Reading and Writing Character Data

Craig Dickstein, ASG Inc., Weare, NH
Marge Scerbo, Univ. of Maryland Baltimore County, Baltimore, MD

ABSTRACT

This is the information age. Client server, dynamic data exchange (DDE), object linking and embedding (OLE), and open database connectiviity (ODBC) have been added to our vocabulary. Each of these allow quick access to data in a variety of formats. But with all these advances, the need to read character data files and to create ASCII text files is still a major function in the data processing environment. The necessity to share data across platforms, operating systems, and software packages adds to the complexity of daily tasks and often the most direct solution is character data. This paper will discuss the process of reading and writing character data. Examples will be provided with each step. Beginning programmers will be able to follow the conceptual process quite well, while the more experienced analysts may gain some insight into problem solutions. So, the short and tall of this paper will include the simple and the complex.

INTRODUCTION

Just like people, data come in all shapes and sizes. In a perfect world, sharing data across systems and software packages would be a simple trick, requiring little of the programmer's time. Even in a single organization, data may take many forms and reside on a variety of platforms. The challenge for the data processor is to be able to read, manipulate, and write data in any form. Often times the solution is to treat the data as character (text), even if it appears to be numeric in content. The SAS® System provides many tools for this task. This paper will present some of these tools and suggest a philosophy for handling new and unknown data sources.

A set of data will be tracked from input to output and a practical approach built to deal with a variety of problems inherent in new data sources. This data will consist of customer level purchases containing the following elements: Name, Store, Product, Items, and Dollars expended. The tasks are to understand the data file, read the data into a SAS data set, manipulate it into a useable form, and to write the data out for use in another system. Not all of the techniques demonstrated herein are unique to handling character data. For the novice, a process flow is established for translation to other situations. The more advanced reader should find some interesting techniques not often utilized.

For the purposes of this discussion, 'character' data values will be defined as the SAS System defines them: simply a sequence of letters, numbers, and/or special characters. A character type variable is defined on INPUT when a $ follows the variable name or a character informat is used. It can also be predefined by a LENGTH or ATTRIBute statement. Assignment statements can create a new character variable as a result of a 'character' operation. The default length of character variables is 200 and the values are left justified. The variable's length can be alternately defined in a variety of ways. By default, character data is read in its original case. The CAPS/NOCAPS system option will have an effect on how the case of certain input sources are treated. This feature should be studied and tested if there is any question on how the data are treated. The UPCASE and LOWCASE functions are also available for affecting the case of character values and will be discussed later.

As a convention, this paper will use uppercase to denote necessary SAS System keywords while lowercase code will suggest user choice. The examples will be in uppercase to suggest tried and true code. Where necessary, in the code, lowercase may be used in quoted character strings or data values.

KNOW THY DATA

Before attempting data manipulation or analysis on a new source of data, it is imperative that the programmer understand the format and content of the file. Many hours of human and machine resource can be conserved by employing a few simple techniques to understand the data. While many new data sources come with some layout description, it should, at a minimum, be verified. This section looks at a few simple programming statements to accomplish this task.

To read a data file, the location of the file must be defined. This definition will be contained in either the FILENAME or INFILE statements, or a combination of both. One interesting distinction is that FILENAME must be used in conjunction with the INFILE statement if an external device is required (e.g., tapes). If the data source is on DASD (direct access storage device) then the INFILE statement alone would be adequate.

Options are available to these statements for additional definition.

The FILENAME statement serves the purpose of defining a resource. In actual usage the statement may define the input file or the output file. A FILENAME statement is needed to define a device such as a tape. The simple format of this statement assigns a file reference (name) to the file, regardless of its intended use:

```
FILENAME fileref 'a:\filename.filetype';
```

The INFILE statement is used to point to an external file from which data is to be read with the INPUT statement. This statement also contains many options for the definition of that file. Generically, an INFILE statement will appear as:

```
INFILE fileref options ;
```

'Fileref' is the link between the FILENAME statement and the INFILE statement. Alternately, without the FILENAME statement, the same task can be accomplished with a simpler form of the INFILE statement. The location and name of the file must be enclosed in quotes as is done in the FILENAME statement.

```
INFILE 'a:\filename.filetype' ;
```

A variety of options is available for this statement and may be platform specific. As one becomes familiar with the data source through an iterative approach, one or more of these options may become useful. Some of the more useful ones are defined here and will be utilized as necessary.

The reference *fileref* may also be used in the FILE statement to point to the output file which will receive the action of PUT statements. Please note, FILE is the antithesis of INFILE and shares many of the same options. Multiple FILENAME statements will most likely be employed to define a variety of input and output resources.

Options are available to define the physical layout of the data. LRECL= defines the logical record length. RECFM= defines the record format. LINESIZE= limits the record length to be read. The default physical layout of the file to be read is system dependent. For example, in SAS 6.10 for Windows the file default is a line size of 256 (LINESIZE=256) and is in variable length format. Refer to host specific manuals often!!

Often data extracted from a spreadsheet or database will include delimiters. The INFILE option DELIMITER= (optionally, DLM=) will allow for the definition of the delimiter used. With Version 6.07 the DSD option was added to the INFILE statement. Several useful default actions occur with this option. First, DSD will cause the reading of missing values between consecutive delimiters. In the past, two commas side by side in a comma delimited record would have been treated as a single comma; with this option the variable which would have occurred between the commas is set to missing. Second, DSD allows for character data to be read within quotes without including the quotes as data. When data contains the defined delimiter within the character string, the string must be quoted. Third, the default delimiter is the comma such that defining DLM="," is redundant. (See Technical Report P-222 for more on this option.)

Flow control, or controlling the flow of data elements, can be established with the following options. MISSOVER prevents the input statement from reading past the end of the line when the data does not entirely fill the line. If the data is not in column format or if the value of the last field is not complete, omitting this option can cause the misreading of data. STOPOVER will cause the processing of the DATA step to terminate if the INPUT statement reaches the end of the current record without finding values for all variables listed in the INPUT statement. FLOWOVER is the antithesis of STOPOVER and is the default. Given the chance, the SAS System will read past the end of the current record for the values of the defined variables.

When accessing external data these options may become very useful and the reader is encouraged to investigate them further.

A simple, yet effective, method for understanding the physical appearance of the contents of a data file is as follows:

```
FILENAME FRED 'A:\NAMES.TXT';
DATA _NULL_;
   INFILE FRED FIRSTOBS=5 OBS=9;
   INPUT;
   PUT _INFILE_;
```

The first and most necessary task, to dump a few records for visual inspection, has been accomplished. The data is on the A drive in a file named NAMES.TXT and will be referred to henceforth as FRED. This arbitrary file reference is chosen to emphasize the ephemeral nature of *fileref.* It is used solely as a pointer for the duration of the job or session. The data need not be stored either permanently or temporarily so the special data set name _NULL_ is used. The INFILE statement points to the data source via the fileref FRED and will read only five records starting with the fifth record. The defaults, without this kind of

pointer control, would be to start with the first data record and to read them all. Also, be aware that the OBS value is not relative to FIRSTOBS. It is equivalent to lastOBS= but is not spelled this way. The null INPUT statement will read the entire record as one long character field. The PUT statement will write each of these records to the SAS log (default action without a FILE statement).

Inspect the contents of the file and begin to formulate a strategy for reading the file. Data which had been converted from a spreadsheet to an ASCII text file may appear as:

```
Mr. John H Doe,Sears,hardware,6,600
"Mr. and Mrs. Fred Saunders, Jr.",J.C. Penneys,,2,$60
"Dr. and Mr. Barb Wire, Ph.D",LL Bean,apparel,1,400
J Price,"Expensive, LTD",kitchen,3,"7,919"
Mr. H. Donald Smith,,,,,
```

This file (NAMES1.TXT) appears to have been created from a spreadsheet and several potential problems (features) become evident. The file is comma delimited with quoted strings containing commas. Some data is missing. Punctuation is not always consistent. The format of the dollar amount and the item quantity fields are variable.

As a quick aside on delimited files, please note that the primary advantage of delimited files is that the data elements, and therefore records, can be of variable length. Since no blanks are used to pad a character field to align the data in columns, the file itself is more efficient and more compact. Note that the end of the record serves as a delimiter so that an explicit one is not required. A delimited file is similar to a list format file where the data are separated by spaces. For those that have used list input technique delimited files are not new. The blank separating values in simple list input is the implied 'delimiter'. Other characters can and will be used by other software.

A file that is delimited with more than one blank would look like:

```
Mr. John H Doe Sears hardware 6 600
Mr. and Mrs. Fred Saunders, Jr. J.C. Penny two $60
Dr. and Mr. Barb Wire, Ph.D LL Bean apparel 1 400
Ms J Price Expensive, LTD kitchen 7,919
Mr. H. Donald Smith
```

This file (NAMES2.TXT) looks simple enough until notice is taken that there is missing data. Special techniques need to be employed here.

A tape file from a mainframe may have a distinct column layout with two records per observation:

```
Mr. John H Doe                          Sears
hardware                    6           600
Mr. and Mrs. Fred Saunders, Jr.         J.C. Penny
                            two         $60
Dr. and Mr. Barb Wire, Ph.D             LL Bean
apparel                     1           400
```

This file (NAMES3) would suggest a wholly different technique for reading the data.

Without the time taken to dump a few records for visual inspection, the next step would have been painful.

READING THE DATA

It is important to understand that there are many ways to 'skin a cat' with the SAS System. The following treatment is used to emphasize some important points. In no way do the authors portend tight, efficient code. Once the concepts are accepted as good, the reader will discover many ways to employ them. So . . .

For the first set of data (NAMES1.TXT) a special case of *list input* is employed. This is a special case because the delimiter is not a blank but consistently a comma and definable as such. Hence the following code would be adequate:

```
DATA NAMETEST;
    INFILE 'A:\NAMES1.TXT' DSD;
    LENGTH NAME STORE $40
        PRODUCT $12
        ITEMS DOLLARS $8 ;
    INPUT NAME $ STORE $ PRODUCT $
        ITEMS $ DOLLARS $;
```

Note that all fields are read as character data due to the problems noted on inspection. As a default, all data can be read as character, debugged, and converted as necessary at a later time. This is not true if it were to be read as numeric data; all manner of chaos may break out. Also, remember that the DSD option both defines the default delimiter as a comma and accounts for missing variable values. A LENGTH statement is good form. Explicitly define the variable lengths, otherwise, the default action may not be appreciated.

For NAMES2.TXT the following code would serve as a basis:

```
DATA NAMETEST;
    INFILE 'A:\NAMES2.TXT' MISSOVER;
    LENGTH NAME STORE $40 PRODUCT $12
        ITEMS DOLLARS $8 ;
    INPUT NAME $ & STORE $ & PRODUCT $
        ITEMS $ DOLLARS $;
```

In this case, the data was generated with two blanks after each data value and missing data was written out as blanks. The format modifier '&' will allow for character data with single embedded blanks to be read without difficulty. Two or more blanks will act as the appropriate delimiter. The apparent presence of missing data will be handled by the MISSOVER option. "Apparent" because the data step will not be aware of missing data until it reaches the end of the record. The offending record will still need to be dealt with since values may not have been populated to the correct variable.

NAMES3 appears to be nicely formatted in specific columns and should be read as such. Note that the data appears as two records per observation of interest. For this example *formatted column* input is appropriate with the liberal use of *pointer controls*.

```
DATA NAMETEST;
    INFILE 'A:\NAMES3' ;
    INPUT @1 NAME $40. @41 STORE $40. /
        @1 PRODUCT $12. @26 ITEMS $8.
        @41 DOLLARS $8.;
```

In this case the LENGTH statement is not required since the informat, for example $40., assigns a length to the character variables. The relative line pointer (/) and the absolute column pointer serve to locate the data. Careful inspection of the record dump suggest that there are always two records per observation and that the dollar and item fields are in variable format. Again, to circumvent these anomalies, all data are read as character type and will be manipulated to numeric prior to analysis or permanent storage.

It takes experience to become comfortable with the different methods of reading data files. There is no one right method of reading every file. This may at times be controlled by the structure of the file to be read.

Several rules of thumb should be remembered: 1) always print a few of the raw data records for careful inspection and iterative design of the input routine 2) all data can be read as character data and processed as such 3) test a small representative sample 4) print the contents of the test SAS data set.

List input statements are the simplest and work well with files containing only one record per observation and those that are consistently delimited. Take care in using this method; explore all options and techniques available with the INFILE and INPUT statements. Always PROC PRINT the input data to test if the created data set indeed contains the data as in the original file. List input is most useful with the new DSD= option. This allows for the reading of many types of spreadsheet and database output files.

A combination of column and formatted input is probably the most used because of generally accepted methods of storing data. Often, especially in older shops or where large amounts of data are transported, data have been stored in columns. A column format data file is easy to eyeball for errors and allows for nonspecific definition of missing data. It is also the easiest to explain to new programmers and users. Use of pointers allows maximum control within the INPUT statement. Informatting data in the INPUT statement is easy to read and is most efficient from a maintenance perspective.

No matter what method is used, it is always advisable to print a limited number of records to validate the input. If frequencies are appropriate, they also provide a proofing opportunity. Due to the vagaries of character data, every opportunity must be taken to proofread the results.

DATA MANIPULATION

In this section a few of the more interesting functions for manipulating character data will be introduced. This is only meant as an introduction to the vast array of functions available with the SAS System. Other functions and additional uses of these functions are left to the readers further digestion of the SAS Language Reference manual.

In the example data the most interesting problem is with the NAME field. There may be all manner of titles, middle initials, and suffixes. The 'project goals' call for retaining only a first name, middle initial if it exists, and last name. The individual's title and suffix are to be ignored for further processing. For the sake of the example suppose that the data were collected by a variety of clerks (with no regard for consistency) and stored as a spreadsheet. The data has been shipped as such on diskette. Once the data are read in the most generic of forms, as character data with all its anomalies, it can then be parsed into cleaner forms prior to further processing. Here is one method for cleansing this field. Examine the following code:

```
DATA NAMETEST;
    INFILE 'A:\NAMES1.TXT' DSD;
    LENGTH NAME STORE $40
        PRODUCT $12  ITEMS DOLLARS $8 ;
    INPUT NAME $ STORE $ PRODUCT $
        ITEMS $ DOLLARS $;
    NAME=LOWCASE(COMPRESS
            (NAME,'&.,:;'));
```

The result to this point is that NAME is now in all *lowercase* with known extraneous punctuation *compress*ed out. It is very useful to arrive at a known starting point such as this prior to attempting the

parsing of the character string. Next, *the program scans* for known titles in the *first word* of NAME.

```
IF SCAN(NAME,1," ")
    IN ('mr','mrs','ms','dr','drs','col','capt','lt','ltc',
    'maj','sgt','messr','messrs','rev','fr','hon',
    'cong','brig','sr','miss','and' )
THEN DO;
    FNAME = SCAN(NAME,2," ");
    MINIT = SCAN(NAME,3," ");
    LNAME = SCAN(NAME,4," ");
    SUFFIX = SCAN(NAME,5," ");
END;
```

If an occurrence of a title is found, the other *words* are *scanned* into the created variables. If not, there appears not to be a title so the other words are scanned into the appropriate variables.

```
ELSE DO;
    FNAME= SCAN(NAME,1," ");
    MINIT = SCAN(NAME,2," ");
    LNAME = SCAN(NAME,3," ");
    SUFFIX = SCAN(NAME,4," ");
END;
```

But, remember that on inspection 'Mr. and Mrs' is found as a valid title. Other compound titles may exist. So, the *second and third words* in NAME must be scanned. If found, the holding variables will be rewritten with what should be the proper content.

```
IF SCAN(NAME,2," ")
    IN ('mr','mrs','ms','dr','drs','col','capt','lt','ltc',
    maj','sgt','messr','messrs','rev','fr','
    hon','cong','brig','sr','miss','and')
THEN DO;
    FNAME = SCAN(NAME,3," ");
    MINIT = SCAN(NAME,4," ");
    LNAME = SCAN(NAME,5," ");
    SUFFIX = SCAN(NAME,6," ");
END;
IF SCAN(NAME,3," ")
    IN ('mr','mrs','ms','dr','drs','col','capt','lt','ltc',
        'maj','sgt','messr','messrs','rev','fr',
        'hon','cong','brig','sr','miss','and')
THEN DO;
    FNAME = SCAN(NAME,4," ");
    MINIT = SCAN(NAME,5," ");
    LNAME = SCAN(NAME,6," ");
    SUFFIX = SCAN(NAME,7," ");
END;
```

At this point there should be a valid first name, middle initial, last name and suffix. Since SUFFIX is not required for retention it will be dropped. If required, validation checking could have been handled as with TITLE:

```
IF SUFFIX IN
    ('jr','sr','iii','iv','dds','d d s','md','m d','c e o',
    'c f', 'd m d','esq','ii','jnt','m d m','mgr','o d',
    'p e','pres','dmd','s r l','v p','phd')
THEN . . .
```

Since the only real guarantee is that there is a good FNAME, begin at the rear and work backwards to arrive at the other two variables of interest, LNAME and MINIT:

```
IF LNAME IN
    ('jr','sr','iii','iv','dds','d d s','md','m d','c e o',
    'c f','d m d','esq','ii','jnt','m d m','mgr','o d',
    'p e','pres','dmd','s r l','v p','phd')
THEN DO;
    SUFFIX = LNAME;
    LNAME = MINIT;
    MINIT = " ";
END;
```

The LENGTH function is very useful for determining the actual length of a variable value and then acting accordingly. Please note that this is <u>not</u> the length of the variable as defined by the LENGTH statement. The result returned is the integer value of the position of the right-most non-blank character in the argument's value. If the *length* of LNAME is 1 then the variable is missing and the middle name must be the last name.

```
IF LENGTH(LNAME) = 1 THEN DO;
    LNAME = MINIT;
    MINIT = " ";
END;
```

Since just a middle initial is required and not a whole name, several other functions might be employed to arrive at the correct result:

```
IF MINIT NE " " THEN
    MINIT = SUBSTR(MINIT,1,1) || '.';
```

The required three components of NAME are ready for further use. Be aware that all anomalies may not have been taken care of. The two that come to mind are the presence of a dual last name and dual middle initial (e.g., Mr. Johan R. J. Von Snitzel). Further proofing of the data and iterative manipulation code development may be called for. With character data, 100% accuracy may not be possible.

There is also a need to attend to the problems found with DOLLARS. The techniques to employ would be to *compress* out all dollar signs and commas, *left* justify the value, and then to *input* the *put* of the character data.

```
SALES = INPUT(PUT(LEFT(COMPRESS
```

```
(DOLLARS,'$,')),CHAR.),8.);
```

The sky is the limit for data manipulation. The proper combination of functions and formats can be effectively used to turn character data into useful information.

WRITING EXTERNAL DATA

There are times when others, both inside and outside your organization, need your data. How to share this will depend on their hardware and software capabilities. If they have the capability of reading a SAS data set, either as is or in transport format, this will provide an easy method to share data. But as with reading data, this is not always appropriate.

In the creation of an output text file, it is important to understand the needs of the recipient of the file. How the data is written to the file, the format of each variable, etc., all should be taken into account before file creation. For example, the format of each date variable may depend on either or both the hardware and the software to be used to read the file. Certain software packages may require the date in a specific format. This can be easily accomplished by using the correct date format in the output data step.

When an output data file is created, indeed all the data fields become 'character'. This includes numeric data. If this data are to be analyzed by software other than SAS, missing values cannot be transferred in SAS format. Instead, by using the system option MISSING=" ", all missing numeric values will be output as a blank rather than a period. Large numbers containing decimals, including non-formatted dollar amounts, may need to be rounded, truncated or formatted for ease of use. By providing clean and useful data to another user, their work might be made easier and more efficient.

An example of this can be shown with the newly calculated field SALES which could contain decimal amounts and has not been formatted. On some platforms, the value of $21.88 unformatted would be stored and appear as 21.879989624. Rather than include this number in the output file, a more useful number will be output by using the statement:

```
FORMAT SALES 10.2;
```

As with verification of an input file, so should output files be carefully validated. If the data are output correctly, it should be easily reread by a simple SAS input data step. Before creating a very large file, use the OBS= option to create a small file and visually check the output. At the creation of the complete file, make sure the output file contains the correct number of records as stated in the log. If the output file contains multiple records or lines of text per observation, the total number of lines in the file should be evenly divisible by the number of records per observation. If this is not the case, unless specifically programmed to contain a record identifier, the output file may be in error.

With the SAS System, data can be written in most any form to a variety of locations. It is easy to simply 'dump' the contents of a SAS data set to a file. Consider the following code.

```
DATA _NULL_;
    SET NAMETEST;
    FILE 'A:/RAWDATA.OUT';
    PUT _ALL_;
```

All SAS data set variables are written in *list* format to the file defined by the FILE statement. Alternately a combination of the FILENAME and FILE statements could have been used as described above. The generic format of these statements does not change whether describing a source or a target. In order to execute a data step without creating a data set, use _NULL_ as the data set name. Use of this special data set name allows for more efficient use of resources.

Similar options as used on the INFILE statement can be used with the FILE statement. The options LRECL, LINESIZE, RECFM, PAD, FLOWOVER and STOPOVER provide the same capabilities as with the INFILE statement. Rather than a MISSOVER option, the FILE statement contains a DROPOVER option which discards data exceeding the line length.

The PUT statement is a powerful tool that will write information to the defined file in a variety of formats. As with the INPUT statement, the PUT statement can write information in *lists*, *columns*, or *formatted* strings.

Consider the following code for writing the example data set to a tab delimited external file that will be ready for input by a spreadsheet:

```
FILENAME OUT2 'A:\EXCEL.TXT';
DATA _NULL_;
    SET NAMETEST;
    FILE OUT2 LRECL=180 RECFM=F PAD;
    DLM='05'X;
    PUT FNAME DLM MINIT DLM
        LNAME DLM STORE ;
RUN;
```

The data will be output in the order of the variables listed. Spacing will depend on the type of information to be output. For regular *list puts* variable values will always be followed by a space (i.e., the pointer will stop at the second position following the end of a variable value). Character strings will not be followed by a

blank; the pointer will reside in the first position following the last character.

Note the use of the PAD option on the FILE statement. The effect here is to generate fixed length records by padding blanks out to the logical record length (LRECL=). Also, the DLM= option is not available on the FILE statement as it was on INFILE. Since there is the need to write a tab delimited file, assign the hexadecimal value to the new variable DLM and include it in the PUT statement. This again is system specific. A somewhat more efficient method would be to simply use the hex constant '05'X directly in the PUT statement:

```
PUT FNAME '05'X  MINIT '05'X . . .
```

An added value of using a hexadecimal delimiter, in this case a tab, is that it ensures compatibility and flexibility when the data is being ported to a different platform.

With a view toward more consistent and documented data files it is recommended that formatted output techniques be utilized.

On completion of the output file, provide complete documentation to the recipient. Provide the number of records, the line size, logical record length, record format, delimiter if applicable, the sequence of fields included, and if in column format, the column locations. If a format is applicable, include that information.

Just as there are many ways, some efficient, some clever, some easy to document, of reading and manipulating character data files, there are multiple methods to create an output text file. The process of writing a SAS program which will create a column format data file can be tedious and often in error. In addition, creating the documentation to match the output is not a fun or interesting task for a busy programmer. The code below, using ARRAYs and DO loops, automates both tasks.

The program to create both the data and documentation file is within one data step. The first section of the program will create a null SAS data set and define the two files to be output: the data file and the documentation file. The data to be output is stored in the earlier created SAS data set, NAMETEST, and the END= option will help identify when the end of the data set has been read. The code for this portion of the program is:

```
DATA _NULL_;
  FILENAME DAT 'A:/DATA.OUT';
  FILENAME DOC 'A:/DOC.OUT';
  SET NAMETEST  END = ENDFILE;
  FILE DAT;
```

The following SAS statements will define which variables are to be RETAINed throughout the run of the program and the three ARRAYs to be used. The RETAIN statement will include the length of each variable and RETAIN this value throughout the run of the program. The first array, LENSC, will name elements which will contain the lengths of each variable. The second array, VARC, will name the variables to be output. The third array, NAMEVAR, is a character array and will contain the names of the variables. This array will be used in the documentation portion of the program.

```
RETAIN COL 1 LNAMELEN 20 FNAMELEN 15
       STORELEN 40 PRODLEN 12
       ITEMLEN  8 DOLLRLEN 8;
ARRAY LENSC (6)  LNAMELEN FNAMELEN
    STORELEN PRODLEN
    ITEMLEN DOLLRLEN;
ARRAY VARC (6) LNAME FNAME STORE
    PRODUCT ITEM DOLLARS;
ARRAY NAMEVAR (6) $8 'LNAME' 'FNAME'
    'STORE' 'PRODUCT'  'ITEM' 'DOLLARS';
```

Each new record output is counted and added to a counter called RECCNT. A DO loop is now used to PUT the data stored in the appropriate array element from the array VARC in the correct column. This line pointer position is held by using the trailing '@' until all array elements are output. The position of the next COLumn pointer is then calculated by adding 1 to the length of the variable stored in the associated array (LENSC). The adding of 1 will place a blank between columns. The RETAIN statement sets the initial column position (COL) to 1. At the completion of the DO loop, the COL variable is again reset to 1.

```
RECCNT + 1;
DO I = 1 TO 6;
    PUT @COL VARCS(I) @;
    COL + LENSC(I) + 1;
END;
COL = 1;
```

When the end of the data set is encountered, the variable defined in the END= option, ENDFILE, is set to 1 and the next portion of code is processed within an IF THEN DO construct. First, the second output file is referenced in the FILE statement. A header is added to this file using a PUT statement with column pointers '@' and line skips '/'. Then, another DO loop is processed which calculates the beginning and ending columns of each variable and PUTs the variable name from the array NAMEVAR, column beginning and ending locations, and length of the variable into the second output file. The total number of records,

RECCNT, is printed at the bottom of the report, thus satisfying the needs of the user.

```
IF ENDFILE=1 THEN DO;
FILE DOC;
PUT 'NAME LIST'// 'VARIABLES AND
LOCATIONS'// @3 'VARIABLE'
    @13 'BEGINNING' @26 'ENDING'
    @36 'VARIABLE' / @3 'NAME'
    @13 'NUMBER' @26 'NUMBER'
    @36 'LENGTH'/ 43*'*';
DO I = 1 TO 6;
    ENDCOL = COL + LENSC(I) - 1;
    PUT @1 NAMEVAR(I) @16 COL
        @28 ENDCOL @39 LENSC(I);
    COL + LENSC(I) + 1;

END;
PUT /@1 'TOTAL NUMBER OF RECORDS: '
    RECCNT;
END;
```

So, one SAS program can be used to accomplish two tasks. This program can easily be edited for use with other data files. By editing the array elements, the RETAIN statement, and changing the DO loops to the correct number, this program can be usable for many files.

CONCLUSIONS

What seems like an easy task might become an all-day project. Reading and writing character data can be made more efficient by doing the up-front work and carefully analyzing the problem ahead of time. Understanding input data files is all important. By incorrectly reading a text file and not catching the inaccuracies, all further analysis can be null and void.

SAS provides multiple methods for both reading and writing files containing character data. This paper touched upon some of those methods, but as in most cases when using SAS software, there are other techniques. Have the SAS Language Reference manual within arms reach and use it often. Test, test, and verify each step until there is no question about the correctness of the code. Data are still the core of all information.

REFERENCES

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

SAS Institute Inc., *SAS Language: Reference, Version 6, First Edition*

SAS Institute Inc., *Technical Report P-222*

You may contact the authors at:

Craig Dickstein
Voice: (603)529-3818 FAX: by request
EMS: cdd@asg-inc.com

Marge Scerbo
Voice: (410)455-6807 FAX: 410-455-6850
EMS: scerbo@umbc.edu