

SAS as a Utility Language: Using SAS for Solving Little Problems

Gary E. Schlegelmilch, U.S. Dept. of Commerce, Bureau of the Census, Suitland MD

ABSTRACT

SAS, from its very conception, was intended to provide methods for determining and handling statistics cleanly, clearly, and effectively. It does so with speed, strength, and versatility. However, the tools that effect its versatility lend themselves with equal ease to nonstatistical projects as well. Small day-to-day tasks – impromptu reports, quick reviews of data, and other small utility programs - are handled in SAS quickly and efficiently. Because SAS provides minimal data field definition, a modular approach in the DATA step, and clearly defined processes in the PROC step, small tasks can be programmed with a minimum of effort. In addition, the turnaround time will frequently be less than the same routine written with a compiled/linked language. When time is of the essence, SAS may very well be the best candidate.

SOLVING A SAMPLE PROBLEM: COMPARING DISK STORAGE AREAS

In the two following sample programs, the purpose is to show a number of ways to solve day-to-day problems using Base SAS tools. The data or specific problems may not be of specific interest; hopefully, the techniques will be.

Let's start with an example that might be found in an office environment. Data files, program files, and source files on a Windows NT system are copied regularly to a network archival disk. In reviewing disk utilization, the Data Management section has observed that the work area, contained on a second disk, is nearly full. So Data Management requests that all users remove all nonessential files from the work disk. The typical user could do a quick DIR, find they have 1,246 files on disk, and start checking them one by one to see which are current working copies, and which are extraneous because they are already contained on the archive disk. Or, with a little imagination; let the computer do the work, and

have a tool ready when the same request comes in next month.

GETTING STARTED

First, a sample program could be laid out with the basic steps to be done, in comment form:

```
/* Read the work disk */
/* Read the archive disk */
/* Consolidate the data */
/* Compare the results and report */
```

Even on a small utility, it is better to use the most effective SAS code. The “/* */” comment construct will execute faster than the “* ;” construct, because the words between “/* */” are not interpreted, and the words in “* ;” are. As with any language, any future maintenance becomes a much less trying exercise if accurate comments are in place. It also permits the program to be more readily made available to a Production environment; any good programmer knows that there's no such thing as “one-time use” if the program works well.

Under the “/* Read the work disk */” header, the code might proceed like this:

```
options noxwait xsync;

/*-----
   get a list of the first disk's
   contents, using a DOS command,
   with the output directed to an
   ASCII file.
   -----*/

x "dir c:\ > c:\user.dat";

filename LOCALDAT "c:\user.dat";

data WORK.USER (drop=SIZE);

/* Use MISSOVER to place missing
   values in records shorter than
   the data lines (e.g., the
   Directory line, only 17 chars
   Long). */
```

```

infile LOCALDAT missover;
length FILENAME $36. SIZEU 8.
      SIZEX $10. VDATEU $16.;
input @1 VDATEU
      @29 SIZEX
      @40 FILENAME;

/*-----
DOS sizes are specified with
commas; without the COMMA
informat, an error will be
generated in the function.
-----*/

/* eliminates <DIR> lines */
if index(SIZEX, '>') = 0;

/*-----
If no filename (e.g., record too
short), no need to process it
-----*/

if FILENAME > '';

/* Volume label line, no need to
process */

if SIZEX ^= 'Volume';
SIZEU=input(SIZEX, comma10.);
FILENAME=upcase(FILENAME);

/* process only if file has
measurable block size */

if SIZEU > 0;
run;

```

It should be noted that all platforms will have slightly different formats for directory listings. Even Windows 95 and 98 have different formats than Windows NT, the format shown here.

When using an 'x' command to perform an operating system, there are a number of system options, which can affect the flow. The XWAIT command will pause and wait for the user to type 'exit', when in interactive mode. To avoid this, change the system default XWAIT to NOXWAIT.

The XSYNC command will tell the system to wait until the operating system command has completed before proceeding. In this application, using it ensures that the DIR command has finished creating the file to be used as input before the SAS program continues to process.

Another way to get the DIR information into the SAS Data Step would be to use the PIPE option from the FILENAME command. By using the line:

```
filename LOCALDAT pipe "dir C:\\";
```

the SAS program would perform the Windows system command, "pipe" or redirect the output to a buffer, and give that buffer the name LOCALDAT. It can then be accessed in the SAS data step as one would any input file.

XSYNC, XWAIT, and PIPE are all documented in the Companion volumes, as they have subtle differences depending on the platform for which they are used; consult them before using the commands for your program.

Perform a virtual repeat of the process for the archive disk. Copying a block of lines is a simple matter; placing 'rr' in the first and last line number of the source code in the Program Editor for v6.12, or cut and paste in either v6.12, v7, or v8.

```

/* Read the archive disk */

/*-----
Don't write to the archive
disk, reuse the same name.
-----*/

x "dir z:\ > c:\user.dat";

filename ARCHDAT "c:\user.dat";

data WORK.ARCHIVE (drop=SIZEX);
  infile ARCHDAT;
  length FILENAME $36. SIZEA 8.
        SIZEX $10. VDATEA $16.;
  input @1 VDATEA
        @29 SIZEX
        @40 FILENAME;

  if index(SIZEX, '>') = 0;
  if FILENAME > '';
  if SIZEX ^= 'Volume';
  SIZEA=input(SIZEX, comma10.);
  FILENAME=upcase(FILENAME);
  if SIZEA > 0;
run;

```

Notice the "not equal" symbol used, "^=" . In many languages, the "<>" symbol is used as the "not equal" – however, this is NOT the case in SAS! The "<>" is a MAX function, and can return a false

value. As an example; if SIZEX was equal to "Q.DAT", the return of the comparison would have been 'Volume' if the "<>" symbol had been used, and interpreted by SAS as "true". A good trap to stay out of!

Reading two files of identical formats could actually be done in a number of ways. The code to read the files can be placed in a separate SAS file, and put in place with the %INCLUDE statement; it can be developed as a Macro, with the input and output names set as macro variables, or set as arguments to the Macro; or both files could be read in one data step, using the FILE statement. In this context, I wanted to illustrate the method that would be the simplest and quickest to code; writing a single routine, then copying the same lines and making minor changes to them. If the program is intended for repeat use, using the %MACRO would be far more efficient.

After this code has executed, there are two small data sets which contain the name, size, and version date of all files on both disks. In each, the ASCII value of the file size was dropped in lieu of the numeric value, to save space. The next step is to merge them, eliminating any nonmutual files.

```
/* Consolidate the data */

proc sort data=WORK.USER;
  by FILENAME;
run;
proc sort data=WORK.ARCHIVE;
  by FILENAME;
run;

/* Compare the results and report */

data WORK.CONSOLDT;
  merge WORK.ARCHIVE WORK.USER;
  by FILENAME;

/* check to see if there is an
   occurrence from each input data
   set */

  if SIZEA>0 and SIZEU>0;

/* check to see if the sizes
   match */

  if SIZEA=SIZEU;
```

```
/* check to see if the version
   dates match */

  if VDATEA=VDATEU;
run;
```

Notice that the IN= option for the MERGE statement would have worked here as well. However, it would have created unnecessary fields in the Program Data Vector, since we have to check the existing size and version date anyway.

PROC COMPARE would have been another method for comparing those datasets. It provides the user with far more detailed data on the comparison of two datasets, including details on nonmatched data.

As you can see, SAS provides a great variety of data-handling tools for projects both large and small.

At this point, we have a data set CONSOLDT containing all the files on both directories with matching version dates and sizes. PROC REPORT would probably give the neatest format, and a DATA step would permit all kinds of flexibility for formatting. However, all we are really looking for is a list of file names that can be deleted.

```
title "Files that may be deleted "
      "from Userdisk, duplicated "
      "on Archive disk";
proc print data=CONSOLDT;

/*-----
The last data set created - in this
case, CONSOLDT - is the default, so
it need not have been specified.
However, it is a good practice to
put in the filename, to maintain
clarity, and self-document the code.
-----*/
  var FILENAME;
run;
```

The program could be further enhanced to produce an actual executable file to delete the duplicate files. The user could still review his list, make any necessary modifications to the batch file, then run it to remove any unneeded files. That code could be written in this way:

```

filename dupdel 'c:\dupdel.bat';
data _null_;
  file dupdel;
  set WORK.CONSOLDT;
  put 'del ' FILENAME;
run;

```

In about 50 executable lines, you have a reusable, self-documented method for comparing two drives, or two directories. Of course, neatness counts; you might want to add these few lines at the bottom of the program, just to clean up the work files.

```

x "del c:\user.dat";
proc datasets library=WORK;
  delete USER ARCHIVE CONSOLDT;
quit;
/* notice - 'quit' instead of 'run'! */

```

And not a statistic in sight.

REFORMATTING ASCII RECORDS

Another common occurrence in an office is the processing of ASCII, or “flat”, files. ASCII files are used frequently as interfaces between platforms. However, there are times when a file will be received by the processing system from an interfacing system that will be in an incorrect format, or contain invalid data. That data must be corrected and realigned quickly so no Production time is lost.

Another possibility is that a series of files was upgraded to a new system, but a historical file must be processed in to recover old data – and the historical file is in the old transaction format. SAS’s speed and power to manipulate strings rivals that of many languages, because field formats do not need to be defined in detail in order for the data to be parsed.

In the following example, I had to read a file that consisted of blocks of records, grouped by ID. For the purpose of this example, the old transaction format will consist of three records of retail inventory data, type (RECTYPE) 1-3. Type 1 is a product record, containing serial number (SERNBR) and description of a product (DESC). Type 2 contains the number of items on hand (NBRONHND), number of items on order (NBRONORD), and the number of items that have

been sold to date (SOLDTODT). Types 1 and 2 are mandatory; 3 is an optional comment record, containing up to three lines of comments about the product.

The new transaction format consists of five records, type A-E, of which only the first is mandatory; the others only exist if data exists for them. Type A contains the ID, NBRONHND, SERNBR, and the previous serial number in the event that it had been changed (PREVSR). Type B contains the product description, if one exists. Type C contains the number of items on back order, if any. Type D contains sold-to-date information, if any. And type E contains comments, if any. As with the old files, the data is not necessarily in ID order, or in record type order by ID; but transactions are grouped by ID.

For this example, we could build a DATA step to read in the first file, sort it, and then produce an output file. However, if space is at a premium, we can do it without building a data set at all.

```

/* Reformat from Old into New
   transaction Format */

filename OLD
'c:\schle001\in_work\h199701.dat';
filename NEW
'c:\schle001\in_work\d199701.dat';

data _null_ ;

  length OLDID ID PREVSR $8.;
  retain OLDID ID RECTYPE;
  retain SW01 SW02 0;

  /*-----
    The old file is variable
    length, with the longest
    being 309 characters. So,
    since it's larger than 256,
    define the file's longest
    record with an lrecl option.
    -----*/

  /* only need to do this on the
     first iteration */

  infile OLD end=FILEEND lrecl=309;
  if _n_=1 then do;
    input @1 ID $CHAR8.
          @9 RECTYPE $CHAR1. @;
    OLDID=ID;
  end;

```

```

do until ((FILEEND=2)
  OR (OLDID ^= ID));
  if (ID = OLDID) then do;

    select (RECTYPE);
    when ('1') do;
      input @10  SERNBR      8.
            @20  DESC       $40.;
      SW01 = 1;
    end;
    when ('2') do;
      input @10  NBRONHND   8.
            @18  NBRONORD   8.
            @26  SOLDTODT   8.;
      SW02 = 1;
    end;
    when ('3')
      input @10  COMMENT1 $100.
            @110 COMMENT2 $100.
            @210 COMMENT3 $100.;
    otherwise do;
      file LOG;
      put 'Invalid record type '
        RECTYPE ', ID ' ID;
    end;
  end;
end;

/*-----
  Once the record is read,
  proceed to check and see
  if there is any more data
  for this ID.
-----*/

if (FILEEND = 1) then FILEEND=2;
else do;
  infile OLD end=FILEEND
        lrecl=309;
  input @1 ID          $CHAR8.
        @9 RECTYPE     $CHAR1. @@;
end;

/*-----
  Note the use of the double
  trailing at-sign. This will
  maintain the record position,
  even if the DATA step goes on
  to the next iteration.
-----*/

end;

/*-----
  Now, make sure the mandatory
  record types are present. If
  not, note them in the log
  instead of building the new
  transaction.
-----*/

```

```

OK=0;
if (SW01 = 1) and (SW02 = 1)
then OK=1;
else do;
  file LOG;
  put "Missing record types " @;
  if SW01 = 0 then put "01 " @;
  if SW02 = 0 then put "02 " @;
  put "for ID " OLDID ".";
  OLDID=ID;
end;
SW00=0; SW01=0;

/*-----
  The single trailing at-sign
  used here will allow the next
  PUT statement to continue
  from that position on the
  line. The line would also end
  at the end of this iteration of
  the DATA Step, if no other PUT
  statement had been coded.
-----*/

if OK; /* only need to go
      forward if the data
      is good */

/* Previous Serial Number (PSN)
   used to be part of the
   comments; in the new format,
   it is in a separate field, so
   it must be searched out from
   the comment field.
   According to the standards,
   a Previous Serial Number
   should always be preceded by
   'PSN ', so that will be
   the search parameter. */

CHECKPT = index(COMMENT1,'PSN ');
if (CHECKPT > 0) then do;
  CHECKPT+4;
  PREVSER=
  left(substr(COMMENT1,CHECKPT,8));
end;

/* Now, write out the data
   records in the new format.
   Type 'A' is always written;
   'B' through 'E' are optional,
   and written only if the
   specific data exists. */

file NEW lrecl=312;
put @1  OLDID      $8.
    @10 'A'
    @12 SERNBR     10.
    @25 PREVSER    $8.
    @35 NBRONHND   8.;
if DESC > '' then do;

```

```

        put @1  OLDID  $8.
           @10  'B'
           @12 DESC  $50.;
    end;
    if NBRONORD > 0 then do;
        put @1  OLDID  $8.
           @10  'C'
           @12 NBRONORD 8.;
    end;
    if SOLDTODT > 0 then do;
        put @1  OLDID  $8.
           @10  'D'
           @12 SOLDTODT 8.;
    end;
    if COMMENT1 > '' then do;
        put @1  OLDID      $8.
           @10  'E'
           @12  COMMENT1 $100.
           @112 COMMENT2 $100.
           @212 COMMENT3 $100.;
    end;
    OLDID=ID;
run;

```

Because of the variable number of records in a block, this one took a little longer to write and debug. However, writing the same type of program in other languages would have had to overcome the same problems; checking each ID, retrieving all data for that ID, using a parser to retrieve the previous serial number from the first comment, and writing out a new block of records depending on the data available. In 90 lines of executable, readable, and maintainable code, it makes for a small, yet powerful little program.

CONCLUSIONS

There will always be a need for applications big and small. SAS DATA step and PROC step modules permit ease in breaking down large requirements into small, testable, maintainable modules, which translate into easily manageable milestones. But because of the versatility of the SAS language, it can save both programming and processing time to use it to develop those “one-time” programs as well.

ACKNOWLEDGEMENTS

I'd like to thank Kim Bankard of the Census Bureau for her insights on this paper in its early stages; S. David Riba of Jade Tech Inc., Dana Raiffee of Destiny Corp., and Sam Rozenel of the Census Bureau for their encouragement for me to write it; Ian Whitlock, SESUG2K Section Chair, for his guidance and critique; and Clara Waterman, SESUG2K Section Chair, for her invaluable support.

REFERENCES

Aster, Rick, Professional SAS Programmer's Pocket Reference, 2nd Edition, 1998, Breakfast Books.
SAS Companion for Microsoft Windows Environment, Version 6, 2nd Edition, SAS Institute Inc.

AUTHOR

Gary E. Schlegelmilch, U.S. Dept. of Commerce
 Bureau of the Census, ESMPD/MCDIB
 Suitland Federal Center, Rm. 1200-4
 4700 Silver Hill Road
 Suitland MD 20746
 Email
Gary.E.Schlegelmilch@ccmail.census.gov

SAS and all other SAS Institute Inc. product and service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries.