Paper BI-08-2015

Easy Come, Easy Go — Interactions between the DATA Step and External Files

Andrew T. Kuligowski, HSN

ABSTRACT

Chances are, your raw data was not created within the SAS® System. There is a good likelihood that your data may also need to be packaged and passed along to another non-SAS package.

This presentation will provide basic answers to two questions common to new SAS users:

- How do I get my data into SAS for analysis?
- How do I get my data out of SAS?

The focus for this presentation will be on two pairs of DATA step statements: INFILE / INPUT and FILE / PUT. We will discuss syntax and usage, citing various types of files as examples.

INTRODUCTION

The general processes of moving data from an external file to SAS, and from SAS to an external source are largely mirror images of each other. The commands necessary to perform these tasks share a common syntax, similar options, and even sound similar to each other. As such, this presentation will discuss the two *opposite* tasks in parallel with each other.

In order to trade data between a SAS session and an external source, the user must provide the answers to a couple of simple questions:

- Where is this external source / destination?
  - And, is there anything significant about this external data file?
- What does the data look like coming in / going out?

Once these questions have been answered and translated into valid SAS code, the data transfer can begin.

DEFINING THE EXTERNAL DATA SOURCE

There are two statements that will both define the external data file to the DATA step, and provide a few tidbits of information regarding its form. The **INFILE** statement handles this functionality when moving data into the DATA step. Going in the other direction, the **FILE** statement should be employed to write to an external file. (Note that the mirror image analogy does not apply to the actual terminology – the statement is not called "OUTFILE".) There are several ways that this connection can be made.

The most straightforward method is to hard-code the file name into the INFILE or FILE statement. This eliminates almost all potential for confusion when executing or debugging a routine, and it permits the routine to be self-documenting. The negative side of this approach is that it eliminates flexibility; the next time this routine is invoked, it will point to the same data file as the first time, unless the routine is altered. (Keep in mind that the managerial process / procedure to apply maintenance to a routine is trivial in some shops, but quite involved in others.)

```
DATA SAMPLE ;
  /* full file name -  under Windows */
  INFILE 'c:\sasconf\sasconf.dat';
```

Some flexibility can be recovered by passing a macro variable rather than a fixed value. With the most basic version of macro coding, the only "accomplishment" is that the hardcoding has been moved to a different portion of the program.

```
%LET FILENM = c:\sasconf\sasconf.dat;
DATA SAMPLE ;
  /* macro file name - under Windows */
  INFILE "&FILENM";
```

However, with a little extra creativity, the hardcoded macro variable in this example can be replaced with something that can provide some flexibility. The routine can use cues from the data being processed to build a unique dataset name, then load it into a macro variable so it can be passed to the INFILE / FILE statement. Also, please note the use of double-quotes in this sample code, rather than the single quotes that were used in the first example. Double-quotes permit the resolution of macro variables that are contained within, while single quotations do not; instead, they treat the contents as a literal – `&FILENM` in this instance – which will result in a syntax error upon compilation.

Additional flexibility can be obtained within the DATA step itself, without having to use macro variables. There is an option called **FILEVAR**, which will allow the value of an active variable to be used as the physical file name.

```
DATA SAMPLE ;
  RETAIN  fn  "SASCONF" ;
  File_in  = TRIM( fn ) || ".dat" ;
  File_out = TRIM( fn ) || ".out" ;
  INFILE dummy    FILEVAR=File_in ;
  FILE   dummy2   FILEVAR=File_out ;
```

In our example, the FILEVAR value is calculated in each iteration of the DATA step. Due to the design, it does not actually change its value from observation to observation. This is inefficient coding technique – one would normally use a RETAIN and something like a "`IF _N_ = 1`" loop to handle a situation like this. The example is coded in this way to show how the value *could* change from observation to observation, if the value of the "`fn`" variable used to build the FILEVAR is changed from iteration to iteration rather than being hard-coded to a constant.

Also notice the use of the word "dummy". This is our first look at a **File Reference**, or **fileref** for short. The INFILE and FILE statements do not actually use this File Reference when FILEVAR is in use, but the syntax of these commands requires that a file reference be present. We call the File References "`dummy`" and "`dummy2`" in our example to illustrate the fact that they are ignored by the routine, but any valid File Reference designation could be used. Please note, however, that each of the dummy File References must be unique in cases where more than one is used in a SAS DATA step.

Now that the concept is introduced, it is time to look at File References, also known as FILEREF. A File Reference is a short name which refers back to a full dataset name – a nickname for a file, in essence. Filerefs should be 8 characters in length or less. They should start with an alphabetic character, although letters, numeric values and select special characters can be used in positions 2 and beyond.

There are several methods to define a File Reference. One common method is to perform this action outside of SAS, to let the operating system handle it. In MVS, for example, it is easy to define a File Reference, as follows:

```
//SAMPDATA DD DSN=SAS.GLOBAL.FORUM.SAMPLE.DATA,DISP=SHR
```

(It is also easy to see how the File Reference acquired the alias "DDNAME", as each file is followed by the letters "DD" in its definition under MVS.) Other operating systems have their own parallel commands and procedures to accomplish the same purpose; the reader is directed to the appropriate *SAS Companion* document for their operating system of choice.

A common method to define File References and make external files available to the SAS System is the **FILENAME** statement. The simplest form of the statement is FILENAME <fileref> <filename>, such as:

```
FILENAME  confdata 'c:\sasconf\sasconf.dat';
```

This syntax is valid whether the file is to be used for input, output, or both.

There are several options available which are unique to a given operating system. The reader is again encouraged to research the appropriate *Companion* document for their particular operating system(s). Many of these options are used in the creation of a new file, containing pertinent information such as record length. (Many, perhaps most, of these options can also be included when defining an existing file as well as when creating a new one. This is not encouraged – the statement will work fine without the options being spelled out when pointing to an existing file. However, including the options with an erroneous value that conflicts with the way the file is defined can result in a needless error in the routine.)
There are some special arguments which can be passed into the FILENAME statement:

CLEAR	Removes a previously defined File Reference
LIST	Writes a description of a File Reference to the SASLOG

There is also a special pseudo-File Reference: **`_ALL_`**, which can be used in conjunction with either CLEAR or LIST. When present, it causes the requested action to be taken for ALL active File References, rather than having to use individual commands to specify each one separately.

The INFILE statement can also handle what are known as **aggregate File References**. This term originally referred to the Partition Dataset, or PDS, under MVS. It has subsequently been expanded to include the concept of a directory under Windows and UNIX. When dealing with an aggregate File Reference, the FILENAME statement would point to the entire aggregate File Reference, while the particular PDS member or dataset within the requested directory would be identified parentheses in the INFILE statement. (The File Extension for the requested file is assumed to be .DAT by default under Windows and UNIX.)

```
/* Example using Windows, defines*/
/*    c:\sasconf\sascon07.dat   */
FILENAME  confdata 'c:\sasconf';
DATA SAS_Conf07;
   INFILE  confdata(sascon07);
   /* other statements follow ...*/
```

```
/* Example using MVS, defines         */
/*    'userid.sasconf.data(sascon07)' */
FILENAME  confdata 'userid.sasconf.data';
DATA SAS_Conf07;
   INFILE  confdata(sascon07);
   /* other statements follow ...*/
```

The FILENAME statement is interpreted and executed outside of the DATA step, as has been illustrated by the examples up to this point. It is also possible to define a File Reference within the DATA step itself using a **FILENAME** function. The parameters for this function are similar to the ones available for the FILENAME statement. However, unlike the FILENAME statement, the FILENAME function is executed within the boundaries and control of a SAS DATA step. As such, constants must be enclosed in quotation marks, to ensure that the interpreter does not confuse them with variable names. (Of course, the SAS user can opt to use a variable name in lieu of a hard-coded constant, which increases the flexibility of this alternative!) Additionally, like all functions, FILENAME returns a value – in this case, it is the return code of the operation. A zero can be interpreted as a success, while a non-zero indicates some sort of failure.

```
DATA  _NULL_ ;
   Outfile = "c:\sasconf\sampdata.out";
   RetCd1 = FILENAME("confdata",'c:\sasconf\sasconf.dat');
   RetCd2 = FILENAME("outdata", Outfile);
```

The routine can be enhanced to monitor the return codes, and to take some appropriate action if they return a non-zero value indicating an error.

As an aside, there are three other functions which might be useful when attempting to allocate a File Reference.

- The **FILEREF** function accepts a File Reference as its only argument. A Return Code of 0 indicates that the File Reference is currently assigned. A positive value indicates that the File Reference is not currently assigned, while a negative value indicates that the File Reference is assigned but that the file it is referencing does not currently exist.
- The **FILEEXIST** function accepts a file name as its only argument. It returns a 1 to indicate that the external file exists, and a 0 to show that it does not.

```
DATA   _NULL_ ;
   IF  FILEEXIST("c:\sasconf\sasconf2.dat")  THEN
      RetCd  = FILENAME("confdat2", 'c:\sasconf\sasconf2.dat' );
   ELSE
      RetCd  = FILENAME("confdat2", 'c:\sasconf\sasconf.dat' );
   RetCd2 = FILEREF( "confdat2" ) ;
   IF      RetCd = 0  THEN
      PUT "Fileref confdat2 and its associated file exist";
   ELSE IF RetCd < 0  THEN
      PUT "Fileref confdat2 does not point to a valid file";
   ELSE /* RetCd > 0 */
      PUT "Fileref confdat2 does not exist";
RUN;
```

- **FEXIST** is a cross between the two other functions described above. It accepts a File Reference as its argument, then confirms or denies the existence of the external file defined by that File Reference. As above, a 1 is returned if the file exists, while 0 shows that the file does not.

It should be noted that there is a specialized File Reference that is used for input data: **DATALINES** (and its "alias" of **CARDS**, which dates back to the days when the data was actually stored on physical punch cards). This File Reference informs the SAS DATA step that the input can be found inside the code itself, immediately following the affected DATA step. The input data is separated from the rest of the routine by the DATALINES or CARDS command – in fact, SAS will automatically assume that the routine is looking for the DATALINES File Reference if the DATALINES statement is present; this is one case where the INFILE statement is actually optional. All subsequent lines in the routine are considered to be input data until a line containing a semicolon in position 1 is encountered. (**CARDS4** and **DATALINES4** are also available if the input data is expected to contain semicolons – in this case it requires four consecutive semicolons to terminate the input data and return to the routine.)

```
DATA SAS_Conf07;
   INFILE  DATALINES; /* stmt optional for DATALINES */
   INPUT  @  1  Start_Dt  MMDDYY8.
          @ 10  End_Dt    MMDDYY8.
          @ 19  ConfName  $CHAR16.
          @ 36  ConfLoc   $CHAR16. ;
DATALINES;
04/16/07 04/19/07 SAS Global Forum Orlando       FL
06/03/07 06/06/07 PharmaSUG        Denver        CO
09/16/07 09/18/07 PNWSUG           Seattle       WA
09/30/07 10/02/07 SCSUG            Austin        TX
10/17/07 10/19/07 WUSS             San Francisco CA
10/28/07 10/30/07 MWSUG            Des Moines    IA
11/04/07 11/06/07 SESUG            Hilton Head   SC
11/11/07 11/14/07 NESUG            Baltimore     MD
;
```

By default, the input to DATALINES is assumed to be "card image", that is 80 bytes in length. This is driven by a system option, **CARDIMAGE**. It can be overridden by changing the option to **NOCARDIMAGE**. Normally, CARDIMAGE is associated with MVS and CMS (for those who may still be using that mainframe operating system), while operating systems such as Windows and UNIX that are traditionally associated with servers and desktop devices are better served with the NOCARDIMAGE option.

For the record, despite claims to the contrary, CARDS is not a 100% clone of DATALINES. The difference comes through when attempting to use this specialized File Reference as an output destination. The SAS System will block an attempt to define "`FILE CARDS`", but *will* accept "`FILE DATALINES`" and write the output to the SASLOG.

```
18   DATA  _NULL_;
19      SET SAS_Conf07;
20      FILE  CARDS;
NOTE: The file CARDS cannot be opened
      for UPDATE processing.
21      PUT  _ALL_ ;
22   RUN;
NOTE: The SAS System stopped processing
      this step because of errors.
```

```
62   DATA  _NULL_;
63      SET SAS_Conf07;
64      FILE  DATALINES;
65      PUT  _N_= ConfName ;
66   RUN;
_N_=1 SAS Global Forum
… …     … …
_N_=8 NESUG
NOTE: There were 8 observations read
      from the data set WORK.SAS_CONF07.
```

DESCRIBING THE APPEARANCE OF THE DATA

Once the physical location of the external data is made known to SAS, the next step is to describe what it looks like. Most of the methods to do this are associated with the INPUT and PUT statements. As one might expect, INPUT is associated with incoming data, while PUT describes outgoing data. Unlike the FILE / OUTFILE discussion earlier, there is an OUTPUT statement in SAS. However, this is used for writing data to a SAS dataset, not to an external file.

There are five types of ways that INPUT can be used to describe incoming data. (As might be expected, each has a parallel that corresponds to PUT. In an attempt to maintain some semblance of brevity, this presentation will emphasize the alternatives using INPUT, trusting the reader to apply the concepts to the PUT statement.)

List Input

The first alternative is the simplest, and is called **list input**. In list input, each variable is listed in the INPUT statement without any further descriptions, with the exception of a "**$**" following variables that are to be defined as character. Upon execution, the DATA step will read each line of input data individually, switching from variable to variable when it encounters a delimiter. (By default, the delimiter is considered to be a blank. We will discuss alternatives later in this presentation.)

There are potential problems with this simple approach. To begin with, the length of character variables is assumed to be 8; this becomes a problem when a value has a 9th character. Secondly, since the default delimiter between fields is a blank, this approach will not work for fields that might contain one or more embedded blanks. This can be problematic when reading such common data as names and addresses! Furthermore, there is no allowance for any specialized formatting, such as dates.

As one might expect after even a brief exposure to computer programming of any kind, each of the problems above has multiple potential solutions. One way to solve the problems described above is to define the variables in the DATA step prior to executing the INPUT statement. LENGTH, INFORMAT, and FORMAT statements will easily override the SAS defaults described earlier.

Unfortunately, these solutions will not resolve the issue with the blank delimiter. It might be possible to use a *format modifier* to assist in situations like this – this would result in our approach being redesignated as *modified list input*. (I have yet to meet the programmer who would be so intent to use a concept like "list input" that they decline useful alternatives to help resolve a problem!) The *ampersand* (**&**) is a special format modifier which notifies the INPUT statement that character values in the data might contain embedded blanks. However, the ampersand's usefulness is limited in many situations. It doesn't help when the data contains two or more consecutive embedded blanks; the second is considered to be the delimiter. Conversely, it can become confused when two distinct fields are separated by only a single blank delimiter – it will continue to read that new field's value into the old field.

It is still possible to write a routine that works within the limitations described above.

```
/* Ex A – "Straight" List Input – WILL NOT WORK */
DATA SAS_Conf07;


   INFILE  DATALINES ;
   INPUT  Start_Dt
          End_Dt
          ConfName  $
          ConfLoc   $ ;


DATALINES;
04/16/07 04/19/07 SAS Global Forum Orlando        FL
06/03/07 06/06/07 PharmaSUG        Denver         CO
09/16/07 09/18/07 PNWSUG           Seattle        WA
09/30/07 10/02/07 SCSUG            Austin         TX
10/17/07 10/19/07 WUSS             San Francisco  CA
10/28/07 10/30/07 MWSUG            Des Moines     IA
11/04/07 11/06/07 SESUG            Hilton Head    SC
11/11/07 11/14/07 NESUG            Baltimore      MD
;
```

```
/* Ex B – Modified List Input with corrections */
DATA SAS_Conf07;
   LENGTH  ConfName $ 16  ConfLoc $ 16. ;
   INFORMAT Start_Dt End_Dt  mmddyy8. ;
   INFILE  DATALINES ;
   INPUT  Start_Dt
          End_Dt
          ConfName  $ &
          ConfLoc   $ & ;
   ConfName = TRANSLATE( ConfName, ' ', '-' );
   ConfLoc  = TRANSLATE( ConfLoc,  ' ', '-' );
DATALINES;
04/16/07 04/19/07 SAS Global Forum  Orlando-------FL
06/03/07 06/06/07 PharmaSUG         Denver--------CO
09/16/07 09/18/07 PNWSUG            Seattle-------WA
09/30/07 10/02/07 SCSUG             Austin--------TX
10/17/07 10/19/07 WUSS              San Francisco-CA
10/28/07 10/30/07 MWSUG             Des Moines----IA
11/04/07 11/06/07 SESUG             Hilton Head---SC
11/11/07 11/14/07 NESUG             Baltimore-----MD
;
```

The modified version of the routine, called "Ex B", contains a few simple methods to get around the limitations of basic list input. However, it also contains what should be considered a mortal sin in the world of computing – it alters the input data for the convenience of the inputting routine. (In this example, the multiple embedded blanks between the city and state names in **`ConfLoc`** were replaced with a dash; the dash was subsequently changed back to to a blank by a TRANSLATE function. A second change is less obvious to the naked eye; the **`ConfLoc`** values were moved one position to the right to ensure that there are always 2 or more blanks between it **`ConfName.`**)

Before leaving the subject altogether, here is a list of format modifiers and their purpose:

&	Ampersand	Permit single embedded blanks in character variables.
:	Colon	Ignore its default 8-character maximum on character variables.
?	Question Mark	Bypass error processing should invalid data be expected and acceptable to the application - for example, attempting to read a character field into a numeric variable. Suppress the "Invalid Data" message, allowing the other indications of an error to be dealt with normally.
??	Double Question Mark	Bypass error processing AND conceal every indication of an error. The "Invalid Data" message and the echoing of the input line containing the bad data are omitted from the SASLOG, and the _ERROR_ variable is not set to "true" (numeric 1).
~	Tilde	Read delimiters within quoted character values as characters, and retains the quotation marks.

The use of the Question Mark and Double Question Mark format modifiers is strongly discouraged by this author. They do not discriminate when processing errors – or rather, when *not* processing errors. The error that they ignore may *not* be the one that the coder was anticipating when they utilized the Question Mark(s) format modifier. This could result in an error slipping through processing.

Formatted Input

The problems associated with list input are easily overcome with **formatted** input. Formatted input is similar to list input, except that each variable has an informat or a format associated with it – informats are associated with input data, while formats are tied to output data. Formats allow the INPUT statement to avoid the problems described under list input – contending with default variable lengths, and input formats.

There are five categories of informats and formats available in the SAS System: *character, numeric, date/time, column-binary,* and *user-defined*. (Date/time is sometimes considered to be a specialized subset of numeric, since both deal with numeric data. However, most sources believe that date/time is so specialized as to merit a separate category; the folks who write the documentation that accompanies the SAS System are included in that group!)

This presentation will make use of a few different informats and formats in its examples – a detailed list and discussion of them will be left to other presentations that focus specifically on that topic. It should also be noted that the final category listed above, user-defined, will actually contain formats from each of the other categories! The term “user defined” actually refers to any informat or format that is not provided “in the box” with installation of the SAS System. The good folks in Cary realize that each industry – indeed, each individual site or even user – will have specific data representations which it needs to read in and/or write out. To accommodate these specialized needs, the SAS System provides a utility to create and store customized informats and formats. The interested reader is encouraged to refer to the section on **PROC FORMAT** in the *SAS Procedures Guide*, or to several excellent papers on the topic.

There are several methods to tie an informat or a format to a variable using SAS. The most straightforward is to use an INFORMAT or FORMAT statement, listing each variable with the informat or format to be associated with it. The ATTRIB statement can be used in much the same way, with INFORMAT= and FORMAT= being two of the attributes which can be associated with a variable. These are considered permanent techniques – the variable is permanently tied to its associated INFORMAT and/or FORMAT, at least for as long as the variable exists.

It is not necessary to invoke an external statement to tie a format to a variable. Using formatted input, each variable can be followed by the proper informat. These informats are considered temporary; they must be respecified each time the variable is referenced. However, this can also be used to ones advantage – a permanent format can be associated with a variable using one of the techniques specified, but that can be overridden by including a different format with that variable on the INPUT or PUT statement.

Formatted input does introduce a situation that was not an issue with list input – blank padding. List input by default stops reading one variable and begins the next when it counters a delimiter – which, by default is one or more blank characters. Formatted input, conversely, treats blanks as any other data. If the first variable starts in position 1 and has a length of 8, then formatted input is going to look for the next variable in position 9 – no delimiter, no padding.

Fortunately, formatted input provides another tool which can be used to overcome this “limitation” – **column pointers**. Once a variable is read, the pointer can be repositioned forwards or backwards on the line prior to reading the next variable.

There are two categories of column pointers. **Relative column pointers** are represented by a “plus” sign **+**, and indicate that the column pointer should be moved a specified number of positions to the right. It can also move the column pointer to the left by providing a negative number – please note that the plus sign representing the column pointer must be separated from the negative sign associated with the position with parentheses. **Absolute column pointers** are represented by the “at” sign @. The at sign causes the column pointer to be moved to the specific position referenced on the file.

In both cases, the flexibility can be further increased by using a variable as pointer control, rather than a constant. SAS will move the column pointer to the appropriate absolute or relative position, according to whatever value is contained in the variable. This value can even change from iteration to iteration of the DATA step, or a numeric expression can be provided that will be evaluated prior to each execution – this can provide even more flexibility!

File: 'c:\sasconf\sascon07.dat'

```
04/16/07 04/19/07 SAS Global Forum Orlando       FL
06/03/07 06/06/07 PharmaSUG        Denver        CO
09/16/07 09/18/07 PNWSUG           Seattle       WA
09/30/07 10/02/07 SCSUG            Austin        TX
10/17/07 10/19/07 WUSS             San Francisco CA
10/28/07 10/30/07 MWSUG            Des Moines    IA
11/04/07 11/06/07 SESUG            Hilton Head   SC
11/11/07 11/14/07 NESUG            Baltimore     MD
```

```
/* Relative Column Pointers           */
/* Simple coming in, complex going out */
DATA SAS_Conf07;
   RETAIN  pt_loc 18 ;
   INFILE  'c:\sasconf\sascon07.dat' ;
   INPUT   Start_Dt  mmddyy8. +1
           End_Dt    mmddyy8. +1
           ConfName  $char16. +1
           ConfLoc   $char16. ;
   FILE  'c:\sasconf\sascon07a.dat' ;
   PUT   Start_Dt  date7.    +10
         ConfName  $char16.  +(-25)
         End_Dt    date7.    +(pt_loc)
         ConfLoc   $char16.  ;
RUN;
```

```
/* Absolute Column Pointers           */
/* Simple coming in, complex going out */
DATA SAS_Conf07;
   RETAIN  pt_loc 36 ;
   INFILE  'c:\sasconf\sascon07.dat' ;
   INPUT   @  1 Start_Dt  mmddyy8.
           @ 10 End_Dt    mmddyy8.
           @ 19 ConfName  $char16.
           @ 36 ConfLoc   $char16. ;
   FILE  'c:\sasconf\sascon07b.dat' ;
   PUT   @  1       Start_Dt  mmddyy8.
         @ 19       ConfName  $char16.
         @ 10       End_Dt    mmddyy8.
         @ pt_loc   ConfLoc   $char16. ;
RUN;
```

Speaking of flexibility, there is an additional column pointer option available on the INPUT statement – SAS will accept a character string instead of a number as an absolute column pointer! With character strings, the input line will be parsed for the presence of the requested string. When it is found, the INPUT statement will move to the next position on the line and continue the process of assigning values to variables. The INPUT statement will accept either a variable or a hard-coded string, just like with the more traditional numeric pointers.

```
DATA SAS_Conf07;
   RETAIN  pt_loc "LOC:" ;
   INFILE  DATALINES ;
   INPUT   @  1      Start_Dt  mmddyy8.
           @ '07 '   ConfName  $char16.
           @ 10      End_Dt    mmddyy8.
           @ pt_loc  ConfLoc   $char16. ;
DATALINES;
04/16/07 04/19/07 SAS Global Forum LOC:Orlando      FL
06/03/07 06/06/07 PharmaSUG        LOC:Denver       CO
09/16/07 09/18/07 PNWSUG           LOC:Seattle      WA
09/30/07 10/02/07 SCSUG            LOC:Austin       TX
10/17/07 10/19/07 WUSS             LOC:San Francisco CA
10/28/07 10/30/07 MWSUG            LOC:Des Moines   IA
11/04/07 11/06/07 SESUG            LOC:Hilton Head  SC
11/11/07 11/14/07 NESUG            LOC:Baltimore    MD
;
```

In the example, it should be noted that the column pointer **1** is a numeric value, while **'07 '** is being treated as a character – note the quotation marks. (The routine is looking for what the reader will recognize as the year portion in the End_Dt variable.) Also notice that we include a trailing blank in the character string so that that the blank padding between fields also bypassed when reading the variable. (This could be handled by using the $ informat instead of $CHAR, as $ ignores leading blanks while $CHAR processes and keeps them.) Our routine is even able to include the End_Dt field, since the SAS input statement does not require variables to be listed in order – column pointers can move either to the right or the left.

It would be expected that using a pointer with a character string only works during INPUT. This is true – almost. If you attempt to use a character string as an absolute column pointer, you will get the following error message in your SASLOG:

```
ERROR: The @'CHARACTER_STRING' or @CHARACTER_VARIABLE specification is
       valid on the PUT statement only in conjunction with the FILE ODS
       option.  The execution of the DATA STEP is being terminated.
```

There are several excellent papers, as well as some books available via SASPress, that will cover the topic of ODS should the reader want to further explore this specialized exception case.

Just as the column pointer permits the INPUT and PUT statements to move forward and backward on the current input line, **line pointers** can be used to read and write multiple input lines. As with column pointers, there are absolute and relative line pointers.

An **absolute line pointer** is denoted by a pound sign, "**#**" on the input statement. It can accept a numeric constant or a numeric variable as a parameter. It can also accept a numeric expression, provided the expression is surrounded by parentheses. Unlike absolute column pointers, it cannot accept a character string as a parameter. The **relative line pointer** is denoted by a slash "**/**". It cannot accept parameters; one slash causes the line pointer to move down one line. The only way to move down multiple lines is to specify multiple slashes. As the relative line pointer cannot accept parameters, there is no way to reverse the direction and move upwards, other than to revert to the use of an absolute line pointer.

```
DATA SAS_Conf07;
   RETAIN  line_loc 2 ;
   INFILE  DATALINES ;
   INPUT  #1             Start_Dt   mmddyy8.
          #line_loc      End_Dt     mmddyy8. +1
                         ConfName   $char16.
          #(line_loc+1)  ConfLoc    $char16. ;
   FILE 'c:\sasconf\conftbl.dat' ;
   PUT    @  1 ConfName  $char16.
          @ 21 Start_Dt  date7.
               " to"
        / @  1 ConfLoc   $char16.
          @ 21 End_Dt    date7.
        /      28*'-';
DATALINES;
04/16/07
04/19/07 SAS Global Forum
Orlando       FL
06/03/07
06/06/07 PharmaSUG
Denver        CO
… …    … …
11/11/07
11/14/07 NESUG
Baltimore     MD
;
```

```
File: 'c:\sasconf\conftbl.dat'
SAS Global Forum    16APR07 to
Orlando        FL   19APR07
----------------------------
PharmaSUG           03JUN07 to
Denver         CO   06JUN07
----------------------------
PNWSUG              16SEP07 to
Seattle        WA   18SEP07
----------------------------
SCSUG               30SEP07 to
Austin         TX   02OCT07
----------------------------
WUSS                17OCT07 to
San Francisco CA    19OCT07
----------------------------
MWSUG               28OCT07 to
Des Moines     IA   30OCT07
----------------------------
SESUG               04NOV07 to
Hilton Head    SC   06NOV07
----------------------------
NESUG               11NOV07 to
Baltimore      MD   14NOV07
----------------------------
```

Column Input

Column input is almost – *almost* – like a cross between list and formatted input. Like list input, the variables are individually listed, while the positions of the variables in the record are explicitly defined as with formatted input. However, instead of using the column and line pointers that are employed with formatted input, column input specifies the first and last position of each field in the record.

```
DATA SAS_Conf07;
   INFILE  'c:\sasconf\sascon07.dat' ;
   INPUT   Start_DtC $  1 -  8
           End_DtC   $ 10 - 17
           ConfName  $ 19 - 34
           ConfLoc   $ 36 - 51 ;
   Start_Dt = INPUT( Start_DtC, mmddyy8. ) ;
   End_Dt   = INPUT( End_DtC,   mmddyy8. ) ;
RUN;
```

Column input is appropriate when the data to be processed is neatly aligned in a tabular format. It does not have the flexibility to shift positions dynamically as do the column pointer options with formatted input – column positions must be hardcoded in the routine. (It is possible to use macro variables in lieu of constants in order to recover some flexibility. However, the coding necessary for this alternative is often much more complicated than to simply use formatted input in the first place.) Nor is column input able to deal with the simple "just get the next set of bytes and assign them to a variable" approach which is the philosophy of list input. Further, it cannot deal with anything other than straight numeric or character text – things like date/time variables require additional logic to transform them into a useful form. This author does not normally code with this technique; it must be understood, however, in order to apply maintenance to code written by others who may not share the same preferences.

Named Input

The rarest style of input data is **named input**. The data read with this technique must be in the form *fieldname=value*. SAS documentation used to state that this technique could not be combined with other input styles in the same INPUT statement. That has been proven false and has been corrected; other styles can be used, providing they occur before the first instance of named input. Once the first *fieldname=* is encountered on the INPUT statement, all other variables on the statement must follow that same pattern.

Named input has one big advantage over other forms of input – the variables on the INPUT statement do not need to be in the same order as they occur in the actual data. A second advantage is that variables do not necessarily even have to be listed on the INPUT statement in order to be read in using named input. Any variable previously defined, such as with a LENGTH or INFORMAT statement, will also be identified and processed.

One use for this specialized technique is to accept input parameters into the routine. Note that the routine looks for all variables defined by the routine when using named input, not just the ones explicitly specified on the INPUT statement. In the example below, ProcessDt is not listed on the INPUT statement, but it is processed anyway because it is defined with an INFORMAT statement.

```
File: 'c:\sasconf\param.dat'
PROD  ProcessDt=12/02/06  Analyst=ATK
```

```
DATA Params;
   INFORMAT  ProcessDt  MMDDYY8.;
   INFILE  'c:\sasconf\param.dat';
   INPUT  @ 1 RunType     $CHAR4.
              Analyst=    $CHAR3. ;
RUN;
```

```
DATA  _NULL_;
   SET Params;
   FILE  LOG;
   PUT  RunType=
        ProcessDt= DATE9. Analyst= ;
RUN;
```

```
RunType=PROD ProcessDt=02DEC2006 Analyst=ATK
```

Named format is a little more common on the PUT statement. Its primary use is in debugging and audit messages that are often inserted into code by coders who want to track their results manually. Each variable must be explicitly specified when using this technique for output; SAS will not make assumptions as it does when reading Named Input. This additional piece of intelligence can be specified by using "**PUT _ALL_;**" This will echo everything contained in the SAS Program Data Vector in a *fieldname=value* pair format, including those variables specifically marked as DROP and temporary SAS variables that are not included when outputting the record.

Null Input

The SAS documentation identifies 4 different types of INPUTs and PUTs, which have been discussed in this document. There is a pseudo- 5th type – **null input**. Null input does not have any column pointers, or line pointers – or even any variables. It is a standalone INPUT statement. The utility of such a beast may not seem obvious at first – after all, what good is a statement which does not perform any action? In reality, null input is extremely useful; it is used when conditionally reading or writing part of a line of data.

This paper covered the topic of the "at" sign "@", the absolute column pointer. What was not mentioned at that time is that there are two different flavors of the "at" sign. The first precedes variable names, and specifies the location where they can be read from or written to. The second occurs at the end of an INPUT statement, and is referred to as the "**trailing** @". By default, the line pointer is automatically moved to the next line at the conclusion of the INPUT or PUT statement, and the column pointer is shifted to the first position of that line.. The trailing @ overrides those defaults, and causes the line and column pointer to remain right where they are.

This technique is useful when using a loop to read or write multiple items from a line. The code can be structured to read a single variable, hold the pointer where it is, and read the next variable. Once the last variable in the set is read in, the line and column pointer can be released to move to the next line. The null input statement is the tool used to perform that “release”. It does not process any variables, and it does not take any explicit action with the column or line pointer. As such, it triggers the default action of moving the pointers to the first position of the next line.

The trailing @ can also be used for conditional processing. Let us assume that we only want to process some of the records in an input file, based on the value of the first field of each record. This can be accomplished with an INPUT statement specifying that field, followed by the Trailing @. An IF statement can be employed to evaluate the field. If it passes the specified conditions, a second INPUT statement can read the rest of the line. If it fails, a NULL input will release the “hold” on the column and line pointers, so that they begin to read the next row of the data.

It must be noted that the “hold” on the column and line pointers is released at the end of the DATA step; that is, when the RUN (or DATALINES) statement is encountered. Often, this is advantageous – this feature can be used to eliminate the need for the null input statement in many situations. However, in some cases, this is an undesirable limitation, as the logic is more complex and requires multiple iterations per decision. In these cases, there is a second form of the Trailing @ - the **Double Trailing** @@. The only difference between the two options is that the hold on the column and line pointer is not released at the end of the DATA step iteration by the Double Trailing @@.

```
DATA SAS_Conf07;
   ARRAY ConfDt (2)  Start_Dt  End_Dt ;
   INFILE  'c:\sasconf\sascon07.dat' ;
   INPUT  @ 19   ConfName  $char16.
          @ 36   ConfLoc   $char16.
          @  1   @ ;
   DO  i = 1 TO 2 ;
      INPUT  ConfDt(i)  mmddyy8. +1 @ ;
   END ;
   INPUT ;
   PUT  @  1   ConfName  $CHAR16.
        @ 18   Start_Dt  mmddyy8.
        @ 27   End_Dt    mmddyy8. @ ;
   IF  MONTH( Start_Dt) ^= MONTH( End_Dt )  THEN
       PUT ' Span Month' ;
   ELSE  PUT ;
RUN ;
```

Quick transfer from input to output

Many of the examples in this presentation deal with both external input and output in the same DATA step. There is a temporary SAS variable that can be used to facilitate this process: **_INFILE_**. A null input will populate this field, which will enable a quick transfer between the two files.

```
193  DATA _NULL_;
194     INFILE  DATALINES ;
195     INPUT ;
196     FILE  PRINT ;
197     PUT  _INFILE_;
198  DATALINES;

NOTE: 8 lines were written to file PRINT.
```

Of course, it is probable that the user will want to perform *some* sort of action prior to the output. Perhaps the file should only be written out if it meets some sort of condition, or .maybe a *simple* data transformation is desired. These are also possible using _INFILE_, along with some other features of the SAS Data step – all without actually having to specify the record format of either the input or output data.

```
DATA _NULL_;
  INFILE  DATALINES ;
  INPUT @ 1 Start_Dt mmddyy8.
        @10 End_Dt   mmddyy8.;
  IF  MONTH( Start_Dt ) = Month( End_Dt );
  IF  Start_Dt <= '17Apr07'D  THEN
     _INFILE_ = "HISTORY-" || SUBSTR( _INFILE_, 9 );
  FILE  PRINT ;
  PUT   _INFILE_;
DATALINES;
04/16/07 04/19/07 notSUGI   Orlando
06/03/07 06/06/07 PharmaSUG Denver
09/16/07 09/18/07 PNWSUG    Seattle
09/30/07 10/02/07 SCSUG     Austin
10/17/07 10/19/07 WUSS      San Francisco
10/28/07 10/30/07 MWSUG     Des Moines
11/04/07 11/06/07 SESUG     Hilton Head
11/11/07 11/14/07 NESUG     Baltimore
;
```

```
HISTORY- 04/19/07 notSUGI   Orlando
06/03/07 06/06/07 PharmaSUG Denver
09/16/07 09/18/07 PNWSUG    Seattle
10/17/07 10/19/07 WUSS      San Francisco
10/28/07 10/30/07 MWSUG     Des Moines
11/04/07 11/06/07 SESUG     Hilton Head
11/11/07 11/14/07 NESUG     Baltimore
```

In the example, the only fields read were the two dates. Both were used in a "keep or drop" decision, which resulted in one record being discarded and the rest of them written to the output. One of the date fields was also used to determine if a simple transformation should occur prior to writing a record to our output – the first record in our file was affected by this change. However, it was not necessary for us to even have a file format prior to writing this routine; all we needed was the location and format of the two fields that were part of the decision-making logic in the routine.

Which technique is best?

There is a natural tendency to compare these techniques. Named input and null input are obviously specialized cases, so they would fall outside of any ranking. The author has already discussed his predisposition against using column format in most instances. This leaves the list input and formatted input techniques. After describing both and walking through a few examples, the reader may be tempted to conclude that the formatted method is far more flexible and superior to the more simplistic List approach. However, this supposition is skewed; a casual examination of the sample code snippets and test files in this presentation will show that all of the input and output to this point fall into a nice, even, columnar pattern. Let us look at another type of file – one with variable length records and fields that do not always start and end in the same position on each line – to see that the real answer to the question of "which technique is better" is "it depends".

CSV Files and other Delimiter Separated Data

Almost everyone who has used a personal computer for anything besides surfing the Internet has had exposure to a spreadsheet package like Microsoft Excel. Spreadsheets provide a quick, easy to understand method for even the most casual computer user to enter and analyze data. The most common mechanism to share data between a spreadsheet and an external software package or routine is using the **comma separated value** file, also known as **CSV**.

Comma separated value files look exactly like the name implies. They contain one or more values per line, with each value separated by a comma. Each line in the file corresponds to a column in the spreadsheet, while each value on the line corresponds to an individual cell. In some cases, character values are surrounded by quotation marks while in others they are not. Numeric values are also written out in a straightforward manner. The CSV file only contains values; a cell that contains an equation will see the results of that formula output to the CSV file.

SAS facilitates the processing of CSV files via a special keyword on the INFILE and FILE statements, **DSD** stands for **Delimiter Separated Data**, using "Delimiter" in case a value other than a comma is being used to separate values. The DSD keyword causes a number of default behaviors of the INFILE and FILE commands to be overridden.

- The default delimiter is changed from blank to comma. (As an aside, this default, whether blank or comma, can always be overridden with the **DELIMITER=** option.)
- 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 as part of the variable's value, if desired, by using the tilde format modifier (~) as described earlier in this paper.
- Multiple consecutive commas (or other delimiter as defined) are assumed to represent missing values. Under normal circumstances, the second and subsequent consecutive delimiters are ignored and treated as a single delimiter.

Comma Separated Value files are variable length files; that is, each line of data may contain a different number of characters. Further, unlike the earlier examples in this presentation, each individual value may start in a different position from line to line – one cannot write code expecting a strict columnar table! Furthermore, specifying a format on the INPUT or PUT statement will cause the DATA step to read in or write out *exactly* that many characters – the length of the formats will take precedence over the presence of comma (or other specified) delimiters, as will the necessity to control the column pointer after the value has been read. CSV files do not work that way; each value contains only the number of characters in the value without any leading or trailing blanks to force the file into a column form. Given this features of the CSV file, any attempt to use a strict regiment of column pointers and formats – i.e. using formatted input or put - will result in failure!

It is still possible to override default lengths and formats when processing CSV files. As discussed earlier, this can be done using the LENGTH, INFORMAT, FORMAT, and/or ATTRIB statements in the DATA step rather than specifying those attributes on the INPUT or PUT statement.

File: 'c:\sasconf\sascon07.csv'

```
04/16/07,04/19/07,"SAS Global Forum","Orlando FL"
06/03/07,06/06/07,"PharmaSUG","Denver CO"
09/16/07,09/18/07,"PNWSUG","Seattle WA"
09/30/07,10/02/07,"SCSUG","Austin TX"
10/17/07,10/19/07,"WUSS","San Francisco CA"
10/28/07,10/30/07,"MWSUG","Des Moines IA"
11/04/07,11/06/07,"SESUG","Hilton Head SC"
11/11/07,11/14/07,"NESUG","Baltimore MD"
```

```
39 /* WRONG WAY to handle CSV File */
17 DATA SAS_Conf07;
18   INFILE  'c:\sasconf\sascon07.csv' DSD;
19   INPUT  Start_Dt mmddyy8.
20          End_Dt   mmddyy8.
21          ConfName $CHAR16.
22          ConfLoc  $CHAR16.;
23   FILE  'c:\sasconf\testout2.csv' DSD;
24   PUT  ConfName $CHAR16.
25        ConfLoc  $CHAR16.
26        Start_Dt date7.
27        End_Dt   date7.;
28 RUN;

NOTE: Invalid data for End_Dt in line 1 9-16.
RULE: ----+----1----+----2----+----3----+---4--
      04/16/07,04/19/07,"SAS Global Forum",O …
Start_Dt=16APR2007 End_Dt=.
ConfName=7,"SAS Global Fo
ConfLoc=rum","Orlando FL _ERROR_=1 _N_=1
NOTE: Invalid data for End_Dt in line 2 9-16.
      06/03/07,06/06/07,"PharmaSUG","Denver CO"
Start_Dt=03JUN2007 End_Dt=.
ConfName=7,"PharmaSUG","D
ConfLoc=enver CO" _ERROR_=1 _N_=2
        etc. etc. etc.
```

'c:\sasconf\testout1.csv'

```
7,"SAS Global Forum","Orlando FL16APR07      .
7,"PharmaSUG","Denver CO"       03JUN07      .
7,"PNWSUG","Seattle WA"         16SEP07      .
7,"SCSUG","Austin TX"           30SEP07      .
7,"WUSS","San Francisco CA"     17OCT07      .
7,"MWSUG","Des Moines IA"       28OCT07      .
7,"SESUG","Hilton Head SC"      04NOV07      .
7,"NESUG","Baltimore MD"        11NOV07      .
```

It is obvious from glancing at the output file created by this routine that it is a mess. A careful analysis will reveal that the routine attempted to read "**`End_Dt`**" immediately after "**`Start_Dt`**" was concluded, without consideration for the comma delimiter. This resulted in an invalid date value that began with a comma, and ended with a one-digit year, a "0". The final digit in the year – the "7" – and the next comma delimiter were treated as the first two characters in "**`ConfName`**". "**`ConfName`**" and "**`ConfLoc`**" flow together in the output file, so they do not look erroneous. A quick look at the SASLOG shows that the intended values of the two fields are actually intertwined together rather than separated at the proper point. It also reveals that the comma was not written out as a delimiter, but rather was erroneously treated as part of the value of one or the other of the fields, then simply echoed back.

```
/* CORRECT WAY to handle CSV File */
DATA SAS_Conf07;
  INFORMAT  Start_Dt  End_Dt   mmddyy8.
            ConfName  ConfLoc  $CHAR16.;
  FORMAT  Start_Dt  End_Dt   date7. ;
  INFILE  'c:\sasconf\sascon07.csv' DSD;
  INPUT  Start_Dt
         End_Dt
         ConfName
         ConfLoc ;
  FILE  'c:\sasconf\testout2.csv' DSD;
  PUT  ConfName
       ConfLoc
       Start_Dt
       End_Dt  ;
RUN;
```

`'c:\sasconf\testout2.csv'`

```
SAS Global Forum,Orlando FL,16APR07,19APR07
PharmaSUG,Denver CO,03JUN07,06JUN07
PNWSUG,Seattle WA,16SEP07,18SEP07
SCSUG,Austin TX,30SEP07,02OCT07
WUSS,San Francisco CA,17OCT07,19OCT07
MWSUG,Des Moines IA,28OCT07,30OCT07
SESUG,Hilton Head SC,04NOV07,06NOV07
NESUG,Baltimore MD,11NOV07,14NOV07
```

It also takes only a glance at the second example to see that it worked as intended. Fields were read in and written out, no matter how many characters they happened to contain. Comma delimiters were respected in both input and output.

Of course, this method of interfacing between the spreadsheet and SAS requires that each of them "converse" through the common intermediary step of the CSV file. It is possible to communicate directly between SAS and MS Excel using DDE. Rumors of the impending demise of DDE have been around almost as long as DDE has around. It continues to be available through the various releases of Windows, however, including through the newly offered Windows Vista. A reasonable discussion of DDE usage would take up an entire presentation by itself; several are available in the past Proceedings of what was then called SUGI, as well as from the assorted U.S. Regional User Group conferences.

INPUT request longer than the INPUT data

It is inevitable that a SAS coder will encounter a situation where the INPUT statement requests more information than is available on the line being read in. It could be an error in specifications, or in coding, or simply that the wrong data is being fed into the routine. There are several alternatives to deal with this situation (besides rewriting the code and/or getting different input data!) All of them are options that can be overridden on the INFILE statement.

By default, SAS will use the FLOWOVER option. When it encounters the end of the line in the middle of a read, it simply jumps to the next line of input data and keeps reading. The user is notified that this occurred by the infamous message in the SASLOG:

```
NOTE: SAS went to a new line when INPUT statement reached past the
      end of a line.
```

The SCANOVER option is similar to FLOWOVER. It is only used when an INPUT statement is using the "@ "<char-string>" option, and controls whether or not SAS should go to the next line to search out the requested character string:

It is not always a good idea to automatically jump to the next line if the routine is expecting additional data on the current one. Some coders base their design on this assumption, but others, including this author, believe that the code should be written in such a way to avoid messages like this. After all, one may not know that the message was triggered by a condition other than the one which is normally expected!

TRUNCOVER and MISSOVER are useful alternatives to FLOWOVER. In both, SAS does not move to the next line of input data should the INPUT statement request more data than exists on the current line. Instead, it stops, and sets the remainder of the requested variables on the line to missing. The difference between the two options is how it handles the situation of a line ending in the middle of populating a variable. MISSOVER sets that variable to missing along with the others, while TRUNCOVER "settles for what it can get" and populates the variable with the last X bytes of data that were read in from the line.

There is another alternative for those who prefer more extreme preventative measures – STOPOVER. This option basically ceases execution of the DATA step upon occurrence of the "end of data on a line" condition, and trips the error flag.

ADDENDUM – INPUT and PUT functions

When discussing the INPUT and PUT statements, someone often asks about the INPUT and PUT functions. There is a little parallel between the INPUT statement and the INPUT function, and the PUT statement and PUT function – very little. Both the statements and the corresponding functions convert a value from one form to another using a specified INFORMAT or FORMAT, respectively. The INPUT and PUT functions, however, do not read from an external file as do the INPUT and PUT statements. Their purpose is to convert a SAS variable into another SAS variable with a different representation, based on the informat / format specified. INPUT 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, while PUT is typically used to convert a numeric value into a character one.. Further discussion of these SAS features is outside of the scope of this paper; the reader is encouraged to get further detail and examples from one of the fine presentations on Functions that can be found in the various Proceedings and through SASPress

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 offered such a class a few years ago and always ran out of time before finishing the prepared materials!) It is hoped that the information 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

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

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.

Cody, Ronald (2007). *Learning SAS by Example – A Programmer's Guide*. Cary, NC: SAS Institute, Inc.

Kuligowski, Andrew T. (2006). "DATALINES, Sequential Files, CSV, HTML and More – Using INFILE and INPUT Statements to Introduce External Data into the SAS® System". *Proceedings of the Thirty-First 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.

Microsoft Corporation (2007). “The Windows Vista Developer Story: Application Compatibility Cookbook”. *http://msdn2.microsoft.com/en-us/library/aa480152.aspx*

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

SAS Institute, Inc. (1999), *SAS Language Reference: Dictionary, Version 8.* 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. (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:
KuligowskiConference@gmail.com

ACKNOWLEDGMENTS

Many people have contributed to my understanding of the topic over the years, and specifically to either this paper or one of its predecessors (which only emphasized the INPUT side of the equation). I’d like to acknowledge Debbie Buck, Paul Dorfman, Sue Douglass, Harry Droogendyk, Pete Lund, Dave Riba, Nancy Roberts, Koen Vyverman, Ian Whitlock, and Nat Wooding. I apologize to those who have been left off the list through my oversight.