The Ugliest Data I've Ever Met

Derek Morgan, Washington University Medical School, St. Louis, MO

Abstract

Data management frequently involves interesting ways of doing things with the SAS® System. Sometimes, "interesting" becomes another way of saying "messy", especially when it comes to processing files from outside of your department. This paper will detail the code used to transform raw data we received into raw data that we could use. At first glance, flat file-to-flat file translation wouldn't seem to be a problem. However, this is a list of the programming methods necessary to perform this particular transformation:

- Automation of a process using SAS System macros
- Reading from and writing to multiple files with a single DATA step
- Using the SAS System to write its own code "on the fly"
- Adding sequence numbers for new records in a data set
- "Merging" without sorting
- "Arrays" of SAS System macro variables

The code in this real-life production application will be used to illustrate each of these methods, and to show the reasoning behind their use.

This task would have been much more difficult without the use of any of these tricks. Without using the macro facility, it would have been necessary to run the application 23 different times, changing a filename each time. Being able to process multiple files in the same DATA step avoided writing multiple DATA steps within the application, or some convoluted method of using macro variables to obtain the correct filenames.

Using the arrays of macro variables made it possible to automate the entire process. Instead of passing parameters to a macro call, or having to know how many iterations ahead of time, the application will be able to handle similar situations on its own.

Finally, by using the application to write its own code, it was possible to obtain the variables to be read from one file, then read them with a DATA step in the same application.

All of these methods are documented (somewhere). With the exception of creating an external file containing fully qualified file names, all of this was done within the SAS System for UNIX, version 6.12, and should therefore be portable to other operating systems.

The Problem

The Program Data Center of The Family Blood Pressure Program is responsible for gathering and pooling the data from four separate genetic studies. Pooling the phenotype data involves conversion from four different formats and finding lowest common denominators for each data item.

The genotyping data for each study is done by a single facility, which issues the data in a standardized format. Unfortunately, the cleaning and preliminary analysis of this data is done separately by each study. The data is delivered to the Program Data Center after this stage has been done. Different studies use differing analysis programs, each of which have their own format.

There already was an application to create a SAS System data table from the standardized format flat file data, which had been in use for a while. However, each study sends their cleaned data in the format that suits them the best, which may not be the same as the standardized format. Therefore, translation, in some way, is frequently necessary.

The following table describes the differences between the original file format and the alternate file format for which this application was built.

Standard Format	Alternate Format
1 file per genetic marker	Multiple markers per file
1 record per subject in each file	1 record per subject in each file
The name of each file is the marker name	Each file name is the chromosome number
380-405 markers in a batch, one directory	380-405 markers in a batch across 4 different directories, 96 total data files
2 data items (columns) per marker	2 data items (columns) per marker
Marker names can't automatically be used as variable names (Version 6 limitation)	Marker names can't automatically be used as variable names (Version 6 limitation)
	Which markers are in each data file are specified in a corresponding information file
	Actual data values must be obtained from a translation table
	Translation table is keyed by marker name
Desired Result	
One SAS System data table containing 1 record per subject. Each record must contain 405 columns, each representing a single marker.	

This is a brief description of the overall problem. The solution lay in creating the SAS System data table in two stages. Since there was an existing application that created the desired table from standardized format flat files, there was no need to write this application to do the same. Instead, it was quicker and easier to create this

application to re-format the files into the standardized format. This ultimately saved a lot of programmer time, although two separate programs had to be submitted.

Where to Start?

It's always easier to work with smaller problems. By tackling one problem at a time, the last question was about the integration of parts, not how to automatically reformat all of those files. To start this process, we need to get the information about what variables are in each file. There are 24 data files in each of 4 directories. Each of these files has its own corresponding information file that contains an ordered list of the variables in the data file.

We will create a SAS System table with these fully qualified file names (using the FILENAME statement won't work for this.) This filename table will then be used at different points to read the information and data files. This application starts with an operating system command to make a flat file of the directory listing (①).

Step 1: Obtain external file names, and create a SAS System table containing them

```
X 'ls /path991006/data*/*.data > tmpit';   ①
DATA files;
LENGTH dataname mfname $ 200;
INFILE 'tmpit' PAD MISSOVER;
INPUT dataname $;
quit = INDEX(dataname,'.')                          ②
mfname = SUBSTR(dataname,1,quit) || 'info';
DROP quit;
RUN;
```

Since there are two files associated with the data file, we read the name of one file, and create the other by dropping the letters after the "." in the file name, and replacing them (②). Now we have a table with the fully qualified path names of both the data and information files in all four directories. This will be referred to as the "path table." The next step will use the information files to make a table containing all the variable names in all of those files.

Step 2: Create Data Set of All Marker Names Across All Files

```
DATA biglist;
LENGTH marker $ 10;
SET files;          ①
INFILE in FILEVAR=mfname PAD MISSOVER
          END=eof;      ②
DO UNTIL(eof);
     INPUT marker $ ;                    ③
     marker = UPCASE(marker);
     DROP dataname;
     OUTPUT;
END;
RUN;
```

This is an example of how to read many files with a single DATA step using the FILEVAR option. The SET statement (①) refers to the path table created in step 1. The FILEVAR option for the INFILE statement (②) refers to the information file column in the path table. Note the END option here; this is critical.

③ is the DO UNTIL loop that processes the information files. It is keyed to the end-of-file indicator. Why do we need a loop inside of this DATA step? The DATA step will iterate once for each record in the table *FILES*. Therefore, the input statement will only execute once for each file. Without the loop, only the first record from each information file will be processed.

The OUTPUT statement is in the loop for a similar reason. Without it, only the last record from each information file would be output, since the default output operation would occur at the end of the record from the table specified in the SET statement. To capture all lines from all files, you must use the loop with the output statement. The END option tells the SAS System when to stop processing the information file (at end-of-file.) Finally, the DROP statement is used to remove unneeded columns from this table.

The table created in step 2 is the "column label" table. It will be used to figure out the column names. We will eventually use it to create sequence numbers that will be matched with column names, and we can then read the data from the data files. The sequence numbers are used to get around the eight-character variable name restriction, and would not be necessary were this application developed in versions 7 or above.

The next step is in preparation for automation of the process.

Step 3: Create "Arrays" of Macro Variables for Automation

```
data _null_;
length mfile dfile $ 4;
set files;
mfile = "M" || left(put(_n_,3.));       ①
dfile = "D" || left(put(_n_,3.));
call symput(mfile,trim(left(mfname)));                ②
call symput(dfile,trim(left(dataname)));
call symput('iter',left(put(_n_,4.)));          ③
run;
```

Step 3 uses the path table to create two "arrays" of macro variables: one for information files, and one for data files. It also creates an iteration target because the number of records in the path table is the same as the number of files to be processed. A column name is created in the character variables *mfile* and *dfile* by appending the DATA step iteration counter (_N_) to the letters "M" and "D", respectively (①). The CALL SYMPUT statement (②) uses these to create macro variables named *&M1, &M2, &M3, &ETC*. Each "M" variable contains a full path names for a single information file, while the "D" variables work the same way with respect to the data files. *&ITER* is the iteration target macro variable.

The next step in the process concerns the maintenance of the application-specific genetic marker dictionary, and is of little relevance to the SAS System. However, if a previously unencountered marker is found, the dictionary must be updated and the new record(s) assigned a sequence number. Here's how the sequence number is obtained.

SAS Tip: Adding New Sequence Numbers To An Existing File

```
PROC MEANS DATA=genotype.newrose MAX
NOPRINT;
VAR markno;                                    ①
OUTPUT OUT=himark MAX=maxmark,
RUN;

DATA _NULL_;
SET himark;                    ②
CALL SYMPUT('maxmark',LEFT(PUT(maxmark,4.)));
RUN;

DATA prepnew;
SET newrecs;                   ③
markno = INPUT(SYMGET('maxmark'),4.) + _N_;
RUN;
```

The above code shows a method of adding a sequence number to a record that is to be appended to an existing file with sequence numbers. This method does not re-sequence the entire file. The first step (①) is to find out the highest sequence number in the existing file using PROC MEANS. Next, that number is transferred to a macro variable in ②. The final step is to process the file with the new records. Add the iteration counter to the value of the macro variable. The data table PREPNEW will have sequence numbers starting from the ending point of the existing file. This can be done more economically using the VARSTAT function in SCL, which will allow you to avoid using macros.

At this point in the application, all of the preprocessing necessary to handle the file translation has been completed. We can now automate the remainder of the process with the SAS System Macro Facility.

Automating the Process With Macros

As noted earlier, there are 96 data files, many with different structures, spread across 4 directories. The total number of columns represented in all of these files is unknown, but should be around 400. Since each file has a different structure, 96 different programs will be necessary to read the data. When the next group of data arrives, there may only be 72 files in one directory, with approximately 400 columns, some of which may be different from the ones we've already received.

Changing the INPUT and FILENAME statements by hand seems like a large waste of time, and is prone to error. Since we want the maximum amount of flexibility built into the application, any way of automating this process is highly desirable.

Since the names (and number) of files to be processed is in the table FILES, we created an iteration target macro variable back in step 2 so that we don't have to know how many iterations are needed in advance. Each iteration of the macro should work on one pair (data, information) of files. The macro handles the actual reading of the information and data files, the translation of the aliases, and the writing of the standard format flat files.

Step 4: Get Marker Sequence Numbers

```
OPTIONS NONOTES;
%MACRO doall;
%DO i=1 %TO &iter;    ①

/* namefils is marker list file  */
/* datafile is marker data file  */

FILENAME namefils "&&M&i";
FILENAME datafile "&&D&i";        ②

/* Read marker list for this file */
DATA marklist;
LENGTH marker $ 10;
INFILE namefils PAD MISSOVER;              ③
INPUT marker $ ;
marker = UPCASE(marker);
RUN;

PROC SORT DATA=marklist out=markmrge;
BY marker;                                 ④
RUN;

PROC SORT DATA=chkrose OUT=rosetta;
BY marker;
RUN;

/* Merge to obtain the sequence numbers used to read
the data file. */

DATA markers;
MERGE markmrge (IN=ina) rosetta (IN=inb);
BY marker;
RENAME marker=start;                       ⑤
label = PUT(markno,3.);
fmtname = "$markno";
RUN;

PROC FORMAT cntlin=markers;
RUN;

data marklst2;
set marklist;
markno = INPUT(PUT(marker,markno.),3.);
RUN;
```

The beginning of the macro assigns symbolic SAS System FILENAMEs to the actual path of the input files (②) for use in the INFILE statements used in the macro, one of which is at (③). This avoids any quoting problems. The iteration target is at (①).

The next bit of trickery involves "merging without sorting." The marker names are initially read in the order that they are listed in the data file. This is the order in which the data must be read. Unfortunately, in order to get the marker sequence numbers, this list must be merged with the rosetta stone table, which is sorted by marker name. So we sort the marker names into another table, MARKMRGE (④), which allows us to retain the original ordering in MARKLIST. The merged table MARKERS (which contains the marker sequence numbers matched with their corresponding marker names) also serves as a format control data set. The PROC FORMAT reads this in and creates a format named $MARKNO. Although

MARKERS is sorted by marker name, in (⑥), we create the table MARKLST2 with the original order by using the format MARKNO to associate the marker sequence number with the marker name. The results are shown in the tables below.

TABLE MARKLIST (original order)

obs	marker
1	GATA44F10
2	GATA21G05
3	GATA23B01
4	GATA66B04
5	GGAA2A03
6	MFD235
7	MFD139
8	MFD232
9	GATA29B01
10	MFD238

TABLE MARKMRGE (marker name order)

obs	start	markno
1	GATA21G05	169
2	GATA23B01	176
3	GATA29B01	203
4	GATA44F10	241
5	GATA66B04	281
6	GGAA2A03	369
7	MFD139	394
8	MFD232	397
9	MFD235	398
10	MFD238	399

TABLE MARKLST2 (original order again)

obs	marker	markno
1	GATA44F10	241
2	GATA21G05	169
3	GATA23B01	176
4	GATA66B04	281
5	GGAA2A03	369
6	MFD235	398
7	MFD139	394
8	MFD232	397
9	GATA29B01	203
10	MFD238	399

Now that we have sequence numbers for the markers listed in the information file, we can generate the program that will read the corresponding data file.

Step 5: Generate Code To Read Data File

```
DATA _null_;
SET markers END=eof;        ①
LENGTH str $ 7;
FILE "link2marsh.tmp.sas" NOPRINT;        ②
IF _N_ EQ 1 THEN DO;
  PUT "DATA chr1;";
  PUT "INFILE datafile PAD MISSOVER;";                ③
  PUT "INPUT famid $ sid $ fid $ mid $ sex" @;
END;
str = "m" || PUT(markno,Z4.) || "_1";
put +1 str @;                                         ④
str = "m" || PUT(markno,Z4.) || "_2";
PUT +1 str @;
IF eof THEN
  PUT ";";
  PUT "IF famid EQ ' ' THEN DELETE;";                 ⑤
  PUT "RUN;";
RUN;
%INCLUDE "link2marsh.tmp.sas";        ⑥
```

One important thing to remember about writing SAS System code using a program is that there will be certain parts of your code that needs to be written only once. The static code in this program is at ③ and ⑤. The END= option (①) is used to flag the end of processing for the table MARKERS. The NOPRINT option on the FILE statement (②) Keeps the SAS System from adding carriage control characters to each line. In ③, the first part of the static code is written. This includes the DATA and INFILE statements, as well as the segment of the INPUT statement that is not data-dependent. The trailing at ("@") is used to keep the INPUT statement from breaking across lines.

The code in ④ creates legal column names from the sequence numbers and writes them to the code file. First, an "M" is prepended to the formatted value of the sequence number. Then, since each marker has 2 pieces, "_1" or "_2" is added to the end of the sequence number to indicate first or second piece. The PUT statement advances the column pointer by 1 (to separate words), writes the column name, then holds the line for the next write operation.

The last section of static code is written when the end-of-file condition for the table MARKERS occurs. This puts the semicolon on the INPUT statement, writes the instruction to delete any data line without an identifier, and the "RUN;" statement. The INCLUDE statement (⑥) executes the program just generated. Below is actual code produced by this step during an iteration of the macro.

```
DATA chr1;
INFILE datafile PAD MISSOVER;
INPUT famid $ sid $ fid $ mid $ sex
m0002_1  m0002_2  m0035_1  m0035_2
m0048_1  m0048_2  m0092_1  m0092_2
m0160_1  m0160_2  m0168_1  m0168_2
m0192_1  m0192_2  m0245_1  m0245_2
m0270_1  m0270_2  m0278_1  m0278_2
m0303_1  m0303_2  m0320_1  m0320_2
m0326_1  m0326_2  m0336_1  m0336_2
m0343_1  m0343_2  m0364_1  m0364_2 ;
IF famid EQ ' ' THEN DELETE;
RUN;
```

Taking Care Of Aliases

Now that the data for a particular chromosome has been read, we need to move on to the next step, which is translation of the values in the data file to the values that we use in the standard format. The fact that each marker has its own set of translations for a given alias complicates things. In other words, an alias value of "1" has different meanings for each marker. Fortunately, we were given a translation table keyed by marker name and alias value.

However, we'll need to change the table from its current orientation. There is one record per subject, with several columns. In order to do the alias translation, we need one record per column with two values, ALIAS and MARKNO, while retaining the subject's identifying information on each record. PROC TRANSPOSE was designed for situations like this. The other problem in translation is to turn the sequence numbers back into their names. This whole exercise would have been much more economical and easier under version eight.

Step 6: Translating Aliases

```
PROC SORT DATA=chr1;
BY famid sid mid fid sex;
RUN;

PROC TRANSPOSE DATA=chr1 OUT=trans1;
BY famid sid mid fid sex;          ①
RUN;

/* The transposed data set is prepared for translation.
Merge variables ALIAS and MARKNO are created,
along with an allele number to be used later.  */

DATA tmp1;
SET trans1 (RENAME=(col1=alias));
markno = INPUT(SUBSTR(_name_,2,4),4.);
allelno = INPUT(SUBSTR(_name_,7,1),1.);          ②
DROP _name_;
RUN;

PROC SORT DATA=tmp1;
BY markno;
RUN;

PROC SORT DATA=rosetta;
BY markno;
RUN;

/* Get the marker names (or version 8) */

DATA temp1;
MERGE tmp1 (IN=ina) rosetta;
BY markno;
IF ina;
DROP markno;
RUN;

PROC SORT DATA=temp1;
BY marker alias;
RUN;

PROC SORT DATA=local4.marktran;
BY marker alias;
RUN;

/* Translate LINKAGE aliases to fragment lengths. Alias
values of 0 are translated as fragment lengths of 0. */

DATA tempa1;
MERGE temp1 (IN=ina) local4.marktran (IN=inb);
BY marker alias;
IF ina;
IF alias EQ 0 THEN
    fraglen = 0;
DROP alias;
RUN;
```

There are only two things of note in the above code. The BY statement in the PROC TRANSPOSE (①) associates the record identifying information with each record that is transposed. This means that the identifying information, column name and alias will all be columns in the new data table, rather than having them as records in the new table. The second point is obtaining the sequence number from the column name and the pair count (②). The SUBSTR function selects the character positions where these numbers are located, and the INPUT function does the character-to-numeric translation. Once we have the sequence numbers, we can merge to get the marker names. The pair counter ALLELNO will be used when it comes time to pair the data for output. The translation itself is a plain one-to-one merge.

Now that the data has been translated, and the table TEMPA1 contains marker names and fragment lengths, we are almost ready to write this to the correct files. The marker data still needs to be paired properly. All that needs to be done is to write one record for every two in the table TEMPA1.

Step 6: Pair Data For Output

```
DATA tempb1;
SET tempa1;
LENGTH allele1 allele2 3;
RETAIN allele1;          ①
DROP allelno fraglen;
IF allelno EQ 1 THEN
  allele1 = fraglen;
IF allelno EQ 2 THEN DO;
  allele2 = fraglen;
  OUTPUT;
END;
```

```
RUN;
```

This is a simple method using the RETAIN statement (①) to keep the value from the first record to be paired available when the second record is processed. When the second record is processed (using ALLELNO to determine the pair), it is output. Now we have the table tempb1 in the proper format for writing the flat ASCII files.

Step 7: Write Data To Flat Files

```
PROC SORT DATA=tempb1;
BY marker;                          ①
RUN;

/* This writes out the ASCII file in MGS format. FV is the
name of the file to be written to, which is (marker-
name).out. As the value of FV changes, the output file
changes because of the FILEVAR option. The MOD
option avoids overwriting data in the existing output files.
Any MGS output from a previous session should be
erased before running this program */

OTPIONS NONOTES;        ②
DATA _null_;
LENGTH fv $ 100;
SET tempb1;
fv = "/local4/" || TRIM(marker) || ".out";          ③
FILE writeout FILEVAR=fv MOD;          ④
PUT famid +3 sid +3 fid +3 mid +3 sex +3 allele1 +3
      allele2 +3 '.';
RUN;
OPTIONS NOTES;
%END;
%MEND doall;
```

This final segment demonstrates the use of the FILEVAR option to write multiple files with a single DATA step. There are a few things to be aware of when doing this. The SAS System usually displays a message in the log when an external file is opened. If your data are not sorted so that each output file is only opened once, you will get a message each time the output file changes. In the application, we're writing to 405 different files, a record at a time for each. The log file would be enormous! Fortunately, if you forget to sort properly (①), the NONOTES option (②) will come to your rescue, and prevent the log from filling a disk volume.

The rest of the writing operation is straightforward. In ③, the external file name is assembled using the table column MARKER, and stored in the column FV. The FILE statement (④) has a dummy filename ("writeout"), but the file is actually specified by the column in the FILEVAR option. The MOD option on the FILE statement prevents the file from being restarted each time it is opened. From there, it's just a matter of using whatever PUT statements you need to produce the desired output.

Summary

Without the macro facility and language, this task would have been impossible to automate. Using the FILEVAR option simplified the handling of the multiple input and output files without confusion. Finally, by using the application to write its own code based on data obtained from a flat file, the changing input parameters were also taken care of in an automated fashion.

Although each of these techniques is well documented, this application demonstrates a method for applying them in a sequence that solved a seemingly complex problem.

Acknowledgements

This work was supported by NHLBI grant U10 HL54473.

Further inquiries are welcome to:

Derek Morgan
Division of Biostatistics
Washington University Medical School
Box 8067, 660 S. Euclid Ave.
St. Louis, MO 63110
Phone: (314) 362-3685 FAX: (314) 362-2693
E-mail: derek@wubios.wustl.edu

The sample application and this paper are available via the World Wide Web at:

http://www.biostat.wustl.edu/~derek/sasindex.html