Paper 273-2017

Tales from the Help Desk: Solutions to Common DATA Step Tasks

Bruce Gilsen, Federal Reserve Board, Washington, DC

ABSTRACT

In 32 years as a SAS ® consultant at the Federal Reserve Board, I have seen some questions about common SAS tasks surface again and again. This paper collects the most common questions related to basic DATA step processing from my previous "Tales from the Help Desk" papers, and provides code to explain and resolve them. The following tasks are reviewed:

1. Using the LAG function with conditional statements.
2. Avoiding character variable truncation.
3. Surrounding a macro variable with quotes in SAS code.
4. Handling missing values (arithmetic calculations versus functions).
5. Incrementing a SAS date value with the INTNX function.
6. Converting a variable from character to numeric or vice versa and keeping the same name.
7. Converting character or numeric values to SAS date values.
8. Using an array definition in multiple DATA steps.
9. Using values of a variable in a data set throughout a DATA step by copying the values into a temporary array.
10. Writing data to multiple external files in a DATA step, determining file names dynamically from data values.

In the context of discussing these tasks, the paper provides details about SAS processing that can help users employ SAS more effectively. See the references for seven previous papers that contain additional common tasks.

1. USING THE LAG FUNCTION WITH CONDITIONAL STATEMENTS

Data set ONE has the following data.

```
VAR1    VAR2

  1       1

 -1       2

  1       3

 -1       4

  1       5
```

A user wants to assign values to the variable LAGVAR2 as follows.

- If VAR1 is greater than 0, then LAGVAR2 = the value of VAR2 in the previous observation.
- Otherwise, LAGVAR2 = missing (.).

The user submits the following DATA step. The code might include an ELSE statement after the IF statement, ELSE LAGVAR2 = .;, but this statement does not change the results, so it is not included in this DATA step.

```
data two;
  set one;
  if var1 > 0 then lagvar2 = lag(var2);
run;
```

Data set TWO has the following data. Note that LAGVAR2 is 1 in the third observation; 2 was expected. Also, LAGVAR2 is 3 in the fifth observation; 4 was expected.

```
VAR1   VAR2   LAGVAR2
 1      1        .
-1      2        .
 1      3        1
-1      4        .
 1      5        3
```

Users make this error because the intuitive definition of a LAG function is that it always returns the value in the previous observation. In fact, the LAG function works as follows. Every time LAG(VAR2) is executed,

- the current value of VAR2 is stored on a queue
- the value of VAR2 from the last time LAG(VAR2) was executed is retrieved

LAG functions that are executed conditionally (as in DATA step TWO above) only store and retrieve values from observations for which the condition is satisfied. This is best illustrated by example. In DATA step TWO, the condition is satisfied (VAR1 is greater than 0) in the first, third, and fifth observations. LAGVAR2 is assigned as follows.

- Observation 1.

  The current value of VAR2, 1, is stored. No previous value of VAR2 is available to retrieve, so LAGVAR2 is set to missing (.).

- Observation 2.

  The IF statement is false, so the LAG function is not executed, and nothing is stored or retrieved.

- Observation 3.

  The current value of VAR2, 3, is stored. The most recently stored value of VAR2, 1, is retrieved and assigned to LAGVAR2.

- Observation 4.

  The IF statement is false, so the LAG function is not executed, and nothing is stored or retrieved.

- Observation 5.

  The current value of VAR2, 5, is stored. The most recently stored value of VAR2, 3, is retrieved and assigned to LAGVAR2.

To get the expected results, move the LAG function outside of the conditional code, so that it executes in every observation, as in the following DATA step.

```
data two;
  set one;
  lagvar2 = lag(var2);
  if var1 <= 0 then lagvar2 = .;
run;
```

Data set TWO has the following data.

```
VAR1   VAR2   LAGVAR2
 1      1        .
-1      2        .
 1      3        2
-1      4        .
 1      5        4
```

This DATA step executes as follows.

- Observation 1, statement with LAG function.

  The current value of VAR2, 1, is stored. No previous lagged value of VAR2 is available to retrieve, so LAGVAR2 is set to missing.

- Observation 1, IF statement.

  The IF statement is false, so no action is taken.

- Observation 2, statement with LAG function.

  The current value of VAR2, 2, is stored. The most recently stored value of VAR2, 1, is retrieved and assigned to LAGVAR2.

- Observation 2, IF statement.

  The IF statement is true, so LAGVAR2 is set to missing.

- Observation 3, statement with LAG function.

  The current value of VAR2, 3, is stored. The most recently stored value of VAR2, 2, is retrieved and assigned to LAGVAR2.

- Observation 3, IF statement.

  The IF statement is false, so no action is taken.

- Observation 4, statement with LAG function.

  The current value of VAR2, 4, is stored. The most recently stored value of VAR2, 3, is retrieved and assigned to LAGVAR2.

- Observation 4, IF statement.

  The IF statement is true, so LAGVAR2 is set to missing.

- Observation 5, statement with LAG function.

  The current value of VAR2, 5, is stored. The most recently stored value of VAR2, 4, is retrieved and assigned to LAGVAR2.

- Observation 5, IF statement.

  The IF statement is false, so no action is taken.

The LAG function returns values that are lagged one time. The related functions LAG2, LAG3,..., LAG100 return values that are lagged 2, 3, ..., 100 times. These functions behave analogously to the LAG function in conditional code.

Another way to correctly code the DATA step is as follows.

```
data two;
  set one;
  drop _templagvar2;
  _templagvar2 = lag(var2);
  if var1 > 0 then lagvar2 = _templagvar2;
run;
```

2. AVOIDING CHARACTER VARIABLE TRUNCATION

In data set ONE, created by the following DATA step, CHARVAR1 has the expected value, “fffggghhhiii”, but CHARVAR2 has the value “jjj”, not ”jjjkkklllmmm”.

```
data one;
  numvar = 100;
  charvar1 = "aaabbbcccddd";
  charvar2 = "eee";
  if numvar > 0 then
  do;
    charvar1 = "fffggghhhiii";
    charvar2 = "jjjkkklllmmm";
  end;
run;
  /* Display results and related info */
proc print data=one;
run;
proc contents data=one;
run;
```

Here is output from PROC PRINT.

```
Obs   numvar     charvar1     charvar2
 1      100    fffggghhhiii     jjj
```

To help understand the result, here is output from PROC CONTENTS.

```
#    Variable    Type    Len    Pos
-----------------------------------
2    charvar1    Char     12      8
3    charvar2    Char      3     20
1    numvar      Num       8      0
```

Note that the length of CHARVAR1 is 12 and CHARVAR2 is 3. The length of a character variable is determined the first time it is used (“first usage”). Subsequent statements in the DATA step cannot change the length.

In the DATA step above, first usage of CHARVAR1 and CHARVAR2 is in the following statements. The lengths of "aaabbbcccddd", 12, and “eee”, 3, determine the lengths of CHARVAR1 and CHARVAR2.

```
charvar1 = "aaabbbcccddd";
charvar2 = "eee";
```

To avoid this problem, use a LENGTH statement to create the variables. Code the LENGTH statement before other usage of the variables in the DATA step, so it is the first usage of the variables. Set the length of each variable to its largest possible value.

```
data one;
  length charvar1 $12 charvar2 $12;
  numvar = 100;
  charvar1 = "aaabbbcccddd";
  charvar2 = "eee";
  if numvar > 0 then
  do;
    charvar1 = "fffggghhhiii";
    charvar2 = "jjjkkklllmmm";
  end;
run;
```

Here is output from PROC PRINT. CHARVAR2 now has the correct value.

```
Obs   numvar    charvar1      charvar2
 1      100   fffggghhhiii  jjjkkklllmmm
```

Here is output from PROC CONTENTS. The length of CHARVAR2 is now 12.

```
#    Variable    Type    Len    Pos
----------------------------------
2    charvar1    Char     12      8
3    charvar2    Char     12     20
1    numvar      Num       8      0
```

3. SURROUNDING A MACRO VARIABLE WITH QUOTES IN SAS CODE

Beginning users usually do not use macros, but might use macro variables. A common mistake is surrounding a macro variable with single quotes, so that it does not resolve, as in the following DATA step statement.

```
newvar = '&macvar1';
```

The simple solution is to use double quotes, as in the following DATA step statement.

```
newvar = "&macvar1";
```

One reason that this error occurs is that single and double quotes are equivalent in some instances, such as the following DATA step statements, so users assume that single and double quotes are equivalent in all instances.

```
newvar = 'mynamehere';
newvar = "mynamehere";
```

4. HANDLING MISSING VALUES (ARITHMETIC CALCULATIONS VERSUS FUNCTIONS)

Data set ONE has the following values.

```
Obs   country   gnp2008   gnp2009   gnp2010
 1    usa          1         2         3
 2    japan        4         .         6
 3    china        .         .         9
```

Initially, we do an arithmetic calculation to calculate the mean of GNP2008, GNP2009, and GNP2010.

```
data two;
  set one;
  averagegnp = (gnp2008 + gnp2009 + gnp2010)/3;
run;
```

Data set TWO has the following values of AVERAGEGNP. In the second and third observations, at least one term in the calculation is a missing value (.), so the result is set to missing.

```
Obs   averagegnp
 1        2
 2        .
 3        .
```

Now, we replace the arithmetic calculation with the MEAN function. This generalizes the application because we no longer need to know how many values are used to calculate the mean.

```
data two;
  set one;
  averagegnp = mean(gnp2008, gnp2009, gnp2010);
run;
```

Data set TWO now has different values of AVERAGEGNP in the second and third observations, as follows.

```
Obs   averagegnp
 1        2
 2        5
 3        9
```

To understand why these answers differ, it is important to understand how SAS handles missing values.

- In an arithmetic calculation, SAS sets the results to missing if any of the values are missing.
- For DATA step functions such as MEAN, SUM, and MAX, SAS performs the operation on the non-missing values. For example, the second observation has two non-missing values, so the MEAN is the average of these two values. The result is set to missing only when all the values are missing.

If there are a large number of variables, it can be more convenient to specify them in an abbreviated form. For data set ONE, the following statements are equivalent.

```
(1) averagegnp = mean(gnp2008 + gnp2009 + gnp2010);
(2) averagegnp = mean(of gnp2008 - gnp2010);
(3) averagegnp = mean(of gnp:);
```

In (2), a numbered range list specifies a group of variables whose names differ by a numeric suffix. All variables in the range (GNP2008, GNP2009, and GNP2010) are used to calculate the mean. Any variables in the range that do not exist are created with a value of missing. As discussed in Williams (2006), if OF is omitted, GNP2008-GNP2010 is treated as a difference, not a range list, and SAS calculates the mean of that difference, as illustrated for the first observation.

```
averagegnp = mean(gnp2008 - gnp2010);
             mean(1 - 3);
             mean(-2);
             -2;
```

In (3), the name prefix operator “:” tells SAS to calculate the mean of all variables that begin with “GNP”. The OF keyword is required; otherwise an error occurs.

5. INCREMENTING A SAS DATE VALUE WITH THE INTNX FUNCTION

The DATA step function INTNX returns a SAS date value incremented by a specified number of intervals (days, weeks, months, quarters, years, etc.).

In the following DATA step, DATE1 is set to the SAS date value for October 18, 2005, which is 16,727. INTNX is used to increment the date by two days, two months, and two years. DATEPLUS2DAY has the expected value, but DATEPLUS2MONTH and DATEPLUS2YEAR do not.

```
data one;
  date1 = ‘18oct2005'd;
  dateplus2day = intnx(‘day',date1,2);    * want to increment by 2 days;
  dateplus2month = intnx(‘month',date1,2);* want to increment by 2 months;
  dateplus2year = intnx(‘year',date1,2);  * want to increment by 2 years;
run;
```

		Expected		Actual	
Variable	Description	Value	SAS date	Value	SAS date
dateplus2day	2 days after 10/18/2005	16729	10/20/2005	16729	10/20/2005
dateplus2month	2 months after 10/18/2005	16788	12/18/2005	16771	12/1/2005
dateplus2year	2 years after 10/18/2005	17457	10/18/2007	17167	1/1/2007

To help understand this problem, here is a somewhat informal and incomplete review of the syntax of INTNX. Information is provided for SAS date values, but not for datetime and time values, and sub-arguments to the interval value are omitted. For complete syntax, see the *SAS 9.4 Language Reference*.

INTNX has three required arguments and one optional argument, commonly used as follows for SAS date values.

INTNX(*interval*, *start-from*, *increment* <,*alignment*>);

- *interval* is the unit of measure (days, weeks, months, quarters, years, etc.) by which *start-from* is incremented.
- *start-from* is a SAS date value to be incremented.
- *increment* is the integer number of intervals by which *start-from* is incremented (negative values = earlier dates).
- *alignment* is where *start-from* is aligned within *interval* after being incremented. Possible values are BEGINNING, MIDDLE, END, and (new in Version 9) SAMEDAY. This argument is optional, and defaults to BEGINNING.

INTNX’s default *alignment* is BEGINNING, so by default *start-from* is aligned to the beginning of the period after being incremented. This leads to unexpected results for intervals other than DAY, as in the examples from the previous DATA step. As before, DATE1 is the SAS date value for October 18, 2005.

1. dateplus2day = intnx(‘day',date1,2);

   SAS first increments by two days, then aligns to the start of October 20, 2005 (the beginning of the interval, DAY), which has no effect. DATEPLUS2DAY is 16,729, the SAS date value for October 20, 2005, as expected.

2. dateplus2month = intnx(‘month',date1,2);

   SAS first increments by two months, then aligns to December 1, 2005 (the beginning of the interval, MONTH). The result is 16,771, the SAS date value for December 1, 2005.

3. dateplus2year = intnx(‘year',date1,2);

   SAS first increments by two years, then aligns to January 1, 2007 (the beginning of the interval, YEAR). The result is 17,167, the SAS date value for January 1, 2007.

In Version 9, a new alignment value, SAMEDAY, was added. SAMEDAY preserves the SAS date value’s alignment within the interval after it is incremented, generating the expected results. To prevent the problem shown in this example, always set alignment to SAMEDAY for intervals other than DAY (when interval is DAY, SAMEDAY is not necessary).

Here are the examples from the previous DATA step with the SAMEDAY argument added. As before, DATE1 is the SAS date value for October 18, 2005.

SAS Statement	Description	Value	SAS date
`dateplus2day=intnx(‘day',date1,2,”sameday”);`	2 days after 10/18/2005	16729	10/20/2005
`dateplus2month=intnx(‘month',date1,2,"sameday");`	2 months after 10/18/2005	16788	12/18/2005
`dateplus2year=intnx(‘year',date1,2),"sameday");`	2 years after 10/18/2005	17457	10/18/2007

Here are additional examples for some interesting dates. Note that 2000 and 2004 but not 2003 are leap years.

SAS Statement	Description	Value	SAS date
`date2=intnx('year','29feb2000'd,1,"sameday");`	1 year after 2/29/2000	15034	2/28/2001
`date3=intnx('year','29feb2000'd,4,"sameday");`	4 years after 2/29/2000	16130	2/29/2004
`date4=intnx('month','31mar2003'd,-1,"sameday");`	1 month before 3/31/2003	15764	2/28/2003
`date5=intnx('month','31mar2004'd,-1,"sameday");`	1 month before 3/31/2004	16130	2/29/2004

Note that until SAS 9.2, SAMEDAY should only be used with single, non-shifted date intervals (DAY, WEEK, WEEKDAY, TENDAY, SEMIMONTH, MONTH, QTR, SEMIYEAR, YEAR), because the following intervals might return the wrong answer with no error or warning.

- multiple (e.g., month2 = two-month interval)
- shifted (e.g., month.4 = month interval starting on April 1)
- time
- datetime

6. CONVERTING A VARIABLE FROM CHARACTER TO NUMERIC OR VICE VERSA AND KEEPING THE SAME NAME

This issue arises frequently because users want to merge two data sets by a variable that is numeric in one data set and character in the other. To merge, the variable must have the same type and name in both data sets.

CONVERT BETWEEN NUMERIC AND CHARACTER WITHOUT KEEPING THE SAME NAME

You can easily convert numeric variables to character variables with the PUT function, and convert character variables to numeric variables with the INPUT function, as in the following code.

```
data one;
    /* Create numeric variable */
  stcode = 33 ;
    /* Convert numeric variable to character variable */
  state = put(stcode, 2.) ;
    /* Convert character variable to numeric variable */
  stnumv = input(state,2.);
run;
```

CONVERT A SINGLE VARIABLE AND KEEP THE SAME NAME

It’s less obvious how to convert a variable from numeric to character or vice versa in a DATA step and keep the same name. Suppose we want to convert character variable STATE in data set ONE to a numeric variable. One way to do it is as follows.

```
data two ;
  set one (rename = (state = Z_state)) ;
  drop Z_state ;
  state = input (Z_state, 12.);
run;
```

Let's review how this code is processed.

- The DATA statement specifies that DATA step results are written to data set TWO.
- The SET statement controls how data set ONE is read. The RENAME option has the syntax oldname = newname. It does not change the name of STATE in data set ONE, but causes the name to be Z_STATE in this DATA step.
- The DROP statement prevents the variable Z_STATE, which is available in the DATA step, from being written to the output data set, TWO. This prevents an extra variable from cluttering data set TWO.
- Since data set TWO does not have an incoming character variable called STATE (it was renamed to Z_STATE in the SET statement), we can create a new numeric variable called STATE in data set TWO with the INPUT function. The 12. informat allows us to convert values with a width up to 12 characters.

An equivalent way to code this step is to remove the DROP statement and add DROP=Z_STATE to the DATA statement, as follows.

```
data two (drop=Z_state);
```

The following code does not work because character variable STATE from data set ONE exists during the DATA step and is renamed to Z_STATE when output to data set TWO, so the INPUT function fails because Z_STATE does not exist.

```
data two;   /* Incorrect code #1 */
  set one;
  rename state = Z_state;
  state = input (Z_state, 12.);
run;
```

The following code does not work because STATE is an existing character variable and the statement with the INPUT function tries to create a numeric variable called STATE.

```
data two;   /* Incorrect code #2 */
  set one;
  state = input (state, 12.);
run;
```

7. CONVERTING CHARACTER OR NUMERIC VALUES TO SAS DATE VALUES

Character values can be directly converted to SAS dates, but some users find it confusing that numeric values cannot be directly converted.

Suppose data set ONE has one observation and two variables: CHARVAR1 is character and NUMVAR1 is numeric. We want to convert CHARVAR1 and NUMVAR1 to SAS dates in a DATA step.

```
CHARVAR1     NUMVAR1
20141229     20141229
```

In general, we can convert numeric values to character values with the PUT function, and convert character values to numeric values with the INPUT function. SAS dates are numeric values.

The INPUT function converts a character value to a SAS date value as follows. The first argument is the character value to convert, and the second argument is an informat that corresponds to the appearance of the data.

```
sasdate1 = input(charvar1, yymmdd8.) ;
```

But, converting a numeric value is not as straightforward, for the following reasons.

- The first argument to the INPUT function must be a character value, so the following statement is not valid.

  ```
  sasdate1 = input(numvar1, yymmdd8.) ;
  ```

- The result of the PUT function is always a character value, so the PUT function cannot create a SAS date, which is numeric.

To create a SAS date from a numeric value, first convert the numeric value to a character value with the PUT function. Then, convert the character value to a SAS date with the INPUT function.

```
tempchar = put (numvar1, 8.) ;
sasdate2 = input(tempchar, yymmdd8.) ;
```

This statement is equivalent to the previous two statements. It avoids creating the variable TEMPCHAR, but is somewhat less readable.

```
sasdate2 = input(put(numvar1,8.),yymmdd8.) ;
```

You can also convert character and numeric values to SAS dates with PROC SQL. The following code converts CHARVAR1 and NUMVAR1, the variables listed above, to SAS dates. To control how the SAS dates are displayed, different formats are assigned to the two variables containing SAS dates. This is done for illustrative purposes.

```
proc sql;
   create table two as select
     input(charvar1,yymmdd8.) as sasdate1   format=yymmddn8.,
     input(put(numvar1,8.),yymmdd8.) as sasdate2   format=date9.
   from one;
quit;
```

8. USING AN ARRAY DEFINITION IN MULTIPLE DATA STEPS

An array definition is only in effect in the DATA step in which it is defined and cannot be stored in a data set.

If you use the same array in multiple steps, as users often do, it can be helpful to define it in only one place. Then, if the array definition changes, you only need to change the code once.

To define the same array in more than one DATA step, you can do one of the following.

- Create a macro variable containing the array definition with a %LET statement, and use the macro variable in each DATA step.
- Create a macro containing the array definition, and execute the macro in each DATA step.

Here is an example with a macro variable.

```
  /* Create a macro variable called ARRAYDEF.  It contains an
     array definition that is used in subsequent steps. */
%let arraydef= array gnp (*) consume invest gov tax;

  /* Use the array definition in a DATA step */
data two;
```

```
  set one;
  &arraydef; /* Define array GNP */

   /* more SAS statements */

run;
     /* Use the array definition in another DATA step */
data three;
  set one;
  &arraydef; /* Define array GNP */

   /* more SAS statements */

run;
```

Here is an example with a macro.

```
  /* Create a macro called ARRAYDEF.  It contains an
     array definition that is used in subsequent steps. */
%macro arraydef;
  array gnp (*) consume invest gov tax;
%mend arraydef;

  /* Use the array definition in a DATA step */
data two;
  set one;
  %arraydef /* Define array GNP */

   /* more SAS statements */

run;
     /* Use the array definition in another DATA step */
data three;
  set one;
  %arraydef /* Define array GNP */

   /* more SAS statements */
run;
```

9. USING VALUES OF A VARIABLE IN A DATA SET THROUGHOUT A DATA STEP BY COPYING THE VALUES INTO A TEMPORARY ARRAY

Suppose we want all values of variable BANKID from data set ONE available while we read and process data set TWO and create data set THREE but do not need to write the values of BANKID to THREE.

Users often merge data sets in this situation, but a simple alternative is to copy the values of BANKID to a temporary array at the beginning of the DATA step. Values in a temporary array are automatically retained, and are thus available throughout the DATA step.

Here are data sets ONE and TWO.

```
            ONE                                TWO

obs  bankid   var1   var2           var3   var4   var5

 1     111     10     11              1      2      3

 2     222     20     12              4      5      6

 3     333     30     13              7      8      9

 4     444     40     14
```

METHOD 1

First, use PROC SQL to create two macro variables.

- ALL_BANK_IDS contains the values of variable BANKID, space-separated.
- NUM_OBS contains the number of values of BANKID in macro variable ALL_BANK_IDS. It is copied from the automatic macro variable SQLOBS, which contains the number of rows selected by the last PROC SQL statement.

In this example, the macro variables have the following values:

```
ALL_BANK_IDS: 111 222 333 444

NUM_OBS:      4
```

Then, in a DATA step, define temporary array ALL_BANKIDS using the two macro variables created with PROC SQL to specify the array size and values, then process data set TWO. The temporary array values will be available throughout the DATA step. In this example, the temporary array has a size of 4 and contains the following values:

```
111

222

333

444
```

```
  /* Create macro variables with the values of BANKID
     and the number of values of BANKID */
proc sql noprint;
  select bankid into :all_bank_ids separated by ' '
  from one;
  %let num_obs = &sqlobs;
quit;

data three;
     /* Create temporary array with values of BANKID */
   array all_bankids (&num_obs) _temporary_ (&all_bank_ids);

    /* Read data set TWO.  The values of BANKID in the ALL_BANKIDS
       temporary array are available throughout the DATA step. */
  set two;

     /* Standard DATA step processing of TWO goes here. */

run;
```

METHOD 2

This method is not as simple as Method 1, but could be helpful to review because it can be generalized to store multiple variables in a 2-dimensional array.

First, use PROC SQL to create macro variable NUM_OBS containing the number of observations in data set ONE, 4. Then, in a DATA step, read the values of BANKID into the temporary array ALL_BANKIDS, then process data set TWO. The temporary array values will be available throughout the DATA step. As with Method 1, the temporary array has a size of 4 and contains the following values:

```
111
222
333
444
```

```
  /* Set macro variable NUM_OBS to the number of observations in ONE.
     It is used as the size of the temporary array. */
proc sql noprint;
   select count(*)
   into :num_obs
   from one;
quit;

data three;
  array all_bankids (&num_obs) _temporary_;
  drop bankid;
```

```
    /* Read the values of BANKID into the temporary array */
  if _n_=1 then do;
    do _i = 1 to &num_obs;
      set one (keep=bankid) ;
      all_bankids(_i) = bankid;
    end;
  end;

    /* Read data set TWO one observation at a time in the standard way.
       The values of BANKID in the ALL_BANKIDS temporary array are
       available throughout the DATA step. */
  set two;

     /* Standard DATA step processing of TWO goes here */

run;
```

10. WRITING DATA TO MULTIPLE EXTERNAL FILES IN A DATA STEP, DETERMINING FILE NAMES DYNAMICALLY FROM DATA VALUES

Data set ONE has the following data.

```
Obs     bankid    income
 1      bank1       10
 2      bank1       20
 3      bank3       80
 4      bank3       90
 5      bank3      100
 6      bank2       40
 7      bank2       50
```

We want to write data values to multiple output files in the Linux directory /my/home/directory/out1. The file names and records to write to those files are determined by the value of the BY variable, BANKID, for example:

Value of BANKID	File that the record is written to
bank1	/my/home/directory/out1/bank1
bank2	/my/home/directory/out1/bank2

Linux files are used in this example, but the code can be used on other platforms by changing the file names to platform-appropriate values.

The code below includes the following steps.

- Create a retained character variable, FILEID, that contains the name of the current output file. The length of FILEID must be large enough to hold the longest possible file name.
- Change FILEID by appending the current value of BANKID to the directory path.
- Use a FILE statement that references the current output file with the clause FILEVAR=FILEID. The FILE statement is an executable statement that must be executed in every observation, so it must not be within a conditional (e.g., IF/THEN) statement. The FILE statement includes a placeholder file specification, TEST, that is ignored but required – an error occurs if it is omitted.

```
data three ;
  set two;
  by bankid;
    /* FILEID must be large enough to hold longest possible file name */
  length fileid $40;

    /* Specify the output file for this observation */
  fileid = "/my/home/directory/out1/" || bankid;

    /* The FILE statement executes in every observation and includes a
       placeholder file specification, TEST, that is ignored but required
    */
  file test filevar=fileid;
  put income;  /* Write to an external file */
run;
```

This program creates three external files with the following values.

```
/my/home/directory/out1/bank1:
            10
            20
/my/home/directory/out1/bank2:
            40
            50
/my/home/directory/out1/bank3:
            80
            90
           100
```

CONCLUSION

This paper provided SAS code for some simple, common SAS DATA step tasks, and in the context of discussing these tasks, provided details about SAS system processing. It is hoped that this paper enables users to better understand SAS system processing and thus employ SAS more effectively in the future.

REFERENCES

Gilsen, Bruce (2003), "Deja-vu All Over Again: Common Mistakes by New SAS Users," *Proceedings of the Sixteenth Annual NorthEast SAS Users Group Conference.* <http://www.lexjansen.com/nesug/nesug03/bt/bt010.pdf>

Gilsen, Bruce (2007), "More Tales from the Help Desk: Solutions for Common SAS Mistakes," *Proceedings of the SAS Global Forum 2007 Conference*. <http://www2.sas.com/proceedings/forum2007/211-2007.pdf>

Gilsen, Bruce (2009), "Tales from the Help Desk 3: More Solutions for Common SAS Mistakes," *Proceedings of the SAS Global Forum 2009 Conference*. <http://support.sas.com/resources/papers/proceedings09/137-2009.pdf>

Gilsen, Bruce (2010), "Tales from the Help Desk 4: Still More Solutions for Common SAS Mistakes," *Proceedings of the SAS Global Forum 2010 Conference*. <http://support.sas.com/resources/papers/proceedings10/146-2010.pdf>

Gilsen, Bruce (2012), "Tales from the Help Desk 5: Yet More Solutions for Common SAS Mistakes," *Proceedings of the SAS Global Forum 2012 Conference*. <http://support.sas.com/resources/papers/proceedings12/190-2012.pdf>

Gilsen, Bruce (2015), "Tales from the Help Desk 6: Solutions to Common SAS Tasks," *SESUG 2015: The Proceedings of the SouthEast SAS Users Group, Savannah, GA, 2015*. <http://www.lexjansen.com/sesug/2015/72_Final_PDF.pdf>

Gilsen, Bruce (2016), "Tales from the Help Desk 7: Solutions to Common SAS Tasks," *SESUG 2016: The Proceedings of the SouthEast SAS Users Group, Bethesda, MD, 2016*. <http://www.lexjansen.com/sesug/2016/PA-105_Final_PDF.pdf>

SAS Institute Inc. (2015), "Base SAS 9.4 Procedures Guide, Fifth Edition," Cary, NC: SAS Institute Inc.

SAS Institute Inc. (2006), SAS Note 016184, "INTNX function with SAMEDAY alignment does not support multiple, shifted, time, or datetime intervals." <http://support.sas.com/kb/16/184.html>

SAS Institute Inc. (2015), "SAS 9.4 Statements: Reference, Fourth Edition," Cary, NC: SAS Institute Inc.

SAS Institute Inc. (2015), "SAS 9.4 Macro Language: Reference, Fourth Edition," Cary, NC: SAS Institute Inc.

Williams, Christianna S. (2006), "Those Sneaky SAS® Functions: Beware of Unexpected Handling of Missing Data and Variable Lists," *Proceedings of the Nineteenth Annual NorthEast SAS Users Group Conference*. <http://www.lexjansen.com/nesug/nesug06/cc/cc31.pdf>

ACKNOWLEDGMENTS

The following people contributed extensively to the development of this paper: Heidi Markovitz and Donna Hill at the Federal Reserve Board. Their support is greatly appreciated.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Bruce Gilsen
Federal Reserve Board, Mail Stop N-122, Washington, DC 20551
202-452-2494
bruce.gilsen@frb.gov