Paper 30

How to DoW

Paul M. Dorfman, Independent Consultant

SUMMARY

The DoW-loop is a nested, repetitive DATA step structure enabling you to isolate instructions related to a certain break event before, after, and during a DO-loop cycle in a naturally logical manner. Readily recognizable in its most ubiquitous form by the DO UNTIL(LAST.ID) construct, which readily lends itself to control-break processing of BY-group data, the DoW-loop's nature is more morphologically diverse and generic. In this workshop, the DoW-loop's logic is examined via the power of example to reveal its aesthetic beauty and pragmatic utility. In some industries like Pharma, where flagging BY-group observations based on in-group conditions is standard fare, the DoW-loop is an ideal vehicle greatly simplifying the alignment of business logic and SAS® code. In this workshop, the attendees will have an opportunity to investigate the program control of the DoW-loop step by step using the SAS DATA step debugger and learn of a range of nifty practical applications of the DoW-loop.

I. DOW-LOOP: THE CIRCUITRY

1. Intro

What is the DOW-loop? It is not a "standard industry term". Rather, it is an abbreviation that has become common in the SAS-L circles in the last few years or so. Here is the short story.

Once upon a time on SAS-L, the renowned Master of the SAS Universe Ian Whitlock, responding to an inquiry [1], casually used a DO-loop in a certain peculiar manner, of which one of the authors (*PD*) took notice - primarily because he failed to fully comprehend it at once. However, knowing that Ian never codes anything (let alone posting it on a public forum) without a good reason, he kept thinking about the construct, eventually realizing the potential, elegance and programmatic power hidden in such a structure, and started using it in his own daily work and propagandizing it on SAS-L and beyond. Thus this loop structure has become the center of a number of his own applications, as well as application suggested by the other author (*LS*). At some point the construct seemed to started meriting its own distinctive name. The combination of DO*[-loop]* and W*[hitlock]* into DOW appeared natural enough due to already established name recognition of the moniker, so *PD* offered it on SAS-L, and after a while, the name took root.

Extensive practical usage and experimenting have helped understand the structure still better and given birth to a number of other bells and whistles *PD* has subsequently added to the structure himself. The enthusiastic embrace of the DOW-loop by SAS experts of the caliber of Jack Hamilton, David Cassell, , Richard DeVenezia, Toby Dunn, Venky Chakravarthy, Judy Loren, Ken Borowiak, Howard Schreier, and Ian Whitlock *per se*, has made the method a de facto standard SAS Data step programming technique.

2. The General Scheme

Let us consider the following Data step structure:

```
Data ... ;
   <Stuff done before break-event> ;
   Do <Index Specs> Until ( Break-Event ) ;
      Set A ;
      <Stuff done for each record> ;
   End ;
   <Stuff done after break-event... > ;
Run ;
```

The code between the angle brackets is, generally speaking, optional. We call the structure the DOW-loop.

The intent of organizing such a structure is to achieve logical isolation of instructions executed between two successive *break-events* from actions performed before and after a break-event, and to do it in the most programmatically natural manner. In most (although not all) situations where the DOW-loop is applicable, the input data set is grouped and/or sorted, and the break-event occurs when the last observation in the by-group has been processed. In such a case, the DOW-loop logically separates actions which can be performed:

1. Between the top of the implied loop and before the first record in a by-group is read.
2. For each record in the by-group.
3. After the last record in the by-group has been processed and before the bottom of the implied loop.

3. The Scheme by Example

Let us assume that an input SAS data file A is sorted by ID. The step below does the following:

1. Within each ID-group, multiplies and summarizes VAR values for all observations within the group.
2. Counts the number of all records and records with non-missing values of VAR for each ID-group.
3. Computes the average value of VAR across the each ID-group.
4. Writes a single record (with COUNT, SUM, MEAN, PROD) per each ID-group to file B.

```
Data B ( Keep = Id Prod Sum Count Mean) ;

   Prod = 1 ;

   Do Count = 1 By 1 Until ( Last.Id ) ;
      Set A ;
      By  Id ;
      If Missing (Var) Then Continue ;
      Mcount = Sum (Mcount, 1) ;
      Prod = Prod * Var ;
      Sum = Sum (Sum, Var) ;
   End ;

   Mean = Sum / Mcount ;
Run ;
```

Now let us see how it all works in concert:

1. Between the top of the implied Data step loop and the beginning of an ID-group: PROD and COUNT are set to 1, and the non-retained SUM, MEAN, and MCOUNT are set to missing by the default action of the implied loop (program control at the top of the implied loop).
2. Between the first iteration and break-point: DOW-loop starts to iterate, reading the next record from A at the top of each iteration. While it iterates, control never leaves the Do-End boundaries. If VAR is missing, CONTINUE passes control straight to the bottom of the loop. Otherwise MCOUNT, PROD and SUM are computed. After the last record in the group is processed, the loop terminates.
3. Between the break-point and the bottom of the step: Control is passed to the statement following the DOW-loop. At this point, PROD, COUNT, SUM, and MEAN contain the group-aggregate values. MEAN is computed, and control is passed to the bottom of the implied loop, where the implied OUTPUT statement writes the record to file B. Control is passed to the top of the implied loop, where non-retained, non-descriptor variables are re-initialized, and the structure proceeds to process the next ID-group.

Note that contrary to the common practice of doing this sort of processing, the accumulation variables need NOT be retained, nor nullified before each BY-group. Because the DOW-loop passes control to the top of the Data step implied loop *only* before the first record in a by-group is to be read, this is the only point where non-retained variables are reset to missing values. But this is exactly where we need it, so that the programmatic form of the construct and its business logic work together towards the purpose of the step as a synchronized pair.

4. Programmatic-Logical Coherence

What makes the DOW-loop special? It is all in the logic. The construct programmatically separates the before-, during-, and after-group actions in the same manner and sequence as does the stream-of-the-consciousness logic:

1. If an action is to be done before the group is processed, simply code it before the DOW-loop. Note that is unnecessary to predicate this action by the IF FIRST.ID condition.
2. If it is to be done with each record, code it inside the loop.
3. If is has to be done after the group, like computing an average and outputting summary values, code it after the DOW-loop. Note that is unnecessary to predicate this action by the IF LAST.ID condition.

In the example above, an iterative Do-loop is nested within the implied Data step loop and fits in it like a glove. As a result, there is no need to retain summary variables across observations. While the DOW-loop iterates, they hold their values. When the last record in the BY-group has been processed, program control hits the implicit OUTPUT statement at the bottom of the implied loop and output the single record per group as required without the need of any conditional statements. When and only when program control is then passed back to the top of the implied loop, the non-retained variables are reset to missing values, so before the next by-group starts processing, they turn out to be naturally reinitialized.

5. Control-Break Processing

A closer look at the general DOW-loop scheme outlined above reveals that its programmatic and logical scope is not limited to by-processing. Rather, it is a natural tool for control-break processing. By control-break processing, programmers usually mean an arrangement where a group of data elements (records, observations, array items, etc.) has a boundary expressed by some condition.

In the example above, the condition was in the form of LAST.ID, but it does not have to be. For instance, imagine that data set CB has a variable CB occasionally taking on a missing value, for example:

```
Data CB ;
   Do CB = 9, 2, .A, 5, 7, .Z, 2 ;
      Output ;
   End ;
Run ;
```

Suppose we need to summarize the values of CB until it hits a missing value, and print the rounded average of CB across consecutive non-missing records each time it happens. Here is an excerpt from the log:

```
 Data CBSum ;
    Do _N_ = 1 By 1 Until (NMiss(CB) | Z) ;
       Set CB End = Z ;
       Sum = Sum (Sum, CB) ;
    End ;
    Mean = Round (Sum / (_N_- 1 + Z), .01) ;
    Put (Sum Mean) (= 4.2) ;
 Run ;
-----------------------------------------
Sum=11.0 Mean=5.50
Sum=12.0 Mean=6.00
Sum=2.00 Mean=2.00
```

Above, the control-break event has nothing to do with BY-processing, and hence there is no BY statement anywhere in the step. Instead, the DOW-loop boundaries are set by the condition that CB is missing (or else the end-of-file is reached).

Note the use of the automatic variable _N_ as the DOW-loop counter. We shall dwell on the subject more in the subsection dedicated to the effect later on.

II. DOW-LOOP: SPECIAL FORMS AND EFFECTS

1. Intro

Effects associated with DOW-loop programming are practically as diverse as Data step programming itself. In this section, we will be able to discuss only an small fraction of them, more or less randomly selected according to our likings.

2. Do _N_ = 1 By 1 Until…

Earlier, we have already mentioned the use of automatic variable _N_ as a DOW-loop counter. Let us consider another example:

```
data dow ;
   input id num @@ ;
   cards ;
3 1   3 2   3 3   2 4   2 5   1 6
;
run ;
```

```
Line 1. data dow_agg (keep = id summa average) ;

Line 2.    put 'before DoW: ' id= _n_ = ;

Line 3.    do _n_ = 1 by 1 until (last.id) ;
Line 4.       set dow ;
Line 5.       by descending id ;
Line 6.       summa = sum (summa, num) ;
Line 7.    end ;

Line 8.    average = summa / _n_ ;

Line 9.    put 'after  DoW: ' id= _n_= / ;

Line 10.  *implied OUTPUT here ;
Line 11. run ;
---------------------------------------
before DoW: id=. _N_=1
after  DoW: id=3 _N_=3

before DoW: id=3 _N_=2
after  DoW: id=2 _N_=2

before DoW: id=2 _N_=3
after  DoW: id=1 _N_=1

before DoW: id=1 _N_=4

---------------------------------------
before DoW: id=. _N_=1
after  DoW: id=3 _N_=3

before DoW: id=3 _N_=2
after  DoW: id=2 _N_=2

before DoW: id=2 _N_=3
after  DoW: id=1 _N_=1

before DoW: id=1 _N_=4
---------------------------------------
NOTE: There were 6 observations read from the data set WORK.DOW.
NOTE: The data set WORK.DOW_AGG has 3 observations and 3 variables.
```

Now let us walk statement by statement through the step to see why exactly it prints to the log what it prints and outputs to DOW_AGG what it outputs:

- Program control (PC) starts at the DATA statement, then proceeds to line 2. This is the top of the implied loop, so the non-retained variables SUMMA and AVERAGE are set to missing values. At this point, no records have been read yet from DOW, so ID=. As this is the first iteration of the implied loop, _N_=1. Thus the 'before' PUT statement prints ID=. and _N_=1.
- PC enters the DOW-loop at line 5 and sets _N_=1. It was the same anyway, so this time, no change here. Then SET reads the first record from DOW at line 6 and sticks its values into the PDV. Next, PC proceeds to line 8. At this point, SUMMA=. because it was set this way automatically before the loop. The SUM function adds NUM=1 to it, making SUMMA=1. Next, PC is at the bottom of the DOW-loop, so it has to check the UNTIL condition LAST.ID; since this is not the last record in the BY-group, the condition is false, and PC goes back to the DO statement.
- _N_ is incremented by 1 to _N_=2, and SET reads the next record. NUM=2 is added to SUMMA=1. (Note that SUMMA at this point has the value it acquired during the previous iteration despite the fact that it is not retained. This is because PC did not reach the top of the implied loop where non-retained variables are reset to missing values, but only the top of the DOW-loop, where there is no such default action.) This makes SUMMA=3, and again, PC loops back to the DO statement.
- _N_ is bumped up to _N_=3, SET reads the last record in the BY-group, and the SUM function makes SUMMA=3+3=6. Now this IS the last record in the group, so LAST.ID=1, the UNTIL condition is true, and PC leaves the loop.
- At this point we have ID=3, SUMMA=6, and _N_=3, so the average computed at line 11 results in 6/3=2. Accordingly, the diagnostic PUT at line 13 reports ID=3, SUMMA=6.

- Now PC is at the bottom of the implied Data step loop. *Since there is no explicit OUTPUT statement anywhere in the step, the implicit OUTPUT statement at the bottom of the step is executed*, and the first record (corresponding to the first BY-group with ID=3) is written to data set DOW_AGG.
- After that, PC has nothing else to do at the bottom of the step. Since there is still unread input from DOW in the step (3 records left to be read), PC loops back to the top of the step. Here SUMMA and AVERAGE are obviously set to missing, but what will happen to _N_? Remember that at the bottom of the implied loop we had _N_=3. If, as some think, _N_ is incremented at the top of the step, then 'before' PUT would now print _N_=4, right? But this does not happen, and what PUT actually prints is _N_=2. For the time being, let us just note this remarkable fact and continue; we will explain it at due time later.
- PC is now at the 2nd iteration of the implied loop and enters the DOW-loop for the second time. Now its actions repeat those during the first iteration to the tee, except that the values read from DOW and computed as a result are different: after PC leaves the DOW-loop, we have _N_=2 (since there are 2 observations in the ID=2 BY-group) and SUMMA=9, so AVERAGE=4.5. The implicit OUTPUT writes the second record with ID=2 and the aggregates to DOW_AGG, and PC loops back to the top of the step.
- The same pattern repeats in the 3d iteration of the implied loop, at the bottom of which now _N_=1 (just one record in the BY-group); the record for ID=1 with SUMMA=6 and AVERAGE=6 is output to DOW_AGG, and PC loops back to the top of the step again.
- Here the default actions reinitialize non-retained variables, _N_ is apparently set to _N_=4, PC enters the DOW-loop again and executes the SET statement. This time, however, all records have already been read, the buffer set for DOW is empty, and the attempt to read from the empty buffer terminates the Data step immediately.

A printout of the output file DOW_AGG shows:

```
id    summa    average
----------------------
 3      6        2.0
 2      9        4.5
 1      6        6.0
```

Let us get back to the _N_-question. It is apparent that no matter matter what happens to _N_ within the step - here it is used just as another numeric variable for the purpose of counting records in each BY-group - at the top of the step it is invariably set to the number of iterations of the implied loop. This can only happen if the Data step has an internal counter of the implied loop iterations and simply moves its next value to _N_ once program control is once again at the top of the step.

This also means that if the top-of-the-implied-loop value of _N_ is not specifically relied upon within the step boundaries, it can be used absolutely safely for any purpose just like any other numeric variable. We will see other examples of making use of _N_ in such manner (to synchronize consecutive DOW-loops).

Such use has the added benefit of not having to worry about dropping _N_ - as an automatic variable, it is dropped automatically. From the standpoint of defensive programming, this is an advantage because loop counters accidentally left in an output file are known to make people pull their hair trying to find out what is wrong with a step trying to process the file afterwards.

3. DO _N_=1 BY 1 UNTIL (EOF): a DOW-Loop?

Consider the following construct:

```
data something ;
   <...SAS statements...>
   do _n_ = 1 by 1 until (eof) ;
      set input end = eof ;
      <...more SAS programming statements...>
   end ;
   <...still more SAS statements...> ;
   stop ;
run ;
```

It is an extremely useful construct in its own right, and was expounded upon in great detail in [1]. Since this form of the DO-loop had existed and been in fairly wide use before the momentous SAS-L post by Ian, the coiner of the DOW-loop moniker did not originally include it in the DOW-loop realm. However, a number of active SAS experts (notably, David Cassell) eventually started doing so due to the apparent similarities between the two.

From the general point of view, there is indeed a good reason for the inclusion. After all, both constructs:

- Contain an explicit DO-loop wrapped around a statement reading sequentially from some data source.

- Terminate when a break-point specified in the UNTIL condition is reached.

Ultimately, the similarity between the two constructs appears even more intimate if we consider that essentially, a Data step views each BY-group as a separate virtual file, with its boundaries defined by the beginning of the group and LAST.ID=1. But DO UNTIL(EOF) structure effectively treats the entire file as a single BY-group, its boundaries defined by the beginning of the file and EOF=1. Thus, DO UNTIL(EOF) can be truly considered as a single BY-group limit case of the "true" DOW-loop.

4. Double DOW-Loop

Let us consider the example from the section above once again, but with an extra twist: This time, in addition to computing the aggregates for each BY-group (ID value), we need to merge these statistics with the original file in the same step. In other words, in the new output we want all the original DOW records augmented with aggregate columns, like so:

```
id    num    summa    average
-----------------------------
 3     1       6         2.0
 3     2       6         2.0
 3     3       6         2.0
 2     4       9         4.5
 2     5       9         4.5
 1     6       6         6.0
```

Of course, this can be easily achieved using PROC SQL or another DATA step with MERGE, but our purpose here is not to merely solve the problem but, more importantly, demonstrate the concept.

It seems pretty obvious that to attain the goal we will have to read the file twice no matter what (SQL does it behind-the-scenes, but still does). It is also fairly apparent that algorithmically, it should be as simple as this:

- Read the first BY-group one observation at a time and compute the aggregates, just as it was done in the preceding section.
- Read the group one record at a time once again, only now instead of writing the output after the group is over, write out each record right after it is read, dragging along the aggregate values computed during the first pass through the group.

Seems simple, but how can we make the Data step read each BY-group twice in succession and each time - precisely the same number of times? The concept behind the following step was offered by Howard Schreier:

```
Line 1.  data dow_agg () ;

Line 2.    do _n_ = 1 by 1 until (last.id) ;
Line 3.       set dow ;
Line 4.       by descending id ;
Line 5.       summa = sum (summa, num) ;
Line 6.    end ;

Line 7.    average = summa / _n_ ;

Line 8.    do until (last.id) ;
Line 9.       set dow ;
Line 10.      by descending id ;
Line 11.      output ;
Line 12.   end ;
Line 13. run ;
--------------------------------------------
NOTE: There were 6 observations read from the data set WORK.DOW.
NOTE: There were 6 observations read from the data set WORK.DOW.
NOTE: The data set WORK.DOW_AGG has 6 observations and 4 variables.
```

How does THIS work? Basically, exactly the same way as it was outlined in the algorithmic sketch above. The secret here is to make use of TWO separate, identical SET statements, each wrapped in its own DOW-loop.

Remember that for each SAS data set named in a syntactically correct I/O statement (SET, MERGE, UPDATE, or MODIFY), the Data step compiler sets up its own buffer. The effect of having data set DOW named twice in the step above is thus the same as if there were two completely independent virtual data sets DOW1 and DOW2, both identical to data set DOW. Now let us pick this step's modus operandi apart:

- Program control (PC) starts the implied loop at line 1. As this is the top of the Data step, non-retained variables SUMMA and AVERAGE are set to missing values, _N_ is set to 1.
- PC enters the first DOW-loop at line 3 and proceeds to iterate through the first BY-group (ID=3) exactly as described in the sequence of events in section 2. Note (important!) that every time PC hits the SET statement, it reads the next record from DOW1, the first virtual copy of data set DOW. Thus after PC exited the DOW-loop, the first 3 records from DOW1 have been read, and the statistics for that group is accumulated. PC proceeds to line 9 and calculates the average.
- PC enters the second DOW-loop and hits the SET statement. However, now SET reads a record from DOW2, the second virtual copy of DOW. Since nothing has been read from that copy yet, SET reads the 1st record from the 1st BY-group (ID=3). In this DOW-loop, though, instead of computing the sum, we have the explicit OUTPUT statement, so it ejects the current PDV content to DOW_AGG. Now remember that before the loop started, SUMMA and AVERAGE had already been computed and are now sitting in the PDV. The iterations of the second DOW-loop have no effect on their values, since they are modified neither inside this DOW-loop nor by the implied Data step loop - for in the process, PC never reaches its top. Hence, the record is output with its current values read from DOW2, and the aggregate values computed before the loop started. The second DOW-loop then continues to iterate, reading the rest of the records from the DOW2's first BY-group (ID=3), spitting them out to DOW_AGG together with the aggregates computed beforehand. Thus, at the end of the second loop the content of DOW_AGG is as follows:

```
id    num    summa    average
-----------------------------
 3     1       6        2.0
 3     2       6        2.0
 3     3       6        2.0
```

- At this point, PC is at the bottom of the implied Data step loop, and both buffers DOW1 and DOW2 have been depleted by 3 records with ID=1 (first BY-group). This time, however, the implicit OUTPUT statement is turned off by virtue of the explicit OUTPUT statement being present elsewhere in the step (in the second DOW-loop), so no output happens at this time, and PC simply loops back to the top of the implied loop.
- There, non-retained aggregates SUMMA and AVERAGE are set to missing values, _N_ is set to 2 (which is totally inconsequential for the purpose of the step), and program control repeats the same action as it performed in the first iteration of the implied loop: reads the 2nd BY-group (ID=2) from buffer DOW1 and computes aggregates using the first DOW-loop; then reads the 2nd BY-group (ID=2) from buffer DOW2, writing out each record to DOW_AGG; finally PC exits the second DOW-loop and goes back to the top of the Data step, at which point, the second BY-group has been added to DOW_EGG.
- In the same manner, in the third iteration of the implied loop the third BY-group from each buffer DOW1 and DOW2 are processed. Before PC moves back to the top of the implied loop again, DOW_AGG arrives at its final content:

```
id    num    summa    average
-----------------------------
 3     1       6        2.0
 3     2       6        2.0
 3     3       6        2.0
 2     4       9        4.5
 2     5       9        4.5
 1     6       6        6.0
```

- At this point, after the standard motions at the top of the implied loop, PC enters the first DOW-loop for the 4th time. But by the time, all records from both buffers (virtual files) DOW1 and DOW2 have been read, and the buffers are void. The SET statement in the first DOW-loop tries to read from the empty buffer DOW1, and it immediately causes the step to cease and desist.

This course of control flow is indirectly supported by the two LOG notes testifying that 6 records were read by 2 different SET statements.

5. Double-DOW: The Synchronism

The most important question arising from the sequence of events described above is: How does the double DOW manage to ensure the synchronism between the two consecutive DOW-loops? In other words, how does it make certain that the second DOW-loop reads from DOW2 exactly the same number of observations the first DOW-loop has just read from DOW1? To understand why it is important, let us imagine for a moment that for some BY-group, the second DOW-loop read fewer records from its buffer DOW2 than the first DOW-loop read from its buffer DOW1. It that happened, DOW1 would be exhausted sooner than DOW1, and the first SET would try reading from the empty buffer DOW1 whilst the buffer DOW2 still has records in it. Needless to say, that would not only result if fewer records in DOW_AGG than necessary, but also the aggregates computed in the first DOW-loop would be attached to wrong records read from the second DOW-loop.

So what makes the two DOW-loops synchronize? It is the fact that each SET statement has its own BY statement, and the BY-statements are identical. Suppose for a moment that the BY statement is left out from the second loop. By the time PC enters the second loop, the value of LAST.ID=1 because the first DOW-loop has just read the last record from its current BY-group. But that in turn means that the second DOW-loop will end immediately after the very first iteration: once program control is at the bottom of the DO-UNTIL loop, the UNTIL condition is found to be true, passing PC past the closing END.

Now it should be easy to understand that the second BY is not necessarily the only way of keeping the double DOW-loop in synchronism. After all, we only have to ensure that the second DOW-loop reads the same number of records from the second buffer. For example, here is an alternative, resulting in the same DOW_AGG1 as DOW_AGG before:

```
data dow_agg1 () ;
   do _n_ = 1 by 1 until (last.id) ;
      set dow ;
      by descending id ;
      summa = sum (summa, num) ;
   end ;

   average = summa / _n_ ;

   do _n_ = 1 to _n_ ;
      set dow ;
      output ;
   end ;
run ;
---------------------------------------
NOTE: There were 6 observations read from the data set WORK.DOW.
NOTE: There were 6 observations read from the data set WORK.DOW.
NOTE: The data set WORK.DOW_AGG1 has 6 observations and 4 variables.
```

How does this take care of business in lieu of the extra BY statement and even without the second UNTIL? This is how:

- The first DOW-loop runs its business as usual, but what is important is that after program control (PC) has left the loop, _N_ is set to the number of records just read from the first DOW buffer (called above DOW1). For instance, after the first BY-group and before PC enters the second DOW-loop, _N_=3 (there were 3 records in the group with ID=3).
- After computing the mean, PC now enters the second DOW-loop. Because of the way the DO-loop work, first the upper index bound is set to the value to which the upper bound expression resolves. Because the upper bound expression is coded as _N_, it resolves to 3, and hence that is what the upper bound will remain for the entire duration of the loop (across all of its iterations). Now although in the first iteration of the DOW-loop index control sets _N_ to 1, it has no effect on the already set upper bound value.
- From this point on, the loop index, varyting from 1 to 3, will cause the loop to iterate 3 times, i.e. exactly the number of times equal to the value _N_ had had before PC entered the loop. Quod erat demostrandum.

As a practical proof, the step produces data set DOW_AGG1 exactly identical to DOW_AGG, and the log notes and the number of records read from the two input DOW streams are also identical.

6. Multi DOW-Loop

The concept of the double DOW-loop is now not difficult to extend. Suppose that for whatever whimsical reason, in one of the double-DOW steps above, we want to print in the log the value of ID for each BY-group *before* the first DOW-loop has even started. (The example may seem contrived, yet often times this is a very useful and necessary thing to do; for instance, table-driven SAS applications generating SAS code by reading control tables come to mind.) The problem here is of course that the needed value of ID is not available to the PDV before the SET statement is executed, because the statement is located i*nside* the DOW-loop!

What we can do, though, is run yet another, essentially idle, DOW-loop before the first DOW-loop begins (which of course will at once make it second):

```
data dow_agg1 () ;
```

```
   do _n_ = 1 by 1 until (last.id) ;
      set dow ;
      by descending id ;
   end ;
   putlog ID= ;

   do _n_ = 1 to _n_ ;
      set dow ;
      summa = sum (summa, num) ;
   end ;

   average = summa / (_n_ - 1) ;

   do _n_ = 1 to _n_ - 1 ;
      set dow ;
      output ;
   end ;
run ;
-----------------------------------------------
id=3
id=2
id=1
NOTE: There were 6 observations read from the data set WORK.DOW.
NOTE: There were 6 observations read from the data set WORK.DOW.
NOTE: There were 6 observations read from the data set WORK.DOW.
NOTE: The data set WORK.DOW_AGG1 has 6 observations and 4 variables.
```

Because we now have 3 identical data streams being read from DOW, the SAS log duly informs us that 6 observations were extracted from each. Also, and alert reader will notice that now the denominator in the formula for the average and the upper index bound in the last DOW-loop are set to (_N_-1) instead of the former _N_. Figuring out why this is necessary is offered to the reader as a DOW-comprehension exercise.

Needless to say, a multiple DOW-loop can contain practically any number of consecutive single DOW-loops. From the performance standpoint, it is an obvious drawback because for each new loop, the input data set will have been read in full. However, sometimes there is no avoiding it (as in the example of merging statistics back with data), and sometimes the programming convenience of the multiple DOW-loop structure far overweights performance setbacks. For example, if a multiple DOW-loop is used as a SAS code generator (as mentioned above), the amount of data it has to read and write is practically totally inconsequential, so programming efficiency tramps machine efficiency.

7. DOW-Loop and Hashing Applications

Because of the programmatic structure of the DOW-loop, it is ideally suited for applications where hash tables are used to track keys or accumulate aggregates *separately for each BY-group*. To better understand why, let us consider an example:

```
data dow ;
   input id transid amt ;
   cards ;
1  11   40
1  11   26
1  12   97
2  13    5
2  13    7
2  14   22
3  14    1
4  15   43
4  15   81
5  11   86
5  11   85
;
run ;
```

Imagine that we need to output five SAS data files, amongst which the records with ID=5 belong to a SAS data set OUT1, records with ID=2 belong to OUT2, and so on. Imagine also that there is an additional requirement that each partial file is to be sorted by TRANSID AMT.

In Version 9, not only the hash object is instantiated at the run-time, but its method are also are run-time executables. Besides, the parameters passed to the object do not have to be constants, but they can be SAS variables. Thus, at any point at run-time, we can use the .OUTPUT() method to dump the contents of an entire hash table into a SAS data set, whose very name is formed using the SAS variable we need, and write the file out in one fell swoop:

```
data _null_ ;
   dcl hash hid (ordered: 'a') ;
   hid.definekey  ('_n_') ;
   hid.definedata ('id', 'transid', 'amt'   ) ;
   hid.definedone ( ) ;

   do _n_ = 1 by 1 until ( last.id ) ;
      set dow ;
      by id ;
      hid.add() ;
   end ;

   hid.output (dataset: 'OUT' || put (id, best.-l)) ;
run ;
------------------------------------------------------
NOTE: The data set WORK.OUT1 has 3 observations and 3 variables.
NOTE: The data set WORK.OUT2 has 3 observations and 3 variables.
NOTE: The data set WORK.OUT3 has 1 observations and 3 variables.
NOTE: The data set WORK.OUT4 has 2 observations and 3 variables.
NOTE: The data set WORK.OUT5 has 2 observations and 3 variables.
NOTE: There were 11 observations read from the data set WORK.SAMPLE.
```

Above, the purpose of making the group record enumerator _N_ the hash key is to make the key within each BY group distinct under any circumstances and thus output all records in the group, even if they contain duplicates by TRANSID AMT. If such duplicates are to be deleted, we only need to recode the DoW-loop as:

```
 do _n_ = 1 by 0 until ( last.id ) ;
      set dow ;
      by id ;
      hid.replace() ;
    * rc = hid.add() ; *rc = to avoid log errors on dupes ;
  end ;
```

BY 0 lets the loop iterate with *_N_* set to constant 1. To the eyes of alert readers knowing their Data steps but having not yet gotten their feet wet with the hash object in Version 9, the step above must look like a heresy. Indeed, how in the world is it possible to produce a SAS data set, let alone many, using Data _Null_? It is possible because, with the hash object, the output is handled completely by and inside the object when the .OUTPUT() method is called, the Data step merely serving as a shell and providing parameter values to the object constructor.

Now let us see how the step works in detail:

- Before the first record in each BY-group by ID is read, program control encounters the hash table declaration. It can be executed successfully because the compiler has provided the host variables for the keys and data.
- Then the DOW-loop reads the next BY-group one record at a time and uses its variables and _N_ to populate the hash table. Thus, when the DOW-loop is finished, the hash table for this BY-group is fully populated. Now PC moves to the HID.OUTPUT() method. Its DATASET: parameter is passed the current ID value, which is concatenated with the character literal OUT. The method executes writing all the variables defined within the DefineKey() method from the hash table to the file with the name corresponding to the current ID.
- The Data step implied loop moves program control to the top of the step where it encounters the hash declaration. The old table is wiped out and a fresh empty table is instantiated. Then either the next BY-group starts, and the process is repeated, or the last record in the file has already been read, and the step stops after the SET statement hits the empty buffer.

Many more examples of doing nifty SAS things by teaming DOW-loops with hash tables can be found in [2].

CONCLUSION

Despite its relatively short public SAS history, the DOW-loop has already become a powerful tool of aligning code with logic in SAS Data step programming in hands of quite a few experts. Although at first it appears peculiar to a novice SAS programmer or one who has not year given it the thought it deserves, the DOW-loop, once understood, is usually readily embraced as a tool of choice in numerous situations involving Data step BY-processing.

ACKNOWLEDGEMENTS

The author would like to thank *Ian Whitlock*, who, by using the (nameless at the time) DO***W***-loop construct on SAS-L, inspired the author to undertake serious research in the matter. He has subsequently learned that *Don Henderson* at the end of the 1980s [4] was the first to make use of the structure in a more limited sense in order to gain a certain performance edge in the DATA step. We therefore salute ***Do****n* as the pioneer. My thank you list would be incomplete without *Howard Schreier,* whose insights have greatly enriched the DOW-loop development and apparently spurred *Richard DeVenezia* to come up with the fantastic concept of the double DOW-loop, after which expanding it to multi DOW-loop was a matter of simple induction. Needless to say, Richard merits a special place in the DOW-loop history. We thank *David Cassel*l for having prodded *Paul Dorfman* to share his DOW-loop definition with the SAS community (heretofore the definition - given in this paper as part of the abstract - had been used only as a program comment in personal consulting work). *Venky Chakravarthy* deserves my gratitude for being the first to make the DOW-loop moniker part of a SUGI paper title, thereby making it "official" beyond the confines of SESUG 2002 Proceedings. *Jack Hamilton* has earned my appreciation for the enthusiasm with which he embraced the concept and for having been the first to dedicate an entire SESUG paper to the programmatic utility and aesthetics of fusing the DOW-loop with the SAS hash object. Finally, I also thank all those who went on to admit the technique into their programming arsenals, thus helping to further develop the DOW-loop into a valuable SAS productivity instrument, whose programmatic utility and beauty directly stems from its structural coherence with the DATA step's internal logic.

REFERENCES

1. Ian Whitlock. Re: *SAS novice question.* Archives of the SAS-L listserve, 16 Feb. 2000.
http://www.listserv.uga.edu/cgi-bin/wa?A2=ind0002C&L=sas-l&P=R5155.
2. Paul M. Dorfman. *The Magnificent DO*. Proceedings of SESUG 2002, Savannah, GA, 2002.
http://www.devenezia.com/papers/other-authors/sesug-2002/TheMagnificentDO.pdf*.*
3. Paul M. Dorfman, Koen Vyverman. *Data Step Objects as Programming Tools.* Proceedings of SUGI 31, San Francisco, CA, 2005.
http://www2.sas.com/proceedings/sugi31/241-31.pdf
4.Don Henderson and Merry Rabb. *The SAS Supervisor*. Proceedings of NESUG 1988, New York, NY, 1988.
http://www.lexjansen.com/nesug/nesug88/sas_supervisor.pdf

AUTHOR CONTACT INFORMATION

Paul M. Dorfman
4437 Summer Walk Ct.,
Jacksonville, FL 32258
(904) 260-6509 (land)
(904) 226-0743 (cell)
sashole@gmail.com