Paper 228-31

DATALINES, Sequential Files, CSV, HTML and More – Using INFILE and INPUT Statements to Introduce External Data into the SAS® System

Andrew T. Kuligowski, Nielsen Media Research

ABSTRACT / INTRODUCTION

The SAS® System has numerous capabilities to store, analyze, report, and present data. However, those features are useless unless that data is stored in, or can be accessed by, the SAS System.   This presentation is designed to review the INFILE and INPUT statements. It has been set up as a series of examples, each building on the other, rather than a mere recitation of the options as documented in the manual. These examples will include various data sources, including DATALINES, sequential files, CSV files, and HTML files.

GETTING STARTED – BASIC INFILE / INPUT with DATALINES

In order to bring data from an external source into your SAS session, the user must provide the answers to a couple of simple questions: Where is the data, and what does it look like? The **INFILE** statement will define the data source and provide a few tidbits of information regarding its form, while the **INPUT** statement will define the format of the data to be processed.

```
/* INTRO EXAMPLE */
DATA SasConf;
  INFILE DATALINES;
  INPUT ConfName
        ConfYear
        ConfCity
        ConfST  ;
DATALINES;
SUGI      2006 San Francisco  CA
PHARMASUG 2006 Bonita Springs FL
NESUG     2006 Philadelphia   PA
WUSS      2006 Irvine         CA
SESUG     2006 Atlanta        GA
SCSUG     2006 Irving         TX
MWSUG     2006 Dearborn       MI
PNWSUG    2006 Seaside        OR
SUGI      2007 Orlando        FL
;
```

```
NOTE: Invalid data for ConfName in
      line 9 1-9.
NOTE: Invalid data for ConfCity in
      line 9 16-18.
NOTE: Invalid data for ConfST in
      line 9 20-28.
RULE: ----+----1----+----2----+----3---
9    SUGI      2006 San Francisco  CA
ConfName=. ConfYear=2006 ConfCity=.
ConfST=. _ERROR_=1 _N_=1
NOTE: Invalid data for ConfName in
      line 10 1-9.
NOTE: Invalid data for ConfCity in
      line 9 16-18.
NOTE: Invalid data for ConfST in
      line 9 20-28.
10   PHARMASUG 2006 Bonita Springs FL
ConfName=. ConfYear=2006  ConfCity=.
ConfST=. _ERROR_=1 _N_=2
        <… messages repeated for each
            line of input data …>
```

In this first example, we are reading in a list of SAS Conferences around the United States from data actually embedded inside of our code. The INFILE statement references **DATALINES** as the source of its data; DATALINES is a special *file reference* that tells SAS there will be instream data following the conclusion of the DATA Step. As would be expected, the separation between the SAS routine and the actual input data is a keyword, also called **DATALINES**. (Veteran SAS users may be familiar with the original name for this keyword: **CARDS**. CARDS is a throwback to the early days of data processing, when routines and data were fed into the computer via physical punched cards. It is still valid syntax under Release 9 of the SAS System.) Please note that DATALINES is very useful when trying to illustrate an example in a SUGI paper or related publication, but it has the built-in disadvantage in the real world that the data are hard-coded into a routine. Under most circumstances, this limitation is unacceptable in an professional environment.

When using DATALINES, the DATA step detects that the end of the data has arisen when it encounters a semicolon in Position 1 on the line following the last line of legitimate data. As an aside … class attendees often ask what to do if their data contains a semicolon in the 1st position. The early designers of SAS foresaw this possibility, and created a special command to deal with it. The first line of data must be preceded by **DATALINES4** instead of DATALINES, and 4 consecutive semicolons in Positions 1 to 4, rather than just a single semicolon, must follow the last line of data. (The INFILE statement can be followed by either DATALINES or DATALINES4 – the command controls how many semicolons that the

DATA step is anticipating, not the file reference.) If the user has a special data situation where the first 4 characters of input data might be semicolons … use another method to process your request, you have exceeded the capabilities of DATALINES!

The INPUT statement in our example is coded in its simplest form, known as *list input.* List input simply says that we are providing a list of variables to the INPUT statement, without any "complications" such as pointers or formats.

Unfortunately, the first version of our code does not work. SAS will attempt to read everything as numeric values by default. It will reject the alphabetic values for 3 out of our 4 variables as a result. The correction is simple – add the Dollar Sign format modifier, so that SAS can distinguish between Character and numeric data.

```
/* INTRO EXAMPLE – 1st Revision */
DATA SasConf;
  INFILE DATALINES;
  INPUT ConfName $
        ConfYear
        ConfCity $
        ConfST  $ ;
DATALINES4;
SUGI       2006 San Francisco  CA
PHARMASUG 2006 Bonita Springs FL
NESUG      2006 Philadelphia   PA
WUSS       2006 Irvine         CA
SESUG      2006 Atlanta        GA
SCSUG      2006 Irving         TX
MWSUG      2006 Dearborn       MI
PNWSUG     2006 Seaside        OR
SUGI       2007 Orlando        FL
;;;;
```

```
NOTE: The data set WORK.SASCONF has 9
observations and 4 variables.


          < Results of a PROC PRINT >

            Conf
Obs ConfName Year ConfCity  ConfST
 1  SUGI     2006 San       Francisc
 2  PHARMASU 2006 Bonita    Springs
 3  NESUG    2006 Philadel  PA
 4  WUSS     2006 Irvine    CA
 5  SESUG    2006 Atlanta   GA
 6  SCSUG    2006 Irving    TX
 7  MWSUG    2006 Dearborn  MI
 8  PNWSUG   2006 Seaside   OR
 9  SUGI     2007 Orlando   FL
```

OK, upon further review, maybe it wasn't as simple as first hoped. We're able to read in the input data without encountering any error messages, but we've encountered two logical issues. First, the embedded blanks in two of the city names were incorrectly interpreted to represent the delimiters between two values. (SAS considers the blank (" ") to be the default delimiter between fields. We will see in a subsequent example how to override this default.) Secondly, some of the character fields were incorrectly truncated to 8 characters in length. Both of these problems can be fixed.

Breaking down and going to the manual, we might read about *modified list input.* The "book" (using a term which became obsolete in this context, replaced with CDs and .pdf files in the 1990s), says that we can use a *format modifier* to provide the DATA step with additional information about the input variable. In this case, the *ampersand* (**&**) should allow us to read a character variable with a single embedded blank, coupled with a **LENGTH** Statement to override the 8-character default on the two fields with values that exceed that length.

The example on the next page will show that this "corrected" version of the routine did not work, either.   It almost worked; the blank between "San" and "Francisco" was treated as part of the whole value, as was the blank between "Bonita" and "Springs".   However, the INPUT statement also treated the blank delimiter between "Springs", and "FL" as an allowable single embedded blank, and kept reading until it got to the end of the input line. (You don't see this in the output, because the variable was cut off at 14 characters by the LENGTH statement. This is the type of challenge that can arise during debugging.) With one more variable still to read, SAS moved to the next line, grabbed the first set of characters it encountered to complete its processing, and ignored the rest of that input line altogether.

The ideal solution would be to set up the input data with 2 blanks between fields, but such things are not always within the coder's control. Therefore, let us assume that changing the data format is not within our power, and make some additional alterations to our code to bring this example to a successful completion.

Here is the initial attempt to use the "&" format modifier:

```
/* INTRO EXAMPLE – 2nd Revision */
DATA SasConf;
  LENGTH ConfName $ 9.  ConfCity $ 14.;
  INFILE DATALINES;
  INPUT  ConfName   $
         ConfYear
         ConfCity & $
         ConfST    $ ;
DATALINES;
SUGI      2006 San Francisco  CA
PHARMASUG 2006 Bonita Springs FL
NESUG     2006 Philadelphia   PA
WUSS      2006 Irvine         CA
SESUG     2006 Atlanta        GA
SCSUG     2006 Irving         TX
MWSUG     2006 Dearborn       MI
PNWSUG    2006 Seaside        OR
SUGI      2007 Orlando        FL
;
```

```
NOTE: SAS went to a new line when INPUT
      statement reached past the end of
      a line.
NOTE: The data set WORK.SASCONF has
      8 observations and 4 variables.

        < Results of a PROC PRINT >
                               Conf Conf
Obs ConfName   ConfCity        Year ST

 1  SUGI       San Francisco   2006 CA
 2  PHARMASUG  Bonita Springs  2006 NESUG
 3  WUSS       Irvine          2006 CA
 4  SESUG      Atlanta         2006 GA
 5  SCSUG      Irving          2006 TX
 6  MWSUG      Dearborn        2006 MI
 7  PNWSUG     Seaside         2006 OR
 8  SUGI       Orlando         2007 FL
```

Introducing some additional code and concepts, here is the final working version of our first example:

```
/* INTRO EXAMPLE – 3rd Revision */
DATA SasConf;
  LENGTH ConfName $ 9.  ConfCity $ 17.;
  INFILE DATALINES  TRUNCOVER ;
  INPUT  ConfName   $
         ConfYear
         ConfCity & $
         ConfST    $ ;
  IF  ConfST = " "  THEN  DO;
    ConfST   = SUBSTR(ConfCity,16);
    ConfCity = SUBSTR(ConfCity,1,14);
  END;
DATALINES;
SUGI      2006 San Francisco  CA
PHARMASUG 2006 Bonita Springs FL
NESUG     2006 Philadelphia   PA
WUSS      2006 Irvine         CA
SESUG     2006 Atlanta        GA
SCSUG     2006 Irving         TX
MWSUG     2006 Dearborn       MI
PNWSUG    2006 Seaside        OR
SUGI      2007 Orlando        FL
;
```

```
NOTE: The data set WORK.SASCONF has 9
observations and 4 variables.


        < Results of a PROC PRINT >

            Conf
Obs ConfName   Year ConfCity       ConfST
 1  SUGI       2006 San Francisco  CA
 2  PHARMASUG  2006 Bonita Springs FL
 3  NESUG      2006 Philadelphia   PA
 4  WUSS       2006 Irvine         CA
 5  SESUG      2006 Atlanta        GA
 6  SCSUG      2006 Irving         TX
 7  MWSUG      2006 Dearborn       MI
 8  PNWSUG     2006 Seaside        OR
 9  SUGI       2007 Orlando        FL
```

The revised code allows for the Conference City variable to be up to 17 characters long – the length of the actual variable PLUS the Conference State variable PLUS 1, to allow for the embedded blank. We've had to add some logic so that the value for State is removed from the City and placed into its own variable in the one instance where that problem was occurring. In addition, we inserted an option on the INFILE command – **TRUNCOVER**. The default value, FLOWOVER, is what triggers SAS to move to the next input line to complete its current INPUT statement. Truncover specifically tells SAS not to jump to next line; the statement should settle for what it can get, and set the remaining values on the line to missing values. (**MISSOVER** is a similar option to TRUNCOVER. The difference occurs when SAS reaches the end of an input line in the middle of a variable. MISSOVER ignores the partial value and sets that value to missing, while TRUNCOVER "settles for what it can get" and keeps the partial data.)

As one might expect, there are multiple ways to handle each problem. Quite frankly, this example jumps through a few extra "hoops and loops" in order to illustrate modified list input. I would normally prefer to use formatted input in a situation like this; we will address that in the next sections of this presentation.

CONDITIONAL INPUT

Data does not always come in a straightforward format. Sometimes, the records in a file do not always share the same layout; most of those files have a "record type" field or other unique identifier to ensure that the data are read in properly.   Our second example contains similar data to our first example, with a few differences:

a) The data is stored in an external dataset. (Our example uses the MVS operating system for demonstration purposes). We will use the FILENAME command to assign a *file reference* to our dataset.   The file reference, which is an alias or nickname for the dataset, will be used in our INFILE statement.
b) A Record Type has been added in Position 1, denoting if the Conference Type is "I" for "International", "S" for "Special Interest", or "R" for "Regional". Our example will use records with a consistent format for clarity sake; the "conditional" aspect will be to selectively process records for Regional User Group Conferences only.
c) Blank padding between values has been removed.

```
‘USERID.SASCONF.DATA’
ISUGI     2006San Francisco CA
SPHARMASUG2006Bonita SpringsFL
RNESUG    2006Philadelphia  PA
RWUSS     2006Irvine        CA
RSESUG    2006Atlanta       GA
RSCSUG    2006Irving        TX
RMWSUG    2006Dearborn      MI
RPNWSUG   2006Seaside       OR
ISUGI     2007Orlando       FL


/* "RECORD TYPE" EXAMPLE */
FILENAME SASCONF ‘USERID.SASCONF.DATA’;
DATA SasConf;
  INFILE SASCONF ;
  INPUT @ 1 RecordType $CHAR1. @ ;
  IF  RecordType = ‘R’  THEN DO;
     INPUT @  2 ConfName $CHAR9.
           @ 11 ConfYear     4.
           @ 15 ConfCity $CHAR14.
           @ 29 ConfST   $CHAR2. ;
     OUTPUT;
  END;
  ELSE  INPUT;  /* optional */
RUN;
```

```
NOTE: The infile SASCONF is:
      File Name=USERID.SASCONF.DATA,
      Lrecl=80,Recfm=FB,Blksize=960

NOTE: 9 records were read from the
      infile SASCONF.
NOTE: The data set WORK.SASCONF has
      6 observations and 5 variables.

         < Results of a PROC PRINT >

     Record Conf   Conf           Conf
Obs   Type  Name   Year ConfCity    ST
 1     R    NESUG  2006 Philadelphia PA
 2     R    WUSS   2006 Irvine       CA
 3     R    SESUG  2006 Atlanta      GA
 4     R    SCSUG  2006 Irving       TX
 5     R    MWSUG  2006 Dearborn     MI
 6     R    PNWSUG 2006 Seaside      OR
```

This example uses a style known as *formatted input*, with formats and column pointer controls working in tandem to perform the task of reading data. Note the "@" (known as the "at" sign) *column pointer control* accompanying each variable. This symbol causes the DATA step to move the column pointer to an *absolute* position on the input dataset, and to continue reading from that point. "@ 1", for example, tells SAS to move its internal column pointer to position 1, and read the specified variable beginning at that location.   (This concept will be further addressed in the next section of this presentation.)

The INPUT statement in this example concludes with another "@" sign – which has an opposite effect from the one just discussed. By default, the internal *line pointer* will move to the next line of data at the end of the INPUT statement, as denoted by the semicolon. However, this movement can be suppressed – the trailing "@", as it is known, causes the line pointer to remain on the current input line until a subsequent INPUT statement causes the line to move, or until the RUN statement terminates the current iteration of the DATA step. Our example shows how the trailing "@" can be used to perform conditional processing – the value read from the first column of each record is used to determine whether the rest of the record is read and retained. If the first character is an "R" (for "Regional"), we process the remainder of the record. If the first character is anything else, we will invoke an INPUT statement that has no variables associated with it. This *null input* statement has the sole purpose of freeing the line pointer. (In this example, it is unnecessary; the RUN statement will automatically cause the line pointer to increment.

There are other instances where the routine's design may require the coder to release the line pointer manually, as in the example.)

The inquisitive user might wonder if there might be more flexibility; perhaps they might not want to release the line of input data even though the routine has hit the RUN statement. SAS, as one might expect, provides that capability (otherwise I wouldn't have brought the topic up in the first place!)   Two consecutive "@" signs at the end of the line, known as a "double at", will keep the line pointer on the current input line even if RUN is encountered. It becomes the users' responsibility to release the hold via a null INPUT statement; otherwise the routine may continue to read and re-read the same input line; this causes a situation known as an "infinite loop". People who are using a shared processor and cause this phenomenon to occur are likely to incur the wrath of their fellow users and the machine's administrators – not to imply that the author has any first-hand experience with this situation!

USING COLUMN AND LINE POINTERS

The *column pointer* keeps track of the physical location on the current input record that SAS is reading. There are two types of column pointer controls – *absolute* and *relative*. We have already touched upon the absolute column pointer control "@". Its parallel symbol is "**+**" (plus sign), which is known as the relative column pointer. The absolute column pointer control causes the column pointer to be moved to a specific position, while the relative column pointer control triggers it to be moved to a location relative to its current position. Please note that the fact that the relative column pointer control shares a symbol with the representation of positive numbers should not be interpreted to mean that the column pointer can only move forward. Negative values are allowed. To cite an example, +(-5) is valid syntax, and will cause the column pointer to be moved 5 positions to the left of its current location. The parentheses are required when specifying a negative column position, otherwise the DATA step will abend with a syntax error.

It is also possible to control the line pointer, with absolute and relative line pointer controls. The Pound sign (**#**) is the *absolute line pointer control*.   It will move the line pointer to a specified line number within the input buffer. For example, #4 will move to the 4th line of the input buffer. (By default, the input buffer will contain the maximum number of lines specified by using the Pound sign in an INPUT statement. This can be overridden by the N= option of the INFILE statement.)   The Slash (/) performs the function of relative line pointer control. It is not very flexible; the Slash simply signifies that the column pointer should move forward to the next line of input. It is not currently possible to include a parameter with the slash to represent multiple lines; the code must contain 3 slashes in order to jump down 3 lines for example. Since parameters are not accepted, it follows that it is not possible to use a relative line pointer to go backwards in a file. (Note that the column pointer is automatically moved back to Position 1 when the line pointer is invoked.)

We will look at an example that will exercise the capabilities of the column and line pointers. Please make note in advance that this is ugly code, and was done specifically to demonstrate a point (no pun intended). The code will move the column pointer around the record, jumping forwards and backwards within the current record, and will then skip to the next record and do the same type of thing again! This scatter-shot approach is NOT recommended for production code! Anyone interested in pursuing this "coding style" to its logical conclusion and then beyond should seek out one of Art Carpenter's presentations on "Programming for Job Security", which have been offered at numerous SUGI and Regional User Group conferences.

```
'USERID.SASCONF2.DATA'

 I SUGI      2006 LOC=San Francisco  CA
 S PHARMASUG 2006 LOC=Bonita Springs FL
 R NESUG     2006 LOC=Philadelphia   PA
 R WUSS      2006 LOC=Irvine         CA
 R SESUG     2006 LOC=Atlanta        GA
 R SCSUG     2006 LOC=Irving         TX
 R MWSUG     2006 LOC=Dearborn       MI
 R PNWSUG    2006 LOC=Seaside        OR
 I SUGI      2007 LOC=Orlando        FL
 R SESUG     2007 LOC=Hilton Head    SC


/* "Pointers" EXAMPLE */
FILENAME SASCONF2 'USERID.SASCONF2.DATA';
DATA  UGLYINPUT;
  LENGTH  ConfName $ 9. ;
  Pos1 =  9;
  Pos2 =  3;
  Str1 = "LOC=";
  INFILE SASCONF2 ;
  INPUT #1 ConfCity   $ 22-35 +1


          ConfST      $        +(-39)

          RecordType $        +Pos1



          ConfYear             +(-1*(Pos1+6))


          ConfName    $

     / @ "20"       Year2Digit2




      @    1        RecordType2 $


      @ Pos2        ConfName2   $  9.


      @ Str1        ConfCity2   $ 14.




      @ (Pos2*12+1) ConfST2     $ ;


  ConfYear2 = 2000+Year2Digit2 ;
RUN;
```

Go to the first record in the current input buffer, and read a character variable between columns 22next 8 and 35.   NOTE: This is called *column input.* Move the column pointer 1 position to the right.

Move the column pointer 39 positions to the left.

Move the column pointer 9 positions to the right. (POS1 is a numeric variable assigned the value 9.)

Move the column pointer 15 positions to the left. (POS1 is set to 9, add 6 to it and multiply by negative 1 to get -15.)

Skip down to the next record in the current input. Locate the character string "20" in the input data, read the value immediately following that string. (The initial – and incorrect – reaction from the casual reader would be to expect the column pointer to be moved to the 20th position in the input file.)

Move the column pointer to position 1.

Move the column pointer to Position 3. (POS2 is a numeric variable assigned the value "3".)

Locate the character string "LOC=" in the input data, read the value immediately following that string. (STR1 is a character variable assigned the value "LOC=".)

Move the column pointer to Position 37.   (POS2 is set to "3", multiply by 12 and add 1 to get 37.)

Convert from a 2-digit to a 4-digit year. (The INPUT statement used the first two digits to locate the proper location on the input file – it never actually read those 2 digits!)

6

There was considerable interest among veteran programmers when SAS implemented the ability to use character strings as text searches within the INPUT statement. The enhancement seemed to be overshadowed by many of the other improvements in the SAS System over the years. As such, we will take the time to review a separate example which truly exercises this feature.

Our earlier example showed two potential pitfalls to using the character string in conjunction with the column pointer. First of all, the string you are searching for is NOT included in the value read from the file – if you want it included in the variable's value, you will have to take extra measures to do so. Secondly, the string is interpreted very literally. In our case, we looked for numeric characters – the casual reader might have expected the INPUT statement to move to the column denoted by the number rather than the character representation of that number, which could cause issues with debugging and with subsequent maintenance of the routine!

We will read the same input line several times in this example; the character string contains several different variations of the letters "the". Make special note of how capitalization and/or the presence / absence of the trailing blank affects where the column pointer is moved. (In all but one case, we specifically use the line pointer to control the column pointer – remember, any time the line pointer is specified, the column pointer is repositioned back to the first column in the specified line, even when the specified line is still the current line!)

```
/* Example of Column pointer controls
   using character values. */
DATA  THE_data;
   INFILE CARDS;
   INPUT # 1 @"the"  NEXT8_1 $CHAR8.◄


        # 1 @"the " NEXT8_2 $CHAR8.◄


        # 1 @"The"  NEXT8_3 $CHAR8.◄


        # 1 @"The " NEXT8_4 $CHAR8.◄


            @"The"  NEXT8_5 $CHAR8.◄
CARDS;
The theme of the Thebes theater's <etc.>
;
```

Find the *first* lower-case occurrence of the string "the" (lower case without trailing blank), read the next 8 characters.
NEXT8_1 = "me of th"

Find the *first* lower-case occurrence of the string "the " (lower case with trailing blank), read the next 8 characters.
NEXT8_2 = "Thebes t"

Find the *first* lower-case occurrence of the string "The" (capitalized without trailing blank), read the next 8 characters.
NEXT8_3 = " theme o"

Find the *first* lower-case occurrence of the string "The " (capitalized with trailing blank), read the next 8 characters.
NEXT8_4 = "theme of"

Find the *next* lower-case occurrence of the string "The" (capitalized without trailing blank), read the next 8 characters.
NEXT8_5 = "bes thea"

DEALING WITH INFORMATS

A sequential file may contain several types of data. The reader can opt to input this data as ordinary numeric or character data, then transform it programmatically according to its characteristics. However, in many cases, this task can be streamlined by reading the data using an *informat*.   Informats provide instruction for the reading of data into a SAS variable. They specify whether an input value is character or numeric, its length, the number of decimal places (if applicable), and other special conditions.

We have already addressed the $CHAR. informat, which allows us to distinguish and read character data as opposed to numeric values. Alert readers will wonder why anyone would want to spell out $CHAR*w*., when $ *w*. works just as well with less keystrokes. (*w*, of course, represents a positive integer – the *width* of the requested value.) The two formats do not function exactly the same way - $ *w*. ignores leading blanks, while $CHAR*w*. includes them. Either format could be used in the examples in this presentation. However, circumstances may dictate that one be selected over the other when processing certain input fields in the "real world".

There are two methods to specify an informat for a variable. The first is the method already used in this presentation, to specify the informat along with the variable on the INPUT statement. The other method is to use either the **INFORMAT** or **ATTRIB** statement. Either of these statements can be specified in the DATA step to permanently assign an informat to variables within a SAS dataset. The syntax for each of these statements is:

```
INFORMAT variables <informat>
         <DEFAULT=default informat>;
ATTRIB variables INFORMAT=informat;
```

There are five categories of informats available in the SAS System: *numeric, character, date/time, column-binary,* and *user-defined*. A few common examples are listed below; more extensive lists and descriptions of formats in the first four categories can be found in the *SAS Language: Reference* manual. Still more examples that are specific to a given operating system can be found in the *SAS Companion* for that operating system. Those options never seem to be sufficient, however – almost everyone has some shop-specific or routine-specific informats that are not part of the standard set. SAS even provides a tool for folks who need their own specialized user-defined INFORMATs – **PROC FORMAT**. As the name suggests, PROC FORMAT creates user-written formats, and does informats as well. This topic is outside the scope of this paper; there are several others devoted solely to this topic. One short but informative presentation that the reader might find useful is "IN & OUT of CNTL with PROC FORMAT", which can be found in several SAS conference Proceedings from 1998. (Ignore the age of the article; the topic is timeless and the material still current.)

A few selected examples of SAS Informats

INFORMAT	DESCRIPTION
$ASCII*w*.	On IBM Mainframe, convert from ASCII to EBCDIC.
	On other systems, treat as $CHAR.
$EBCDIC*w*.	On IBM Mainframe, treat as $CHAR.
	On other systems, convert from EBCDIC to ASCII.
COMMA*w.d*	Read numeric value, ignore embedded commas (and other accounting symbols, such as "$", percent signs, etc.). NOTE: A left parenthesis at the beginning of the value causes the subsequent numeric value to be treated as a negative.
DATE*w*.	Read date (ddMMMyy or ddMMMyyyy) into numeric field.
JULIAN*w*.	Read Julian date (yyddd or yyyyddd) into numeric field.
w.d	Read standard numeric value, total width = *w* including *d* decimal places.

VARIABLE LENGTH FILES

Up to this point, all of our examples have focused on fixed length files where each variable occurs in the same position on each record of the file. Our next example will deal with variable length files; files in which the length can change from record to record.

Let us look at a file containing one record for each member of a family. Our routine will be based on a record format where the 10 characters contain the person's name, followed by one or more 2-byte state abbreviations representing each state where that person lived for some period of time in their life.

```
'USERID.FAMILY.DATA(STATE)'
Angus      NY
Trixie     ARNYGARIOK
Vladmir    NYFL
Swen       NYMIPA
Petula     NY
Caleb      NYPACA


/* VARIABLE LENGTH FILE EXAMPLE #1*/
FILENAME FAMILY 'USERID.FAMILY.DATA';
DATA FAM_ST;
  ARRAY  ST (5) $ ;
  INFILE FAMILY(STATE) LENGTH=linelen ;
  INPUT @ 1 Person $CHAR10. @ ;
  idx = (linelen-10) / 2;
  DO loop = 1 TO idx;
     INPUT ST(loop) $ 2. @ ;
  END;
RUN;
```

```
NOTE: The infile library FAMILY is:
      Dsname=CUSATK.FAMILY.DATA,
      Unit=3390,Disp=OLD,Blksize=27998,
      Lrecl=80,Recfm=VB
NOTE: The infile FAMILY(STATE) is:
      Dsname=USERID.FAMILY.DATA(STATE),
      Unit=3390,Disp=SHR,Blksize=27998,
      Lrecl=80,Recfm=VB

NOTE: A total of 6 records were read
      from the infile library FAMILY.
      The minimum record length was 12.
      The maximum record length was 20.
NOTE: 6 records were read from the
      infile FAMILY(STATE).
      The minimum record length was 12.
      The maximum record length was 20.
NOTE: The data set WORK.FAM_ST has
      6 observations and 8 variables.

           < Results of a PROC PRINT >

Obs ST1 ST2 ST3 ST4 ST5 Person  idx loop
 1   NY                  Angus    1   2
 2   AR  NY  GA  RI  OK Trixie   5   6
 3   NY  FL              Vladmir  2   3
 4   NY  MI  PA          Swen     3   4
 5   NY                  Petula   1   2
 6   NY  PA  CA          Caleb    3   4
```

Like in the prior example, the data are stored in an external MVS dataset. Again, we will use the FILENAME command to assign a file reference to our dataset. However, there is one difference regarding data storage between these two examples. The current example refers to an "aggregate storage location", known under MVS as a *partition dataset* or *PDS*. (It is my understanding that there are parallel constructs in some other operating systems). In this case, the INFILE statement is provided with a file reference to an external file, and which is followed by a specific member reference, enclosed in parentheses. This allows a single file reference to be used for multiple INFILE statements, each pointing to a different PDS member.

The last field on the file can contain one or more 2-character values. Our example is set up for a maximum of 5 such values, which is capped by the ARRAY's dimension. (As an aside, our routine is lacking an edit check to handle a situation where we reach the 6th occurrence; true production code would be prepared for that situation.) We establish the number of occurrences to process by determining the total length of the input line, then subtracting away the length of the key information at the beginning of the line. We conclude this process by dividing the resultant number by 2, since each value is 2 bytes long.

All of this leads into the critical question of this example – how do we determine the length of the current input line? The answer is amazingly simple; there is a **LENGTH=**<variable> option available on the INFILE statement. <variable> is a temporary SAS variable specified by the coder. It is set to the length of the current line of INPUT when the line is read in. Note that it is a temporary variable, and is not written to the output dataset.

Our first example using variable length files still maintains the construct of having each field clearly lined up in the same column in each record, with the only difference being the number of values stored on each line. It is also possible to encounter a file where the values do not line up neatly in columns. In this case, SAS provides a useful informat called **$VARYING** that provides increased flexibility.

```
'USERID.FAMILY.DATA(CITY)'
San Francisco CA
Bonita Springs FL
Philadelphia PA
Irvine CA
Atlanta GA
Irving TX
Dearborn MI
Seaside OR
Orlando FL


/* VARIABLE LENGTH FILE EXAMPLE #2*/
FILENAME FAMILY 'USERID.FAMILY.DATA';
DATA CITY (KEEP=city state varlen) ;
  INFILE FAMILY(CITY)  LENGTH=linelen ;
  INPUT @ ;
  INPUT name $VARYING40. linelen;
  city  = SUBSTR( name, 1, linelen-2 );
  state = SUBSTR( name, linelen-1 );
  varlen = linelen ;
RUN;
```

```
NOTE: The infile library FAMILY is:
      Dsname=CUSATK.FAMILY.DATA,
      Unit=3390,Disp=OLD,Blksize=27998,
      Lrecl=80,Recfm=VB
NOTE: The infile FAMILY(CITY) is:
      Dsname=CUSATK.FAMILY.DATA(CITY),
      Unit=3390,Disp=SHR,Blksize=27998,
      Lrecl=80,Recfm=VB
NOTE: A total of 9 records were read
      from the infile library FAMILY.
      The minimum record length was 6.
      The maximum record length was 14.
NOTE: 9 records were read from the
      infile FAMILY(CITY).
      The minimum record length was 6.
      The maximum record length was 14.
NOTE: The data set WORK.CITY has
      9 observations and 4 variables.

          < Results of a PROC PRINT >

Obs city            state varlen
  1  San Francisco   CA     16
  2  Bonita Springs  FL     17
  3  Philadelphia    PA     15
  4  Irvine          CA      9
  5  Atlanta         GA     10
  6  Irving          TX      9
  7  Dearborn        MI     11
  8  Seaside         OR     10
  9  Orlando         FL     10
```

The observant reader will immediately notice that we perform two consecutive INPUT statements. The first is a null input, while the second actually performs the INPUT process against our data. This raises the question why a null input would be necessary - we stated earlier in this presentation that its function was to terminate a trailing @, which is obviously not the case here! In this example, the null input allows SAS to set the value of the LENGTH= variable. The second INPUT actually uses that information as specified by the *linelen* variable in our example; it has to be passed in from an earlier INPUT. (Our earlier example also did this, but performed the process subtly – the first INPUT was not simply a null input, but rather actually read part of the record, and set *linelen* at that time.)

The $VARYING informat provides a greater level of flexibility than the traditional $ format used earlier in this presentation. It uses a second parameter to set the maximum length of the variable being processed in the current line – this avoids the "**INPUT statement reached past the end of a line**" messages we encountered earlier in this paper. (SAS processes negative and missing values as zeros, considering it to represent a null input line.)   Our example also specifies a maximum length of 40 for the resultant variable with a LENGTH statement. This is not required; however, its absence will cause SAS to default the field to a length of 8 characters, as we have seen in earlier examples.

The observant reader will also have noticed that our output dataset contains the line length. This does not contradict my earlier statement that the value is stored in a temporary SAS variable and not written to the output dataset. In our example, we create a new (externally available) variable, and assign it the value of our internal variable – *this* is the field displayed in our output, not the temporary variable!

CSV (*C*omma *S*eparated *V*alue) FILES

The *CSV,* or *Comma Separated Value* File is a special variety of variable length sequential file, typically used for importing or exporting data from a spreadsheet. Data values are separated by commas, as is implied by the name, and character values are typically surrounded by double quotation marks (").

```
C:\Papers\ConfList.CSV
"SUGI",2006,"San Francisco","CA"
"PHARMASUG",2006,"Bonita Springs","FL"
"NESUG",2006,"Philadelphia","PA"
"WUSS",2006,"Irvine","CA"
"SESUG",2006,"Atlanta","GA"
"SCSUG",2006,"Irving","TX"
"MWSUG",2006,"Dearborn","MI"
"PNWSUG",2006,"Seaside","OR"
"SUGI",2007,"Orlando","FL"


/* CSV EXAMPLE */
DATA SasConf;
  LENGTH ConfName $  9.
         ConfCity $ 14.;
  INFILE 'C:\Papers\ConfList.CSV' DSD;
  INPUT ConfName $CHAR9.
        ConfYear      4.
        ConfCity $CHAR14.
        ConfST   $CHAR2. ;
RUN;
```

```
NOTE: Invalid data for ConfYear in
      line 11 10-13.
RULE: ----+----1----+----2----+----3---
11    "SUGI",2006,"San Francisco","CA"
ConfName="SUGI",20 ConfCity=San Francisco"
ConfYear=. ConfST=," _ERROR_=1 _N_=1
NOTE: Invalid data for ConfYear in
      line 12 10-13.
12    "PHARMASUG",2006,"Bonita Springs","FL"
ConfName="PHARMASU ConfCity=006,"Bonita Sp
ConfYear=. ConfST=ri _ERROR_=1 _N_=2
NOTE: Invalid data for ConfYear in
      line 13 10-13.
13    "NESUG",2006,"Philadelphia","PA"
ConfName="NESUG",2 ConfCity="Philadelphia"
ConfYear=. ConfST=," _ERROR_=1 _N_=3
NOTE: Invalid data for ConfYear in
      line 14 10-13.

     <… messages repeated for each
        line of input data …>
```

In the example above, INFILE contains the full name of the external file, within quotes, rather than a file reference as in our earlier examples. This method provides a clear explanation to subsequent coders and users as to the source of the external data. This strength is also its biggest weakness; there is no flexibility regarding input files! If a user wants to run the routine against a different file, then he or she must go into the code and change the file name manually. In many shops, this direct manipulation of a production routine raises testing concerns and auditing issues. An alternative would be to clone the routine and change the file name in the new copy. Unfortunately, this type of behavior can cause maintenance nightmares down the road – a change to the processing of this routine might require parallel changes to several clones. Personal experience has shown that the support personnel will only find and fix a fraction of the clones that actually exist, resulting in further maintenance issues down the road.

The INFILE statement in our example contains a special keyword – the **DSD** parameter. DSD, which stands for "Delimiter Separated Data", modifies a few of the default assumptions that accompany the INFILE statement:
- The default delimiter is changed from blank to comma. This default, whether blank or comma, can always be overridden with the DELIMITER= option, which will be discussed shortly.
- Character values are assumed to be enclosed in double quotes; these quotation marks are stripped off the input before storing the value. NOTE: You can retain the quotation marks in the variable, if desired, by using the tilde format modifier (~). We will not be needing it in our example.
- Multiple consecutive commas (or other delimiter as defined by DELIMITER=) are assumed to represent missing values. Under normal circumstances, the second and subsequent consecutive delimiters are ignored and treated as a single delimiter during INPUT.

It is obvious that the routine did not work as expected during this first iteration. A quick examination of the SASLOG shows that the problem was caused by the use of lengths on the formats in our INPUT statement. The presence of the $CHAR9. format on the first variable caused the DATA step to read the first 9 characters on each line, treating the comma as a part of the field value rather than as a delimiter between fields. (Find the reference to `ConfName="SUGI",20` in the SASLOG for a specific example. The field value incorrectly contains both quotation marks, plus the comma delimiter, plus the first two characters of the year!) This issue can be resolved by simply using the "$" format modifier without referencing a length to denote character variables, and including a LENGTH statement to override the 8 byte default when appropriate.

11

```
C:\Papers\ConfList.CSV
"SUGI",2006,"San Francisco","CA"
"PHARMASUG",2006,"Bonita Springs","FL"
"NESUG",2006,"Philadelphia","PA"
"WUSS",2006,"Irvine","CA"
"SESUG",2006,"Atlanta","GA"
"SCSUG",2006,"Irving","TX"
"MWSUG",2006,"Dearborn","MI"
"PNWSUG",2006,"Seaside","OR"
"SUGI",2007,"Orlando","FL"


/* CSV EXAMPLE */
DATA SasConf;
  LENGTH ConfName $  9.
         ConfCity $ 14.;
  INFILE 'C:\Papers\ConfList.CSV' DSD;
  INPUT ConfName $
        ConfYear
        ConfCity $
        ConfST   $  ;
RUN;
```

```
NOTE: The data set WORK.SASCONF has 9
observations and 4 variables.


            < Results of a PROC PRINT >

              Conf
Obs ConfName  Year ConfCity       ConfST
 1  SUGI      2006 San Francisco  CA
 2  PHARMASUG 2006 Bonita Springs FL
 3  NESUG     2006 Philadelphia   PA
 4  WUSS      2006 Irvine         CA
 5  SESUG     2006 Atlanta        GA
 6  SCSUG     2006 Irving         TX
 7  MWSUG     2006 Dearborn       MI
 8  PNWSUG    2006 Seaside        OR
 9  SUGI      2007 Orlando        FL
```

As previously mentioned, and as should be obvious from the name, the most common delimiter used in Comma Separated Value formatted files is, of course, the comma. BUT, other delimiters can work just as well and can be easily handled by the SAS system. Take, for example, the a real-world situation of a shop that decided the Tilde "~" was the only field that never showed up in their character data, and as such they wanted to use it as the delimiter in their "CSV" files. This change requires a very minor alteration to our existing routine. We simply add the **DELIMITER=**"~" (abbreviated **DLM=**"~") option on the INFILE statement to inform the DATA step that the Tilde will act as the boundary between field values. (We leave the DSD parameter as is, since our file is still considered to contain "delimiter separated data.") Mirroring the example above, we illustrate that we can process the tilde instead of the comma with minimal changes to the code.

```
C:\Papers\ConfListTilde.CSV
"SUGI"~2006~"San Francisco"~"CA"
"PHARMASUG"~2006~"Bonita Springs"~"FL"
"NESUG"~2006~"Philadelphia"~"PA"
"WUSS"~2006~"Irvine"~"CA"
"SESUG"~2006~"Atlanta"~"GA"
"SCSUG"~2006~"Irving"~"TX"
"MWSUG"~2006~"Dearborn"~"MI"
"PNWSUG"~2006~"Seaside"~"OR"
"SUGI"~2007~"Orlando"~"FL"


/* CSV EXAMPLE */
DATA SasConf;
  LENGTH ConfName $  9.
         ConfCity $ 14.;
  INFILE 'C:\Papers\ConfListTilde.CSV'
         DSD DLM="~";
  INPUT ConfName $
        ConfYear
        ConfCity $
        ConfST   $  ;
RUN;
```

```
NOTE: The data set WORK.SASCONF has 9
observations and 4 variables.


            < Results of a PROC PRINT >

              Conf
Obs ConfName  Year ConfCity       ConfST
 1  SUGI      2006 San Francisco  CA
 2  PHARMASUG 2006 Bonita Springs FL
 3  NESUG     2006 Philadelphia   PA
 4  WUSS      2006 Irvine         CA
 5  SESUG     2006 Atlanta        GA
 6  SCSUG     2006 Irving         TX
 7  MWSUG     2006 Dearborn       MI
 8  PNWSUG    2006 Seaside        OR
 9  SUGI      2007 Orlando        FL
```

DDE

The next logical step in dealing with spreadsheets would be to bypass the CSV file and to get the information directly from the spreadsheet itself. It is possible to accomplish this in SAS with the INFILE and INPUT statements using *Dynamic Data Exchange*. The Windows Help Facility states that Dynamic Data Exchange, or DDE, is *"A form of interprocess communication (IPC) implemented in the Microsoft*

Windows family of operating systems. Two or more programs that support Dynamic Data Exchange (DDE) can exchange information and commands." In other words, DDE allows a client application to request information from a server application in a Windows environment.

The SAS System can interact with Microsoft Excel using DDE. SAS acts as a client application in this relationship. It can request data from a server application, with the requirement that the server application must be running. (It can also send commands and data to a server application, but that is a topic for another presentation.)

```
X 'C:\EXCEL SASCONF.XLS';

FILENAME SASCONF DDE
'Excel|[Book1]Sheet1!R1C1:R9C4';

/* DDE EXAMPLE – Excel */
DATA SasConf;
  INFILE SASCONF;
  INPUT ConfName $
        ConfYear
        ConfCity $
        ConfST   $ ;
RUN;
```


< no SAS output displayed >

Two questions usually come to mind after seeing this example:
- What the heck is that bizarre string of characters following the word DDE in the FILENAME statement? It sure doesn't look like any file name I've ever seen!
- Where'd my SAS session go? Nothing's happening!

A side query will help to address the first question – What is the biggest difference between this example and every other one discussed in this presentation? The answer is, in this instance, we are not reading from a file in the traditional sense – not a sequential file, not a variable length file, not a partition dataset. Rather, we are actually interacting directly with another software package, which in this case is Microsoft Excel. We have to establish a connection between the client application (SAS) and the server application (MS Excel). This is accomplished by issuing a **FILENAME** statement with the keyword "**DDE**". The syntax for this statement, in this context, is:

> **FILENAME** *fileref* **DDE** *'DDE-triplet'* **;**

The DDE-triplet is a specialized argument, and is made up of three components:

> **application|topic!item**

Application is the name of the server application, Excel in this example. *Topic* is defined as the "topic of conversation"; this is typically the file to be processed. When dealing with Excel, the topic contains the Workbook name and the Worksheet name. *Item* is the "item of conversation"; this is the range of cells that is to be included when processing a spreadsheet like Excel

The DDE-triplet for the Excel worksheet in our example is:

> **'Excel|[Book1]Sheet1!R1C1:R9C4';**

Walking through each component of the triplet, we can see that we are going to Excel, the 1st worksheet in the default file of the spreadsheet, and we will be processing the cell at Row 1 Column 1 through the cell at Row 9 Column 4. Note that the application and topic are separated by a vertical bar (|), while the topic and item are separated by an explanation point (!). The brackets are actually part of the worksheet definition, and the colon is the separator between the start and end cell.

The specific format of a DDE triplet for an application should be defined in the documentation for that application. (Our example uses MS Excel, but DDE is available for other software products, as well.) Most people find it easier to let SAS determine the proper DDE triplet. The following is a step-by-step method to obtain the proper DDE triplet for an application using the SAS System:
- Ensure SAS and Excel (or other application) are both active.
- Toggle to your application, and use the standard PC "cut" techniques to store the portion of the client application to be processed in the Windows Clipboard. (For example, use the mouse to highlight the area to be "cut", then select CUT or COPY on the EDIT pop-up menu of most Windows applications.)
- Toggle to your SAS session, and click on the "Options" menu on the Menu Bar in SAS.
- The Options menu will contain an option called "DDE Triplet". Click on it.
- This will display an Information Box, which will contain the DDE-triplet.
- Enter this DDE-triplet into the FILENAME statement of your SAS routine.

Now that the subject of the DDE triplet is resolved, we can begin to debug our application. In this instance, we have a major issue – nothing whatsoever appears to be happening! As you might expect, this is because SAS is doing exactly what we told it to do, but not what we wanted it to do. We used the **X** statement to issue a command directly to the operating system. In this case, we asked Windows to launch Excel and open our requested spreadsheet. SAS did exactly that … and then went into a wait state until the Operating System command was completed, that is, until Excel was terminated. This is obviously not what we want – we can't interface between two applications if only one of them is active!

The solution is to use two SAS options: **NOXSYNC** and **NOXWAIT**. XSYNC is the system option which "specifies that the operating system command execute synchronously with your SAS Session." More to the point, SAS waits until the Operating System command is complete before resuming. XWAIT specifies that the user must specifically exit the command processor before SAS resumes i.e. type EXIT or equivalent. It is necessary to turn both options off before invoking the call to Excel. (Of course, it is also possible to open Excel prior to beginning your SAS session!)

```
OPTIONS  NOXSYNC NOXWAIT;

X 'C:\EXCEL SASCONF.XLS';

FILENAME SASCONF DDE
'Excel|[Book1]Sheet1!R1C1:R9C4';

/* DDE EXAMPLE – Excel */
DATA SasConf;
  INFILE SASCONF;
  INPUT ConfName $
        ConfYear
        ConfCity $
        ConfST   $  ;
RUN;
```


< Results of a PROC PRINT >

```
Obs ConfName  Year ConfCity   ConfST
 1  SUGI      2006 San        Francisc
 2  PHARMASUG 2006 Bonita     Springs
 3  NESUG     2006 Philadel   PA
 4  WUSS      2006 Irvine     CA
 5  SESUG     2006 Atlanta    GA
 6  SCSUG     2006 Irving     TX
 7  MWSUG     2006 Dearborn   MI
 8  PNWSUG    2006 Seaside    OR
 9  SUGI      2007 Orlando    FL
```

It looks like this approach solved all of our problems! We were able to open Excel, and we were able to return to SAS and use the interface to populate our dataset. Unfortunately, we have introduced new errors – some of the values have been truncated, and embedded blanks caused the input to be parsed incorrectly. Both of these problems should look familiar – we encountered and resolved the same type of issues in our first example reading from a sequential file! The corrections to resolve the problems are also similar – inform the DATA step that embedded blanks are expected and acceptable, and that variables might be longer than 8 characters.

```
OPTIONS  NOXSYNC NOXWAIT;

X 'C:\EXCEL SASCONF.XLS';

FILENAME SASCONF DDE
'Excel|[Book1]Sheet1!R1C1:R9C4'
 NOTAB  MISSOVER ;

/* DDE EXAMPLE – Excel */
DATA SasConf;
  LENGTH ConfName $  9.
         ConfCity $ 14.;
  INFILE SASCONF '09'x;
  INPUT ConfName $
        ConfYear
        ConfCity $
        ConfST   $  ;
RUN;
```
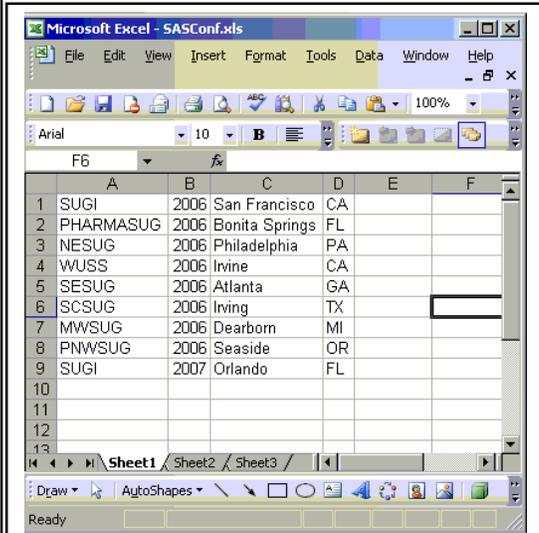
< Results of a PROC PRINT >

```
Obs ConfName   Year ConfCity      ConfST
  1  SUGI       2006 San Francisco  CA
  2  PHARMASUG  2006 Bonita Springs FL
  3  NESUG      2006 Philadelphia   PA
  4  WUSS       2006 Irvine         CA
  5  SESUG      2006 Atlanta        GA
  6  SCSUG      2006 Irving         TX
  7  MWSUG      2006 Dearborn       MI
  8  PNWSUG     2006 Seaside        OR
  9  SUGI       2007 Orlando        FL
```

The actual methods to resolve these issues are not necessarily the same as the methods originally used in our first example. In particular, the embedded blanks cannot be resolved with the ampersand format modifier as they were in the "modified list input" example earlier in this paper. Rather, it is necessary to add a parameter to the INFILE statement, **NOTAB**, to allow the SAS DDE interface to include blank lines. The LENGTH statement is sufficient to process values longer than the default 8 characters, just as it was in our first example. However, this is the only acceptable method to deal with this situation! As we observed in a previous example, use of a character format with a length on the INPUT statement itself, causes SAS to read past the embedded blank *and* possibly past the column boundary itself, until it read in the maximum number of characters specified in the format!

The alert reader will note that we also added the MISSOVER option to the FILENAME statement. This option was discussed earlier in this presentation. It is not actually necessary in our example as it currently stands; it would be very useful, however, should any of the cells be empty.

As an aside, it is possible to use DDE without knowing the name of the DDE triplet. However, there is some manual intervention required.
- Toggle to your application, and use the standard PC "cut" techniques to store the portion of the client application to be processed in the Windows Clipboard.
- Toggle to your SAS session, and replace the FILENAME statement with the following:
  ```
  FILENAME fileref  DDE CLIPBOARD ;
  ```

15

SAS will obtain its data from the clipboard, rather than the server application. The inherent weakness in this approach is that the data to be processed must be stored in the Clipboard prior to each invocation of your SAS routine. However, it does get you around having to work with DDE-triplets.

It is even possible to use DDE to get real-time updates from Excel to SAS! The HOTLINK option on the FILENAME statement will inform SAS of your intent to work real-time. Time and space limitations prevent us from going into details in this presentation. (Besides, real-time examples rarely convey a sense of excitement when transcribed onto paper or other "fixed" medium!)

It is also possible to use DDE with applications other than MS Excel. The DDE triplet is still required, but its form will change depending on the requested application. For example:

MS Word	`winword|"filename.doc"!bookmark`
MS Access	`MSAccess|filename.mdb;Table TableName!All`

Space and time considerations prevent us from delving into these features during this presentation. Readers are encouraged to seek out the writings of Koen Vyverman, who is a frequent contributor to SUGI, and Nat Wooding, who has been an occasional contributor to SAS-L, for additional information and examples of DDE usage.

HTML

The author apologizes – due to poor time management, it was not possible to complete the research and subsequent writing necessary to do a quality job documenting the use of the INFILE and INPUT commands to process HTML. This section is targeted for completion in time to be included in the lecture component of this presentation at SUGI 31 in March 2006. The completed version of this section will be made available to interested attendees at this time. Any subsequent publication of this material will include the completed version of this material, as well.

ADDENDUM – INPUT function

When discussing the INPUT statement, someone often asks about the INPUT function. There is a little parallel between the INPUT statement and the INPUT function – very little. Both the statement and the function convert a value from one form to another using a specified INFORMAT. The INPUT function, however, does not read from an external file as the INPUT statement does. Its purpose is to convert a SAS variable into a different representation, based on an informat. It is typically used to convert a character value into a numeric value, but it can also be used to convert a character value into a different character value. Further discussion of this SAS feature is outside of the scope of this paper.

CONCLUSION

There are a number of ways to use INFILE and INPUT to introduce external data into the SAS System, just as there are number of different file types and formats that need to be read. It would be impossible to provide in-depth information on all of them in the limited space of this presentation; in fact a ½ day class could not cover them all! (I know that from personal experience – I've tried to do just that!) It is hoped that the material contained in this paper will serve to stimulate the curiosity of the reader, and that they will continue their education by researching the appropriate manuals and technical papers devoted to the specific topics discussed within this paper. Ultimately, however, it will be through real-life trial and error that true comprehension and retention of this knowledge will be attained.

REFERENCES / FOR FURTHER INFORMATION

Beatrous, Steve, and Clifford, Billy. (1998). "Sometimes You Get What You Want: I/O Enhancements for Version 7". *Proceedings of the Twenty-Third Annual SAS Users Group International Conference.* Cary, NC: SAS Institute, Inc.

Boling, John C. (1997). "SAS Data Views: A Virtual View of Data". *Proceedings of the Twenty-Second Annual SAS Users Group International Conference*. Cary, NC: SAS Institute, Inc.

Burlew, Michele M. (2002). *Reading External Data Files Using SAS®: Examples Handbook. Cary, NC.* SAS Institute, Inc.

Carpenter, Arthur L. and Payne, Tony (2005). "A Bit More on Job Security: Long Names and Other V8 Tips". *Proceedings of Alaska Day 2005 SAS Users Group Conference*. Tacoma, WA. PNWSUG.

Cody, Ronald (1998). "The INPUT Statement: Where It's @". *Proceedings of the Twenty-Third Annual SAS Users Group International Conference*. Cary, NC: SAS Institute, Inc.

DelGobbo, Vincent (2003). "Techniques for SAS® Enabling Microsoft® Office in a Cross-Platform Environment". *Proceedings of the Twenty-Seventh Annual SAS Users Group International Conference.* Cary, NC: SAS Institute, Inc.

Kuligowski, Andrew T. (2005). "Getting Data Into SAS: INFILE and INPUT". *Proceedings of the Eighteenth Annual NorthEast SAS Users Group Conference*. Cary, NC: SAS Institute, Inc.

Kuligowski, Andrew T. (2001). *Class Notes: Turning External Data into SAS Data.* Dunedin, FL. Self-published.

Kuligowski, Andrew T., and Roberts, Nancy (1997). "From There to Here: Getting Your Data Into the SAS System". *Proceedings of the Twenty-Second Annual SAS Users Group International Conference*. Cary, NC: SAS Institute, Inc.

Levine, Allison. (1997). "The What, When, Why, and How of PROC FORMAT". *Proceedings of the Tenth Annual NorthEast SAS Users Group Conference.* USA.

Microsoft Corporation (1999). *Microsoft Windows 2000 Professional Help Facility*. Redmond, WA: Microsoft Corporation.

Patton, Nancy K. (1998). "IN & OUT of CNTL with PROC FORMAT". *Proceedings of the Sixth Annual SouthEast SAS Users Group Conference*. Cary, NC: SAS Institute, Inc.

Riba, S. David (1996), *Course Notes: Connecting With Your Data.* Clearwater, FL: JADE Tech, Inc.

SAS Institute, Inc. (1993), *SAS Companion for the Microsoft Windows Environment.* Cary, NC: SAS Institute, Inc.

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

SAS Institute, Inc. (1999). *SAS Online Documentation, Version 8.* Cary, NC: SAS Institute, Inc.

SAS Institute, Inc. (2006). *SAS Online Documentation, Version 9.* Cary, NC: SAS Institute, Inc.

SAS Institute, Inc. (1990). *SAS Procedures Guide, Version 6, Third Edition.* Cary, NC: SAS Institute, Inc.

SAS Institute, Inc. (1994), *SAS Software: Abridged Reference, Version 6, First Edition.* Cary, NC: SAS Institute, Inc.

SAS Institute, Inc. (1999). SAS Technical Report 325, The SAS System and DDE. Cary, NC: SAS Institute, Inc.

SAS Institute, Inc. (1997). *Window by Window: Capture Your Data Using the SAS System.* Cary, NC: SAS Institute, Inc.

The author can be contacted via e-mail as follows:
    A_Kuligowski@msn.com
    Andy.Kuligowski@NielsenMedia.com