Paper 256-2011

Finding Your Way Through the Wilderness: Moving Data from Text Files to SAS® Data Files

Charley Mullin, SAS Institute Inc., Cary, NC, USA

ABSTRACT

As a SAS® software user, you have undoubtedly encountered raw data files that have defied your attempts to read them into SAS and make them into usable SAS data files. Working with such files might make you feel a little lost—like you are wandering around a remote wilderness. This paper helps you to get back on course by presenting solutions to the challenging problems that sometimes occur when you try to read external files into SAS.

The paper also addresses advanced data problems and provides creative techniques that enable you to manipulate the data so that it is easier to read. The discussion covers everything from hierarchical structures to the logic that is required for using regular expressions in the input buffer in order to make the data readable.

INTRODUCTION

This paper demonstrates how to move complicated text files into SAS so that your SAS data files are usable and readable. Sometimes clear direction for accomplishing this task is not always evident. You can compare this process to hiking on a well-marked path. Somehow you veer off course and realize that you are standing in the middle of the wilderness. It is not immediately obvious how to get back to the right path. This scenario is evident in the calls that SAS Technical Support frequently receives from customers who have problems reading data files into SAS. Some common problems that customers have reported involve incorrect or inconsistent end-of-record (EOR) markers, complex delimiters, hierarchical files, occurs files, and foreign encoding.

In many cases, you can read really complicated text files in a single DATA step. However, coding the DATA step can be complex and as intimidating as being off course in the middle of the wilderness—even to the most experienced SAS programmers who might think that they need to hire someone to do the task for them.

However, this task is not as complicated as it seems, especially when you take the right approach. Breaking the task into multiple steps can make each step easier to understand. Employing this modular approach to the DATA step also makes it easier to reuse parts of the code for the next task rather than simply starting over. Writing multiple simple DATA steps is typically faster than writing one complex DATA step. The overall benefits of this approach are that testing, debugging, and maintenance are simplified, and the resulting program is modular, flexible, and easy to maintain.

To make the data easier to read, you might need to make multiple passes through the text to preprocess the data. It might be necessary to insert, remove, or change some characters, or to split the text file into multiple pieces. Necessary is a relative term, but in this case, these steps are necessary because they will simplify a difficult and sometimes problematic task—and keep you on the right path. This paper elaborates on the following techniques for resolving challenging problems that occur when reading external files:

- preprocessing files before they are read
- modifying records as they are read
- applying hierarchical file logic
- using tools for foreign encoding
- reading an occurs file

Advanced data problems and creative solutions for these problems are also discussed.

SOLUTIONS FOR CHALLENGING PROBLEMS

Where possible, the techniques that are presented in this section use new functionality that is available in SAS 9.2.

PREPROCESSING FILES BEFORE THEY ARE READ

There are two ways to preprocess external files:

- by updating the file in place, meaning that you change individual bytes but do not add to or subtract from the total number of bytes in the original text file
- by rewriting the existing file into a new file with the required changes

Before you decide whether to update a file in place or create a new one, consider the ramifications of your choice. Creating a new file doubles the disk space that is used. However, the original file is always available. Modifying the file saves disk space, but requires re-creating or reloading the original file if regression becomes necessary. In most cases, modifying a file is faster than rewriting it.

The following examples show two different methods of eliminating extraneous line feeds or EOR markers.

Updating Files in Place

When you look at your output, the most common symptom of having too many or inconsistent EOR markers in a file is seeing records that split when the text is read. The SAS log will not always have errors in this situation. In the output data set, one or more observations might abruptly end, frequently in the middle of a character variable. The subsequent observation will have incorrect data in the first few variables and missing data in the latter variables. This problem occurs if there is a line feed in a comment field. This situation is common when you are getting data from Excel files and the character fields have soft returns in them. The following program is one example of how to solve the problem with SAS releases prior to SAS 9.2.

```
data _null_;
infile 'c:\_today\mike.csv' recfm=n sharebuffers;
file 'c:\_today\mike.csv' recfm=n;
input a $char1.;
retain open 0;

a='"' then open=not open;
if a = '0A'x and open = 1 then put ' ';
run;
```

Because there is no way to type a character for a line feed, you must specify the line feed by another method. The most common method is to use a hexadecimal value, which looks like '0A'x. The SHAREBUFFERS option tells SAS to use a single data buffer for both input from and output to the external file that is being modified. This eliminates unnecessary data movement, which improves performance. RECFM=N tells SAS to treat the entire file as a single record. Control characters, such as an EOR marker, are also treated as data.

The external file is opened for both reading and writing at the same time. This way, SAS can update the file in place without creating a new file. The INPUT statement reads through the file character-by-character. When double quotation marks (") are found, the value of the OPEN variable is changed. The OPEN variable is a standard numeric variable that will have only one of two values: **0** or **1**. It is used as a logical flag. When a line feed is found, the value of **OPEN** is tested to see whether the line feed was found between double quotation marks. If so, it is overwritten by a blank space.

Rewriting a File into a New File

The following program shows how to remove extraneous line-feed characters that do not have quotation marks around them. The program also creates a new file from the old one. The strategy is to count the number of delimiters between EOR markers. If the count is low when an EOR is found, you know that the line feed should not be there so it is not written to the new output file. In the following example, the data file is supposed to have 16 variables on each record. This means that there are 15 delimiters on each record. Notice that the delimiter is not a comma.

```
data _null_;
infile 'c:\_today\sample.dat' recfm=n;
file 'c:\_today\sample_.dat' recfm=n;
input a $char1.;
if a = "^" then c+1;
if a= '0A'x then do;
   if c = 15 then do;
     c=0;
     put '0A'x;
   end;
end;
else put a $char1.;
run;
```

Beginning with SAS 9.2, problems with extraneous line feed characters are typically handled by using the TERMSTR= option in the INFILE statement. If you are working on a PC and have a file that was created on a PC, then you can solve this problem by using TERMSTR=CRLF. This tells SAS that the only valid EOR marker is the

combination of a carriage return (CR) and a line feed (LF) together (CRLF) and in that order. By default, CRLF is the way in which a PC writes an EOR marker.

This technique is also perfect for solving the problem of reading PC-formatted files in a UNIX operating environment. The difference is that the EOR marker on UNIX platforms is a single line-feed character. As stated previously, the EOR marker on PC-generated files is CRLF. When SAS reads data from files that were created on PCs in a UNIX operating environment, a carriage return is treated like data. If the last variable in the INPUT statement is a character variable, then you will have an extra non-blank character at the end of your variable. If the last variable is numeric, you will get an invalid data message and the variable will have a missing value on every observation, even though the data looks fine in the text file when it is viewed with a text editor.

Prior to SAS 9.2, the recommended fix was to add the carriage return to the delimiter list, requiring the programmer to specify both delimiters by their hexadecimal values. The hexadecimal value of a comma on UNIX and PC systems is "2C"x. Adding the value "0D"x for the carriage return to the DLM= option in your INFILE statement looks like this:

```
infile "/u/sasxxx/SGFdata/sample.csv" dlm='0D2C'x truncover lrecl=1023;
```

With SAS 9.2, you only need to specify TERMSTR=CRLF and the problem resolves.

MODIFYING RECORDS AS THEY ARE READ

With SAS 9.2, additional features were added to enable you to parse input more easily. For example, one addition is the new INFILE option, DLMSTR=. This option tells SAS which character or characters are used to delimit variables in each record. When using DLM=, each individual character that is listed is treated as a delimiter. Using DLMSTR= specifies that the entire string is a single delimiter. So if you are reading a file in which each variable is delimited by a double pipe (||), use DLMSTR='||' to parse the records.

If SAS 9.2 is not available on your system, you can use the PRXCHANGE function to change the double pipe to a single pipe (|) so that a typical INPUT statement will read the record correctly. Here is what the modifications look like when you change a record while it is still in the input buffer and before the INPUT statement parses the data into individual variables:

```
infile 'c:\_today\rodney.txt' dlm='|' dsd truncover lrecl=4095;
input @;
_infile_=prxchange("s/\|\|/|/",-1,_infile_);
input a b c d e;
```

The first INPUT statement with the trailing at sign (@) reads the record into the input buffer and holds the input pointer on the same record at column 1. PRXCHANGE replaces every pair of double pipes with a single pipe while the data remains in the input buffer so that the second INPUT statement can parse it normally. Note that the escape character (\) is required on the matching side of the Perl regular expression, but not on the side of the substitution.

Again, with SAS 9.2, it is even easier to modify records with the DLMSTR function. Add the DLMSTR function to the INFILE statement like this:

```
infile 'c:\_today\rodney.txt' dlmstr='||' dsd truncover lrecl=4095;
```

Another tool, which is available in SAS 9.2, that could be useful in this situation is the TRANSTRN function. TRANSTRN is like TRANWRD, which will replace one string with a different string, even of a different length. The difference is that TRANSTRN will do the substitution for all matches that are found in the source variable instead of for only the first match that is found. Here is an example of using the TRANSTRN function:

```
_infile_=transtrn(_infile_,'||','|');
```

APPLYING HIERARCHICAL FILE LOGIC

There are two different types of hierarchical files. The first type of file has a code in the same position on each record that indicates to which group the record belongs. An example of this type of file is a product survey output. The second type of file indicates which group the record belongs to by the position of the data. Stated in a different way, the presence, or absence, of data in a particular column or position in the record indicates the group in which that record belongs. The following discussion includes examples of both types of hierarchical files.

In this first example, the indicators are numbers. The number 1 signifies the start of demographic information, the participant ID, and the survey date. The number 2 in the first column indicates product information. The number 3 indicates the survey data. Note that one person can be surveyed on more than one product. The logic for generating a data set with one survey per observation is straightforward—RETAIN everything that is read for record types 1 and 2, then output an observation for every record type 3. If the goal is to have all surveys for an individual on a single

observation, do a simple read, then later collapse the observations by participant ID. Separating the tasks simplifies the logic. Here is the sample data and code:

```
1~0012345~06/17/2010
2~G17~9-Conv
3~1~1~3~3~4~1
3~2~2~2~4~3~1
3~3~2~2~2~5~1
2~G21SF~45-Conv
3~1~2~2~2~1~2
3~3~2~2~3~1~4
```

```
data survey;
infile 'c:\_today\charley.dat' truncover lrecl=1023 dsd dlm='~';
input @1 t 1. @;
If t=1 then input id date :mmddyy10.;
if t=2 then input model $ desc :$20. ;
if t=3 then do;
   input config acc feel site size rec
   output;
end;
retain _all_;
run;
```

In the next example, records are grouped by the position of data. The positional hierarchical file might look more complex, but it is not. This example involves reading a rejected credit card transaction file. The following sample data was modeled after actual customer data that was sent to SAS Technical Support:

```
100074                                  02                                  Transaction failed
                0000450003010920334 3   501-10025       Product Type    VISA    Product can not...
                                                        Product Role    9
----+----1----+----2----+----3----+----4----+----5----+----6----+----7
100143                                  02                                  Transaction failed
                0000450060010564385 6   500-10045       Field Name      Street  Invalid input -
                                                        Field Value     C/O CIBC TRANSIT 03
                                                        Unacceptable WorCIBC
                0000450060010564385 6   500-10045       Field Name      Street  Invalid input -
                                                        Field Value     C/O CIBC TRANSIT 03
                                                        Unacceptable WorC/O
                0000450060010564385 6   500-10045       Field Name      Street  Invalid input -
                                                        Field Value     C/O CIBC TRANSIT 03
                                                        Unacceptable WorCIBC TRANSIT
```

In the preceding file, there are three areas of each record that need to be tested: columns 1, 17, and 56 (highlighted in yellow). For each record that has data in column 1, there will be at least one record that has data in column 17 and up to 5 parameters that are related to this individual's credit card transaction. In your output, you want all transactions for each credit card to appear in a single observation that also includes the ID of the account. Start by declaring all of the variables and arrays that are needed, as shown in this sample code:

```
data debby;
infile "c:\_today\debby.txt" truncover end=done;
length a b c $1 req_id  stat_cd 8 except $60 card_num $20 stat_cd1 $10 except1 $60;
array prm_n(5)$12 parm_nm1-parm_nm5;
array prm_v(5)$20 parm_val1-parm_val5;
```

Just as you did in the first example, start each iteration of the DATA step by determining the type of the record. Do that by testing each position of each record. If the first character is not blank, then this is an ID record. If column 17 is not blank, then this is a credit card number record and the first set of parameters. If column 56 is not blank, then this is a parameters record that goes with the previous credit card. One more piece of the logical puzzle is knowing when all of the information that is needed for a complete observation has been gathered, so that you can commit the observation to the output data set. When a new credit card number is read, if blanks are read from all three test positions, or if the end of the file is reached, it is time to output the information that is currently in the Program Data Vector.

When the observation is output to the data set, it is time to clear the data that has accumulated for the observation that was just written to the output data set. Because the ID information can appear on more than one observation, and the credit card information will be read for each observation, the only variables that need to be cleared are the two arrays.

In the following code, the MISSING call routine assigns missing values to the variables that are listed inside the parentheses:

```
input a $1 b $17 c $56 @;
if a ^= ' ' or b ^= ' ' then do;
   if _N_ > 1 then output;
   if a ^= ' ' then input @1 req_id @40 stat_cd @91 except $60.;
   pct=1;
   call missing(of parm_nm1-parm_nm5);
   call missing(of parm_val1-parm_val5);
   input card_num $17-36 stat_cd1 $40-48
         parm_nm1 $56-71 parm_val1 $72-90
         except1 $91-141 ;
end;
```

The ELSE statement indicates that if column 57 is not empty, then there is information about this transaction to read, as shown below

```
else if c ^= ' ' then do;
   pct+1;
   input prm_n(pct) $56-71 prm_v(pct) $72-90;
end;
else delete;
drop a b c i pct;
retain _all_;
if done and pct>0 then output;
run;
```

The resulting output data set is shown in Display 1:

	req_id	stat_cd	except	card_num	stat_cd1	parm_nm1	parm_val1
1	100023	2	Transaction failed due to service error	0000450440001866846 2	002-11001	Description	No record afftected
2	100018	2	Transaction failed due to service error	0000450228693100747 0	500-10045	Field Name	Street
3	100017	2	Transaction failed due to service error	0000450228693100746 2	500-10045	Field Name	Street
4	100074	2	Transaction failed due to service error	0000450003010920334 3	501-10025	Product Type	VISA
5	100143	2	Transaction failed due to service error	0000450060010564385 6	500-10045	Field Name	Street
6	100143	2	Transaction failed due to service error	0000450060010564385 6	500-10045	Field Name	Street
7	100143	2	Transaction failed due to service error	0000450060010564385 6	500-10045	Field Name	Street
8	100162	2	Transaction failed due to service error	0000450060000767035 3	501-10025	Product Type	VISA
9	100224	2	Transaction failed due to service error	0000450553002377860 6	501-10025	Product Type	VISA
10	100207	2	Transaction failed due to service error	0000450550000727015 1	002-11001	Description	No record afftected
11	106775	2	Transaction failed due to service error	0000450553002714921 8	502-10073	Card Number	0000450553002714921
12	106779	2	Transaction failed due to service error	0000450065505333589 7	502-10073	Card Number	0000450065505333589

Display 1. Output Data Set

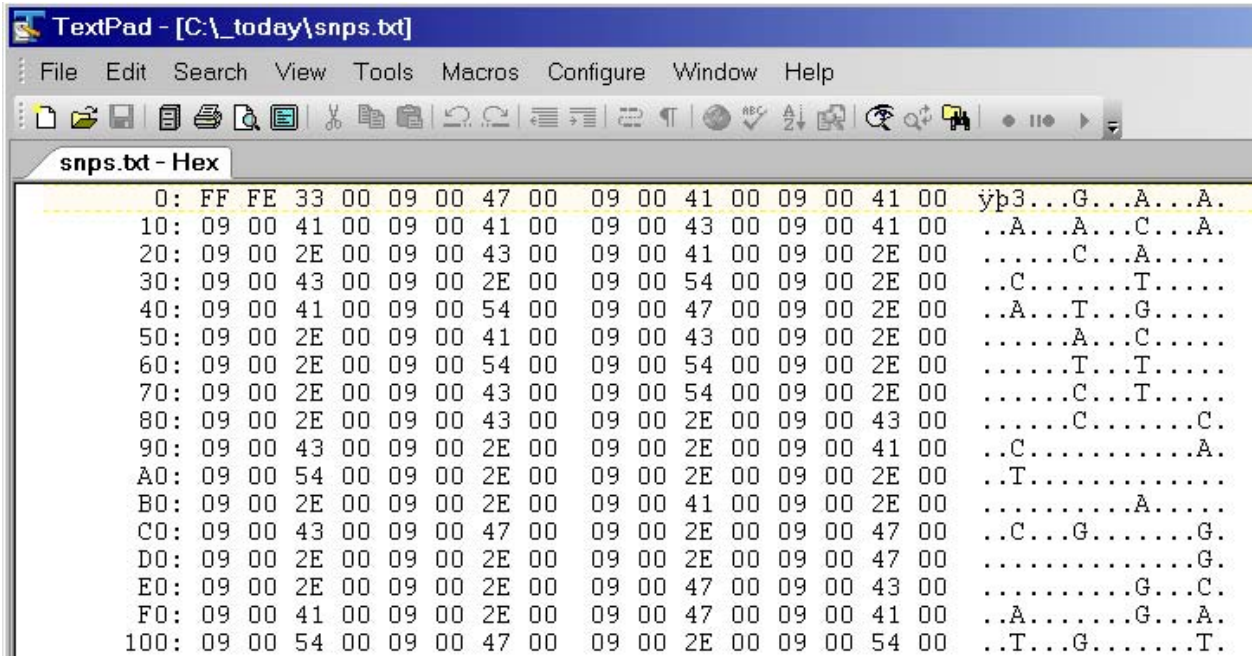
USING TOOLS FOR FOREIGN ENCODING

SAS 8.2 included a new option called ENCODING=, which is added to the INFILE statement. This option enables you to dynamically change the character set encoding for processing external data. Valid values for this option are listed in the "SBCS, DBCS, and Unicode Encoding Values for Transcoding Data" section of the *SAS® National Language Support (NLS): Reference Guide* (SAS Institute Inc. 2011). In order to make the ENCODING= option easier to use, three aliases were created as values for this option: UNICODE, EBCDIC, and ASCII. You might see these aliases in various examples in SAS documentation, but they are not fully documented at this time. They map to the values that are documented in the *SAS® National Language Support (NLS) Reference Guide* as follows: UNICODE maps to UTF-8; EBCDIC maps to EBCDICANY; and ASCII maps to US-ASCII. The following discussion focuses on these ENCODING= values: UNICODE, EBCDIC, and ASCII.

ENCODING=UNICODE

Two types of encoding are common to the general term Unicode: UTF-8 and UTF-16. SAS Technical Support most frequently sees UTF-8 encoding, which uses 1 byte for any ASCII character and a null character after the 1 byte. UTF-16 encoding is used for Multi-Byte Character Sets (MBCS)—previously referred to as Double-Byte Character Sets (DCBS). When you specify ENCODING=UNICODE, SAS looks at the first 2 bytes of the file to determine the actual encoding. These 2 bytes are known as *byte order mark* (BOM).

When viewing a file that has a UTF-8 encoding, null bytes are not visible in an editor because the editor discovers the BOM and adjusts to only show the single-byte data. Some editors display the null characters as spaces, which makes the data appear to have a space between each character and three spaces between each word. If you use an editor that shows the hexadecimal values, you will see something similar to Display 2 (note that this particular file has the BOM and is delimited by tabs):



Display 2. Editor Window of a File with the Hexadecimal Values for the Data

In this file, the first 2 bytes, 'FF'x and 'FE'x, combine to form the BOM, which indicates the UTF-8 encoding. '33'x is the first character of the data in the file—a number 3. This is followed by a null character, the tab delimiter '09'x, and another null character. The '47'x character is the ASCII hexadecimal value for "G." It is followed by another null character. This is the pattern that the entire file follows: character, null, character, null.

The simple way to handle Unicode files is to use the option ENCODING=UNICODE in your INFILE statement and code the INPUT statement as you would normally. SAS internally doubles the specified LRECL= option value so that the SAS user does not need to be aware of it or compensate for it. Beginning with SAS 9.2, SAS examines the file before reading it to determine the encoding and adjusts for it automatically. As a result, you might see notes such as the following in your log:

```
NOTE: No encoding was specified for the file "…".  A byte order mark in the file indicates
that the data is encoded in "utf-8".  This encoding will be used to process the file.
```

This note is provided only for your information and does not imply a problem unless the encoding that is chosen is incorrect.

Occasionally, SAS Technical Support gets a call from a customer who either has a UTF-8 file with no BOM or a plain ASCII file with a BOM, but has no encoding or the encoding does not match the BOM. In SAS 9.2, the way to address this issue is to modify or rewrite the file, because, at this time, the automatic adjustments cannot be disabled. The following program reads the original file and writes the plain ASCII text to the new file (note that you cannot overwrite the original file because the total byte count will not be the same as the original):

```
data _null_;
infile "c:\_today\gladys.txt" recfm=n;
file "c:\_today\gladys_.txt" recfm=n;
input a $char1.;
if a in('FF'x,'FE'x,'00'x) then delete;
put a $char1.;
run;
```

ENCODING=EBCDIC

This option was created to read simple EBCDIC data files on an ASCII system. *Simple* means that the file contains only characters that can be typed on an ordinary keyboard and does not contain nonstandard numeric data formats such as integer binary (IB) or packed decimal (PD). You still need to provide information about the file using the RECFM= and LRECL= options. However, you can use a standard INPUT statement with standard character and numeric informats where required.

If the data that you want to read contains nonstandard numeric data, use the traditional method of reading the data using the value for the RECFM= option that is appropriate for the file—F (fixed format) or S370VB (variable block [VB] S370 record format). Use $EBCDICw. for character data and the S370xxxxw.d informats that match the data that is being read. This is required if the data file is downloaded in binary form from the host computer or read directly from the host computer via the FTP engine.

Experience has shown that the most problem-free way to move a variable block file from a z/OS system to a UNIX system or a PC system is to preprocess the file with the z/OS utility IEBGENR. This utility converts the structure from VB to unformatted before the file is downloaded. This is necessary because nearly every FTP program that is available removes the record descriptor word (RDW) and block descriptor word (BDW) from a VB file during the transfer. By using IEBGENR to change the file's format, FTP programs can move the file from system to system with the RDW and BDW intact. Without these descriptors, reading the file can be quite difficult.

TS-642: "Reading EBCDIC Files on ASCII Systems" (SAS Institute Inc. 2000) provides more information about this topic.

ENCODING=ASCII

ENCODING=ASCII is the mainframe counterpart to using ENCODING=EBCDIC on the ASCII systems. Contrary to the preceding problems that are discussed, some difficulties occur when moving a text file from PC or UNIX systems to the mainframe with translation. In most cases, the data reads cleanly with a standard INPUT statement on z/OS systems. If you run into a problem here, call SAS Technical Support.

READING AN OCCURS FILE

The record structure in a simple occurs file has a fixed-length section followed by one or more counter variables that contain the number of times that a specific record segment occurs—hence the name *occurs* file. A complex occurs file can have multiples of the simple structure. There is no limit to the number or size of the segments in an occurs file. Writing code to read this type of file can be somewhat cumbersome and tedious, but it is not difficult.

Every occurs file has a record layout file. This is the map for the file. It indicates the variable names and lengths, the segment groupings, and how many digits are in the index variables. It is possible to have zero occurrences of repeating segments on some or all of the records in a file. There is no shortcut for writing the INPUT statements to read the data. Use arrays for each of the variables in the recurring segments. The dimension of the arrays for a segment is the maximum number of occurrences for that segment. If only part of the record is needed, it is still necessary to include code to read through all of the data on each record. This action is necessary to position the input pointer at the beginning of the next record.

The most common scenario is an EBCDIC file on an ASCII system with no RDW or BDW. The solution is to read the file using RECFM=N, which treats the entire file as a single record and every character, including control characters, as data. Because the entire file is a single record, at signs (@) must not appear in the INPUT statement. If moving the input pointer to a different location in the input buffer is necessary, use a plus sign (+) in the INPUT statement to advance the input pointer some number of columns. This is one alternative if you want to skip a particular repeating segment. If the total length of the segment is, for example 45 characters, then use a DO loop to step forward that number of segments on any given record. Remembering that the count can be zero, the code would look like this:

```
do I = 1 to index_var;
  input +45;
end;
```

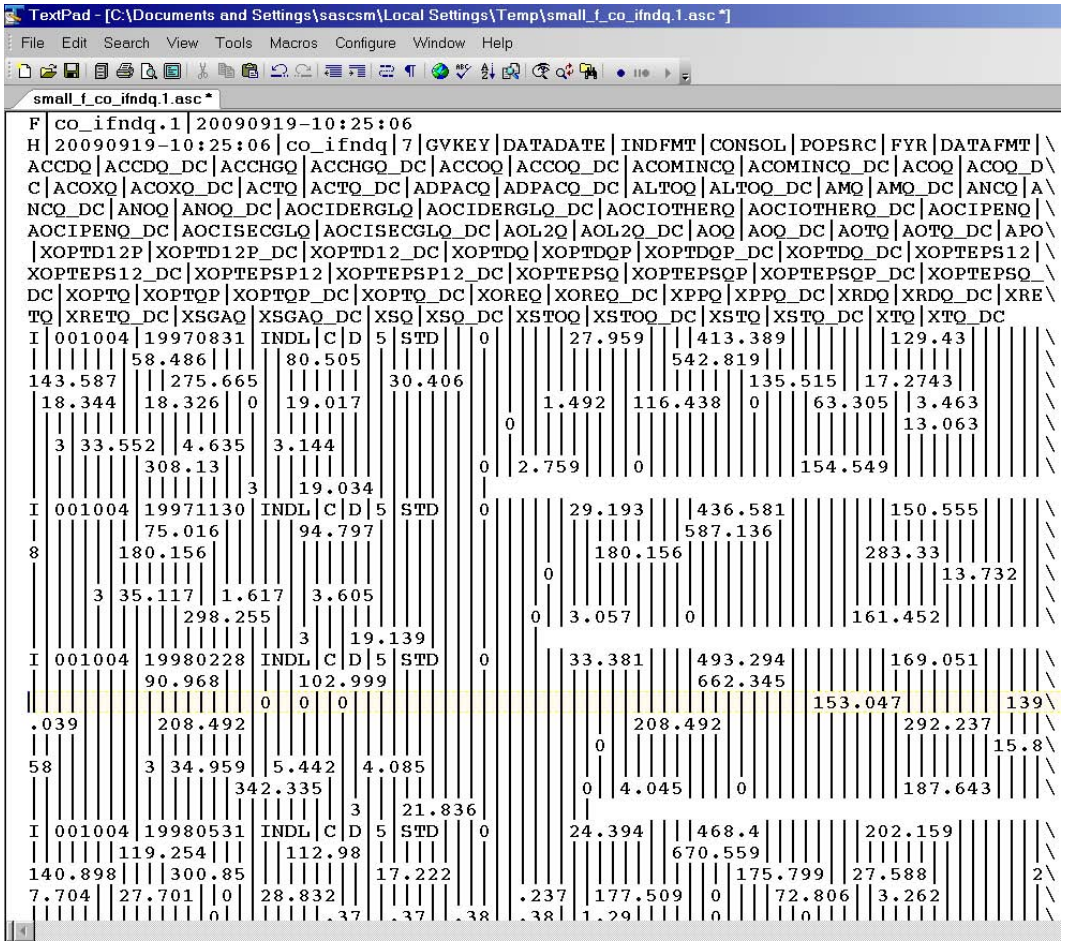
TS-642: "Reading EBCDIC Files on ASCII Systems" (SAS Institute Inc. 2000) provides more information about this topic.

ADVANCED PROBLEM SOLVING

This section presents real examples of advanced data-reading problems that have been submitted to SAS Technical Support. Creative solutions are presented using the tools and techniques that are discussed in the preceding section.

EXAMPLE 1: COMBINING PHYSICAL RECORDS TO FORM A LOGICAL RECORD

Display 3 is a partial view of a file that was sent to SAS Technical Support from a customer who asked for help with reading the file:



Display 3. Partial View of a Customer Text File

Although it does not look like it, this is actually a hierarchical file. The record type indicators are in column 1 and are "F", "H", and "I". Notice that there is a backslash (\) in column 80 of nearly every line. This is a *continuation* character. Each backslash tells the program to join the next record to that record. The simplest approach to reading this file is to create a new file by joining the physical records together into a much longer record. Then you write that new variable to a new file. Finally, you can perform a simple hierarchical read of the new file.

To join the physical records together, read a line from the input, concatenate it to the output line, and check the input line for a continuation character. If there are no continuation characters, remove the backslash from the output record and write it to the output file. Here is the sample code:

```
data _null_;
infile 'c:\_today\jai.asc' truncover;
   file 'c:\_today\jai_.asc' lrecl=32767;
length otlin $32767 ;
input inlin $char80.;
otlin=cats(otlin,inlin);
if substr(inlin,80,1)^='\' then do;
   otlin=compress(otlin,'\');
   put otlin ;
   otlin=' ';
end;
retain otlin;
run;
```

Display 4 is a sample of the reshaped file:

jai_.asc

```
F|co_ifndq.1|20090919-10:25:06
H|20090919-10:25:06|co_ifndq|7|GVKEY|DATADATE|INDFMT|CONSOL|POPSRC|FYR|DATA
I|001004|19970831|INDL|C|D|5|STD|||0|||||27.959||||413.389|||||||129.43||
I|001004|19971130|INDL|C|D|5|STD|||0|||||29.193||||436.581|||||||150.555
I|001004|19980228|INDL|C|D|5|STD|||0|||||33.381||||493.294|||||||169.051
I|001004|19980531|INDL|C|D|5|STD|||0|||||24.394||||468.4||||||202.159||
I|001004|19980831|INDL|C|D|5|STD|||0|||||28.371||||509.808||||||197.887
I|001004|19981130|INDL|C|D|5|STD|||0|||||29.674||||547.5||||||189.916|||
```

Display 4. Reshaped Customer Text File

Now, read this file with the following program (in this example, the input variables and informats are stored in macro variables):

```
data cstat;
   infile "T:\small_co_ifndq.asc" Dsd truncover dlm="|" lrecl=32767;
   input @1 t :$1. @;
   if t="F" then input process :$40. datetime :$17. ;
   else if t="E" then input ;
   else if t="H" then input &H_lines. ;
   else if t="I" then do;
     input @1 &I_lines. ;
     output;
   end;
   retain _all_;
run;
```

EXAMPLE 2: USING PRXCHANGE IN THE INPUT BUFFER

Another SAS customer had a file that was transferred from a mainframe to a PC. The data was simple character and numeric, so, when the file was transferred, the data was converted from EBCDIC to ASCII. The data transferred cleanly and the resulting file is exactly the way it was on the mainframe. The problem is that the COBOL format that was used to write the numeric data embedded minus signs (-) in the numeric field after the leading zeros instead of in front of the leading zeros. Here is a sample of the data from the problem file:

```
000001224.12
00000000-500
0000-1263.23
000000000.92
000000000-21
000000-154.21
000008773.51
```

At this time, SAS does not have an informat to read numeric data with an embedded sign. To solve the problem, use PRXCHANGE to move the minus sign, if there is one, to the front of the leading zeros. This part of the program looks like this:

```
input @;
_infile_=prxchange("s/(0+?)-/-$1/o",-1,_infile_);
input   @1  Record_ID  $3.  @;
if record_id = '100';
input  @4  Artiva_acct_id  9.
       @13  Account_Number  $30. ... ;
```

The Perl regular expression looks for any number of zeros followed by a minus sign. In simple terms, if a minus sign is found, it is moved to the front of the leading zeros and the zeros are shifted to the right one space. The rest of the digits are not affected.

EXAMPLE 3: READING POSITIONAL HIERARCHICAL FILES

The following sample file has been modified from its original spacing to fit on the page. It is a sales report that was formatted as a positional hierarchical file.

```
102/05/2009                XXXXXXXXXXXXXXX            PAGE   1
                       TRANSACTION HISTORY REPORT
0TRANSACTION TYPE                      TYPE      RESULT    MATCHING

 110  WHITE MAIL                     29,582
       A  - ADD                                   9,768
       AP - ADD PRENATAL                          4,443
       AR - ADD TRIPLETS NEW                          6
       AS - ADD SINGLE TWIN                         174
             110  WHITE M                                        10
             117  JEFFERS                                         1
             123  AM BABY                                        28
             126  KC ON-L                                        23
             145  MOTHER                                         89
             167  BABY TA                                        22
             182  NCOA                                            1
       AT - ADD TWIN                              128
       A2 - ADD 2 SINGLE BIRTHS                    16
             123  AM BABY                                         3
             126  KC ON-L                                         1
             145  MOTHER                                         11
             167  BABY TA                                         1
       A3 - ADD 3 SINGLE BIRTHS                     2
             126  KC ON-L                                         1
             145  MOTHER                                          1
       B  - IMPROVED BIRTH DATE                12,324
             110  WHITE M                                     1,080
             112  STORK A                                         6
             113  F MOMEN                                         1
             117  JEFFERS                                        44
             118  PREG OR                                        49
             119  BK FULF                                        24
             123  AM BABY                                     2,062
             126  KC ON-L                                     2,045
             137  BABIES                                         97
```

In this code, the key locations are underlined in the first group, column 1, columns 2-4, columns 8-9, and columns 14-16. The first key location that is not blank tells what variables are present on the record. Those variables that are unique to that type of record are read and the variables from subsequent groups are initialized. For example, the test variable from columns 8-9 has data and you want to initialize the variables that come from the column 14-16 group, but not from the column 2-4 group. Each column group builds on the previous group. All records, except the print headers, are saved to the output data set.

Here is the code to read the data:

```
data kathy;
infile "c:\_today\feb_transhist_orig.txt" truncover;
input a 1 b $ 2-4 c $ 8-9 d $ 14-16 @;
length list_id 8 list_co $11 trans_code $2 transaction $41
       trans_src_id 8 trans_src $7 quant 8;
if a = 1 then do;
   input / / / ;
   delete;
end;
else if a = 0 then do;
   input / ;
   delete;
end;
else if b ^= ' ' then do;
   input list_id 2-4 list_co $ 7-17 @ 57 quant comma10.;
   trans_code=' ';
   transaction=' ';
   trans_src_id = . ;
   trans_src = ' ';
end;
```

```
else if c ^= ' ' then do;
   input trans_code $ 8-9 transaction $ 13-53 @78 quant comma10.;
   trans_src_id = . ;
   trans_src = ' ';
end;

else if d ^= ' ' then do;
   input trans_src_id 14-16 trans_src $ 19-25 @95 quant comma10.;
end;
else delete;
drop a b c d;
retain _all_;
format quant comma10.;
run;
```

EXAMPLE 4: READING DATA BASED ON COBOL COPYBOOK INFORMATION

The next two items are the COBOL copybook and the DATA step that reads the data that the copybook describes. Here is the copybook that was sent to SAS Technical Support from a customer:

```
01  STATEMENT-RECORD.
    05  FILLER                 PIC X(02).
    05  CLIENT-ID              PIC X(04).
    05  ACCOUNT-NUMBER         PIC X(18).
    05  SEQUENCE-NUMBER        PIC X(14).
    05  NUM-ZZZZ-STMTS         PIC 9(01) COMP-3.
    05  NUM-XXXX-TRADES        PIC 9(01) COMP-3.
    05  NUM-YYYY-RECORDS       PIC 9(01) COMP-3.
    05  XXXX-INFO        OCCURS NUM-XXXX-TRADES TIMES.
        10  XXXX-SUBCODE       PIC X(08)
        10  XXXX-ACCT-NUM      PIC X(18)
        10  XXXX-CODE          PIC X(03)
    05  YYYY-INFO      OCCURS NUM-YYYY-RECORDS TIMES.
        10  YYYY-SUBCODE       PIC X(08)
        10  YYYY-DOCKET-NUM    PIC X(11)
        10  YYYY-CODE          PIC X(02)
    05  ZZZZTMT-INFO     OCCURS NUM-ZZZZ-STMTS TIMES.
        10  QQQQ-LENGTH        PIC 9(02) COMP-3.
        15  QQQQ-STATEMENT OCCURS QQQQ-LENGTH TIMES.
            20  QQQQ-CHAR      PIC X(01)
```

This is a typical COBOL copybook for a fairly simple occurs file. The fixed-segment variables and segment counters are labeled **05**. The occurring segment variables are labeled **10**. The 15-level record is the counter for the occurring segment within an occurring segment. The **20** label is the embedded segment. Although this is not an uncommon structure, it does require uncommon INPUT statements.

There are two data types listed in the copybook: PIC x(XX) is character data; PIC 9(YY) COMP-3 is Packed Decimal data. Adding to the complications of reading this particular file that it is in EBCDIC format and is being read on an ASCII system. Due to the nonstandard numeric data, using ENCODING= is not an option here. The corresponding SAS informats are $EBCDICw. (where w=XX) and S370FPDUw. (where w=YY). For a complete list of COBOL data types and the corresponding SAS informats, refer to SAS Note 3714 "SAS informats that correspond to COBOL data descriptions" (SAS Institute Inc. 2010).

Here is the code that reads the file.

```
data cave;
   infile  "c:\_today\xyzzy.ebc" recfm=N lrecl=65536 ;
   input    filler1 $ebcdic2.
            rec_length $ebcdic2.
            account_number $ebcdic16.
            tu_edit_seq_nbr $ebcdic14.
            num_zzzz_stmts s370fpdu1.
            num_xxxx_trades s370fpdu1.
            num_yyyyic_records s370fpdu1. @;

   array xxxx_subcode(9) $ebcdic8 ;
   array xxxx_acct_num(9) $ebcdic18 ;
   array xxxx_code(9) $ebcdic3 ;
```

```
   if num_xxxx_trades gt 0 then do i = 1 to num_xxxx_trades;
      input xxxx_subcode(i) $ebcdic8.
            xxxx_acct_num(i) $ebcdic18.
            xxxx_code(i) $ebcdic3. @;
   end;

   array yyyy_subcode(9) $ebcdic8 ;
   array yyyy_docket_num(9) $ebcdic11 ;
   array yyyy_code(9) $ebcdic2 ;
   if num_yyyyic_records gt 0 then do i = 1 to num_yyyyic_records;
      input yyyy_subcode(i) $ebcdic8.
            yyyy_docket_num(i) $ebcdic11.
            yyyy_code(i) $ebcdic2. @;
   end;

   array qqqq_statement(256) $256;
   if num_zzzz_stmts > 0 then do i=1 to num_zzzz_stmts;
      input qqqq_length(i) s370fpdu2. @;
      do j = 1 to qqqq_length(i);
         input a $ebcdic1.;
         substr(qqqq_statement(i),j,1)=a;
      end;
   end;
run;
```

Notice the 15- and 20-level records in the copybook. qqqq_statement is one character and occurs qqqq_length times. This means that the 20 level is a character string with a length specified in the 15th level. If the file was ASCII, the $VARYINGw. informat would have been perfect to use. Instead, one byte at a time is read with $EBCDICw. and each byte is inserted into the qqqq_statement in sequence. This is done with the SUBSTR function to the left of the equal (=) sign.

The COB2SAS program is also useful when reading COBOL copybooks. SAS Note 22377 "Converting COBOL data descriptions to a SAS® INPUT statement" (SAS Institute Inc. 2008) provides information about how to download this program. This program reads a COBOL copybook and generates the INPUT statement, with informats, to read the data that is described in the copybook. The only difficulty with the COB2SAS program is that it was not designed to process an occurs file, so the output needs to be edited for this type of file.

EXAMPLE 5: SPLITTING DATA INTO MULTIPLE VARIABLES CONDITIONALLY

This example involves reading data that is too long to store in a single-character variable. The maximum length of a character variable is 32K. There is a variable in the input record that can exceed 32K characters. In most cases, however, the variable will not exceed the limit. What do you do to prevent the loss of data?

The best solution is to preprocess the file and add another variable for the over-flow or an empty character variable if the content of the first variable is less than the 32K limit. This involves counting delimiters to find the correct variable, counting characters, and deciding what to write out. Here is the code for resolving the problem:

```
data _null_;
infile 'c:\_today\claude.csv' recfm=n;
file 'c:\_today\claude_.csv' recfm=n;
input a $char1.;
put a $char1.;
if a = ',' then c+1;
if c=3 then do;
   d+1;
   if d=32767 then put ',';
end;
if c=4 then do;
   if d<32767  then put ',';
   d=32768;
end;
if c=5 then do;
   d=0;
   c=0;
end;
run;
```

After the third delimiter is found, count each character until the counter reaches 32767 or until the fourth delimiter is found. Whichever happens first, a delimiter is written out. C counts the delimiters, and D counts the characters in the

fourth variable. If the fourth delimiter is found first, D is set to 32768. After the modifications are made, a simple DATA step is used to read the new file into a SAS data set. By adding the delimiter, either an over-length variable is split to the maximum that is allowed and the excess is put into another variable, or a null variable is added to the new file.

EXAMPLE 6: SPLITTING ONE FILE INTO MANY

The next sample comes from a hospital productivity report. The file actually has three different sections. Only the boundary of the first and second sections is shown below. The line of Xs marks the transition. Everything below the Xs is the second section of the report. In the second section, the length of the records is double that of the first section, so there appears to be twice as many lines in the second section. The following data sample shows the entire second section split as a whole instead of line-by-line.

```
A:LIFT   |    0.00    0.00    0.00    0.00    0.00    0.00|    0.00    0.00    0.00    0.00    0.00    0.00    0.00|
A:TRAV   |    0.00    0.00    0.00    0.00    0.00    0.00|    0.00    0.00    0.00    0.00    0.00    0.00    0.00|
A:ASOR   |  139.59    0.00    0.00    0.00  139.59    0.00|    0.00    0.00    0.00    0.00   30.66  170.27  170.27|
A:3OR    |  189.21    0.00    0.00    0.00  189.21    0.00|    0.00    0.00    0.00    0.00   41.53  230.72  230.72|
A:CVOR   |  201.60    0.00    0.00    0.00  201.60    0.00|    0.00    0.00    0.00    0.00   44.26  245.75  245.75|
A:ORAD   | 4303.28    0.00    0.00    0.004303.28    0.00|    0.00    0.00    0.00    0.00  440.74 4962.964962.96|
A:CCMS   | 2124.27    0.00    0.00    0.002124.27    0.00|    0.00    0.00    0.00    0.00  254.71 2430.002430.00|
A:MCH    | 3962.52    0.00    0.00    0.003962.52    0.00|    0.00    0.00    0.00    0.00  333.82 4469.124469.12|
A:MLOA   |    0.00    0.00    0.00    0.00    0.00    0.00|    0.00    0.00    0.00    0.00    0.00    0.00    0.00|
A:CLIN   |    0.00    0.00    0.00    0.00    0.00    0.00|    0.00    0.00    0.00    0.00    0.00    0.00    0.00|
A:NICT   |    0.00    0.00    0.00    0.00    0.00    0.00|    0.00    0.00    0.00    0.00    0.00    0.00    0.00|
```

```
                                                                                                X X X X X X X X X X X X X X

                                                                                              PRODUCTIVITY REPORT FROM ??/??

DIV XXXXXX-XXMC          /PICU
HOURS     |--------------PRODUCTIVE HOURS--------------| OVER  |------SUPPLEMENTAL-------|---------OTHER HOURS A
LEVEL     |  BUDGET  TARGET  ACTUAL BU-AC TA-AC %BG %TR| TIME  |  FLOAT  REGIS  OTHER %SU|   ANCL NONPRO   TOTAL
RN   DAYS |       51      88      96   -45    -8  53  92|       0|       0     40       0  42|       0      52      148
     EVES |       51      80      92   -41   -12  55  87|       0|       0     28       0  30|       0       0       92
     NIGHT|       51      80      88   -37    -8  57  91|       0|       0     16       0  18|       0      32      120
     TOTAL|      152     248     276  -124   -28  55  90|       0|       0     84       0  30|       0      84      360
          |                                            |        |                          |
MT   DAYS |        0       0       0     0     0   0   0|       0|       0      0       0   0|       0       0        0
     EVES |        0       0       0     0     0   0   0|       0|       0      0       0   0|       0       0        0
     NIGHT|        0       0       0     0     0   0   0|       0|       0      0       0   0|       0       0        0
     TOTAL|        0       0       0     0     0   0   0|       0|       0      0       0   0|       0       0        0
          |                                            |        |                          |

                                                                                                ??/??/????

     H O S P I T A L

  '/???? TO ??/??/????
```

```
 ND TOTAL----------|-------PRODUCTIVE HOURS PER PATIENT DAY-------||      |  ----------Patient Data-----------
 .  BUDGET BU-TO %NP|  BUDGET  TARGET  ACTUAL  SUPLEM  BU-AC  TA-AC||      |  DAYS   EVES   NIGHT  AVERG     %
 :      67   -81  35|    4.20    5.50    6.00    2.50  -1.80  -0.50|| CEN |   16.0   16.0   15.0   15.7   100.0
        67   -25   0|    4.20    5.00    5.75    1.75  -1.55  -0.75|| 1   |    0.0    0.0    0.0    0.0     0.0
 '      67   -53  27|    4.20    5.33    5.87    1.07  -1.67  -0.53|| 2   |    0.0    0.0    0.0    0.0     0.0
 '     200  -160  23|   12.60   15.83   17.62    5.36  -5.02  -1.79|| 3   |   11.0   14.0   13.0   12.7    80.9
                    |                                             || 4   |    5.0    2.0    2.0    3.0    19.1
 '       0     0   0|    0.00    0.00    0.00    0.00   0.00   0.00|| 5   |    0.0    0.0    0.0    0.0     0.0
 '       0     0   0|    0.00    0.00    0.00    0.00   0.00   0.00||
 '       0     0   0|    0.00    0.00    0.00    0.00   0.00   0.00||
 '       0     0   0|    0.00    0.00    0.00    0.00   0.00   0.00||
                    |                                             ||
```

The technique for solving this problem is to divide and conquer. Each section is a hierarchical structure. Because the data was going to three different data sets, it makes sense to split the file into three separate files and read each one individually.

The following code splits the original file into three pieces so that it can be easily read. Each of the new files is a simple hierarchical file.

```
data _null_;
infile 'c:\_today\han.txt' lrecl=4096 truncover end=finish;
length test $20;
do until(first);
   input test &$20.;
   file 'c:\_today\part1.txt' lrecl=4096;
  if test='DIV KAISER-LAMC' then first=1;
   else put _infile_;
end;
do until(second);
   input test &$20.;
   file 'c:\_today\part2.txt' lrecl=4096;
   if test='GRAND SUMMARY' then second=1;
   else put _infile_;
end;
do until(finish);
   input ;
   file 'c:\_today\part3.txt' lrecl=4096;
   put _infile_;
end;
run;
```

CONCLUSION

This paper discussed problems that are commonly reported to SAS Technical Support when customers are trying to move data from text files into SAS data files. The topics included preprocessing files before they are read, modifying records as they are read, applying hierarchical file logic, using tools for foreign encoding, and reading an occurs file. Advanced data problems, such as using PRXCHANGE in the input buffer, reading positional hierarchical files, and splitting data into multiple variables, were also discussed. Problems such as these can seem overwhelming at times, but as the examples in this paper demonstrate, there are clear paths that you can take to resolve these problems.

Customers who are having problems reading data into SAS often call SAS Technical Support looking for the way out of their situation; they express thoughts such as, "If I could *just* change or fix or remove or insert this *one* thing, *then* I'd be able to read this file with no trouble." Well, you *can* fix that one thing—and then some. SAS 9.2 includes tools and functionality that make some of the more difficult tasks easier than ever. With SAS 9.2, you have many tools and resources at your disposal. But don't be afraid to use these tools and your problem-solving skills in new ways. Think creatively—get off the beaten path. When you explore the surrounding wilderness, you learn more about the path you're on and you naturally branch out in new directions. The more times you go down any path, the easier it is to make subsequent trips. Besides, the wilderness is beautiful when you are properly prepared!

With so many tools at the SAS programmer's disposal, it is hard to remain lost in the wilderness and therefore nothing should seem impossible.

RECOMMENDED READING

SAS Institute Inc. 2011. "SBCS, DBCS, and Unicode Encoding Values for Transcoding Data". *SAS® National Language Support (NLS): Reference Guide*. Cary, NC: SAS Institute Inc. Available at support.sas.com/documentation/cdl/en/nlsref/61893/HTML/default/viewer.htm#a002607278.htm.

SAS Institute Inc. 2010. SAS Note 3714. "SAS informats that correspond to COBOL data descriptions". Available at support.sas.com/kb/3714.

SAS Institute Inc. 2008. SAS Note 22377. "Converting COBOL data descriptions to a SAS® INPUT statement." Available at support.sas.com/kb/22/377.html.

SAS Institute Inc. 2000. SAS Technical Support Document. "TS-642: Reading EBCDIC Files on ASCII Systems". Available at support.sas.com/techsup/technote/ts642.html.

ACKNOWLEDGMENTS

The author thanks the following individuals for their contributions to this paper:

- Amber Elam and Ginny Piechota for mentoring and sharing their wealth of knowledge over the years
- Jan Squillace and Sally Walczak for their peer review of this paper
- Kathleen Walch for the long hours editing this paper
- an anonymous SAS customer whose data-reading challenges were the basis of three of the samples in this paper

CONTACT INFORMATION

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

Charley Mullin
SAS Institute Inc.
SAS Campus Drive
Cary, NC 27513
E-mail: support@sas.com
Web: support.sas.com