**PharmaSUG 2018 - Paper EP-16**

# Discover the Deeper DATA Step

Timothy J Harrington, DataCeutics, Inc.

## ABSTRACT

All SAS® Programmers will be familiar with the SAS DATA Step, however, there many SAS users who may be unfamiliar with or even unaware of facilities in the DATA step which, if exploited, would result in easier programming, code that is easier to understand, better run-time performance, and easier QC and verification. This paper lists and describes the most common of these DATA Step 'extras', describes how and when to use them, using examples of SAS code, and compares with alternative methods of achieving the same result.

## INTRODUCTION

The SAS DATA Step is undoubtedly the heart of the SAS System, it gives the programmer the greatest control compared to SAS PROCedures. Each new release of SAS has more functions and facilities with increasing refinement and improved performance, so much so that there are many programmers who may not be aware of the current 'best' technique for solving a given problem. This paper assumes the reader has at least a basic working knowledge and at least a beginner level of experience working with the DATA Step. The contents described in this paper are all based on SAS v9.4, but many of the facilities described are available in some earlier versions.

## AUTOMATIC VARIABLES

All SAS System variables begin and end with a single underscore, for this reason users should not create their own variables starting and ending with an underscore, future releases of SAS are may introduce new automatic variables.

There are three basic automatic run time variables provided by the SAS DATA Step, these are _N_, _ERROR_, and _LAST_. All automatic variables may be typed in upper, lower, or mixed case. The observation count _N_ is the sequential number of the observation currently being processed, that is the current observation number in the *Program Data Vector* (PDV). _N_ is initially set to zero and is incremented each time an observation is read into the PDV at a SET or MERGE statement. In a MERGE all columns from any one or more of the input data sets with matching BY variables are placed in the PDV and _N_ is incremented. Note that _N_ is a temporary variable that is automatically assigned when a new data set is generated, _N_ cannot be manually changed. _N_ is always dropped after the DATA step completes and it cannot be given a different name using the RENAME option, neither can _N_ be used with sub-setting criteria with a WHERE clause. This rule applies to all system variables. To see the value of _N_ at any given point it should be displayed using put _n_=;, or be assigned to another numeric variable.

The second automatic runtime variable is _ERROR_. This is zero when no error occurs but is set to 1 when a runtime error happens. Examples of such runtime errors are a subscript out of range in a SUBSTR function or an INPUT statement using an inappropriate format.

The third automatic runtime variable is _LAST_. This contains the full name, including the library name, of the most recently created data set and has that value in any following SAS PROCedure or the next DATA step. _LAST_ is missing during the execution of the first DATA step being run and in any PROCedures before the first DATA step. This variable should be used with care because its value will change whenever DATA step names and order of processing data sets change.

Another value provided by the SAS system is the IN variable. An IN variable is defined in the data set options as IN=<any valid SAS variable name>. An IN variable is 1 when an observation is read from the data set to which it applies. When reading from one data set only the IN variable will always be 1, but when using a MERGE or SET with multiple input data sets the IN variable indicates if an observation was read from that data set. In a MERGE, when IN variables are used in both (or all) input data sets the IN variable is 1 when an observation from that data set has matching BY variables and 0 when it does not. In the example below two data set, DEMO and VITALS are sorted by the patient ID, PATNUM. An IN variable, A, is defined for DEMO, and an IN variable, B, is defined for VITALS. The created data set PTWGHT would have only observations for patients in *both* DEMO and VITALS, because an output observation is generated only when both A and B are 1. If the conditional statement was changed to ‘if a;’ then output observations would be created for all of the patients in DEMO, whether they had an observation in VITALS or not. When a patient does not have an observation in VITALS a=1 and b=0 so an observation is output in PTWGHT with values from DEMO and the variables from VITALS, but these have missing values, i.e. WTKG is missing for that patient.

```
data ptwght;
  merge demo(in=a) vitals(in=b keep=patnum wtkg);
  by patnum;
  if a and b;
run;
```

IN variables can also be used with SET statements to indicate which data set an observation originated from, in this example the variable ORIGSTDY in ALLPATS is set to 1 for the observations from the STUDY1 data set, to 2 for observations from the STUDY2 data set, and to 3 for observations from the STUDY3 data set.

```
data allpats;
  set study1(in=a) study2(in=b) study3(in=c);
  origstdy=a+2*b+3*c;
run;
```

Programmers should note that IN variables are only valid within the DATA step in which they are defined, they are always dropped when the DATA step completes. IN variables cannot be used with a KEEP or RENAME statement.

Something else which is often useful is the DATA step END flag. This is not an option in the options list contained in parenthesis, but is a flag defined on the SET (or MERGE) statement. An example shown below where the DEMO data is sorted by PATNUM and AGE and the END flag OLDEST is used to output the observation for the oldest patient:

```
data eldest;
  set demo end=oldest;
  by patnum age;
  if oldest;
run;
```

Another useful DATA Step programming tool is the FIRST and LAST facility. When a data set is sorted changes in the values of the BY variables are indicated by the system provided FIRST.<by variable> and LAST.<by variable>. The FIRST.<by variable> is set to 1 when the value of that variable is different from the value in the prior observation. The LAST.<by variable> is set to 1 when the value of that variable is different from the value in the next observation. The example below illustrates this, this DATA step is sorted by TREATMENT and PATNUM. The variables F_TRT, L_TRT, F_PAT, and L_PAT are assigned to the corresponding FIRST and LAST system values. These system values are always dropped on completion of the DATA step; however, they can be displayed using a PUT statement. Also shown is the END flag stored in ENDFLG, which is only set on the very last observation.

```
data stdywght;
   set demwght end=finalrec;
   by treatment patnum;
   f_trt=first.treatment;
   l_trt=last.treatment;
   f_pat=first.patnum;
   l_pat=last.patnum;
   endflg=finalrec;
 run;
```

```
Obs TREATMENT PATNUM  F_TRT L_TRT F_PAT L_PAT ENDFLG
 1     A       100      1     0     1     0      0
 2     A       100      0     0     0     0      0
 3     A       100      0     0     0     1      0

 4     A       101      0     0     1     0      0
 5     A       101      0     0     0     0      0
 6     A       101      0     0     0     0      0
 7     A       101      0     0     0     0      0
 8     A       101      0     0     0     1      0

 9     A       102      0     0     1     0      0
10     A       102      0     1     0     1      0

11     B       200      1     0     1     0      0
12     B       200      0     0     0     0      0
13     B       200      0     0     0     0      0
14     B       200      0     0     0     1      0

15     B       201      0     0     1     1      0

16     B       202      0     0     1     0      0
17     B       202      0     0     0     0      0
18     B       202      0     0     0     0      0
19     B       202      0     1     0     1      1
```

The F_ flags indicate the FIRST value was set to 1 for the first new value in the applicable BY group. Similarly, the L_ flags indicate the LAST value was set to 1 at the end of the applicable BY group. All FIRST flags are set to 1 on the very first observation (_N_=1) and all LAST flags are set to 1 on the very last observation, when END is 1. A convenient way to test for an observation with duplicate BY variables is to check for the FIRST flag or LAST flag for each variable being 0. When there is only one observation with particular BY values the FIRST and LAST flags are both 1. An important point to mention here is that, if selecting observations based on whether they are (or are not) the first or last of a BY group, the IF THEN or SELECT construct should be used since these flags cannot be used with a WHERE clause in data set options. Another fact to note is that if the higher order BY variable changes but the next order BY variable happens to be the same, the FIRST and LAST flags are still set on the lower order BY variable when the higher order BY variable changes. In the above example, if PATNUM 200 also had had observations in TREATMENT A but had subsequently switched to TREATMENT B, these would be the values of the flags, first and last PATNUM change when TREATMENT changes:

```
TREATMENT PATNUM  F_TRT L_TRT F_PAT L_PAT
     A       200     1     0     1     0
     A       200     0     0     0     0
     A       200     0     1     0     1

     B       200     1     0     1     0
     B       200     0     0     0     0
     B       200     0     0     0     0
     B       200     0     0     0     1

     B       201     0     0     1     0
```

Other reserved SAS names useful in a DATA step are _ALL_, _NUMERIC_, and _CHARACTER_. Using _ALL_ with a PUT statement will show all the variables in the data set, including the automatic variables, and their values. Using _NUMERIC_ in a KEEP statement will keep all and only the numeric variables. Using _CHARACTER_ in a KEEP statement will keep all and only character variables. _NUMERIC_ and _CHARACTER_ cannot be used with a PUT statement.

## AUTOMATIC MACRO VARIABLES USEFUL IN A DATA STEP

There are many system macro variables provided by SAS. These are resolved by using an ampersand, just like with user defined macro variables. The system macro variables which are most useful inside a DATA step are SYSDATE9, the current date formatted as ‘ddmmmyyyy’ (date9. SAS format), SYSTIME the system time, and SYSDAY, the day of the week. Other system macro variables often of value are SYSCPL, the operating system platform, SYSVAR, the version of SAS being run, and SYSMACRONAME, the name of the currently executing macro (this is blank when execution is in open code). SYSNOBS is the total number of observations in the current data set (_N_ at the last observation) or output from the most recently created (or modified) data set. SYSPROCNAME is the most recent SAS procedure executed. These PUT statements show the results below in the SAS LOG file:

```
   put "&sysprocname Compiled on &sysday &sysdate9 &systime";
   put "On Host &syshostname &syshostinfolong";
   put "Platform is &sysscpl";
   put "SAS Version is &sysver";
   put "Number of Observations read &sysnobs";
```

```
DATASTEP Compiled on Friday 02MAR2018 20:52

On Host Campus1 Linux LIN X64 2.6.32-696.1.1.el6.x86_64 #1 SMP Sat Oct 17
10:48:55 EDT 2015 x86_64 Wild Fox Research Unit Linux Server release 6.6
(Santiago)

Platform is Linux

SAS Version is 9.4

Number of Observations read 1761
```

A list of all currently available automatic SAS macro variables is written to the SASLOG file by this using this statement

%put _automatic_;

## OBSERVATION SELECTION AND OUTPUT

By default the SAS DATA step reads every observation in a DATA Step. A range of observation numbers can be specified using the FIRSTOBS and OBS options, FIRSTOBS is the first observation to read (default is the first observation in the data set) and NOBS is the total number of observations to read. In this example only observations 25 through 39 are input to the PDV:

```
data ptid2539;
  set demo(firstobs=25 nobs=15);
run;
```

Note: Even though specific observations are selected the internal variable _N_ refers to the sequence of observations read into the PDV, and hence will have the values 1 through 15 and not 25 through 39.

A few words now about the SET statement. When the SET statement is used as shown below, the data sets are appended to each other in the same order, any variable not in all of the data sets is set to missing in the observations from the data sets that do not have that variable. For variables that exist in two or more of the data sets that are SET together their attributes are taken from the first data set. If a character variable of the same name has different lengths the length is taken from the first data set containing that variable and a WARNING may be written to the LOG file. Unless the first length is the longest, truncation of texts longer than that first length will occur.

```
data qall;
  set a b c d;
run;
```

When data set names are on separate SET statements, they are joined (like a MERGE) by observation number. Each observation in the output data set now contains *all* of the variables from *all* of the input data sets. There are two points to note here. First, a variable of the same name present in two or more of the input data sets has its attributes, such as length and label, taken from the first data set and its contents taken from the last data set. In the example below the variable SCORE in data set QALL would have its attributes the same as in data set A and its value the same as in data set D. As with MERGEs, when variables with the same name overwrite, attributes are taken from the first data set and values from the last (including missing values). This is the “Attributes come from the left and values come from the right” rule. The second point is the DATA step stops at the end of the iteration when the last observation in the input data set with the *fewest* observations has been read, any remaining observations in the other data sets are not processed. Sort order in the individual data sets does matter. For these reasons, using SET like this is recommended only for data sets with only one observation or for joining two tables with the same number of observations by the same key values sorted in the same order and their non-key variable names are unique. An example may be joining a data set of patients’ demographics to their baseline laboratory results, both sorted by patient ID. If there are multiple key values in one of the datasets a MERGE must be used with the data sets sorted by the key variables (patient ID in this case). Even when observation key values are unique a MERGE is the preferred method because IN variables can be used to indicate if an observation which should be present in both data sets is only in one.

```
data qall;
  set a(keep=patnum testid score);
  set b(keep=patnum testid mark);
  set c(keep=patnum testid score mark);
  set d(keep=patnum testid score);
  set e(keep=patnum stdtest result);
run;
```

Having looked at SET for input, observation output should now be considered. When observations are output there is an implied output at the end of the DATA step, just before the RUN statement. However, the OUTPUT statement outputs the PDV contents at that point. OUTPUT may be used multiple times in the DATA step. The use of the OUTPUT statement overrides the implied output. If one output is intended for a subset of the observations, the OUPUT keyword can be omitted. For example, this DATA step creates a data set containing only the observations for female patients:

```
data ptfemale;
  set demo;
  if sex='F';
run;
```

In this next example, also shown earlier, the IN variables A and B are being tested. The implied output, that is the output of the contents of the PDV at that point, takes place when both A=1 and B=1 (There are matching observations from both input data sets), no output takes place when this condition is not satisfied.

```
data ptwght;
  merge demo(in=a) vitals(in=b keep=patnum wtkg);
  by patnum;
  if a and b;
run;
```

The OUTPUT statement can specify a specific output data set, and hence can be used to direct an observation to a specific output data set. In the following example the observations are written to a data set of male patients or a data set of female patients depending on whether SEX is 'M' or 'F'. If SEX is blank or has any value other than 'M' or 'F' (including lower case 'm' and 'f') that observation is not output. (An ELSE condition should be added to output erroneous values to an 'error' data set.).

```
data ptfemale ptmale;
  set demo;
  if sex='F' then output ptfemale;
  if sex='M' then output ptmale;
run;
```

The DELETE statement removes the current PDV contents, it is not the same as 'do not OUTPUT', since the variables in the DATA step are no longer available for the rest of the DATA step. A good case for the use of the DELETE statement is in a DATA step structured like shown below and when there is a large or complex amount of processing to be performed, but only a few of a large number of observations in the input data set are expected to satisfy the criteria to be processed and output.

```
data <output data set with the results from a few selected observations>;
  set <input data set with many observations>;
  if <exclusion criteria met / inclusion criteria not met> then do;
    delete;
  end;
  else do;
   <large amount of processing>;
  end;
run;
```

## OBSERVATION SORT STATUS OPTIONS

The use of FIRST and LAST flags has been mentioned above, these depend on the BY variables, that is the variables by which the data set is sorted. Usually a data set has to be sorted prior to performing a DATA step that will use those BY variables. Such sorting can be performed with either a PROC SORT or an ORDER BY clause in PROC SQL. However, if the data set observations are already in the required order, the SORTEDBY option can be set to the list of BY variables to be referenced in the DATA step, just as if a prior sort had taken place using these BY variables, but in this case the sort would have been redundant. The SORTEDBY flag is only available in the current DATA step and has to be re-specified for any subsequent DATA steps. Also, the SAS system does not verify the sort order, hence if the observations in the data set are not sorted in the same manner as specified in the SORTED BY option a 'BY Variables Not Properly Sorted' error will occur.

Another situation where BY groups can be used without a prior sort, even if the data set is not ordered by the intended BY variables is to specify NOTSORTED on the BY statement. This is most useful in a situation where there are a relatively small number of observations and there is a requirement to have those observations sorted in a different order in different DATA steps (or in certain SAS procedures such as PROC PRINT). There is still an internal sort performed at run time but the use of NOTSORTED eliminates the need to use a PROC SORT. However, for large numbers of observations and when multiple DATA steps or SAS procedures are used with the same BY variables, using PROC SORT once is a more efficient option.

## PRIOR AND NEXT OBSERVATION VALUES

There is sometimes a need to carry forward prior values of a variable to following observations. An example of this may be in recording a patient's weight at certain time points during a clinical study. The study rules may specify that if, for some reason, the patient's weight was not recorded at a particular visit (value is missing) the weight recorded at the prior visit should be carried forward. This is achieved by the use of the RETAIN statement. Normally, at the end of a DATA step (after the implied OUTPUT if there is one) the PDV contents are all set to missing. The next observation read then re-populates the PDV at the SET (or MERGE) statement during the next iteration of the DATA step. The RETAIN statement keeps the specified variables (does not set them back to missing) so they are in the PDV at the start of the next iteration. Variables that are in the input data set, that is not created in the current DATA step, are overwritten with the values in the next observation read at the SET (or MERGE) statement. In the example below the observations are sorted by TREATMENT PATIENT and VISITNUM (nominal time order) and the prior patient weight is carried forward in the variable PRIORWT. If the current observation has a missing value for WEIGHT it is set to PRIORWT and a flag IMPWT is set to 1. This flag is missing under normal conditions when this imputation does not take place.

```
data ptweight;
  set vitals(keep=treatment patnum visitnum visit weight);
  by treatment patnum visitnum;
  retain priorwt 0;
  if first.patnum then do;
    priorwt=.;
  end;
  if weight>0 then do;
    priorwt=weight;
  end;
  else do;
    weight=priorwt;
    impwt=1;
  end;
run
```

The zero in the RETAIN statement indicates PRIORWT is numeric and has an initial value of zero. PRIORWT is set to missing for the first observation for each patient since there is no prior weight to carry forward. If WEIGHT is greater than zero (and is hence non-missing) that value is stored in PRIORWT. Assuming there is no variable of the same name in the VITALS data set, PRIORWT will still have this value after the next observation in VITALS has been read. If WEIGHT is missing (or zero) WEIGHT is set to PRIORWT and IMPWT is set to 1. IMPWT will be 1 only on that observation, it is otherwise missing since it is not being retained. Variables not retained are set to missing at the start of the next iteration of the DATA step. This method for carrying forward will also work for two or more successive missing values of WEIGHT. Note: This example uses a numeric variable, if a character variable is being carried forward the variable to hold the carried forward result (corresponding to PRIORWT) must be at least the same length (A LENGTH statement with $ should be specified) or truncation may occur.

This use of RETAIN is just one method of carrying forward prior values to later observations, a second method is to use the LAGn function. This retains the value of the variable n observations ago. Below is an example of the use of the LAGn function on the same data set as above, again using the variable WEIGHT. This time the value lagging by one observation (LAG1) is stored in PRIORWT and the value lagging by two observations (LAG2) is stored in PRIORWT2. If WEIGHT is missing, it is set to the value of PRIORWT, if PRIORWT is also missing WEIGHT is set to PRIORWT2. The variable BYOBSN is the observation number in the current PATNUM BY group. This is reset to zero when FIRST.PATNUM=1 and is incremented at every observation.

```
data ptweight;
  set vitals(keep=treatment patnum visitnum visit weight);
  by treatment patnum visitnum;
  retain byobsn 0;
  byobsn=byobsn*(first.patnum=0)+1;
  if byobsn>1 then do;
    priorwt=lag1(weight);
  end;
  if byobsn>2 then do;
    priorwt2=lag2(weight);
  end;
  if weight=. Then do;
    weight=priorwt;
    if weight=. Then do;
      weight=priorwt2;
    end;
    if weight ne . then do;
      impwt=1;
    end;
  end;
run;
```

```
TREATMENT PATNUM  VISITN  WEIGHT  PRIORWT  PRIORWT2

     A       100     1      72.4      .         .
     A       100     2      75.3    72.4        .
     A       100     3      72.6    75.3     72.4
     A       100     4      69.7    72.6     75.3
     A       100     5      68.5    69.7     72.6

     A       101     1      84.9      .         .
     A       101     2      84.4    84.9        .
     A       101     3      83.6    84.4     84.9
     A       101     4      84.0    83.6     84.4
```

A point to note here is that for the LAGn function, the first n observations in the data set have missing values. However, here the requirement is to consider each PATNUM separately, the first value of PRIORWT for each PATNUM (when FIRST.PATNUM=1) has to be set to missing since there is no prior value for that patient at their earliest observation. The variable BYOBSN maintains a count of the current observation number within the PATNUM BY group (note the use of RETAIN), when BYOBSN is 1 the observation is the first so there is no prior WEIGHT value for this patient. If this first value was set to LAG1(WEIGHT) the first value of PRIORWT would be the last value from the preceding patient, in the above example PATNUM 101 at VISITN 1 would have PRIORWT equal to 68.5kg . (In a DATA step all non-retained variables are set to missing at the top of the DATA step, hence if they are not assigned a value they remain as missing for that iteration of the DATA step). The same ruling must also be applied to PRIORWT2, except this time the first *two* observations now need to be left as missing.

The consideration here is carrying forward missing values of WEIGHT; this method does not carry forward more than two successive missing values, whereas the prior method using RETAIN with PRIORWT carries across any number of successive missing values.

Whilst considering RETAIN, here is another tip. RETAIN can be used for display ordering some or all of the columns from left to right in a data set. To achieve this the RETAIN statement must be the first statement after the DATA statement, before any SET or MERGE. Also, care must be taken with variables created within the DATA step not to assume they are missing at the start of the DATA step (because they are now being retained).

A somewhat less common need in a DATA step is to carry backwards from the next or following observation. Unfortunately, there is not currently a SAS provided function to complement LAGn, such as LEADn. However, a work around is possible. One method is to sort the data set in reverse order and use one of the two methods described above. To achieve this in the above examples the keyword DESCENDING is needed in front of VISITNUM in the BY statement (in both a preceding PROC SORT and the DATA step BY statements). After completing the DATA step the observations would need to be resorted into their original ascending order. A second method is to use the _N_ variable. This method works by assigning a variable such as KEY0 to _N_ and another created variable KEY1 to _N_-1, so that when KEY0 is 1 KEY1 is 0, when KEY0 is 2 KEY1 is 1 and so on, so KEY1 is 1 always less than KEY0. The next step is to create a separate data set with KEY1 and the variable to be carried backwards (WEIGHT), then rename KEY1 in this data set as KEY0 and rename the variable to be carried (e.g. NEXTWT), then sort both data sets by their KEY0 and finally MERGE by KEY0. An example macro to perform this is shown on the next page, here values of the variable being carried backwards are carried back within equal values of a specified BY group, using the observation sequence number within each set of such values instead of _N_ for the whole data set.

The input parameters to the macro are:

INDS, the name of the input data set, the default is the most recently accessed data set.

OUTDS, the name of the output data set. If not specified the new column created with the carried backwards values added to INDS. (Otherwise, the contents of INDS are unchanged).

BYVARS, the names of the BY variables to order and group the observations. If no BY variables are specified all of the observations in the data set are taken as a single grouping.

GRPVAR, the name of a variable in BYVARS to perform the lead within each set of successive observations with the same value of GRPVAR (FIRST. to LAST.). GRPVAR must be specified if BYVARS are used, GRPVAR must be left blank if there are no BYVARS.

N, the number of observations to look ahead, the default is 1, the next observation.

VAR, the name of the variable to be carried backwards (within each GRPVAR), VAR itself is unchanged by this macro.

NXTVARCB, the name of the created variable to hold the carried backward result. The default is the name of VAR with the suffix ‘_CB’ added.

```
/*** Macro to simulate a LEADn function ***/

%macro leadn(inds=&syslast, outds=, byvars=, grpvar=, n=1, var=, nxtvarcb= );

/*** If no output data set &nxtvarcb is added to the input data set */

%if &outds= %then %do;
  %let outds=&inds;
%end;
%if &nxtvarcb= %then %do;
  %let nxtvarcb=&var._cb;
%end;

/*** Set up a dummy sort variable BVAR1 in case no BY groups are specified */
/*** when the whole data set is taken as one group */

%local sortkey;
%let sortkey=bvar1 &byvars;
%if &sortkey=bvar1 %then %do;
  %let grpvar=&sortkey;
%end;

 data _t1;
   set &inds;
   bvar1=1;
 run;

/*** Add a current by group key, current observation in the by group key, */
/*** KEY0, and an nth observation within the group key KEYn – subtract n  */
/*** from KEY0*/

 proc sort data=_t1;
   by &sortkey;
 run;

 data _t2;
   set _t1;
   by &sortkey;
   retain grpkey key0 0;
   grpkey=grpkey+first.&grpvar;
   key0=key0*(first.&grpvar=0)+1;
   key&n=key0-&n;
 run;

/*** Create a copy of the data set sorted by group (GRPKEY) and the   */
/*** observation within the group (KEY0). Do not keep KEYn here       */

 proc sort data=_t2 out=_t3(drop=key&n);
   by grpkey key0;
 run;

/*** Create a key values data set sorted by group (GRPKEY) and the nth   */
/*** observation (KEYn) within group renamed as KEY0. Rename VAR as the */
/*** column to contain the carried backward values. No need to specify  */
/*** a length when using a character variable */
```

```
 proc sort data=_t2(keep=grpkey key&n &var) out=_t4(rename=(key&n=key0
&var=&nxtvarcb));
   by grpkey key&n;
 run;

/*** Join by the keys, the first n observations in each group in _t4 are */
/*** dropped, the last n will have missing NXTVARCB (no following        */
/*** observations to carry back) */

 data _t5;
   merge _t3(in=a) _t4(in=b);
   by grpkey key0;
   if a;
 run;

/*** Resort by original BY variables, drop the temporary key variables */

 proc sort data=_t5 out=&outds(drop=bvar1 grpkey key0);
   by &sortkey;
 run;

/*** Remove the temporary data sets */

 proc datasets nolist;
   delete _t1-_t5;
 run;

%mend leadn;

/*** Example macro call ***/

%leadn(inds=weight,
       outds=ptwtcb,
       byvars=treatment patnum visitnum,
       grpvar=patnum,
       n=1,
       var=wtkg,
       nxtvarcb=nxtwtcf);
```

In this example, a data set PTWTCB is created as a copy of the input data set WEIGHT. A new column, NXTWTCF, which will contain the carried backwards values, is added to PTWTCB. The BY variables used to sort the observations are TREATMENT, PATNUM, and VISITNUM. The grouping variable GRPVAR is PATNUM, so the carrying backwards will be performed within each group of like PATNUMs. N is set as 1 because the value from the next observation, that is the first following observation, is being referenced. The column whose values are to be used for carrying backwards is WTKG.

## DATA _NULL_

The SAS System allows a programmer to run a DATA step without creating an output data set. This is done by using the keyword _NULL_ instead of a data set name on the DATA statement. DATA _NULL_ can be useful for printing or manually manipulating variables and texts. This gives the programmer a great deal of control, though for printed output PROC REPORT and SAS ODS in recent versions of SAS have become easier options.

## PLACING A RUN TIME VALUE INTO A MACRO VARIABLE

Assigning a variable in a DATA step a value held in a macro variable is straightforward, the statement below will assign the resolved value of the macro variable STUDY into the character variable STUDYID.

```
studyid=”&study”;
```

Note the use of the ampersand to resolve the macro variable and the use of double quotation marks because STUDY is alphanumeric. In SAS, macro compilation takes place before DATA step, or SAS procedure compilation and execution hence the value of a macro variable is known ahead of run time. Placing a run time value in a macro variable may appear as not possible since the run time value of a variable is not known until the DATA step executes. However, at macro compilation time, the SAS system creates a symbol table of macro variables referred to in the program at macro compile time. Later, at run time the contents of a SAS variable can be placed in a macro variable set up in the macro symbol table. This is achieved using the SYMPUT function.

In the example below the number of distinct patients in the data set DEMWGHT is counted using the variable PC. At macro compilation time the symbol NPATS is created in the macro symbol table and it is populated at run time with the value of PC. Note the use of DATA _NULL_ so an output data set is not created, and of the end flag FINALREC. On reading the last observation FINALREC is set and the value of PC is placed in the macro variable NPATS. The LEFT function is to remove leading spaces after converting the numeric PC to character. If the PUT function was not used an implicit conversion note would be written to the LOG file because the formatting is undefined (Any valid SAS format may be used with PUT, hence a formatted value can be put into the macro variable.). The %PUT line following this DATA step prints the value of NPATS to the LOG file. An important point to note here is that NPATS does not have its value assigned until after the DATA step has completed, hence NPATS will not have a resolution later in the same DATA step.

```
data _null_;
   set demwght end=finalrec;
   by treatment patnum;
   retain pc 0;
   pc=pc+first.patnum;
   if finalrec then do;
     call symput('npats',left(put(pc,8.)));
   end;
run;

%put Number of Patients in DEMWGHT is &npats;
```

Many SAS programmers will be familiar with the SELECT INTO: construct in PROC SQL. Coding this is somewhat cumbersome for selecting a single value into a macro variable, however this PROC SQL construct, used with SEPARATED BY, is most probably a better option for selecting a whole sequence of values, such as a string of patient IDs, as a list into a single macro variable.

## IMPROVING RUN TIME PERFORMANCE WITH INDEXES

Run time performance becomes important when large volumes of data have to be processed quickly. Efficiently structured code certainly helps but there is a DATA step option, INDEX which allows the programmer to specify ‘preferential treatment’ to one or more of the data set columns. An INDEX may be defined for one or multiple variables. Examples are:

```
data qall(index=(patnum)) /* Builds an internal index using PATNUM */;
  set qol;
run;
```

```
data qall(index=(qlrefind=(patnum testid))); /* Builds an index QLREFIND with
  PATNUM and TESTID */
  set qol;
run;
```

A simple index uses a single variable, a composite index uses two or more variables. The variable names and the index name must be enclosed in parenthesis. The index created is internal to the SAS system, although some run time overhead is involved creating the index structure, once this has been built accessing the variables in subsequent DATA steps and SAS PROCedures is much faster. The best time to use an index is when setting up a data set with a very large number (1000+) of observations which will subsequently be repeatedly accessed and resorted (An index should be used on the BY variables). Once an index is created it does not have to be referred to again in any subsequent DATA steps or PROCedures.

There are a few points to note about INDEXes. PROC SORT does not use the index and so will not execute any faster, but any subsequent DATA steps or other SAS PROCedures using an indexed BY variable will benefit. INDEXes can also be used with PROC SQL, however, benchmark tests have shown a PROC SQL index and a SAS DATA step index often conflict with other and do not improve run time performance. There is a /UNIQUE option which only allows unique values of the variable(s) specified as the index. A data set with such an index cannot have any observations with duplicate index variable values added to it. There is also a /NOMISS option which excludes missing values of the index variable(s) from the index, this should be used when a data set has most of the index variable(s) values missing. /UNIQUE and /NMISS cannot be used together in the same index.

## CONCLUSIONS

This paper has demonstrated some extra methods for processing data set observations, it is not exhaustive, but shows insight into useful and perhaps less well known programming techniques. Additional information about the topics discussed and information on other SAS DATA step articles is available on the SAS HELP webpages.

## ACKNOWLEDGMENTS

The author would like to express gratitude to the following for their assistance in writing and reviewing this paper:

Jeffrey Dickinson, Sharon Hall, and Kathy Greer at DataCeutics, Inc. 215 West Philadelphia Avenue, Boyertown, PA, 19512 | 610.970.2333 | info@dataceutics.com.

Paul Slagle, Amie Bissonett and Eric Kammer, PharmaSUG 2018 Academic Chair, at AcademicChair@PharmaSUG.org

## REFERENCES AND SUGGESTED FURTHER READING

Lora D. Delwiche and Susan J. Slaughter, 1998, “My Little SAS Book”, © SAS Institute, Cary, NC, USA.

Henri Theuwissen, BI Knowledge Sharing, Belgium, SAS Global Forum 2011,Denver CO, USA.”Don’t Waste Too Many Resources to get your Data in a Specific Sequence”

H Ament, MSD, Oss, The Netherlands, Phuse international conference 2011, Brighton, UK, “Learning to Love the SAS Lag Function”

Christianna Williams, PhD, North-East SAS Users Group conference 2004, Baltimore, MD, USA,”SYMPUT and SYMGET: Getting DATA Step Variables and Macro Variables to Share”.

Timothy J. Harrington, Pharamasug 2017, Baltimore, MD, USA, “A Macro to Carry Values Through Observations Forwards or Backwards over missing or null values within a BY Group or a SAS Data Set”

## CONTACT INFORMATION

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

Timothy J. Harrington
DataCeutics, Inc.
215 West Philadelphia Avenue,
Boyertown,
PA 19512
(610) 970 2333
harringt@dataceutics.com