Reading an Entire Line of Data of Unkown Length Into a Character Variable

Mel Widawski, Office of Academic Computing, UCLA, Los Angeles, California

ABSTRACT

At times when reading complex files it is useful to read a data line into a single character variable and then use string functions to, parse the line of data. When the data is in variable-length records, this can cause problems.

The solution is to use the **$VARYING#.** format on the input statement along with the **LENGTH=** option on the **INFILE** statement. This paper will show how to do this, and discuss implications. A small application using this method to create condensed output will be shown. Concepts used in this presentation are:

- **INFILE 'path' LENGTH=len;** set length
- **INPUT a $VARYING##. len;** to read the line
- **PUT _INFILE_;** to output the line
- **INDEX(a,'string')** to find a string imbedded in the test string, a

INTRODUCTION

There are times when simple input statements are not sufficient to read your data. At these times one solution is to read your data into a single character variable and then use string functions to parse the data into understandable components. These components are then converted to variables as we know them in data analysis.

When the data are read as fixed-length records, there is no problem reading a data line into a character variable for parsing. However, variable-length records cause a problem because the record might end before a given length string is filled. The solution to this problem is to use the **$VARYING##. lvar** format to read the string. Using this format requires that the length of the line be known. Thus using this format requires an additional step. A variable containing the length of the line has to be set on the **INFILE** statement. The following program segment demonstrates the components in context.

```
DATA A;
     INFILE '.\vpagex.lst' LENGTH=len;
     INPUT line $VARYING200. len;
RUN;
```

LENGTH= on the **INPUT** statement sets the variable len to the length of each line as it is read. **$VARYING200.** allows reading lines up to 200 characters long. Specifying len after the format provides the actual length for that line.

I have used this format for reading a group schedule and producing an individual schedule. This requires reading lines of varying lengths and creating variables out of pieces of those lines.

Another application involved reading play-by-play sheets for basketball games and extracting the time, the relative scores, and when a given player entered and left the game. The data was in variable-length records and key words identified which lines of data and what portions of the line needed to be read.

A SIMPLE EXAMPLE

Let us look at a simple situation where reading fixed-length strings will not work. The problem we have is to read some output and determine the number of lines on each page. We detect a page by testing for the automatic title string, **The SAS System**. The data is the output of SAS® procedures.

```
01The SAS System  17:35 Monday, July 26,1999  1
02
030BS    MAKE     PRICE
04
05  1    AMC       4099
06  2    AMC       4749
07  3    AMC       3799
08
09
10The SAS System  17:35 Monday, July 26,1999  2
11
12
13MAKE     Frequency   Percent
14------------------------------
15AMC              3      11.5
16Audi             2       7.7
17BMW              1       3.8
18
19
20
21The SAS System  17:35 Monday, July 26,1999  3
22
23                              Cumulative Cumulative
24REP78  Frequency  Percent  Frequency  Percent
25-----------------------------------------------
26    2         3     11.5          3     11.5
27    3        15     57.7         18     69.2
28    4         6     23.1         24     92.3
29    5         2      7.7         26    100.0
30
31
32
33The SAS System   17:35 Monday, July 26,1999  4
34
35                              Cumulative Cumulative
36FOREIGN  Frequency  Percent Frequency Percent
37-----------------------------------------------
38     0        19     73.1         19     73.1
39     1         7     26.9         26    100.0
40
```

This file has the following number of lines per page:

Page	Number of Lines
1	9
2	11
3	12
4	8

The line numbers inserted on the left make it easy for you to verify the counts.

Fixed Character Input

Here is a program that reads this data with fixed character formats:

```
DATA b;
  INFILE  '.\vpagex.lst'  END=eof;
  INPUT test $CHAR30. @;
  IF INDEX(test,'The SAS System')>0
   THEN DO;
      IF nf>0 THEN OUTPUT ;
      nf=0;
   END;
  nf+1;
  IF eof       THEN OUTPUT;
  KEEP nf ;
RUN;
```

This program reads the lines of output, but does not count the lines correctly. It skips lines less than 30 characters long. It misses outputting the number of lines for the last paragraph because the last line is too short. The following is the comparison of the number of lines in each page and the number of lines that this program detects:

Page	Number of Lines	NF
1	9	5
2	11	7
3	12	9
4	8	.

The **NF** column contains the program's count of the number of lines for each page. In each case an undercount results for each line with a length under 30 characters long.

Character-varying Input

Here is a program that reads this data with character-varying formats (**$VARYING###. Len**):

```
DATA a;
  INFILE  '.\vpagex.lst' LENGTH=len  END=eof;
  INPUT test $VARYING200. len  @;
  IF INDEX(test,'The SAS System')>0 THEN DO;
      IF nv>0 THEN OUTPUT ;
      nv=0;
   END;
  nv+1;
  IF eof       THEN OUTPUT;
  KEEP nv ;
RUN;
```

This program reads the lines of output, and does count the lines correctly. Each line is counted and none are skipped. The following is the comparison of the number of lines in each page and the number of lines that this program detects:

Page	Number of Lines	NV
1	9	9
2	11	11
3	12	12
4	8	8

The **NV** column contains the program's count of the number of lines for each page. In each case the results are as you would expect.

A Caution

In the above examples you can obtain the correct answer for the fixed character case by including **MISSOVER** on the **INFILE** statement. If we were searching for additional key strings on shorter lines, then this shortcut would not work. It also would not work for centered output. Thus it would be to your advantage to learn the method shown here.

A PROGRAM FOR CONDENSING OUTPUT

Output is usually chunked so that every new procedure starts on a new page. Each page may contain only a small number of lines, with the bulk of lines on the page wasted. A number of programmers have come to me to ask for a program that will place more than one procedure on a page.

They also request that we keep the information on any given original page together. In this way, a table never spans a page break.

This requires a two-step process. First, the number of lines per page must be determined. Next, the lines are output to a file, and at the beginning of each page the program determines if the next page will fit on the current output page. If it doesn't, then the page is padded to the printer page length for your output font. To find out how to determine this for window see below. To preserve spacing I use **PUT _INFILE_**.

The following program is a rather simple way to accomplish this task.

```
/* ********************************************** */
/* ***          OUTPUT CONDENSE                 *** */
/* ***                                          *** */
/* ***  Set the line length PS= and the search  *** */
/* ***  string STR= at the beginning of the     *** */
/* ***  program using Macro Variables.          *** */
/* ********************************************** */

%let ps=78;    /* This should equal # of lines in  */
               /* an output page for your printer. */
%let str=The SAS System;

/* ********************************************** */
/* ***  This DATA step determines the lenght    *** */
/* ***  in lines of each input page, and saves  *** */
/* ***  that value in a data set called "a"     *** */
/* ********************************************** */
data a;
    infile  '.\vpagx.lst' length=len end=eof;
    input test $varying200. len  @;
    if index(test,"&str")>0 then do;
         if n>0 then output ;
         n=0;
    end;
    n+1;
    if eof then output;
    keep n;
run;

/* ********************************************** */
/* ***  This DATA step determines how many      *** */
/* ***  input pages fit on an output page.      *** */
/* ***  It then fills the rest of the page      *** */
/* ***  with blank lines up to the output page  *** */
/* ***  size.                                   *** */
/* ********************************************** */
data _null_;
    infile  '.\vpagx.lst' length=len end=eof;
    file    '.\vpagxo.lst' ;

    input test $varying200. len  @;
    if index(test,"&str")>0 then do;
                set a;
                if (nn+n)>&ps then do;
                      do i=1to (&ps-nn);
                          put ;
                        end;
                    nn=0;
                    end;
       end;
    nn+1;
    put _infile_;
    if nn=&ps or eof then do;
         nn=0;
    end;
run;

/* ********************************************** */
/* ***  Just open the output file "vpagxo.lst"  *** */
/* ***  in the program editor. Do a print       *** */
/* ***  preview to insure that the paging is    *** */
/* ***  correct and then print the output.      *** */
/* ********************************************** */
```

Macro Variable: PS

This variable contains the number of lines of printing set by your system. It is set in the **%LET** statement, and the value is used everywhere you see **&PS** in the program.

Macro Variable: STR

This variable contains the string that identifies the title line of the original output. The program uses it to determine the start of each page. By using character-varying formats and the **INDEX** function, the program will work just as well if the output is centered or not. The string value used here is the default title.

This program will also work with alternate titles as long as there is a unique string to identify the title. One option would be the date that appears on the title line of the output.

Notice that the macro variable **&STR** is replaced by its value. When a macro variable is enclosed in double quotes, it is replaced by its value; that is not true if it is enclosed in single quotes.

Additional Program Elements Defined

You may not be familiar with a number of the program elements I use in this program. I will present a short discussion of some of the program elements that may be unfamiliar. I assume that everyone is familiar with the **DATA** statement, and the input statement.

INDEX(string,search) This function returns the position in the **string** of first character of **search**. If the search string is found in **string**, then the result is greater than zero. If the search string is not found, then the result is zero.

END=eof This parameter sets the value of the variable **eof** to one for the last line in the input file.

OUTPUT This statement causes a line to be written to the output **SAS File**. If there is an **OUTPUT** statement, then a case is output only when that statement is executed. There is no longer an automatic output at the end of the data step. Without it a case is written on every pass through the **DATA** step.

PUT _INFILE_ This statement causes an entire input line to be written as is to either the log or the file defined by the **FILE** statement. It insures a verbatim transfer.

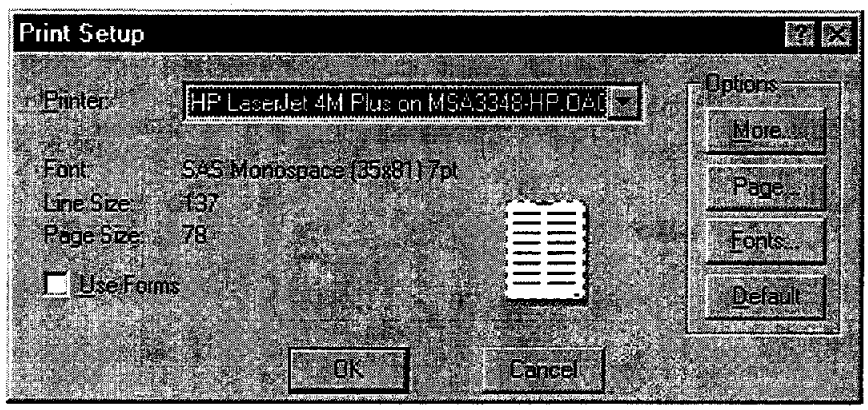
SET You may not know that you can use a **SET** statement along with an input statement. The variables in the data set are retained until the next **SET** statement is executed.

Thus the variable **n** is available until the next execution of **SET** replaces it.

Determining Page Size

This variable contains the number of lines of printing set by your system. In MS Windows® this is determined by your printing font which is set in **PRINT SETUP...** on the **FILE** menu in SAS.

To determine the page size on your PC, choose **PRINT SETUP...** from the **FILE** menu and you will see something that looks like the following window.



Page Size : in the **PRINT SETUP** window is the value that you should use to define **PS** in the **%LET** statement. In this case it is **78** so the statement looks like the following:

```
%LET ps=78;
```

The actual value that you choose will depend on the font size you have chosen for printing. If you are unhappy with the number of lines you can choose a different font size. A smaller size allows more output per page.

PARSING DATE DATA

Let us consider a data set that has variable-length records and date variables entered in different formats. Such a data set follows:

```
mel   03/07/45 red
john  3/8/45 blu
mel   7 mar 45 grn
don   7mar45 grn
mel   3/7/45 grn
mike  10/27/45 vio
mary  55200 vio
```

The task can be broken down into segments. First, the input lines have to be read into variable-length records. Second, the components need to be separated. Third the date types need to be determined, and date variables need to be created.

The first task involves the use of **$VARYING###. Len.** The possibility of embedded blanks makes this necessary, since the data cannot just be read freefield.

The second task is rather easy because the date portion always starts in the same column, and the color is always three characters. This is not always that easy. Sometimes you have to search for the first delimiter and then pull off a name.

The third task is the hardest task, as sometimes the type of date is ambiguous. In our case, it is a little easier as we can make use of the fact that day-month-year dates will always have character months. Also, month-day-year dates will always have numeric values, and some type of delimiter. Julian dates are all numeric with no delimiter, and are always of length 5 in our data set.

The Program and Output

The following program attempts to accomplish the task of reading and parsing this data set.

```
DATA a;
  INFILE '.\vdate.dat' LENGTH=len;
  INPUT rest $VARYING60. len;

  name=SUBSTR(rest,1,5);
  bdatec=TRIM(LEFT(SUBSTR(rest,6,len-(3+5))));
  color=SUBSTR(rest,len-2,3);

  IF INDEXC(UPCASE(bdatec),
           'ABCDEFGHIJKLMNOPQRSTUVWXYZ')>0
       THEN bdate=INPUT(bdatec,DATE9.);
    ELSE IF VERIFY(TRIM(LEFT(bdatec)),
              '0123456789')=0
       THEN DO;
         IF LENGTH(TRIM(bdatec))=5
            AND 90>SUBSTR(bdatec,1,2)>20
            AND    SUBSTR(bdatec,3,3)<=366
         THEN bdate=INPUT(bdatec,julian5.);
       END;
    ELSE  bdate=INPUT(bdatec,MMDDYY8.);

PROC PRINT;
   FORMAT bdate DATE8.;
RUN;
```

The output produced by this program demonstrates the successful parsing and interpreting of the data. Dates are printed using a **DATE8.** format.

NAME	BDATEC	COLOR	BDATE
mel	03/07/45	red	07MAR45
john	3/8/45	blu	08MAR45
mel	7 mar 45	grn	07MAR45
don	7mar45	grn	07MAR45
mel	3/7/45	grn	07MAR45
mike	10/27/45	vio	27OCT45
mary	55200	vio	19JUL55

Program Elements Defined

You may not be familiar with a number of the program elements I use in this program. I will present a short discussion of some of the program elements that may be unfamiliar.

SUBSTR(string,begin,nchar) This function takes the portion of the **string** starting at **begin** and will have **nchar** characters. Thus, **SUBSTR(rest,1,5)** will yield the first five characters of the string **rest**.

TRIM(LEFT(str)) These functions pare off leading and trailing blanks.

VERIFY(str,charlist) This function returns the position of the first character in **str** that is not in **charlist**. A value of zero is returned if all of the characters in **str** are in the **charlist**.

INDEXC(str,charlist) This function returns the position of the first character in **str** that is in **charlist**. A value of zero is returned if none of the characters in **str** are in the **charlist**. We use it here to determine if there is a character coded month imbedded in the date.

LENGTH(str) This function returns the length of **str**. that is in **charlist**.

INPUT(str,format) This function reads a string using the format specified. If you can determine the appropriate date format for a string then you can use this function to read it.

CONCLUSION

We have demonstrated various uses of the **$VARYING##. len** format. You should find it useful any time you have variable record length data that needs to be read into a string variable for manipulation. In addition, you now have a small program that may be used to condense output.

One last caution: at times you may want to do some arithmetic on the length variable **len** before using it with the **$VARYING##.** format for example, you may want to read the string starting at some point after the beginning of the line. A variation of the above example would be to read the **name** separately from the **rest** of the line.

```
DATA a;
  INFILE '.\vdate.dat' LENGTH=len;
  lenn=len-5;
  INPUT name $ @6 rest $VARYING60. lenn;
```

This will not work. In order to do this you will have to include a null **INPUT** statement to set the value of **len**.

```
DATA a;
  INFILE '.\vdate.dat' LENGTH=len;
  INPUT @;
  lenn=len-5;
  INPUT name $ @6 rest $VARYING60. lenn;
```

Other more complex examples are available on request from the author. Please make requests by email as shown below.

REFERENCES

SAS Institute Inc. (1990), *SAS® Language: Reference, Version 6, First Edition*, Cary, NC: SAS Institute Inc.

ACKNOWLEDGMENTS

I would like to acknowledge a number of clients whose desire for saving trees led me to design the example used here. Also, I would like to thank my colleagues, Casey Cantrel and Jonah Schlackman with whom I discussed the solution to the condensing output problem. And finally, thanks to Barbara Widawski without whose editing this manuscript would be illegible.

SAS is a registered trademark or trademark of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

Windows, and Windows NT are registered trademarks or trademarks of Microsoft Corporation in the USA. ® indicates USA registration.

CONTACT INFORMATION

Mel Widawski
Office of Academic Computing
UCLA
Los Angeles, CA

mel@ucla.edu