Paper 059-29

Using Text Files to Enhance SAS® Code

David J. Austin, Quintiles, Inc., Kansas City, MO

ABSTRACT

One fundamental strength of SAS is its ability to convert raw data into analytical information. The files do not have to be particularly well structured for SAS to read and use. SAS has the flexibility to read data from practically any format. In fact, certain efficiencies can be gained by keeping raw data raw. Beginners and intermediate SAS users may find the following methods of utilizing raw data helpful.

INTRODUCTION

Generally the basic model proposed keeps the raw data in a separate directory from the programs. Usually the raw data is an ASCII text file and the data is subject to frequent change; however, the file structure itself should never change. The data can be maintained and updated by anyone: the programmer, the project manager, or the client. Multiple SAS programs may access this "real-time" data. Advantage number one: Every update to the data is immediately available and processed by the SAS program(s). The need to generate current data sets from the raw data is eliminated. The need to *remember* to generate current data sets from the raw data is eliminated. Any and all changes to the data are automatically available to the SAS program.

The programmer must know the file structure employed in order to utilize the data. Data Definition Tables (DDT) define the file structure and often accompany the raw data files. The most common data file structures are delimited, columnar, and hierarchical. For some users, Microsoft Excel files are the only format used to transport and store data. Although not covered in this paper, XML data files contain a definition of their data structure as well as the actual data. All of the above file structures can be accessed using SAS. Data sets are easily created using the IMPORT procedure and the data step statements INFILE, INPUT, and %INCLUDE. It takes only a few statements to write text files using the EXPORT procedure, while DATA _NULL_ gives users more control over text file output.

The five models presented include the directory model, the footnote model, the hierarchical model, the flat file model, and the MS Excel model.

THE DIRECTORY MODEL

THE SITUATION

This particular endeavor involved approximately 100 studies and 5 projects. Every study had its own unique directories for raw data, programs, analysis files, and outputs. Each project combined the data from the defined studies into one project data set. Projects A, B, C, D, and E originally consisted of 25, 60, 10, 15, and 100 studies respectively. What frustrated the programmers was how the client regularly redefined which studies were in each project. Each study was similar enough that the same code could be reused.

THE SOLUTION

The first step was to create a text file listing all of the directory paths for Project E. You will probably notice the file contains complete SAS statements. It is in effect a SAS program, but the structure is easy enough for any non-SAS user to edit and maintain. Any row can be commented out with an asterisk, if needed. The following is the sample text file **project_e.txt**:

```
libname S001_R 'c:\sugi29\study001\raw_data';
libname S001_P 'c:\sugi29\study001\programs';
libname S001_A 'c:\sugi29\study001\analysis';
libname S001_L 'c:\sugi29\study001\listings';
libname S002_R 'c:\sugi29\study002\raw_data';
libname S002_P 'c:\sugi29\study002\programs';
libname S002_A 'c:\sugi29\study002\analysis';
libname S002_L 'c:\sugi29\study002\listings';
 . . .
libname S100_R 'c:\sugi29\study100\raw_data';
libname S100_P 'c:\sugi29\study100\programs';
libname S100_A 'c:\sugi29\study100\analysis';
libname S100_L 'c:\sugi29\study100\listings';
```

The Project E list was created first since it contained all 100 studies. It became the template to create the text files for projects A through D. The suffixes "R", "P", "A", and "L" allow for subgrouping based on the directory type. The second step is to create file references to the text files using FILENAME statements. These could be predefined in a setup file or at the AUTOCALL level.

```
filename proj_a 'c:\sugi29\programs\project_a.txt';
filename proj_b 'c:\sugi29\programs\project_b.txt';
filename proj_c 'c:\sugi29\programs\project_c.txt';
filename proj_d 'c:\sugi29\programs\project_d.txt';
filename proj_e 'c:\sugi29\programs\project_e.txt';
%include proj_e;
```

Immediately after the %INCLUDE statement - each statement in the text file is executed by SAS and is written to the SASHELP dictionary. The step below uses the INDEX function on the LIBNAME suffix (_A) to subgroup analysis directories only. Using INTO creates a space delimited macro variable named LIBLIST.

```
proc sql noprint;
  select distinct libname into :liblist separated by ' '
  from dictionary.tables
  where index(upcase(libname), '_A')
  order by libname;
quit;
```

The INDEX function searches a string for a substring and returns the beginning position of the substring. For example, INDEX ("ARKANSAS CITY", "KANSAS CITY") finds the substring "KANSAS CITY" beginning at the 3rd character of the given string, and returns the value of 3. All positive values evaluate to TRUE.

The value of the macro variable LIBLIST now looks something like this: `S001_A S002_A … S100_A`, i.e., a space delimited list of library names pointing only to the analysis directories. When the contents of a delimited macro variable are combined with the power of the looping logic below your programming power increases greatly.

```
%let j=1;
%let lib=%scan(&liblist, &j);
%do %while(%length(&lib));

  <<repeating code here>>

  %let j=%eval(&j+1);
  %let lib=%scan(&liblist, &j);
%end;
<<optional code here>>
```

After processing the last LIBNAME, `S100_A`, %SCAN returns NULL to the value of the &LIB macro variable. The %LENGTH function returns zero which evaluates to FALSE.

WHY I LOVE THIS SOLUTION

Simple programs can be repeated on any scale! Want to backup all of your data analysis files? Just insert the DATASETS procedure into the loop. Want to create one data set from all of the data sets? Just use the APPEND procedure. Does your client want to see all actual data values for a certain variable? Then use the PROC APPEND in the repeating code section and PROC FREQ in the optional code section. PROC CONTENTS? Not a problem. Let the client request a dozen ad hoc reports! Let the client endlessly redefine the studies for each project. If a study is canceled or put on hold, it can be commented out in the text file. Once the programs are validated, only the text files need to be maintained.

THE FOOTNOTE MODEL

THE SITUATION

Your project has five programmers working to produce 200 tables, listings, and graphs. You know from experience footnotes constantly change throughout the development period. Some tables have more than ten footnotes. You want the footnotes to have a consistent appearance across all outputs regardless of programmer; to be easily updated; and to have all phrasing standardized. Can you do it?

THE SOLUTION

One solution is to create a text file containing all of the unique footnotes and assigning them a reference code. Notice the format in the file **footnote.txt** - the first eight characters are the reference codes:

```
ae_sae   $    = Serious Adverse Event
ae_nos   NOS  = Not Otherwise Specified
ae_teae  TEAE = Treatment-Emergent Adverse Event
cl_lln   LLN  = Lower Limit of Normal
cl_lln2  LLN  = Low Normal Range
cl_uln   ULN  = Upper Limit of Normal
```

Anyone can update the file without disrupting the programmers. The reference codes help to deter duplicate footnotes from being entered. This increases program consistency, efficiency, and maintainability. These reference codes will be passed into the "footer" macro defined below:

```
filename footlist 'c:\sugi29\project001\programs\footnote.txt';
%macro footer(codelist=)
  data textfile;
    infile footlist missover pad;
    retain num 0;
    input @01 code $8.
          @09 text $200.;
    if indexw("&codelist", code);
    num + 1;
%mend;
```

The INDEXW function searches a string for a delimited substring and returns the beginning position of the delimited substring. If INDEX had been used in the code above, the parameter "cl_lln" would return true for both reference codes cl_lln and cl_lln2. INDEXW avoids finding partial "words" inside the source string.

The next step creates macro variables at each iteration for each footnote. The actual text is placed into FOOT1, FOOT2, etc., while the total number of footnotes is placed into NUMFOOT. This method allows any number of footnotes to be processed.

```
call symput('foot' ||  left(put(num, 2.)), text);
call symput('numfoot', left(put(num, 2.)));
```

After ending the data step the last lines of the macro output each footnote processed from the text file:

```
%do num=1 %to &numfoot;
  %if %length(&&foot&num) %then
  %do;
    footnote&num "&&foot&num";
  %end;
%end;
```

To use, simply call the footer macro before the final data step:

```
%footer(codelist=cl_lln cl_lln2)
```

By utilizing the ability to share common footnotes, this method was used on a clinical trial to successfully maintain consistency among hundreds of footnotes.

THE HIERARCHICAL MODEL

Hierarchical structures exist in the military chain of command, directory trees, and in the organizational charts found in most corporations. The structure of a hierarchical file requires less disk space than an uncompressed flat file holding the same data. Laboratory data usually shows multiple patients taking multiple tests over multiple visits. Elevated test results often trigger more testing to be performed on a patient. The hierarchical structure does not change when extra tests or extra visits are unexpected added. XML files can have the hierarchical form.

THE SITUATION

Your raw laboratory data file has been delivered in a hierarchical data structure and you need to access the data immediately. After a cursory glance at the data you notice most patients have taken three lab tests, RBC, WBC, and SGOT, but the second patient has only taken two lab tests. A printout of **lab.csv** reveals the structure:

```
H,JDOE,16-Apr-04,LABS
P,1,101,FDR
V,Visit 1,10-Jan-04,16:04
R,RBC,4.3,10**6/uL
R,WBC,6.8,10**9/L
R,SGOT,23,IU/L
V,Visit 2,20-Feb-04,14:23
R,RBC,4.4,10**6/uL
R,WBC,7,10**9/L
R,SGOT,21,IU/L
P,2,102,JFK
V,Visit 1,11-Mar-04,10:25
R,WBC,7.2,10**9/L
R,SGOT,19,IU/L
V,Visit 2,22-Apr-04,11:09
R,RBC,5,10**6/uL
R,WBC,10,10**9/L
R,SGOT,25,IU/L
F,19,,
```

Look at the first and last rows. The "H" and "F" denote the Header and Footer rows. The first character of each row denotes the row type: "P" for Patient Demographics, "V" for Visit, and "R" for Results. Each row type can have different numbers of fields, so the SAS input variables must be conditional upon the value of this first character.

THE SOLUTION

Use the SAS INPUT statement trailing @ operator to hold the pointer position while a conditional statement evaluates the value of the first character. Retain the values input from each row as needed. In the final flat file, all these retained values will be duplicated after each successive row type is input. Initialize the retained values and be sure you do not inadvertently carry a value across patients or test results.

```
filename lab 'c:\sugi29\raw_data\lab.csv';
libname  dir 'c:\sugi29';

data sasfile(drop=h_user h_datemade h_filename p_count);

    attrib  rowtype      label='RowType'    length=$1
            h_user       label='User'       length=$4
            h_datemade   label='DateMade'   length= 8
            h_filename   label='FileName'   length=$4
            p_count      label='PatCount'   length= 8
            p_patient    label='PatNum'     length=$3
            p_initials   label='PatInit'    length=$3
            v_visnum     label='VisNum'     length=$7
            v_visdate    label='VisDate'    length= 8
            v_vistime    label='VisTime'    length=$5
            r_testcode   label='TestCode'   length=$10
            r_result     label='Result'     length=$10
            r_units      label='Units'      length=$10
            f_records    label='NumRecords' length= 8;
```

You'll notice above the two date variables were assigned numeric lengths. These use the DATE9 INFORMAT to input the characters into SAS date values. Below the DLM option tells SAS what character to look for as a delimiter. The MISSOVER option tells it to enter a missing value if it finds two consecutive delimiters. The LRECL option stands for logical record length and should be set to the length of the longest record plus 4 bytes.

```
infile lab dlm=',' missover lrecl=32767;
input @1 rowtype $ 1. @;                    *** notice trailing at sign;

retain h_user h_datemade h_filename
       p_count p_patient p_initials
       v_visnum v_visdate v_vistime
       r_testcode r_result r_units;

if rowtype='H' then do;
  input @3 h_user
           h_datemade date9.
           h_filename;

end;
else if rowtype='P' then do;
  input @3 p_count
           p_patient
           p_initials;

  * initialize variables for new patient;
  v_visnum='';
  v_visdate='';
  v_vistime='';
  r_testcode='';
  r_result='';
  r_units='';

end;
else if rowtype='V' then do;
  input @3 v_visnum
           v_visdate date9.
           v_vistime;

  * initialize results for new visit;
  r_testcode='';
  r_result='';
  r_units='';

end;
else if rowtype='R' then do;
  input @3 r_testcode
           r_result
           r_units;

end;
else if rowtype='F' then do;
  input @3 f_records;
end;

format h_datemade v_visdate date9.;

run;
```

The last statement above assigns the SAS date values with the DATE9 FORMAT. The DATE9 FORMAT and the DATE9 INFORMAT are different. The DATE9 FORMAT is shown below as four digit years or DDMONYYYY. If you look back to the **lab.csv** printout, you will see the dates were provided in the DD-MON-YY style. The INFORMAT algorithm intelligently inputs two digit years with dashes or slashes. It also correctly inputs four digit years without slashes or dashes.

Displaying the interim data set at this point shows how the values from each row type build into a complete record for the results "R" row:

Row Type	Pat Num	Pat Init	VisNum	VisDate	Vis Time	Test Code	Result	Units	Num Records
H				.					.
P	101	FDR		.					.
V	101	FDR	Visit 1	10JAN2004	16:04				.
R	101	FDR	Visit 1	10JAN2004	16:04	RBC	4.3	10**6/uL	.
R	101	FDR	Visit 1	10JAN2004	16:04	WBC	6.8	10**9/L	.
R	101	FDR	Visit 1	10JAN2004	16:04	SGOT	23	IU/L	.
V	101	FDR	Visit 2	20FEB2004	14:23				.
R	101	FDR	Visit 2	20FEB2004	14:23	RBC	4.4	10**6/uL	.
R	101	FDR	Visit 2	20FEB2004	14:23	WBC	7	10**9/L	.
R	101	FDR	Visit 2	20FEB2004	14:23	SGOT	21	IU/L	.
P	102	JFK		.					.
V	102	JFK	Visit 1	11MAR2004	10:25				.
R	102	JFK	Visit 1	11MAR2004	10:25	WBC	7.2	10**9/L	.
R	102	JFK	Visit 1	11MAR2004	10:25	SGOT	19	IU/L	.
V	102	JFK	Visit 2	22APR2004	11:09				.
R	102	JFK	Visit 2	22APR2004	11:09	RBC	5	10**6/uL	.
R	102	JFK	Visit 2	22APR2004	11:09	WBC	10	10**9/L	.
R	102	JFK	Visit 2	22APR2004	11:09	SGOT	25	IU/L	.
F	102	JFK	Visit 2	22APR2004	11:09	SGOT	25	IU/L	19

The number of records above can be validated against the CONTENTS procedure. Since the values for the rows of type "H", "P", and "V" have been retained into the result records, only the "R" records need to remain in the data set and the variable rowtype can be dropped. Likewise, the number of records has been validated and the final row "F" can be removed and the variable f_records dropped:

```
data dir.sasfile(drop=rowtype f_records);
  set sasfile(where=(rowtype='R'));
run;
```

The final SAS output would appear as follows:

Pat Num	Pat Init	VisNum	VisDate	Vis Time	Test Code	Result	Units
101	FDR	Visit 1	10JAN2004	16:04	RBC	4.3	10**6/uL
101	FDR	Visit 1	10JAN2004	16:04	WBC	6.8	10**9/L
101	FDR	Visit 1	10JAN2004	16:04	SGOT	23	IU/L
101	FDR	Visit 2	20FEB2004	14:23	RBC	4.4	10**6/uL
101	FDR	Visit 2	20FEB2004	14:23	WBC	7	10**9/L
101	FDR	Visit 2	20FEB2004	14:23	SGOT	21	IU/L
102	JFK	Visit 1	11MAR2004	10:25	WBC	7.2	10**9/L
102	JFK	Visit 1	11MAR2004	10:25	SGOT	19	IU/L
102	JFK	Visit 2	22APR2004	11:09	RBC	5	10**6/uL
102	JFK	Visit 2	22APR2004	11:09	WBC	10	10**9/L
102	JFK	Visit 2	22APR2004	11:09	SGOT	25	IU/L

Another solution would be to create multiple data sets using output statements instead of inputting hierarchical files into a flat file.

THE FLAT FILE MODEL

THE SITUATION

You need to output a data set into a column defined flat file and create a data definition (DDT) table in MS Excel. Additionally, you want to create an audit trail to track file versions.

THE SOLUTION

First use the CONTENTS procedure with the NOPRINT option to create a data set containing the variable names, lengths, types, and labels. Create and assign a format to display the types as "Numeric" and "Character."

```
libname  dir 'c:\sugi29\analysis';
proc contents data=dir.sasfile noprint
              out=ddt_info(keep=name label type length);
run;

proc format;
  value type 1='Numeric' 2='Character';
run;
```

In a column defined flat file every field has a definite starting and ending column. Use the data set created by PROC CONTENTS to derive the starting and ending columns based on each variable's length. The LAG function is used since the starting column is based on the length of the previous variable. Each starting column is converted into a macro variable using the CALL SYMPUT function. For this data set eight macro variables (&col1 to &col8) are created. The SAS AUTO variable _N_ is concatenated onto our root macro variable name after each variable record is processed.

Using a date-stamp as a versioning tool is useful since it immediately answers the question of "When was the file created?" The International Organization for Standardization (ISO) date notation is YYYY-MM-DD (ISO 8601) and works particularly well as a file suffix because computers sort it chronologically. We convert the system variable "&sysdate9" into in the international format, which is YYMMDD10 in SAS. CALL SYMPUT turns it into a macro variable.

The purpose of the first retain statement is to write the variable list to the PROGRAM DATA VECTOR (PDV) in the same order as given and with the same letters in uppercase. The EXPORT and PRINT procedures both use the PDV variable order and case in their default outputs.

```
data work.ddt_info(drop=laglen);
  retain Variable Label Type StartColumn Length EndColumn;
  set ddt_info(rename=(name=variable)) end=last;
  format type type.;
  retain startcolumn 1 ;
  laglen=lag(length);
  if _n_ = 1 then startcolumn = 1;
  else startcolumn = startcolumn + laglen + 1;
  endcolumn = startcolumn + length;

  call symput('col'||left(_n_), startcolumn);
  if last then call symput ('datestamp', left(put("&sysdate9"d, yymmdd10.)));
run;
```

We now use the datestamp macro variable to uniquely name our output file in the FILENAME statement.

```
filename flatfile "c:\sugi29\listings\flatfile_&datestamp..dat";
```

The column defined file is produced in a DATA _NULL_ step. The macro variables col1 to col8 contain the starting columns for the output variables.

```
data _null_;
  set dir.sasfile;
  file flatfile;

  put @&col1 p_initials
      @&col2 p_patient
      @&col3 r_result
      @&col4 r_testcode
      @&col5 r_units
      @&col6 v_visdate
      @&col7 v_visnum
      @&col8 v_vistime;
run;
```

The column defined output from **flatfile_040115.dat**:

```
FDR 101 4.3         RBC         10**6/uL    10-Jan-04 Visit 1 16:04
FDR 101 6.8         WBC         10**9/L     10-Jan-04 Visit 1 16:04
FDR 101 23          SGOT        IU/L        10-Jan-04 Visit 1 16:04
FDR 101 4.4         RBC         10**6/uL    20-Feb-04 Visit 2 14:23
FDR 101 7           WBC         10**9/L     20-Feb-04 Visit 2 14:23
FDR 101 21          SGOT        IU/L        20-Feb-04 Visit 2 14:23
JFK 102 7.2         WBC         10**9/L     11-Mar-04 Visit 1 10:25
JFK 102 19          SGOT        IU/L        11-Mar-04 Visit 1 10:25
JFK 102 5           RBC         10**6/uL    22-Apr-04 Visit 2 11:09
JFK 102 10          WBC         10**9/L     22-Apr-04 Visit 2 11:09
JFK 102 25          SGOT        IU/L        22-Apr-04 Visit 2 11:09
```

To output the DDT we use the EXPORT procedure. The following code produces a Comma Separated Values (CSV) file which is easily read by MS Excel.

```
proc export data=work.ddt_info
            outfile="c:\sugi29\listings\flatfile_ddt_&datestamp..csv"
            dbms=dlm replace;
    delimiter=',';
run;
```

Macro variables begin with an ampersand and can end with other macro variables as shown earlier, e.g., &footnote&num. In order to make the file name with the extension, ".csv" above, two dots are needed. The first designates the end of the macro variable, while the second becomes literal. The file below shows the data definition table dressed up in MS Excel.

Variable	Label	Type	StartColumn	Length	EndColumn
p_initials	PatInit	Character	1	3	4
p_patient	PatNum	Character	5	3	8
r_result	Result	Character	9	10	19
r_testcode	TestCode	Character	20	10	30
r_units	Units	Character	31	10	41
v_visdate	VisDate	Character	42	9	51
v_visnum	VisNum	Character	52	7	59
v_vistime	VisTime	Character	60	5	65

THE MS EXCEL MODEL

THE SITUATION

Your client provides a list of codes and their corresponding decodes in an Excel spreadsheet. New data is regularly added to the list. You need to turn this data into a SAS format.

THE SOLUTION

Getting this data into SAS as a format is not difficult using the IMPORT procedure and the FORMAT procedure using the CNTLIN=option. Sample MS-Excel file **formlist.xls**:

Code	Airport
MCI	Kansas City
RDU	Raleigh-Durham
LAX	Los Angeles
PHL	Philadelphia
ATL	Atlanta
MIA	Miami
YUL	Montreal
SAN	San Diego
CLE	Cleveland

First use the following code to convert **formlist.xls** into a data set:

```
libname inlib "c:\sugi29\raw_data";
proc import out=inlib.formlist
            datafile="C:\sugi29\raw_data\formlist.xls"
            dbms=EXCEL2000 replace;
  getnames=yes;
run;
```

The second step is to add a few variables and to rename the variables *code* and *airport* before PROC FORMAT can use them. It is important to understand there are four keywords needed. These are START, LABEL, FMTNAME, and TYPE. START is easier to conceptualize when working with numeric formats. This is the left-hand column under the VALUE statement when assigning data directly with PROC FORMAT. Therefore we rename *code* to START and *airport* to LABEL. As FMTNAME and TYPE are constants, simply retain them for efficiency:

```
data newfmt(rename=(code=start airport=label));
  set inlib.formlist;
  retain fmtname '$aircode' type 'c';
run;
```

You have created a SAS data set filled with all of the formatting variables and values you need. The hard work is finished. The final step is to run PROC FORMAT with the CNTLIN=option:

```
proc format library=inlib cntlin=newfmt;
run;
```

CONCLUSION

When data is subject to constant revision consider using a text file to help you maintain and track changes. Practically all types of text files can be incorporated into SAS programs. One advantage of using a text file over a SAS data set is the text file does not have to be regenerated after every update. Therefore, revisions to the text file are automatically included in each calling program. Also, there is no delay waiting for the text file to regenerate a new data set, and thus negatively impact system resources. Another advantage is that users unfamiliar with SAS can maintain and update the text files. Other advantages include producing ad hoc requests easily, handling large numbers of directories, processing more than ten footnotes, and creating output especially designed for MS Excel. Furthermore, placing text files in a central location reduces redundancy and gives gatekeepers control over data integrity.

REFERENCES

Anderson, Pete, June 10, 2003, "Macro Design Concepts", Kansas City Area SAS Users Group, Overland Park, KS, USA.

Lund, Pete, April 14, 2002, "More than Just Value: A Look Into the Depths of PROC FORMAT", SUGI 27, Orlando, FL, USA.

ACKNOWLEDGMENTS

Special thanks to everyone at Quintiles for their contributions, support, and development of this paper.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Please contact the author at:

David J. Austin
Quintiles, Inc.
10245 Hickman Mills Dr.
Mailstop: F3-M3522
Kansas City, MO 64137
Email: Available through my web page
Web: http://home.kc.rr.com/existentialist