF EOF THEN STOP;
RETURN ; * LOGICAL END OF SIMULATE LOOP;

LOOKER:
* ASSIGN MORTALITY AND LIFE EXPECTANCY;
SET MORT83(DROP=AGE) POINT=I ;
IF _ERROR_ = 1 THEN ABORT ;
RETURN;

```

EXAMPLE 3.

```

...
/* **
DATA mortable ; SET SAVE.MORT83 ;
IF _N_ =1 THEN
    SET PARMS(KEEP=KNOB1 KNOB2 KNOB3);

* ADJUST MORTALITY/ MORBIDITY
  WITH KNOBS (ETC) ;
  MALE  = MALE  * KNOB1 ;
  MAL   = MALE  /1000;
  FEMALE = FEMALE * KNOB1 ;
  FEMAL = FEMALE/1000;
  MMO_NX = MMO_NX * KNOB2 /1000 ;
  MMO_PS = MMO_PS * KNOB2 /1000 ;
  FMO_NX = FMO_NX * KNOB2 /1000 ;
  FMO_PS = FMO_PS * KNOB2 /1000 ;
  H_M = H_M * KNOB3 /400 ;
  H_F = H_F * KNOB3 /400 ;
* LIFE EXPECTANCY CALCULATIONS
  FOR ADJUSTED TABLE ;
  RETAIN MALEX FEMALEX 0 ;
  MALEX  = (1-MAL ) * (1 + MALEX) ;
  FEMALEX = (1-FEMAL ) * (1 + FEMALEX);
RUN PGM=SAVE.mort;
** */

DATA PGM=SAVE.mort ; RUN ;

PROC SORT DATA=mortable ; BY AGE ;
...

```