The SAS® System - A Full-Featured Utility and Language For Programmers

Steven First
Systems Seminar Consultants
Madison, WI 53716 (608) 222-7081

Abstract

SAS software is certainly well known as an end user tool for the more casual computer user. Data processing staffs however can exploit SAS as a full featured programming language to produce both one-shot applications and full production systems. This paper will concentrate on the features of SAS that make it an excellent tool for programmers. It will present a series of short applications that show how SAS can fit in with the other languages used in your shop and make the programmer's job easier.

Introduction

Working with both programmers and non-programmers over the years, I have noticed that programmers are often the hardest to train and also difficult to get to try SAS software. An in-house programmer from a consulting client of ours recently told me that he didn't want to learn SAS because "the learning curve was too long". While I would admit that learning SAS takes some time, it's learning curve is nowhere as long as with COBOL or other languages. I maintain that SAS is a language that can compete effectively in both a production mode and as a programmer's utility.

Is SAS a good "programmers language"?

Some of the reasons I feel that SAS makes the programmers job easier follow:

- SAS runs in batch, foreground, and interactive environments.
- SAS provides flexible input and output though the INPUT statement, conversion procs, database engines.
- SAS provides enormous reporting and analysis capabilities.
- The data step provides a full programming language with:
  - Looping, logic, attribute setting
  - An extensive function library
  - Debugging statements
  - Date arithmetic
  - Access to system control blocks
- SAS is fairly simple for programmers to learn
- Programmers don't need more extensive user interfaces than SAS provides.
- SAS Procs certainly save time.
- Transporting of data and programs is easy using SAS.

My Introduction to SAS Software

My last COBOL programs were written in 1977 under these circumstances:

Our company used two different data centers and communication between them consisted of RJE stations that appeared as card punches and readers to the two hosts. They were actually tape drives, but a lot of card image data would be "punched" from one system and then "read" into the other system.

One particularly critical system was passing thousands of 80 byte records, but only using 20 columns of each record. This process took several hours and one night at 4:55 my boss told me that I needed to write two programs. On the transmitting system I was to read 4 records, stack the used portions onto one 80 byte record that was now completely used before transmitting in to the RJE. A second program would be needed to reverse the process on the receiving system to create the original records. The idea was to cut the transmission time to 25% of the original which they did.

These two trivial COBOL programs took 4 hours of time to write, compile and link, and store the programs in the two data centers. While I don't have the programs any more, each program was at least 100 lines of mostly uninteresting code.

After spending that night writing COBOL and being basically a lazy programmer, I decided to find a utility to help me with these "one-shot" type application.

I found that the following SAS code will accomplish the same task as above. (JCL defined both files with LRECL=80).

```
DATA _NULL_;                        /* DONT NEED DS      */
  INFILE IN;                        /* RAW FILE IN       */
  FILE OUT;                         /* RAW FILE OUT      */
  INPUT @1 TWENTY $CHAR20.;         /* READ 20 CHAR      */
  PUT      TWENTY $CHAR20. @;       /* 20 OUT, HOLD PTR  */

RUN;                                /* END OF STEP       */
```

The Reversing Program

Reversing the logic requires another program that can produce the original output. This program relies on the @@ parameter which keeps the input pointer positioned in the same record even though the DATA step is exited. NOTE: If @@ is used with a constant pointer (ex. @1) or column (ex. 20), an infinite loop occurs. It is very

easy to forget the period on an informat and have SAS interpret the columns as an absolute column, so be careful.

```
DATA _NULL_;                    /* DONT NEED DATASET */
 INFILE IN;                     /* RAW FILE IN       */
 FILE OUT LRECL=80;             /* FILEOUT           */
 INPUT                          /* READ TWENTY/LOOP  */
     TWENTY  $CHAR20.           /* FIXED PTRS CAUSE  */
    @@ ;                        /* LOOPS, BE CAREFUL */
 IF TWENTY NE ' ' THEN          /* IF NONBLANK ?     */
      PUT                       /* OUTPUT 20         */
      @1 TWENTY  $CHAR20.; /* $CHAR SAVES BLANKS*/
 IF _N_ > 20 THEN               /* A GOOD IDEA WHILE */
    STOP;                       /* TESTING           */
RUN;                            /* END OF STEP       */
```

Processing Foreign Tapes

One of the most painful of programming tasks is the dreaded "foreign tape". Very often appearing with no documentation, labeling, or even a hint of what kind of system it was created on, these datasets can be very difficult to process.

If you are confident that the tape is IBM standard labeled, PROC TAPELABEL can be used to map the datasets on the reel. You may have to specify the first dataset name on the reel if you cannot specify BLP label processing. You should be careful if the tape is not labeled, as TAPELABEL may run past the end of a reel looking for the nonexistent label. In any event the program is very simple, and provides some interesting dataset level information.
Here is an example of PROC TAPELABEL under MVS.

```
//  EXEC SAS
//TAPE1  DD UNIT=TAPE, VOL=SER=ABCDE,DISP=OLD

PROC TAPELABEL DONAME=TAPE1;
RUN;
```

For each file on the reel, TAPELABEL will list DSNAME, RECFM, LRECL, BLKSIZE, block count, estimated length in feet, creation date, expiration date, job name of creating job, and more.

Copying Tapes

PROC TAPECOPY provides the capability to copy one or more tapes to a single volume. This can be useful for consolidating volumes, making backup copies of reels etc. PROC TAPECOPY uses a very simple syntax. It should be noted that if the input and output tapes are both standard labeled, the user must have security clearance to read and write all the datasets copied. Specifying LABEL=(,BLP) allows the user to bypass security, and is usually restricted at most sites. Use of BLP processing should be used with great caution, as it is very easy with PROC TAPECOPY to write over system labels accidentally. Here is an example of PROC TAPECOPY under MVS. (CMS processing is also available).

```
//  EXEC SAS
//VOLINDDDSN=first.dsn.on.tape,UNIT=TAPE,VOL=SER=xxxxxx,
//          DISP=OLD
//VOLDUT  DD UNIT=TAPE,VOL=SER=yyyyyy,DISP=(,KEEP)
 PROC TAPECOPY;
 RUN;
```

A Hex Dumping Program

After getting by the label processing, the next step is to try and read some data. The data step LIST statement will print any record read, and will use HEX over/under printing if necessary. If the file is not standard labeled you must provide DCB information. If you have absolutely no idea what is on a tape, RECFM=U can be used to let SAS read a physical block and list it out. Keep in mind that IBM allows BLKSIZE values of up to 32760 bytes, so printing just a few records can produce considerable printed output. Note however that any data step statement can be used to control the LISTing. The following program will read any sequential dataset and print a few records in the log.

```
DATA _NULL_;                    /* DONT NEED DATASET */
  INFILE IN;                    /* RAW FILE IN       */
  INPUT;                        /* READ A RECORD     */
  LIST;                         /* LIST BUFFER IN LOG*/
  IF _N_ > 50 THEN              /* STOP AFTER 50     */
     STOP;                      /* ADJUST AS NEEDED  */
RUN;                            /* END OF STEP       */
```

A Copying Program

Adding output JCL and a FILE, and PUT statement can create a copy of any sequential file.

```
DATA _NULL_;                    /* DONT NEED DATASET */
  INFILE IN;                    /* RAW FILE IN       */
  FILE OUT;                     /* RAW FILE OUT      */
  INPUT;                        /* READ A RECORD     */
  PUT _INFILE_;                 /* WRITE IT OUT      */
RUN;                            /* END OF STEP       */
```

Changing DCB Attributes While Copying

If the INPUT record is shorter than the record length of the output file, it will be padded with blanks. If INPUT length is longer than the output record, the INPUT will be broken into several output records.

```
DATA _NULL_;                    /* DONT NEED DATASET */
  INFILE IN;                    /* RAW FILE IN       */
  FILE OUT LRECL=90             /* INCREASE DCB AS   */
           BLKSIZE=9000         /* NEEDED            */
           RECFM=FB;
  INPUT;                        /* READ A RECORD     */
  PUT _INFILE_;                 /* WRITE IT OUT      */
RUN;                            /* END OF STEP       */
```

A Subsetting Program

Adding IF statements could be used to introduce selection logic and create a subset of any input file.

```
DATA _NULL_;                    /* DONT NEED DATASET */
  INFILE IN;                    /* RAW FILE IN       */
```

```
  FILE OUT;                 /* RAW FILE OUT       */
  INPUT @5 ID $CHAR1.;      /* INPUT FIELDS NEEDED*/
  IF ID='2';                /* WANT THIS RECORD?  */
  PUT _INFILE_;             /* YEP, WRITE IT OUT  */
RUN;
```

Selecting a Random Subset

The RANUNI function can be used to select a uniform percentage of a file for testing of other applications.

```
DATA _NULL_;              /* NO DATASET NEEDED  */
  INFILE IN;              /* RAW FILE IN        */
  FILE OUT;               /* RAW FILE OUT       */
  INPUT;                  /* READ A RECORD      */
  IF RANUNI(0) LE .10;    /* TRUE FOR APP. 10%  */
  PUT _INFILE_;           /* WRITE OUT OBS      */
RUN;                      /* END OF STEP        */
```

Adding Sequence Numbers

A data step can easily add sequence numbers to records as they pass by.

```
DATA _NULL_;              /* NO DATASET NEEDED  */
  INFILE IN;              /* RAW FILE IN        */
  FILE OUT;               /* RAW FILE OUT       */
  INPUT;                  /* READ A RECORD      */
  SEQ=_N_*100;            /* COMPUTE SEQ NO     */
  PUT _INFILE_            /* OUTPUT INPUT REC   */
      @73 SEQ Z8.;        /* OVERLAY WITH SEQ   */
RUN;                      /* END OF STEP        */
```

Writing Literals in Every Record

Constants can also be placed in the record as needed.

```
DATA _NULL_;              /* NO DATASET NEEDED  */
  INFILE IN;              /* RAW FILE IN        */
  FILE OUT;               /* RAW FILE OUT       */
  INPUT;                  /* READ A RECORD      */
  PUT _INFILE_            /* OUTPUT INPUT REC   */
      @10 'SSC';          /* OVERLAY WITH CONST.*/
RUN;                      /* END OF STEP        */
```

Printing a File with Carriage Control

Print files that have been captured can be printed later using a data step. The NOPRINT option on the FILE PRINT statement tells PUT not to add more carriage control characters. These types of files are fairly common for files bound for microfiche or other special printing needs.

```
DATA _NULL_;              /* DONT NEED DATASET  */
  INFILE IN;              /* INPUT FILE IN      */
  FILE PRINT NOPRINT;     /* DONT ADD CC        */
  INPUT;                  /* READ A RECORD      */
  PUT _INFILE_;           /* WRITE IT OUT       */
RUN;                      /* END OF STEP        */
```

Correcting a Field on a File

This type of program can save the programmer many late hours. Very often a file needs just minor modifications. Again with the DATA step's flexibility to handle any type of input field, and the ability to specify whatever logic is needed, this program can be adapted as needed. At midnight we can use all of the help we can get.

```
DATA _NULL_;              /* DONT NEED DATASET  */
  INFILE IN;              /* INPUT FILE IN      */
  FILE OUT;               /* OUTPUT FILE        */
  INPUT @5 ID $CHAR1.;    /* INPUT FIELDS NEEDED*/
  IF ID='2'               /* CHANGE AS NEEDED   */
     THEN ID='3';
  PUT   _INFILE_          /* OUTPUT FILE        */
     @5 ID CHAR1.;        /* OVERLAY CHANGED ID */
RUN;                      /* END OF STEP        */
```

SAS for "Production Jobs"

I maintain that SAS makes an appropriate language not only for one-shot applications, but also makes sense for production jobs. Just because a job is important doesn't mean it should use a difficult language.

Report Writing

There are not many other languages that can produce a report as quickly as:

```
PROC PRINT;RUN;
```

Since this topic is covered thoroughly in documentation and training manuals, I am not going to list extensive report writing examples, but I would like to consider some different approaches to report writing.

PROC FORMS, CALENDAR, FSLETTER

Other programs available for reporting include: PROC FORMS for repetitive forms such as mailing labels and postcards, PROC CALENDAR for date related display of data, and PROC FSLETTER which can provide mail merge applications such as form letters and mailing labels interactively. While there are many mail merge type programs available on PCs, there are virtually none available for mainframes.

PROC FREQ

PROC FREQ can also be used for reporting, as many production applications are designed to count things and calculate percentages. If you don't like the format of PROC FREQ output, OUTPUT options allow you to route the output to SAS datasets for later printing and formatting.

One COBOL program we converted to PROC FREQ used a two dimensional table to produce a matrix of 150 terminals by 200 possible operators who might key into them around the state. The matrix contained the number of transactions each terminal and operator keyed.

Listed below is a diagram of the required report:

Number of Transactions by Termid and Operator

TERMID	Operator 0001	0002	0003	0004 . . .	0200
0001	55				
0002	24	12			
0003			2		
.					
.					
0150					1

Figure 1: A Table of Transaction Counts

Each time the number of new terminals increased, the program would have to be changed and at one point since COBOL requires the user to predefine possible interactions (150 * 200), the table exceeded the number of allowed table elements in a COBOL program. Not only can PROC FREQ do this task easier, it wasn't even possible in COBOL without special algorithms.

If you examine the problem, this is a classic "sparse matrix" problem, since operators from the northern part of the state would not normally travel 300 miles to use a terminal in the southern part of the state, but they might rarely need to if they needed to cover for sick operators etc.

PROC FREQ on the other hand only allocates cells if they are used, and thus can solve the problem (without sorting), by coding:

```
PROC FREQ;
  TABLE TERMID * OPERATOR / LIST;
RUN;
```

PROC MEANS and TABULATE

A similar case can be made for PROC MEANS, PROC SUMMARY, and PROC TABULATE. Many "Production programs" summarize data, calculate sums and averages and print the results. Again numerous examples abound in the SAS documentation.

PROC RANK

PROC RANK is a personal favorite of mine for calculating the rank statistic or in other words which value is in first place, second place etc. Rankings certainly have their place in athletic standings, test scoring, and other statistical applications, but what about business applications? In a recent training class after covering PROC RANK a programmer commented that he would probably never need a rank. In the very next chapter he requested a way to produce a report showing his companies top 25 customers by sales. Sounds suspiciously like a rank. How can we produce the report?

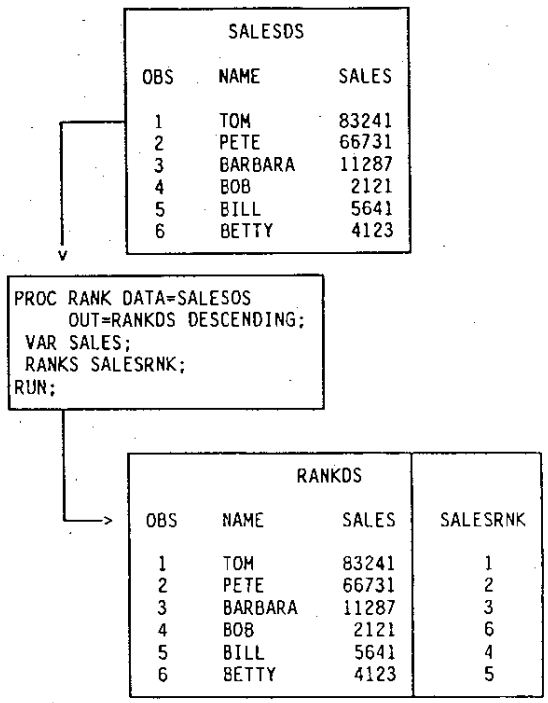

Figure 2: PROC RANK Data Flow

It should be noted that PROC RANK doesn't sort the data, or print it, but a simple PROC SORT and PROC PRINT following the RANK step produces the report with no problem.

DATA Step Programming

The DATA step certainly gives a good SAS programmer all the tools needed to produce virtually any batch system or report. The DATA step offers many features such as end-of-file testing, multi-column reporting, control-break indicators, file merging, and much more. Programmers seem to like the DATA step because it offers lots of tools, but still gives them the freedom to program the way they like. The DATA step is one of the strongest features of SAS, but again the DATA step and it's programming capabilities are well documented, so I won't go any further into the basics here.

Accessing System Control Blocks

One of the most exciting and obscure features is access to several system control blocks through the INFILE and FILE statements. Normally only assembler programs can access these blocks, and it is normally impossible for COBOL and other higher languages.

Using the JFCB

The Job File Control Block is 176 bytes of information stored for each DD card specified in the job step. This

very useful block contains extensive information about each dataset including datasetname, device type, whether the dataset is catalogued, whether the dataset is a SYSIN or SYSOUT dataset, label processing options and much more. By accessing this block, the SAS DATA step can determine what dataset name was provided in the JCL. This is especially useful if the same program is run against several different datasets, and whatever dsname is provided through DD cards is to end up in a title or similar field.

A SMF processing program can determine whether it is reading a live VSAM file, a sequential backup disk file, or a tape file. Since many of the indicators in the JFCB are bit settings, the DATA step may need to do bit testing, which is no problem for the DATA step.

A JFCB example

Determine the DSNAME and DSORG from the JFCB.

```
DATA _NULL_;                        /* DONT NEED DATASET */
 INFILE IN JFCB=JFCBIN;             /* ASK FOR JFCB      */
 LENGTH TITLDSN $ 44;               /* SET LENGTHS AS    */
 LENGTH DSORG1 $1.;                 /* REQUIRED          */
 IF _N_ = 1 THEN                    /* FIRST TIME IN ?   */
  DO;                               /* YES, DO BLOCK     */
   TITLDSN=SUBSTR(JFCBIN,1,44);     /* EXTRACT DSNAME    */
   DSORG1=SUBSTR(JFCBIN,99,1);      /* AND DSORG BYTE 1  */
    IF DSORG1='.1.....'B THEN       /* BIT TEST AS NEEDED*/
       DSORGOUT='PS';               /* MUST BE SEQUENTIAL*/
   END;                             /* END OF BLOCK      */
 INPUT etc. ;                       /* REST OF PROGRAM   */
 .
 .
 .
 RETAIN TITLDSN DSORGOUT;           /* RETAIN            */

RUN;                                /* END OF STEP       */
```

Other Available Control Blocks

Other MVS control blocks available on INFILE and FILE include DEVTYPE containing device type information, DSCB containing the data set control block information for non-vsam disk files, the UCBNAME containing the device address from the UCB, and the VOLUME parameter containing the volume serial of the disk dataset. Another useful INFILE option EOV is set to 1 each time a concatenated datasets boundary is detected. The above mentioned control blocks can then be reread with the new values if desired.

Management Concerns

Some managers may express concern in using SAS for production. Some common comments follow:

SAS software is a "statistical package".

While SAS certainly provides statistical tools, it certainly is much more and qualifies as a full general purpose data processing language.

End users will destroy production databases.

This is a common concern especially in installing the update portions of the database interfaces. Obviously we don't want to allow casual update of vital data. By not installing update portions the installation prevents their use by those people who know what they are doing. Why should database programmers be forced to use primitive languages just to guarantee that naive users don't destroy production data? The obvious answer here is to protect the data, not lock up the tools. Anybody who updates data should know what they are doing and forcing everybody to use second generation languages can cause more damage that it prevents. SAS in the hands of a knowledgeable programmer can be an indispensable tool for emergencies, data base loads, and much more.

SAS is a "compile and go" language.

This is generally true, but not necessarily bad. With compiled languages, the site has two modules to control, the source, and the compiled module. It far too many sites this is not controlled, and the modules are not kept in sync. With SAS only the original source need be controlled since compiled code is not stored until version 6. With the introduction of compiled code SAS programs will have to control compiled code as well.

Probably the most common argument forwarded here is that the cost of compiling at each execution is too high. While this may have some truth, most of the resources consumed by SAS are at execution time, not compile time, so I'm not sure compiled code is worth the bother.

A big advantage to the current method of running SAS gives dynamic program generation through the macro language, and it remains to be seen how much of the macro language will be limited in a compiled DATA step.

SAS code is harder to control.

I'm not sure why source to a SAS job should ever be harder to control than source to any other language. If data and programs need to be secured or controlled, certainly change control and security packages can control SAS as well as any other language.

SAS is a resource hog.

Because of it's structure, SAS takes many passes and does a fair amount of conversion in the DATA step. This can be minimized in several ways. Countless efficiency presentations have been given, and the SAS Applications Guide gives several good tips as well. By minimizing sorting, dataset passes, dataset size, use of numeric variables, and use of other efficiency tips, SAS can be competitive with run times of most languages, not to mention the development time.

Another efficiency tip might be to preprocess large datasets with a faster program to reduce sizes. One

large application used a file containing several hundred bytes per record and had in excess of 6 million records. Processing this file with SAS cost approximately $1200 to pull relatively few records and fields from the file and sort and print a report.

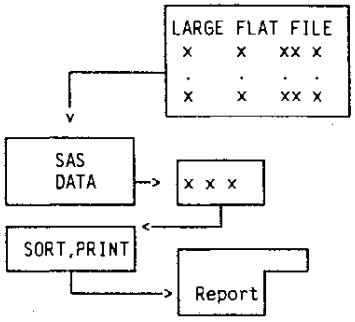

Figure 3: A Large Dataset Read With SAS

This application used very little of the rows and columns of the original dataset, and thus most of the cost in this job was the DATA step.

Another Approach

Since the user would not accept the above costs, we looked for a faster data filter. The fastest program in the shop was probably SYNCSORT(r) which has both row and column selection capabilities, along with sorting. By preprocessing the dataset with this much faster program, even though two extra data passes are required, the resulting datasets are much smaller and the total job cost dropped to $200. SYNCSORT could also do the sorting, and eliminate the SAS PROC SORT step. It should be noted that the control statements for SYNCSORT were much more complex and required 1/2 day of a senior programmers time, and as such may be beyond the capabilities of some end users.

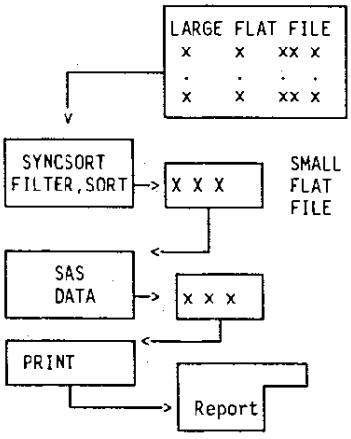

Figure 4: SYNCSORT and SAS together

Programs for Accessing Systems Datasets

Beyond using SAS for application work, SAS certainly does a good job for the data manager or system programmer. These jobs use unusual input, and may be critical and complex applications.

Reading Partitioned Datasets

A cornerstone for OS (MVS) systems since the beginnings of the operating system has been the use of Partitioned Data Sets (PDS). MVS requires that load modules be stored as members in a PDS, and historically source and many other types of data have been stored as PDS members as well. My guess is that the designers of OS used PDSs to minimize the number of VTOC entries on a disk pack. The concept of PDS members has confused countless users however over the years. It has been especially difficult for data managers to control PDSs because historically security, data sharing, and time-stamping have been at the dataset level, with very little available at the PDS member level. For example, it is relatively easy to secure an entire dataset, and determine it's date of last use. Without special software however the same information is not available at the PDS member level. Personally I feel this is the reason so many shops have libraries full of unwanted and unsecured members. However PDSs are probably here to stay, so we may have to just live with them as best we can.

Data managers and others may need to process both PDS directories and the member data as well. This is an especially tricky task, since the internal structures are complex, but luckily there are some programs already written.

The layout of a PDS is as follows: The data set is partitioned into a directory and a data space for the members. Both areas are divided up among the contained members, with unused space as well.

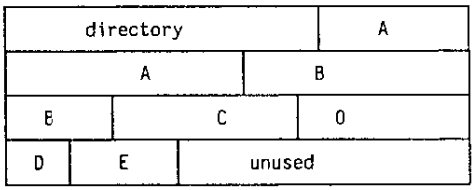

Figure 5: A Partitioned Dataset

The directory consists of one or more fixed-length records called directory blocks usually of length 256. Each block contains a count field, telling how much of the directory block is used, and a varying number of directory entries.

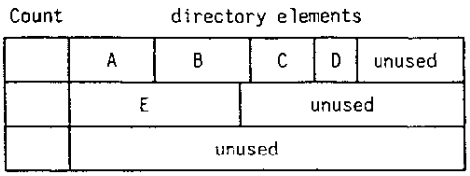

Figure 6: A PDS directory

Finally the directory elements themselves begin with a fixed portion containing the name of the member, the relative location (TTR), and an attribute byte(IND). In load module libraries there is additional information stored for the loader.

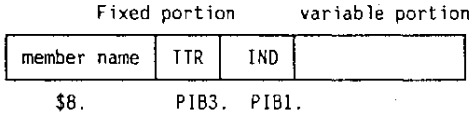

Figure 7: Directory elements

Why bother reading a PDS directory?

Reading PDS directories can be useful to produce listing of the library for printing or to feed into another program. Audit programs may want to list the directory and perhaps match the results with those of another library.

A SAS Program to read a PDS Directory

```
// EXEC SAS
//PDS   DD  DSN=SOME.PDS,DISP=SHR
DATA PDSDIR;                        /* SAS DATASET OUT    */
 INFILE PDS RECFM=F                 /* INFILE TO PDS      */
          BLKSIZE=256;              /* DIRECTORY          */
 INPUT COUNT PIB2. @;               /* READ COUNT FIELD   */
 IF COUNT < 14 THEN STOP;           /* LAST RECORD        */
 DO WHILE(COL<COUNT);               /* LOOP ACROSS BLOCK  */
  INPUT @COL MEMBER $8.             /* READ MEMBER        */
        +3  IND    PIB1. @;        /* SKIP TTR, READ IND */
  COL=COL+12+2*MOD(IND,32);         /* SKIP VARIABLE INFO */
  OUTPUT;                           /* OUTPUT MEMBER      */
 END;                               /* END OF LOOP        */
RUN;                                /* END OF STEP        */
PROC PRINT DATA=PDSDIR;             /* PRINT DATASET      */
 VAR MEMBER;                        /* LIST ONLY MEMBER   */
RUN;                                /* END OF STEP        */
```

Because better utilities such as IBM's ISPF have been made available in recent years, this program is not needed as much as it was in the past. It should work for any PDS though and can be worked into systems as needed.

It should be noted that PROC SOURCE provides a much easier way to list PDS members if the PDS is fixed length and record length of 80. PROC SOURCE will be covered later in this paper.

Free Programs

At MVS sites SAS is shipped with a library called the SAS sample library which contains about 100 assorted programs. By running member INDEX from that library, you can print an index of it's contents.

A variation of the above PDS program is provided to MVS sites in member PDSLIST of the SAS Sample library. PDSLIST also handles load libraries and gives correct space statistics if the library is compressed.

A related program called SPFMLIST is an especially useful variation of PDSLIST which also reports ISPF statistics if present.

Reading VTOCS

Also possible with the SAS data step is the ability to read an MVS volume table of contents (VTOC). This is a dataset residing on each disk pack that contains an entry for each dataset stored on the volume. This dataset is again a very complex system dataset, but it contains very useful information for the space manager. Sample applications might be to inventory disk space usage, determine which datasets need to be backed up, which datasets have unused space and much more. The SAS sample library contains a member called MAPDISK which gives a variety of different reports based on VTOC data.

A Partial Listing of MAPDISK

```
 /*---------- MAPDISK ----------------------------*
THIS PROGRAM READS THE DSCBS IN A VTOC AND PRODUCES A
LISTING OF ALL DATA SETS WITH THEIR ATTRIBUTES AND
ALLOCATION DATA. THE VOLUME TO BE MAPPED MUST BE DESCRIBED
BY A DISK DD STMT.:
//DISK DD DISP=SHR,UNIT=SYSDA,VOL=SER=XXXXXX
*-----------------------------------------------*/
DATA FREE(KEEP=LOC CYL TRACK TOTAL F5DSCB)
     DSN (KEEP=DSNAME CREATED EXPIRES LASTREF LASTMOD
          COUNT EXTENTS DSORG RECFM1-RECFM4 ALOC BLKSIZE
          LRECL SECALOC TT R TRACKS VOLUME)
     FMT1(KEEP=DSNAME CREATED EXPIRES LASTREF LASTMOD
          COUNT EXTENTS DSORG RECFM1-RECFM4 ALOC BLKSIZE
          LRECL SECALOC TT R TRACKS VOLUME CCHHR)
     FMT2(KEEP=CCHHR TOCCHHR)
     FMT3(KEEP=CCHHR ALLOC3);  LENGTH DEFAULT=4;

  RETAIN TRKCYL 0; /* ERROR IF NO FORMAT 4 ENCOUNTERED */
  LENGTH VOLUME VOLSER1 $ 6  CCHHR CCHHR1 $ 5 ;
  FORMAT CCHHR CCHHR1 $HEX10. DSCBTYPE $HEX2. ;

  *-------READ DSCB AND DETERMINE WHICH FORMAT------*;
  INFILE DISK VTOC CVAF CCHHR=CCHHR1 VOLUME=VOLSER1
         COLUMN=COL ;
  INPUT @45 DSCBTYPE $CHAR1. @;  VOLUME=VOLSER1;
  CCHHR=CCHHR1;
  IF DSCBTYPE='00'X THEN DO; NULL+1;
        IF NULL>200 THEN STOP;
        RETURN;  END;  NULL=0;
      IF DSCBTYPE= '1' THEN GOTO FORMAT1;
      IF DSCBTYPE= '2' THEN GOTO FORMAT2;
      IF DSCBTYPE= '3' THEN GOTO FORMAT3;
      IF DSCBTYPE= '5' THEN GOTO FORMAT5;
      IF DSCBTYPE= '4' THEN GOTO FORMAT4;
      IF DSCBTYPE= '6' THEN RETURN;
      _ERROR_=1;RETURN;
```

```
FORMAT1:                    *---REGULAR DSCB TYPE---*;
  INPUT @1 DSNAME $CHAR44.
      @46 SER $CHAR6. @46 YM PIB1. DAYM PIB2. COUNT PIB2.
          IND PIB1.
      @54 YC PIB1. DAYC PIB2.
          YE PIB1. DAYE PIB2.

      FORMAT CREATED EXPIRES LASTMOD LASTREF DATE7.;
      . . . .
```

I should note that most shops have other VTOC reading utilities that may be faster and simpler to run than this program. Most of them have weak reporting capabilities, but it's usually no problem to have the utility direct VTOC data to a report dataset, and then process that "report" with SAS.
Below are a few lines from a program that reads VTOCs with Computer Associates' $RSVP(r) program, and then processes the results with SAS.

$RSVP followed by SAS

```
//TMP EXEC PGM=IKJEFT01,DYNAMNBR=30,REGION=1024K
//SYSTSPRT DD SYSOUT=*
//$RSOUT DD  DSN=&&RSVPOUT,DISP=(,PASS),UNIT=OWKD,
//          SPACE=(TRK,50)
//$OUTPUT  DD SYSOUT=*
//SYSTSIN  DD *          ******** CHANGE VOLSER OF DISK IN
********
 $RSVP VOL(VOL001) -
 TRK PRINT (NEW (OSNAME CDATE LDATE DSORG RECFM BLKSZ -
    LRECL ALLOC USED USECNT VOLUME MDATE CAT LMTIM))
//STEP02   EXEC SAS
//*****************************************************
//* RUN SAS TO GET IT TOGETHER FOR PRINTING
//*****************************************************
//SAS.RSDUT   DD DSN=&&RSVPOUT,DISP=(OLD,DELETE)

/*****************************************************/
/* BUILD SAS DS WITH INDIVIDUAL DS RECS IN IT        */
/*****************************************************/
DATA SASDISK;                    /* DATASET RECORDS    */
 INFILE RSOUT;                   /* FILE FROM $RSVP    */
 INPUT @2 COL2_7 $CHAR6. @ ;     /* DELETE JUNK        */
 IF COL2_7 NE '$RSOA0';          /* HEADERS            */
 IF COL2_7 NE 'DSNAME';          /* MORE               */
 IF COL2_7 NE 'TOTAL ';          /* SUMMARY LINES      */
 INPUT @2  OSN         $CHAR44.  /* DETAIL LINES       */
       @48 CDATE       5.
       @54 LDATE    ?? 5.
       @61 OSORG       $CHAR2.
       @66 RECFM       $CHAR1.
       @72 BLKSIZE     5.
       @78 LRECL       5.
       @87 ALLOC       5.
       @96 USED        5.
      @104 USECNT      6.
      @111 VOLUME    $CHAR6.
      @118 MDATE    ?? 5.
      @125 CAT        $CHAR1.
      @128 LMTIM      TIME5.
           ;
 IF SUBSTR(DSN,1,9) NE 'SYS1.VTOC'; /*  WATCH  VTOC  */
CREATE=DATEJUL(CDATE);              /* CLEAN UP DATES */
 IF LDATE NE . THEN
    LASTUSE=DATEJUL(LDATE);
 IF LASTUSE=. THEN
    LASTUSE = CREATE;
 IF MDATE NE . THEN
    LMDATE=DATEJUL(MDATE);
 IF MDATE=. THEN
    LMDATE = CREATE;
 UNUSED=ALLOC-USED;              /* CALC UNUSED SPACE  */
 FORMAT LASTUSE CREATE LMDATE MMDDYY8.;
 FORMAT LMTIM  TIME5.;
 DAYS=TODAY()-LASTUSE;           /* DAYS SINCE LAST USE*/
 DROP COL2_7 CDATE LDATE MDATE;  /* DROP JUNK VARIABLES*/
RUN;                             /* END OF STEP        */
```

More Reading of Other Program's Output

Many program products store their data in complicated formats that may not be easy for even SAS to read. Many of these programs will often produce bare reports however. It is usually easy to redirect these reports to disk and read the 'report' into a DATA step, throw away the junk, and format as required.

Examples of the above are source management systems, security systems, financial packages, or virtually any program product. I have found that IBM utilities such as IDCAMS, IEHLIST, and many others are easy to post-process in this manner.

Many packages allow "reports" with very little in the way of headers and titles etc. These are usually easy to process, since there is less unnecessary information to bypass.

An example I have written involved a report produced by Panvalet(r):

The user wanted a different sort sequence, and the report programs provided didn't offer that. By using JCL to redirect the Panvalet report to disk followed by a quick DATA step and PROC SORT and PROC PRINT solved the application easily. I have used this technique in many applications with very little effort.

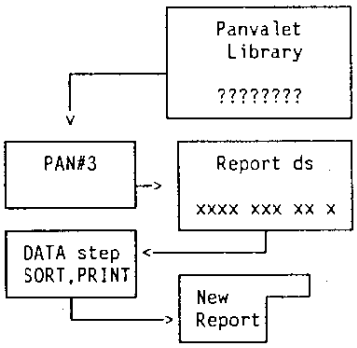

Figure 8: Flow of Panvalet Listing Program

Here is the SAS code needed:

```
// EXEC PGM=PAN#3
//*****************************************/
//*    USE PAN#3 TO CREATE DIRECTORY      */
//*****************************************/
//SYSPRINT DD SYSOUT=*
//SYSPUNCH DD DSN=&&PANDIREC,DISP=(,PASS),
//            UNIT=DISK,SPACE=(TRK,25)
//PANOD1   DD DSN=PANVALET.LIBRARY,DISP=SHR
++CONTROL ....
++PRINT O-UP
//*****************************************/
//*    CREATE REPORTS WITH SAS            */
```

```
//*****************************************/
//SAS   EXEC SAS
//PANDD1   DD DSN=&&PANDIREC,DISP=(OLD,DELETE)
DATA PANDIREC;
INFILE PANDD1 ;
INPUT @01 PNPGMNM  $1D.
      @11 PNLEVEL  3.
      @19 PNPGMTYP $5.
      @27 PNMTDATE MMDDYY8.
      @35 PNACDATE MMDDYY8.
      @24 PNSTATUS $3.
      @43 PNNDBLK  4.
      @48 PNNDBLK  8.;
RUN;
PROC PRINT DATA=PANDIREC;
  TITLE 'PAN DIRECTORY';RUN;
```

Programs that Write Other Programs

A very exciting application of SAS is to use it to create another program which can be stored on disk, or submitted to an internal reader. If the resulting code is SAS, the macro language can do some of this, but this technique can be used to write other languages as well, or maybe just including batch JCL interspersed with SAS code. SAS can in effect generate dynamic programs which can be altered based on some input.

The previous disk management program can be followed by any variety of analyzing step to take various actions. In one system we developed SAS datasets, weren't being backed up automatically. The above program was followed by a series of generated batch jobs which created both the JCL and the PROC COPY jobs to backup the selected datasets. This generated job was directed to an internal reader and scheduled like any other job.

It also was modified to archive old unused SAS datasets by deleting the original after copying. This is a very dangerous program to automate. Initially we set a limit of 50 datasets to be deleted in any one job, as a logic flaw can very easily delete ALL datasets on a pack. If you do this sort of thing, back yourself up thoroughly first and test very carefully.

There goes the JES Queue

Another similar system automated the submission of each nights batch jobs. A scheduling job would determine which JCL members in a job library should be run each night, and would automatically submit 200 or so batch jobs to the internal reader. A logic flaw caused the schedule job to also submit itself, which submitted another 200 jobs plus itself. After cycling 18 times the input queue filled and took down the entire system.

Downloading All SAS Programs Under CMS

If PC SAS is available and there is a host connection, you can RSUBMIT a quick program to download several programs in a single session. By using masking characters in the CMS commands, you can select the programs or data you want, or perhaps logic a DATA step could select desired data. This program can easily be modified for TSO, VMS or other operating systems as well. The process could be reversed to upload several PC datasets to the host for central backup or storage with little trouble. The SAS documentation for PROC UPLOAD and PROC DOWNLOAD give several examples of just these applications.

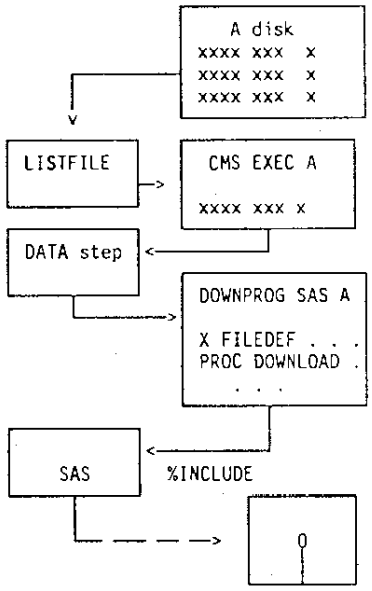

Figure 9: Diagram of a Mass Download Program

The Download Program

```
X LISTFILE * SAS A (EXEC;              /* CMS COMMAND    */
X FILEDEF FLIST DISK CMS EXEC A;       /* LISTFILE OUTPT */
X FILEDEF GENPRG DISK DOWNPRG SAS A;  /* TEMP PROGRAM   */
DATA _NULL_;                           /* BUILD FILEDEFS */
 INFILE FLIST;                         /* PROC DOWNLOADS */
 INPUT @8 FN $8. @17 FT $8.;
 FILE GENPRG;
 PUT 'X FILEDEF HOSTFILE DISK ' FN FT ' L;'
  / 'PROC DOWNLOAD INFILE=HOSTFILE OUTFILE="C:\DOWNDIR\'
FN '.SAS";'
  / 'RUN;';
RUN;
%INCLUDE GENPRG;              /* INC GENERATED CODE  */
RUN;
```

A Mass Change Program

Frequently data managers must scan an entire library and change all members containing some string. This can be somewhat nerve wracking if you need to change all members in an important production library. The first time I needed to do this we were changing SYSOUT classes in all 800 production PROCLIB members. That is, all lines found in a PDS containing "SYSOUT=A" were to be changed to "SYSOUT=*".
The person who helped me said, "It's not any harder than changing 1 member is, in either case you have to

do it correctly. Just back yourself up, and take it slowly.".

If the library needing to be changed is an 80 byte PDS, PROC SOURCE can be used to extract some or all members and put them into a single sequential dataset. It can also put reloading commands at the beginning of each member so that they can be reloaded to a PDS later. This sequential dataset can then be easily read into a DATA step, changed as needed, and then reloaded.

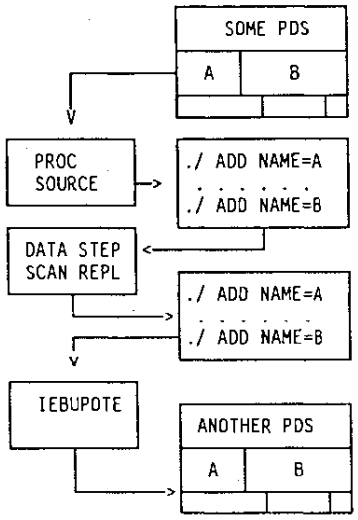

Figure 10: Flow of Mass Change Program

The mass change SAS code follows:

```
// EXEC SAS
//PDSIN  DD  DSN=SOMEPDS,DISP=SHR
//SEQIN  DD  UNIT=DISK,SPACE=(TRK,500)
//POSOUT DD  OSN=SOMEPDS,DISP=SHR
//SEQOUT DD  DSN=&&SEQOUT,DISP=(,PASS),
//         UNIT=DISK,SPACE=(TRK,500)
//PDSOUT DD  DSN=SOMEPOS,DISP=SHR

PROC SOURCE INDD=CLIST OUTDD=SEQIN
 SELECT . . .; /* MODIFY AS NEEDED */
RUN;

DATA _NULL_;
  INFILE SEQIN;
  INPUT @1 WHOLE $CHAR80.;
  IF INDEX(WHOLE,'SYSOUT=A') NE 0 THEN
     SUBSTR(WHOLE,INDEX,8)='SYSOUT=*';
  FILE SEQOUT;
  PUT @1 WHOLE $CHAR80.;
RUN;

// EXEC PGM=IEBUPDTE
//SYSPRINT DD SYSOUT=*
//SYSUT2 DD DSN=SOME.PDS,DISP=OLD
//SYSIN  DD DSN=&&SEQOUT,DISP=(OLD,DELETE)
```

The above program can be modified as needed to print results, add lines etc. PROC SOURCE can also insert command lines for systems such as PANVALET, or LIBRARIAN. Another example of this type of application can be found in the sample library in members PDSFIND and PDSCHANG. These members are not restricted to 80 source input.

Summary

I come from Wisconsin where we say that:
"Point Beer - It's not just for breakfast anymore."

I would rephrase that as:

"SAS Software - It's not just for users anymore."

The author will be glad to answer questions and accept suggestions at the following address:

Steven First
Systems Seminar Consultants
6014 Gateway Green
Madison, WI 53716
(608) 222-7081