Paper 70

Reading External Files Using SAS® Software

Clinton S. Rickards, Purdue Pharma L.P.

ABSTRACT

In the real world, most data are not stored in SAS data sets. Typical data repositories used by business and academia include legacy mainframe files, client/server relational database management systems, and simple text files on desktop computers. While we want to use SAS to examine, to analyze, to display, and to graph this data, SAS works best with data stored in SAS data sets, not with data stored in their native forms.

Fortunately, SAS has very powerful tools for reading a wide variety of external files, from the most rudimentary to the most complex. This paper is an updated and expanded version of similar papers presented by the author at previous SAS conferences and will introduce DATA step techniques for reading standard external files not stored in data base management systems or proprietary file structures.

We will begin with a brief overview of what to do before writing any code, followed by an examination of the SAS statements used to read an external file. Starting with very simple files, we will study programs that utilize more complicated features, including reading into arrays, multiple format files, variable length files, and multiple record files.

INTRODUCTION

Much raw data is stored in “flat files”. SAS procedures usually require a SAS data set or view as input. DATA steps serve as the bridge to transform flat files into SAS data sets. If you understand the features available in the DATA step and how to point the SAS Supervisor to the location of your external data, there are few file formats that cannot be read into a SAS data set.

All of the examples run under Release 6.07 and later releases.

BEFORE YOU START

Writing the actual DATA step is typically no more than 10-20 percent of the work. The remainder is spent exploring your data. Exploration includes designing the structure of the SAS data set to be created and determining the organization of the raw data file to be read.

SAS Data Set Design

SAS data sets are often poorly designed. To design an efficient SAS data set, consider the purpose to which the data set will be used. Determine the variables you need and the range of values to be contained.

Questions to ask when designing SAS data sets include: What are the variable names to be created? Are arrays needed? Is the projected size of the data set so large that using shorter variable lengths or compressing the data set are desirable? Should the data set be split into multiple data sets to reduce the storage of redundant information? Should the data be translated into coded values?

Raw Data Organization

In determining the organization of the raw data file, one must understand how the file is described in its native form. Questions to ask include: Do the records have a single format or are there multiple record formats on the file (e.g., a master record followed by a detailed record)? How are numeric fields formatted (signed or unsigned, packed, floating point, etc.)? How are "group" fields (e.g., personal names, which may contain a title, first name, middle name, last name, and suffix) handled? Are there multiple logical records per physical record or does a single logical record require multiple physical records? Are there repeated fields (i.e. arrays)? Is the file stored on tape or disk?

POINTING SAS TO THE EXTERNAL FILE

There are three ways to point SAS to external files: operating system control statements, FILENAME statements, and INFILE statements.

Operating System Control Statements

This method uses commands or statements interpreted by the operating system rather than the SAS Supervisor to allocate a **fileref**. The SAS Supervisor is the part of the SAS System that interprets SAS statements.

A **fileref** (file reference) is a nickname used in place of the full filename. A fileref created under MVS is sometimes called a “DD Name”. For example, when a SAS program is executed in MVS batch, one might use a DD allocation statement to assign a fileref.

Generally, the fileref is created before a SAS session is started. Some platforms permit the use of the X statement to assign a fileref after the session has started. Please check the SAS Companion for your platform for details. A disadvantage of assigning filerefs with operating system commands that the filerefs will not be available in the Display Manager FILENAME window.

FILENAME Statement
This statement creates a fileref that links a physical file with the SAS system. When the operating system does not provide control statements to assign filerefs (e.g., directory based platforms such as MS-DOS, Windows, and OS/2, and servers running under Unix or Windows NT), FILENAME or INFILE statements must be used. The FILENAME statement is executed within a SAS session prior to the DATA step.

The syntax of the FILENAME statement is:

FILENAME *fileref <device-type> 'external-name'*
<host-options>;

where:
fileref is any valid SAS name

device-type indicates the type of device. It is optional and defaults to DISK.

'external-name' is the name of the file on the host system. The quotes are required.

host-options specify any options that vary from operating system to operating system. Carefully read the SAS Companion for your operating system to understand the meaning and usage of these options.

You may use FILENAME in conjunction with operating system definitions to specify information about the file not covered by the operating system. In general, one or the other alone is sufficient.

INFILE Statement
INFILE is used to tell a DATA step from which file to read when an INPUT statement is executed. If the argument after the keyword 'INFILE' is quoted, the SAS Supervisor treats it as an actual external file name. Otherwise, the SAS Supervisor assumes that it is a fileref.

An INFILE statement should be used in each data step reading an external file and must be executed before the input statement. You can have more than one INFILE statement, which allows you to read multiple files with a single DATA step. When the INFILE statement is absent from a DATA step, the INPUT statement will default to INFILE CARDS.

When SAS begins each iteration of a DATA step, it "forgets" from which file(s) it had read. This means that you must be careful to execute an INFILE for each DATA step that must reference an external file.

The general syntax of INFILE statement is:

INFILE *file-spec <options> <host options>*;

where *file-spec* identifies the source of the data. *file-spec* may have 3 forms:

> *fileref* is the fileref assigned to the required external file. The fileref must be assigned before the DATA step by using an operating system definition or FILENAME statement.
>
> *'external-file'* specifies the name of the required external file. This form is equivalent to specifying the external file with a FILENAME statement.
>
> *CARDS* (or *CARDS4*) indicates that the data immediately follows the CARDS statement at the end of the data step. In Release 6.07 or later, DATALINES (or DATALINES4) may be specified instead.

options specify SAS options to control reading the file or to provide information about the file. Commonly used options will be described in the INFILE STATEMENT OPTIONS section.

host-options are host specific options.

Examples
The following examples are functionally equivalent.

CMS
```
FILEDEF SALESDAT DISK
  NOVEMBER SALES A
```

```
FILENAME SALESDAT
  'NOVEMBER SALES A';

X 'FILEDEF SALESDAT DISK
  NOVEMBER SALES A';

INFILE SALESDAT;
/* with prior FILENAME or */
/* FILEDEF statement      */

INFILE 'NOVEMBER SALES A';
```

MVS JCL

```
//SALESDAT DD
// DSN=F456.SALES.DATA.NOVEMBER,
// DISP=SHR

FILENAME SALESDAT
  'F456.SALES.DATA.NOVEMBER'
   DISP=SHR;

INFILE SALESDAT;
/* with prior FILENAME or */
/* DD statement           */

INFILE
  'F456.SALES.DATA.NOVEMBER'
  DISP=SHR;
```

MVS TSO

```
ALLOC F(SALESDAT)
  DA('F456.SALES.DATA.NOVEMBER')
  SHR

FILENAME salesdat
  'F456.SALES.DATA.NOVEMBER'
  disp=shr;

INFILE salesdat;
/* with prior FILENAME or */
/* ALLOC statement        */

INFILE
  'F456.SALES.DATA.NOVEMBER'
  disp=shr;

X "ALLOC F(SALESDAT)
  DA('F456.SALES.DATA.NOVEMBER')
  SHR";
```

Windows (all versions)

```
FILENAME salesdat
  'C:\SALES\NOVEMBER.DAT;'
INFILE salesdat;
/* with prior FILENAME    */

INFILE 'C:\SALES\NOVEMBER.DAT';
```

Comparison of Methods

The three methods of pointing to an external file have considerable functional overlap. For applications that need resources that must be scheduled (such as tapes), that require options not provided by the FILENAME statement, or that will be handled by support staff not familiar with SAS, it is better (if not required) to assign filerefs through operating system definitions. However, if the application is to be executed interactively through Display Manager or requires SAS specific options not provided by operating system definitions, use of the FILENAME statement may be a better choice.

INPUT STATEMENT BASICS

The INPUT statement is used to instruct the SAS Supervisor how to read the file. When an INPUT statement is executed, it reads from the file pointed to by the most recently executed INFILE statement (since a DATA step may have more than one INFILE statement). With few restrictions, you can use any or all of these styles in a single INPUT statement.

LIST Input merely lists the desired variable names in the INPUT statement. List input assumes that the data values are separated by one or more delimiters (blank is the default delimiter - see INFILE STATEMENT OPTIONS section). It is the easiest style to code but the most prone to problems relating to data location, unwanted imbedded blanks, and wasted cpu resourses. An example of list input would be:

```
INPUT vendor $ apples pears;
```

The $ after VENDOR indicates that VENDOR is a character variable. APPLES and PEARS are numeric data.

Also available are format modifiers, which allow you to combine the functionality of list input and formatted input (see the section labeled **FORMATTED Input**) while altering the standard operation of the informats. The format modifiers are:

: SAS will read data until it encounters a delimiter or the end of the record, or for the length specified by the informat, whichever comes first; unmodified formatted input always reads the length specified in the informat.

& indicates that a character value has one or more single embedded delimiters, so that consecutive delimiters are needed to delimit the value. SAS will read data from the file until it encounters the first of these events: consecutive delimiters, the end of the record, or the column pointer has moved past the length specified by the informat (see the **COLUMN pointer controls** section).

COLUMN Input specifies the columns to be read for each variable. The following INPUT statement illustrates column input:

```
INPUT   vendor   $  1-20
        apples      22-25
        pears       27-30 ;
```

FORMATTED Input specifies the SAS informat to be used in translating the raw data into a variable. This form of input is the only form that will work with non-standard data (e.g. packed, hexadecimal, date/time). The following example illustrates formatted input:

```
INPUT   vendor     $char20.
        apples     5.
        pears      pd5.
        ;
```

NAMED Input is of the form *variable*=.
This style of input requires that the variable name be included with the data, as in:

```
vendor=JOHN apples=5 pears=4
```

This type of data would be read by this program:

```
INPUT   vendor= $
        apples=
        pears=
        ;
```

Examples

Let us assume that the file to be read contains the name of local farms and the number of bushels of apples and pears that your store purchased from them. Further, let us also assume that the data is in text format. A complete program to read this file could be:

```
FILENAME vendors  'external-name';
DATA purchasd.fruit;
    INFILE vendors;
    INPUT  vendor   $   1-20
           apples      21-25
           pears       26-30
        ;
RUN;
```

The FILENAME statement assigns the fileref VENDORS before the DATA step, and the INFILE points to fileref VENDORS. If the numeric data were in a packed format, formatted input would be required as shown in this example:

```
FILENAME vendors  'external-name';
DATA purchasd.fruit;
    INFILE vendors;
    INPUT
        vendor   $  1-20
        apples      pd5.
        pears        pd5.
        ;
```

Variable Lengths

An important consideration when deciding which method of input to use (list, column, formatted, or named) is the desired lengths of the variables in the resulting SAS data set. SAS determines the length of a variable based on how it is referenced in the program, subject to certain defaults.

Numeric variables (including dates) and character variables read by list or named input default to 8 bytes. Character variables read by formatted input or format modified list input are set to the length as specified by the informat. Character variables read by column input are set to the number of columns being read. The LENGTH and ATTRIB statements can be used to set the variables to the length desired (more or less than the default) but must be coded before the INPUT statement.

Invalid Data

When SAS encounters invalid data, it sets the value to missing, places a note in the log, and prints the input record in the log. Depending on the application, these notes may be immaterial. Two format modifiers alter how invalid data elements are handled.

?	suppresses the invalid data message but continues to print the invalid input record
??	suppresses the invalid data message and printing of the invalid input record by setting the automatic variable _ERROR_ to 0

INPUT STATEMENT ADVANCED FEATURES

The last example illustrates two problems with the input techniques examined so far. First, the location of PEARS is not immediately obvious. SAS must add the length of APPLES to the number of columns VENDOR uses. Second, if only the number of PEARS is required, SAS must waste effort reading APPLES and VENDOR even though they are not needed.

Although acceptable in our small example, these problems become material when reading large files. This section provides the tools to overcome these and other problems.

Input Pointer Controls

SAS maintains two pointers, the **column pointer** and the **line pointer**, to track what raw data will be read during the execution of an INPUT statement. You can change these pointers to re-read data, change the order in which data fields are read, or handle logical records that are defined by multiple physical records. Use of these pointer controls also makes the INPUT statement more self-documenting.

COLUMN pointer controls:

@expression

+expression

@ moves the pointer to the column number resulting from *expression.* ***expression*** can be a numeric constant, a variable containing the column number, or an expression enclosed in parenthesis. For example, @44, @START, @(START+COUNT*5) are valid expressions.

expression can also be a character constant, a character variable, or an expression that results in a character string. SAS searches for the character expression, starting at the current column pointer location, and sets the column pointer to the first character following the string. For example, @'FIRSTNAME', @'$', @START and @('$$'||SEARCH4) are all valid expressions.

+ moves the pointer left or right the number of columns resulting from *expression.* ***expression*** can be a numeric constant, a variable containing the number of columns to move, or a numeric expression enclosed in parenthesis. Expressions resulting in positive numbers move the pointer to the right, negative results move the pointer to the left. Negative numeric constants must be enclosed in parentheses. For example, +5, +START, +(-5), and +(START+COUNT*5) are valid expressions.

These examples accomplish the same task:

```
INPUT
    @1   vendor   $char20.
    @21  apples   5.
    @26  pears    5.
    ;

start = 1;
INPUT
    @start  vendor   $CHAR20.
            apples   5.
            pears    5.
                ;

INPUT
    @1      vendor   $20.
    +5      pears    5.
    +(-10)  apples   5.
    ;
```

Note in the last example that PEARS is read before APPLES by using the +5 and +(-10) column pointer controls. Normally, we prefer the form shown in the first example. It clearly shows the starting location, length, and informat of each variable.

LINE Pointer Controls:

#expression
/

moves the pointer to a specific line number based on the result of *expression*. ***expression*** can be a numeric constant, a variable containing the line number, or an expression enclosed in parenthesis.

/ moves the pointer to the beginning of the next line. You can use multiple / to move more than one line.

These INPUT statements are functionally equivalent:

```
INPUT
    #1    @1   lastname   $char25.
    #2    @16  phonenum   9.
    #4    @30  city       $25.
    ;

INPUT
              @1   lastname  $char25.
     /        @16  phonenum  9.
     //       @30  city      $25.
     ;

cityloc  =  4;
INPUT
   #1        @1  lastname   $char25.
   #2        @16  phonenum  9.
   #cityloc  @30  city      $25.
      ;
```

Line Hold Specifiers
Line hold specifiers are used to maintain the position of the line and column pointers on the current line in the external file through multiple INPUT statements or multiple iterations of a single data step. Placed at the end of the INPUT statement, they instruct SAS ***not*** to read a new record when the next INPUT statement is executed. This capability is the key element of techniques used to read more complex files and to improve efficiency.

@ **(trailing at-sign)** tells SAS to keep this record current until either an INPUT is executed without a trailing @ or trailing @@, or until this iteration of the DATA step is completed.

In the following example, only TYPE A observations are written. Since the variables B, C, and D are read only when TYPE is A, wasted processing is avoided.

```
DATA trash;
    INFILE trash;
        INPUT type $ @;
        IF type = 'A' THEN DO;
            INPUT b c d;
            OUTPUT;
        END;
RUN;
```

@@ **(double trailing at-sign)** tells SAS to keep this record current through successive iterations of the DATA step. The line will be released when the first of three events occurs: the column pointer moves past the end of the record; an INPUT statement is executed without a @ or @@; or when the DATA step iteration ends and the last executed INPUT statement did not have a @@. For example, the following program reads multiple observations from each input record:

```
DATA trash;
    INFILE trash;
    INPUT  type $ size @@;
RUN;
```

WARNING: When using @ or @@, care must be taken to avoid infinite loops. For example, the following program may result in an infinite loop:

```
DATA streets;
    INFILE address;
    INPUT @1 street $20.  @@;
RUN;
```

Remember that using line hold specifiers may require you to either explicitly end execution of the DATA step with a STOP or ABORT statement or to release the input line with an INPUT statement without line hold specifiers. If you use @ to hold the input record across multiple INPUT statements within the same iteration of the DATA step, you must execute an INPUT without a line hold specifier to input the next record within the same iteration of the DATA step (although the record will be automatically released at the end of the DATA step iteration). If you use @@ to hold the input record across multiple iterations of the DATA step, you must execute an INPUT or INPUT @ statement to release the current record.

Grouping Variables and Informats
You may group variables and informats to reduce the size of the INPUT statement. This technique, illustrated by Program 3 at the end of this paper, is particularly useful when you are reading into arrays or numbered variables. SAS recycles the informat list until the variable list is exhausted.

Comparison of Methods
We prefer to use formatted input with column and line pointer controls. Some types of data (e.g. packed decimal, signed numeric, hexadecimal, and dates) can be read only with formatted input. Data errors do not cascade beyond the element being read. Confusion related to default processing is avoided. Finally, the code helps document the data structure.

INFILE STATEMENT OPTIONS
The INFILE statement has many options, some specific to the host operating system and some generic to any SAS application. In this section, we will explore the more commonly used SAS options.

END=*variable* sets *variable* to 1 (true) when the current record is the last record in the file. This option is frequently used for efficiency purposes. Imbedding the INPUT statement in a DO loop reduces DATA step overhead.

```
DATA stuff;
    INFILE injunk END=nomore;
    DO UNTIL(nomore);
      INPUT ... ;
      OUTPUT;
    END;
RUN;
```

EOF=*label* defines a label to which the program will ***automatically*** branch when an INPUT statement tries to read past the end of the file. In this example, if INJUNK has 16 records, the 17th read of the file would cause the branching to REPORT.

```
DATA stuff;
  INFILE injunk EOF=report;
  INPUT amount;
  totamt + amount;
  OUTPUT;
RETURN;

REPORT:
  PUT '*** TOTAL READ: ' TOTAMT;
RETURN;
```

FIRSTOBS=*number* specifies the first record to be read. The default value is 1.

OBS=*record-number* is the number of the last record to read. The default value is MAX. To define a range of records to read, use FIRSTOBS and OBS together. This INFILE will read 100 records starting at record 100.

```
INFILE injunk FIRSTOBS=100 OBS=199;
```

End of Record Processing

The following mutually exclusive options define what action SAS will take when the program attempts to move the column pointer beyond the end of a record.

FLOWOVER tells SAS to continue reading succeeding records until all variables in the INPUT statement have been read.

MISSOVER tells SAS to set remaining variables in the INPUT statement to missing.

STOPOVER tells SAS to immediately execute a STOP statement, which will stop the DATA step with _ERROR_ equal to 1.

TRUNCOVER tells SAS to salvage whatever it can from short records without going to the next record.

LENGTH=*variable* defines a variable that is set to the length of the current record. Even though the LENGTH= variable is defined by the INFILE, its value is controlled by the execution of the INPUT statement.

```
FILENAME presinfo
'c:\nesug95\presinfo.dat';
DATA bigcheez;
  INFILE presinfo length=inlength
                  missover;
  INPUT @1  inaugdte yymmdd8.
        @9  termdate yymmdd8.   @;
  namelen = inlength - 16;
  INPUT
    @17 presname $varying50. namelen;
RUN;
```

N=*n* tells SAS how many lines to make available to the INPUT statement. Set this value if there is more than 1 record to be read at a time. The default is 1. The line pointer controls / and # would be used. Program 5 illustrates the use of N=.

DELIMITER= identifies one or more delimiters to be used with list input. Either a variable name or a quoted constant may be specified. The default delimiter is a blank.

DSD enables you to read delimited files correctly when consecutive delimiters are present due to missing values. It also enables you to read quoted text, a common occurence when reading files created on desktop systems.

SPECIFIC APPLICATIONS

Program 1: Reading a Basic Fixed Format File

```
FILENAME intools 'c:\sugi24\input1.dat';
DATA tools;
  INFILE intools;
  INPUT
     @1    item     $char10.
     @11  quantity  5.
     @16  saledate  mmddyy8.
     @24  clerk     $char5.
     @24  clrkgrp   2.
     @26  initials  $char3. ;
RUN;
```

The INPUT statement uses column pointer controls to indicate the beginning of each data item. Formatted input is used to control the translation of the data into SAS variables. The variables CLERK, CLRKGRP, and INITIALS illustrate the technique of re-reading the data to generate different variables. Without the re-reading capabilities, two additional statements would be required to create variables INITIALS and CLRKGRP.

Program 2: Reading a Multi-Format File

This program illustrates the use of the line hold specifier @. The input data has a date record, then several detail records, and a final control record. The first character of each record identifies the record type.

```
Filename intools 'c:\sugi24\input2.dat';
DATA tools;
  INFILE intools missover;
  INPUT  @1 rectype $1. @;
  SELECT (rectype);
    WHEN ('0') LINK daterec;
    WHEN ('1') LINK detail;
    WHEN ('9') LINK control;
    OTHERWISE LINK invalid;
  END;
RETURN;

DATEREC: /* read rest of date record */
  INPUT
    @2   created   mmddyy8.
    @10  asofdate  mmddyy8. ;
  /*  more sas code */
```

```
RETURN;

DETAIL: /* read rest of detail record */
  INPUT
    @2   item      $char9.
    @11  quantity  5.
    @16  saledate  mmddyy8.
    @24  clerk     $char5.
    @24  clrkgrp   2.
    @26  initials  $char3. ;
  /* more sas code  */
RETURN;

CONTROL:  /* read rest of control rec */
  INPUT @2  reccount  6.;
  /* more sas code  */
  RETURN;

INVALID: /* this is a bad record */
  /* more sas code  */
RETURN;

RUN;
```

The technique used is to INPUT the record type first and "hold" the record using the trailing @. A SELECT statement is used to LINK to separate sections to INPUT and process each record type. For example, control totals could be maintained for comparison to the control record in the section CONTROL.

PROGRAM 3: Repeating Field Patterns

Program 3 demonstrates how to read a file with repeated fields. Each record has an ACCOUNT followed by 4 occurrences of a set of repeated fields pertaining to cash flow into the policy. All fields are separated by blanks.

```
FILENAME insales 'c:\sugi24\input3.dat';
DATA sales;
  INFILE insales;
  INPUT
    @1  policy              $char8.
    @10 (fund1-fund3)      ($char3. +14)
    @14 (date1-date3)      (yymmdd6. +11)
    @21 (amount1-amount3) (5.2   +12);
RUN;
```

The INPUT statement reads the data into the remaining variables by using **grouped format lists**. Grouped format lists are a very compact way to read in repeating fields because the format lists are recycled until all of the variables are exhausted. Grouped format lists consist of 2 lists, each enclosed by parentheses: the first is the list of variables and the second is the format list to be recycled. The format lists can also include column pointer controls, vital in this case due to the intermixing of the data.

In Program 3, consider the fund information. FUND1 is read using the $CHAR3. informat and then the +14 moves the pointer past DATE1, AMOUNT1, and the intervening blanks. The pointer is now positioned at the data for FUND2.

Looking at the variable list, SAS then reads FUND2 using the recycled format list, which leaves the pointer at the data for FUND3. This recycling continues until the variable list is exhausted. The column pointer control is ignored when the variable list is exhausted. SAS then processes the next part of the input, which reads the date and amount fields in similar fashion.

Program 4: Repeating Field Patterns

This program shows how to explode a file with repeating fields into a SAS data set with one observation for each group of fields. The data elements are separated by blanks.

```
FILENAME insales 'c:\sugi24\input3.dat';
DATA sales;
  INFILE insales missover;
  FORMAT date yymmdd6.;
  INPUT @1  account $char8. @;
  DO i = 1 to 3;
    INPUT
          +1  fund    $char3.
          +1  date    yymmdd6.
          +1  amount  5.2
          @ ;
    OUTPUT;
  END;
RUN;
```

The first INPUT statement reads the variable ACCOUNT, which will be a common identifier for all observations created from this record. This INPUT statement uses an @ to hold the record for the next INPUT statement.

The DO loop executes 3 times. The INPUT statement uses the +1 column pointer control to move the column pointer to the beginning of each field before reading the actual field. The @ line-hold specifier holds the line for the next execution of the loop. When the data step ends, SAS automatically releases the record, allowing the next iteration of the DATA step to read a new record.

Program 5: Using Array Variables

Array variables may be references directly or by index, as this example illustrates.

```
DATA sales invalid;
  array fund     {8} $3;
  array actdate  {8} 8;
  array amount   {8} 8;

  drop i;
  input policy   $char5.
        accounts 3.     @ ;

  if 0 le accounts le dim(fund) then
    do;
      do i = 1 to accounts;
        input fund    (i) $3.
              actdate (i) mmddyy8.
              amount  (i) 5.2      @;
      end;
      output sales;
    end;
  else
    do;
      output invalid;
    end;

cards;
12345001xxx0101199912345
12345002yyy0202199954321zzz0303199967890
run;
```

Program 6: Multiple Records Per Observation

This program code reads a policy information file that spans multiple physical records.

```
INFILE multirec n=4;
INPUT
    #1  @7   policy   $8.
        @28 issuedte  yymmdd8.
    #2  @33 agent     $40.
    #4  @65 state     $3.
    ;
```

The #1, #2, and #4 line pointer controls moves the pointer to those lines before the next part of the INPUT statement is executed. Line 3 is read from the physical file but is not used.

Program 7: Comma Delimited, Quoted Text File

This data format is frequently called CSV (comma separated values) files. Although this example uses a comma as the delimiter, any other set of delimiters may be used to separate the data values. Note that consecutive delimiters in the file are needed when a value is missing, as is SHOES in the second record.

```
DATA sugi24;
  INFILE datalines dlm=’,’ dsd;
  INPUT
       name  $
       count
       footwear $
       method $
         ;
DATALINES;
"JOHN",123,"SHOES",CAR
"JOE",,"SANDALS","TRAIN"
;
```

Program 8: Variable Length Data

Reading variable length data requires the use of the $VARYING informat and a variable that has the actual length of data to be read. In this example, the record contains the length of the variable to be read immediately before the data.

```
DATA names;
INFILE datalines;
INPUT
      len 2.
      first $varying15. len
      len 2.
      last $varying15. len
      ;
DATALINES;
04JOHN10HUNGERFORD
;
```

CONCLUSION

It is hoped that this paper has provided a good overview of the more commonly used SAS features for reading external files. Please obtain and read the SAS Institute publications appropriate for your system. NESUG and SUGI papers, and the SAS-L Internet distribution list are also good sources of SAS programming tips.

Please remember that there are often multiple ways to solve any particular programming problem. Take the time to experiment with different techniques to improve your skills.

The author may be reached via e-mail at clinton.rickards@pharma.com.

SAS® is a registered trademark of SAS Institute Inc., Cary, NC, USA

REFERENCES

SAS Language: Reference, Version 6.06, First Edition, SAS Institute, Inc., Cary, NC

SAS Technical Report P-222, Changes and Enhancements to Base SAS Software. Release 6.07, SAS Institute, Inc., Cary, NC

SAS Technical Report P-242, SAS Software: Changes and Enhancements. Release 6.08, SAS Institute, Inc., Cary, NC