Take Control:
Understanding and Controlling Your Do-Loops

Sarah Woodruff, Westat Inc., Rockville, MD
Toby Dunn, AMEDDC&S (CASS), San Antonio, TX

ABSTRACT

The term "loop" describes any control structure that causes a set of programming logic to be executed iteratively. Not knowing when and how to use these structures properly cause many to produce SAS® code that is at best an illegible jumble and at worst a virtually useless quagmire. This paper performs an in-depth examination of the underlying concepts involved in DO-loop construct theory, including basic constructs such as DO While and DO Until loops as well as advanced constructs such as the DO-loop of Whitlock (DoW). In addition, the paper describes sound yet simple methods to help determine which loop is needed and rules for how to easily create them.

"Using loops is one of the most complex aspects of programming; knowing how and when to use each kind of a loop is a decisive factor in constructing high-quality software" - Steve McConnell (Code Complete, 2nd Edition).

Keywords: DO-Loop, DO While, DO Until, DoW

Introduction

One of the most basic concepts that every SAS® programmer learns is how data flows through a DATA Step. Most simply, SAS reads one record in to memory, does something to it, writes it out, then takes the next record and repeats these steps. This iterative process is made possible by an unseen do-looping action and underscores the importance that DO-loops play in our everyday programming lives. It could be said that DO-loops are the most important concepts in high level languages, for they free up the programmer's mind and time as well as reducing possible errors.

This paper will move the discussion of DO-loops in SAS beyond only using them for arrays as it begins with the underlying conceptual basis. It will then cover implicit vs. explicit DO-loops, iterative vs. infinite DO-loops, entering and exiting a DO-loop, the DO-loop Of Whitlock, how to write a DO-loop from the inside out and finally wrap up with a brief overview of %DO-loops along with their differences from data step DO-loops. The goal is to cover DO-loops from stem to stern in an effort to enhance the skills needed in your job.

Propaedeutics

Every DO–loop consists of three parts: beginning, middle, and end. Let's begin with the general framework:

1. DO Index = ***start*** <TO ***stop***> <BY ***increment***> <WHILE(***expression***) | UNTIL(***expression***)>;
2. <SAS Code>
3. End ;

Item 1 is the top of the loop; it is here that an index variable will be specified and the exit conditions of the loop will be established. Item 2 is the code that the loop will iterate over; this code can be static or modified in part by the iterations of the loop. Item 3 is the "end" of the loop; however, many people like to refer to this as the bottom rather than the end because the test to exit can be anywhere within the DO-loop.

The basic flow of the DO-loop is as follows:

1. First, the Index, TO, And BY values are resolved and set. If a WHILE or UNTIL expression are present, the exit conditions are set. This means you cannot change the value(s) of those conditions from within the DO-loop itself.
2. If a TO condition is present then the value of the counter is compared to the index starting value. If the counter value exceeds the Index starting value then the DO-loop is exited. If a WHILE condition is present the specified expression is evaluated. If that expression returns false then the loop is exited.
3. If the loop has not been exited prior to this point then the body of loop will execute.
4. If there is a CONTROL or LEAVE statement in the body of the loop and it executes then the DO-loop will terminate.

5. When the DO-loop reaches the end of the statements and an UNTIL expression is present then the exit expression is tested. If it is found to be true then the DO-loop is exited; if not then the DO-loop moves control back to the top.

This covers the three basic parts of the DO-loop, including how it determines where to stop. Now let's turn our attention to the basic types of DO-loops we may encounter in SAS. We have two basic types of DO-loops, implicit and explicit. Within these two main types, there are two subtypes, iterative and infinite. The only implicit DO-loop in SAS that can be controlled to some extent is the DATA step internal loop, which is infinite. Explicit DO-loops can be iterative or infinite. In the next few sections we will take each one of these and explain them further.

Implicit Loops

One of the hallmarks of the SAS language and more specifically the DATA step is its implied looping action to read raw or SAS data. While other advanced programming languages have a similar functionality none have attained the legendary status that it has in the SAS language. While the authors are sure there are many implied processes strewn through other procedures and functions, it is the implied loop of the DATA step that is the most famous and thus it is the one we will use to illustrate implicit DO-loops. Pay close attention to this section because later in the paper we will use this implied looping action to show you how to do some really cool data processing.

An implicit loop is nothing more than a loop performed by the language without the programmer specifically writing it in the code. The need for the DATA step to read each observation, whether that observation comes from an external file or SAS data set, is implied by the fact that a DATA step is being used by the programmer. Since the DATA step either cannot determine the number of loops to perform or cannot do so efficiently, the implied DO-loop is an infinite one, with orders to exit the looping action when there is no more data to be read. Let's look at an example of code with its accompanying log to help illustrate this point:

```
Data One ;
      Do I = 1 To 10 ;
            Output ;
      End ;
Run ;

Data Two ;
      Put 'Before:' _All_ ;
            Set One ;
      Put 'After:' _All_ ;
Run ;
```

```
Before:I=. _ERROR_=0 _N_=1
After:I=1 _ERROR_=0 _N_=1
Before:I=1 _ERROR_=0 _N_=2
After:I=2 _ERROR_=0 _N_=2
Before:I=2 _ERROR_=0 _N_=3
After:I=3 _ERROR_=0 _N_=3
Before:I=3 _ERROR_=0 _N_=4
After:I=4 _ERROR_=0 _N_=4
Before:I=4 _ERROR_=0 _N_=5
After:I=5 _ERROR_=0 _N_=5
Before:I=5 _ERROR_=0 _N_=6
After:I=6 _ERROR_=0 _N_=6
Before:I=6 _ERROR_=0 _N_=7
After:I=7 _ERROR_=0 _N_=7
Before:I=7 _ERROR_=0 _N_=8
After:I=8 _ERROR_=0 _N_=8
Before:I=8 _ERROR_=0 _N_=9
After:I=9 _ERROR_=0 _N_=9
```

```
Before:I=9 _ERROR_=0 _N_=10
After:I=10 _ERROR_=0 _N_=10
Before:I=10 _ERROR_=0 _N_=11
```

From the above log messages, we see the entire looping action of this implied DATA step DO-loop. What we have done is set the PUT statements to write out to the PDV before and after the set statements along with the automatic variable _N_ which, when left unchanged, shows the number of times the implied loop occurs.

All of this code is compiled and executed. For the DATA step, the PDV is set up at the time of compilation with all the information for the variables as well as some automatic variable such as _N_ and _Error_. We can see this by the very first line in the log in which the value of I is missing, the value of _ERROR_ is 0 and _N_ is 1. At this point the DO-loop has been set up but not run. In the next line _N_ = 1 so we are still in our first loop and I = 1. It is the next line that is of interest to us at this junction - Before:I=1 _ERROR_=0 _N_=2. Many people see this and ask why is I still equal to one and _N_ = 2. The answer is simple and can be found by thinking through how the DATA step operates. A DATA step brings in one observation, does something to it then writes it out, and if it can, will bring in another observation. Thus if the programmer has not explicitly written an output statement in the code an assumed one exists right before the RUN statement. In order for the DATA step to be able to retrieve and process the next observation, it needs to pass control back to the top after it writes out the prior observation. While it passes control to the top, the _N_ loop counter gets incremented but the next value of I has yet to be read into the PDV. The reason for this is that all variables coming from a data set will be automatically retained while all variables that are created within the DATA step will be set back to missing unless their values are specifically retained. We will use this bit of information to our advantage and show how proper placement code relying on end-of-file (EOF) markers is important.

The other interesting line is the last one: Before:I=10 _ERROR_=0 _N_=11. Many people ask why _N_ = 11 when we only have 10 observations in the data set. Remember when we stated that the implied loop was infinite because there wasn't an efficient way to determine how many times it should loop, thus it only stops when there is no more data to read? This is exactly what we are seeing here; it loops ten times since there are 10 observations and then loops one extra time, an 11th time, but since there is nothing to retrieve, the loop is terminated.

Earlier we stated that the implied looping of the DATA step can cause problems when code execution is controlled by an EOF marker. Many times we as programmers only keep records that we are interested in or delete records, hence the reason for the WHERE statement or subsetting IF statements. However, when using these one needs to take care; if the last record is deleted or dropped before code that checks for the EOF marker runs then that code does not get executed. To compensate for this, code that requires the EOF marker should be placed before the SET statement. Yet another example would be if you needed something to occur before any data has been read into the PDV, such as writing header information to a file.

Here is an additional example:

```
Data WrongWay ;
      Set One End = EOF ;
            If ( 1 <= I <= 9 ) ;
            If EOF Then I = 99999 ;
      Put I= _N_=  ;
Run ;

I=1 _N_=1
I=2 _N_=2
I=3 _N_=3
I=4 _N_=4
I=5 _N_=5
I=6 _N_=6
I=7 _N_=7
I=8 _N_=8
```

```
I=9 _N_=9
```

In the above example the subsetting IFstatement deletes the record with the marked EOF before it could be used by the IF statement that checks for it. This is a “gotcha” of sorts and something to be aware of. In order to properly test for EOF, there are two possible solutions: one using the WHERE statement and the other with proper placement of the IF statement that tests for EOF.

Here is an example of the first solution:

```
Data RightWay1 ;
       Set One End = EOF ;
              Where ( 1 <= I <= 9 ) ;
              If EOF Then I = 99999 ;
       Put I= _N_= ;
Run ;
```

```
I=1 _N_=1
I=2 _N_=2
I=3 _N_=3
I=4 _N_=4
I=5 _N_=5
I=6 _N_=6
I=7 _N_=7
I=8 _N_=8
I=99999 _N_=9
```

Using a WHERE statement works because data is truncated before it is read into the PDV. This effectively means the SAS supervisor has to read ahead by at least one observation to know when to stop reading the data.

Here is an example of the second solution:

```
Data RightWay2 ;
       If EOF Then Do ;
              I = 9999 ;
       Put I= _N_= ;
End ;
       Set One End = EOF ;
              If ( 1 <= I <= 9 ) ;
       Put I= _N_= ;
Run ;
```

```
I=1 _N_=1
I=2 _N_=2
I=3 _N_=3
I=4 _N_=4
I=5 _N_=5
I=6 _N_=6
I=7 _N_=7
I=8 _N_=8
I=9 _N_=9
I=9999 _N_=11
```

This method places the IF statement before the SET statement, thus allowing for the IF statement to test for the EOF marker, Do what it is it supposed to and then be deleted by the subsetting IF statement.

DO Iterative

The DO Iterative is the most basic type of DO-loop in SAS. It iterates over the values given in its index variable. This can take many forms: list of numeric or character values separated by comma; a range given by a start value, an ending value and a by value; some combination of the previously stated and a WHILE or UNTIL condition. If all this sounds confusing that is because the DO iterative is also the most flexible of the DO-loops with a great many possible applications. To clarify let's look at some examples.

DO Iterative Example #1

```
Data _Null_ ;
      Do I = 1 , 2 , 3 ;
            Put I= ;
      End ;
Run ;
```

```
I=1
I=2
I=3
```

DO Iterative Example #2

```
Data _Null_ ;
      Do I = 'A' , 'B' , 'C'  ;
            Put I= ;
      End ;
Run ;
```

```
I=A
I=B
I=C
```

In the above two examples we see that the DO Iterative will loop for every value in a list that is given to the index variable. It does not matter whether that list is numeric or character. What does matter is that all the values in that list are of the same type. This is due to the fact that the index variable becomes an actual variable in the data set, thus it must follow the rules of SAS variables in that it cannot be both a numeric and character at the same time. When working with a character list, be sure all values are of the same length; you can ensure this by making sure you have the same number of spaces between parentheses or by using a LENGTH statement to set the value of the index variable to the length of the longest value in your list. We believe that using a LENGTH statement is the easiest and safest method.

Incorrect way:

```
Data _Null_ ;
      Do I = 'AAA' , 'BBBB' , 'C' ;
            Put I= ;
      End ;
Run ;
```

```
I=AAA
I=BBB
I=C
```

Correct Way:

```
Data _Null_ ;
      Length I $ 4 ;
            Do I = 'AAA' , 'BBBB' , 'C' ;
            Put I= ;
      End ;
Run ;
```

```
I=AAA
I=BBBB
I=C
```

DO Iterative Example #3

```
Data _Null_ ;
      Do I = 1 To 10 By 2 ;
            Put I= ;
      End ;
Run ;
```

```
I=1
I=3
I=5
I=7
I=9
```

In this example we see that you if you have a contiguous range of numeric values you want to iterate over, you can do so by specifying a starting value, ending value, and a value by which to increment. If no BY value is specified, SAS assumes it to have a value of one. In the above example we start at a value of 1, want to continue to a value of 10 and increment the index variable's value by two with each iteration of the loop.

At the start of the loop the index variable's value is checked against the stop value. Since the index variable's value is smaller than the stop value it is iterated. The value of `I` is written out to the log, control is then passed to the top of the loop, the index variable is incremented by two and is again checked against the stop value. This is repeated until the index variable is greater than the stop value. To prove this, let's see what the value of `I` is after the loop is finished.

```
Data _Null_ ;
      Do I = 1 To 10 By 2 ;
            Put I= ;
      End ;
      Put I= ;
Run ;
```

```
I=1
I=3
I=5
I=7
```

```
I=9
I=11
```

From the log we see that I has a value of 11 after the loop is finished. After the loop where the value of I is 9, control passes back to the top of the loop, I is incremented by two to reach 11. That is tested and found to be greater than the stop value of 10 so the loop stops. However, the value of I is now 11 and not 9, the last value before it would be greater than 10 or equal to 10 as the stop value. This underscores the importance of knowing specifically what is happening before using the index variables later in code.

DO Iterative Example #4

```
Data _Null_ ;
      Do I = 3 To 1 By -1 ;
            Put I= ;
      End ;
Run ;
```

```
I=3
I=2
I=1
```

In the previous example we showed incrementing with a ranged value. In this example we see that we can also decrement these values. To do this the start value must be greater than the stop value and the BY value must be a negative.

Combining Multiple Values and Conditions

The DO Iterative will allow you to combine multiple looping values and conditions. All that is required is separation of each one of these conditions by a comma. If you remember from the previous examples you can also specify a list of values to iterate over by separating them with a comma. This means that the when you combine the code, it can get very confusing. The authors admit that though there are rare times when doing this makes sense, it must be noted that these case should be handled with caution and be accompanied by extensive code documentation.

Here are three examples of how it can work:

```
Data _NULL_ ;
      Do I = 1 , 2 , 3 To 1 By -1 ;
            Put I= ;
      End ;
Run ;
```

```
I=1
I=2
I=3
I=2
I=1
```

```
Data _NULL_ ;
      Do I = 0 to 3 , 4 , 3 By -1 While( I > 0 ) ;
            Put I= ;
      End ;
Run ;
```

```
I=0
```

```
I=1
I=2
I=3
I=4
I=3
I=2
I=1
```

```
Data _NULL_ ;
      Do I = 0 to 3 While( I < 2 ) , 4 , 3 By -1 While(I > 0), 1 By 1 Until( I = 4 )
;
            Put I= ;
      End ;
Run ;
```

```
I=0
I=1
I=4
I=3
I=2
I=1
I=1
I=2
I=3
I=4
```

Do While

The DO While, unlike the basic DO Iterative, is an infinite loop with an exit condition. Thus if the exit criteria is never met the loop will continue until stopped by something outside the looping code. The exit criteria of a DO While loop assumes a true condition and looks for when that true condition becomes false in order to stop the looping process. The DO While loop tests for this exit criteria at the top of the loop. Thus, if the criteria is false the first time it is tested then the loop will not iterate at all.

DO While Example #1

```
Data _Null_ ;
      I = 0 ;
            Do While ( I < 10 ) ;
      Put I= ;
            I + 1 ;
      End ;
Run ;
```

```
I=0
I=1
I=2
I=3
I=4
I=5
I=6
I=7
I=8
```

```
I=9
```

Here our exit criteria states when the value of I is equal to the value of 10 the loop stops. We set the index variable's initial value to 0 (this is called priming the loop), the exit criteria test is done and found to be true so the loop iterates. The value of I is written to the log and then its value is incremented by 1. Control passes to the top of the loop , the test is performed and again found to be true. The steps are repeated until the value of I is equal to the stop criteria thresh hold of 10.

DO While Example #2

```
Data _Null_ ;
      I = 11 ;
            Do While ( I < 10 ) ;
      Put I= ;
            I + 1 ;
      End ;
Run ;
```

Here we primed the loop's initial value to 11 and one can see from the log that no values were written out. The initial test was made and the exit criteria expression was resolved to false. Thus the DO While loop did not execute.

DO While Example #3
We saw in the first example that the loop apparently iterated 10 times; however what exactly is the end value of index variable I in that example?

```
Data _Null_ ;
      I = 0 ;
            Do While ( I < 10 ) ;
      Put 'Inside Loop: ' I= ;
            I + 1 ;
      End ;
      Put 'Outside Loop: ' I= ;
Run ;
```

```
Inside Loop: I=0
Inside Loop: I=1
Inside Loop: I=2
Inside Loop: I=3
Inside Loop: I=4
Inside Loop: I=5
Inside Loop: I=6
Inside Loop: I=7
Inside Loop: I=8
Inside Loop: I=9
Outside Loop: I=10
```

We see from the log that the ending value of I is 10. What happened? The loop initially iterated as expected, the last value from inside the loop printed to the log is 9 which is less than the value of 10 as set in our loop's exit criteria. However, after the loop where I with value of 9 is printed to the log, the value of I gets incremented one more time to 10. Control is passed to the top of the loop, the value is checked one last time and the exit criteria is shown to be false.

In the above example, we incremented a value from within the DO loop while simultaneously using it for our exit criteria. This means you need to be careful about what you are asking to have done; if you needed to use the value of I later in the code, you could end up using the wrong number.

Do Until

The DO Until loop, like the DO While, is an infinite loop with set exiting criteria. Unlike the DO While, the DO Until's exit condition is assumed false and the loop stops only when the criteria becomes true. Yet another difference is that it tests for this exit condition at the bottom of the loop. Thus, it will always iterate at least once.

DO Until Example #1

```
Data _Null_ ;
       I = 0 ;
              Do Until ( I > 10 ) ;
       Put I= ;
              I + 1 ;
       End ;
Run ;
```

```
I=0
I=1
I=2
I=3
I=4
I=5
I=6
I=7
I=8
I=9
I=10
```

We see from the above example that the loop starts with a value of 0 on its first iteration and continues until the value of I is 10. The value of I for the first iteration of the loop is printed out to the log, then incremented by 1, then the test for the exit condition is performed and finally control is passed to the top of the loop. Once the value of I is incremented to a value of 11, when the condition is tested and found to be true, the loop stops iterating.

DO Until Example #2

```
Data _Null_ ;
       I = 11 ;
              Do Until ( I > 10 ) ;
       Put I= ;
              I + 1 ;
       End ;
Run ;
```

```
I=11
```

In this example we see that though we set the initial value of I to 11 the DO Until loop does still execute one time. Remember that because this type of loop tests at the bottom, it will iterate one time regardless of whether the exit condition is true or not so we see the initial value of I written to the log.

DO Until Example #3

```
Data _Null_ ;
      I = 0 ;
            Do Until ( I > 10 ) ;
      Put 'Inside Loop: ' I= ;
            I + 1 ;
      End ;
      Put 'After Loop: ' I= ;
Run ;
```

```
Inside Loop: I=0
Inside Loop: I=1
Inside Loop: I=2
Inside Loop: I=3
Inside Loop: I=4
Inside Loop: I=5
Inside Loop: I=6
Inside Loop: I=7
Inside Loop: I=8
Inside Loop: I=9
Inside Loop: I=10
After Loop: I=11
```

Just like in the DO While loop we see that the last value of I is indeed 11. This makes intuitive sense given that our stated exit condition was to stop loop when I became greater than 10.

The question of when each of these loops tests for the exit condition is often asked. Many people simply memorize it as the DO While tests at the top and DO Until tests at the bottom. We find that a more natural way to remember this is by thinking about the definitions of the words while and until. If we told you to do jumping jacks while we whistled you would first check that we were whistling before you began doing them. However, if we instead told you to do jumping jacks until we said to stop, you would immediately start doing them until you heard us say stop. Thus in the first example you check at the beginning and in the second example you start and then check if you need to stop.

Entering and Exiting A Loop in the Middle

In all the examples we have used so far, we enter at the top of the loop. The two reasons for this are that it is the most common approach any other approach would lead to poor coding. If you enter or exit a loop in the middle you are essentially creating what is known as “spaghetti code”, code that is nearly impossible to read, debug, and update. If you program long enough you will eventually hear horror stories about such code and you would do well to avoid writing it. However, SAS felt compelled to add methods to do just this so we will cover them. This is less of an issue in simple DO-loops than it becomes in longer and/or more complex ones, but the authors find the best rule is to just not use them. Remember there is virtually no DO- loop problem that should *require* doing this. If you use proper exit criteria and conditional statements, you can obviate the pitfalls of entering and exiting in the middle.

The CONTINUE statement stops the current iteration and returns control back to the top of the DO-loop. If the control statement is executed then any code after that will not be executed for that iteration of the DO-loop. To test this let’s look at an example where the CONTINUE statement is executed and check the value of the index variable after the DO-loop is finished.

```
Data _Null_ ;
      Do I = 1 To 5 ;
            If I > 2 Then Continue ;
            Put I= ;
      End ;
      Put I= ;
Run ;
```

```
I=1
I=2
I=6
```

In the example above when the value of I is 1 or 2, the IF statement does not execute and the values of I are written out via the PUT statement in the DO-loop. When the values of I are greater than 2, the IF statement executes, control returns to the top of the DO-loop and the PUT statement is not executed. We can glean from this that any code before a CONTINUE statement will have the possibility to get executed regardless of whether or not the condition of that statement is met. However, any code after the CONTINUE statement will only be executed if that statement is not. This is borne out by the fact that the last value of I is 6 which means that the DO-loop iterated for the entire range specified, but all code after the CONTINUE statement was not executed unless the value of I was less than 3.

The LEAVE statement, unlike the CONTINUE statement, will cause the current DO-loop to stop iterating when executed.

```
Data _Null_ ;
      Do I = 1 To 5 ;
            If I = 3 Then Leave ;
            Put I= ;
      End ;
      Put I= ;
Run ;
```

```
I=1
I=2
I=3
```

When the values of I are less than 3 the PUT statement is executed and the values of I are written out to the log. When the value of I reaches 3, the IF statement is executed and the DO-loop is exited. We can show that this is the case since the last value of I in the log is 3.

The RETURN statement returns control to the top of the DATA step. In all DATA steps there is an implied RETURN statement just before the RUN statement. Using a RETURN statement in the context of a DO-loop is odd, but possible.

```
Data _Null_ ;
      Do I = 1 To 5 ;
            If I = 3 Then Return ;
            Put I= ;
      End ;
Run ;
```

```
I=1
I=2
```

In the above example while the value of I is less than 3, the values of I are written out to the log. When the value of I reached 3, the RETURN statement is executed and control passes out of the DO-loop and back to the top of the DATA step. At this point the DATA step stops execution as there is no incoming data from a data set or file and it has already executed once.

The GOTO statement has become rather infamous, almost to the point of inspiring fanatical programming wars. The general consensus is that while the GOTO statement is not inherently evil, its propensity to create seriously confusing and convoluted code means it should be used in the rarest of cases.

```
Data _Null_ ;
      Do I = 1 To 5 ;
            If I = 3 Then GoTo End ;
            Put I= ;
      End ;
            End: ;
      Put I= ;
Run ;

I=1
I=2
I=3
```

In the above example when the value of `I` is less than 3, the values of `I` are written out to the log. When the value of `I` reaches 3, the GOTO statement is sending control out of the DO-loop and to the part of the DATA step with the label of END. We see by the PUT statement after the END label that the DO-loop indeed was executed only three times.

So far all the examples have demonstrated how to exit from the middle of a DO-loop. However, the GOTO statements is the only way to enter from the middle of a DO-loop.

```
Data _Null_ ;
      I = 3 ;
            If I < 5 Then GOTO MyLoop ;
            Do While ( I < 5 ) ;
            MyLoop:
      Put I= ;
            I + 1 ;
      End ;
Run ;

I=3
I=4
```

Here we see only two values of `I` printed to the log via the DO-loop. `I` is initially set to 3. When the IF statement evaluates as true the GOTO is executed and control passes to the Do While loop. Here the value of `I` is printed to the log, incremented, evaluated and printed to the log again with a value of 4. `I` is once again incremented, now to 5, so control passes to the top of the step where the exit criteria is resolved to be false and the loop is terminated.

While the examples above are kept simple for the sake of clarity, we hope that you will seriously consider the risks in using these statements in your code. Small mistakes can easily create code that is hard to understand, inefficient and has the potential to set up infinite loops.

DO-Loop of Whitlock (DoW)

Another more advanced DATA step technique that takes unique advantage of BY group processing is the DO loop of Whitlock, more commonly called the DoW loop. Several years ago, the respected SAS programmer Ian Whitlock quietly answered a DATA step question on the SAS listserv. While the actual post was of little consequence, the solution he proposed was not. Whitlock wrapped the SET statement inside a DO UNTIL loop and used the BY statement to control the loop. Later, the similarly renowned SAS programmer, Paul Dorfman, recognized the power that this construct brought to the SAS DATA step programming tool kit, and expounded upon its many uses until it

became a commonly known SAS programming technique. His classic paper “The Magnificent Do” is an excellent resource on this topic.

There are many variations of the DoW, but this is the general form:

```
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;
```

This construct is simple and easily coded, yet it has great power when combined with the natural functionality of the DATA step. The normal functionality of the DATA step is to read in one observation, do something to it, and write it out. In this normal flow all variables that are created within the DATA Step itself will, unless told otherwise, be reset to missing with each implicit loop of the DATA Step. When BY group processing is used within this frame work SAS keeps track of the data in each BY group. Thus, if one wanted to do something on the first, middle, or last observation in a BY group, it is the programmers responsibility to code that by using the First.<Variable Name> and Last.<Variable Name>.

In this example, the DATA step sums the X variable values and outputs one observation per BY group.

```
Data Have ;
       Infile Cards ;
       Input X Y $ @@ ;
       Cards ;
       1 A 2 A 3 A 4 A 5 B 6 B 7 B 8 B ;
Run ;

Data Need1 ( Drop = X ) ;
       Set Have ;
       By Y ;

              SumX + X ;

       If Last.Y Then Do ;
              Output   ;
       SumX = 0 ;
       End ;
Run ;
```

```
Y SUMX
A 10
B 26
```

While this construct works, it is fairly complicated. It relies on the programmer coding statements before, during, and after each break event. FIRST. and LAST. BY variables must be used to explicitly control output statements and to explicitly set accumulation variables to zero between groups. In comparison, the DoW works with the natural execution of the DATA step by isolating what happens between two consecutive break events. Statements and functions acting on observation variables and values are placed plainly within the loop, and the implicit action of the DATA step resets calculated values to missing after each BY group.

In the following DoWexample the break events are BY groups, but in other cases could be anything that triggers the DO-loop to stop. It returns the same results as the more standard DATA step example above, but takes advantage of the default actions inherent in the DATA step, thus it yields a simpler program that is easier to understand.

```
Data Need2 ( Drop = X ) ;
       Do Until ( LAST.Y ) ;
              Set Have ;
       By Y ;
       SumX = Sum( SumX , X ) ;
End ;
Run ;
```

```
Y SUMX
A 10
B 26
```

Here there are no statements before the DO-loop, so the first action is the SET statement inside the loop. On each iteration of the DO-loop the SET statement reads an observation into the DATA step. The values of X are then summed until the last observation of the BY group is reached. At the last observation of the BY group a break event is controlled by the automatic LAST. variable and the program passes out of the loop. The program then reaches the end of the DATA step where the record is implicitly output to the dataset Need2. Finally, control returns to the top of the DATA step and the DO-loop instructs SAS to resume reading the next BY group. This process is repeated until there are no more observations in the input dataset.

Since there is no explicit output statement, SAS implicitly executes an output at the end of the DATA step. Also by default, SAS sets all variables created during the DATA step to missing after each implicit output, unless specified in a RETAIN statement, so there is no need to set the variable SumX back to zero for each BY group. The DoW only passes control to the end of the DATA step at each break of the BY groupings, and so uses the implicit output and resetting action of the DATA step to the programmer's advantage.

As an extension, multiple DoW loops can be used in tandem inside a DATA step. The following code appends an aggregated sum of the variable X within BY groups of Y.

```
Data Need3 ;
       Do Until ( Last.Y ) ;
              Set Have ;
              By Y ;
              SumX = Sum( SumX , X ) ;
       End ;

       Do Until( Last.Y ) ;
              Set Have ;
              By Y     ;
       Output  ;
       End ;
Run ;
```

Here the data is also read twice: the first time to get the value of X for each BY group, and a second time to output the observations with SumX attached. As with the previous DoW example, since end of the DATA step isn't reached until after a full BY group is processed by both DoW loops, the values of SumX are not reset before the second loop, and so the value of the summed variable is attached to each output record.

Writing Loops From The Inside Out

If you are like most people the more complex the loop you need to write the harder it is to get correct. In this section we will show a simple methodology for writing complex loops. First consider the basic thing you need to do. Then write the code to do that with only one observation or alternatively write code to perform the task one time. Once that

code is in place you can add to it depending on what else you need to do. Once you have your code working without a loop then add in the looping. If multiple loops need to be added repeat these steps for each loop. Essentially you are building the DO-loop from the inside out and step by step.

Let's say we have a list of five rates and we want to increase them by 5%, placing the new value into another set of variables.

The first step is to create the base values of the rates which will then be increased.

```
Data _Null_ ;
      Array RateA (5) RateA1-RateA5 (10 , 20 , 30 , 40 , 50)  ;
      Array RateB (5) RateB1-RateB5 ;

            RateB1 = ( RateA1 * .05 ) ;

      Put RateB1= ;
Run ;
```

Next we need to add that value to the initial value.

```
Data _Null_ ;
      Array RateA (5) RateA1-RateA5 (10 , 20 , 30 , 40 , 50)  ;
      Array RateB (5) RateB1-RateB5 ;

            RateB1 = RatesA1 + ( RateA1 * .05 ) ;

      Put RateB1= ;
Run ;
```

Once we are satisfied that the math is working for this one simple example we need to add in our looping code.

```
Data _Null_ ;
      Array RateA (5) RateA1-RateA5 (10 , 20 , 30 , 40 , 50)  ;
      Array RateB (5) RateB1-RateB5 ;

            Do I = 1 To Dim( RateA ) ;
            RateB(I) = RateA(I) + ( RateA(I) * .05 ) ;
      End ;
Run ;
```

This simple example shows the thought process that should guide the coding of loops. By following this procedure, you can better ensure the efficiency and accuracy of your code as well as facilitating any necessary troubleshooting.

%DO-Loops in the Macro Facility

Before we get into the details of %DO-loops let's recapitulate some of the macro language philosophy. SAS is essentially a bunch of languages glued together by data sets, each with its own little philosophical twist. In the DATA step words (i.e. Data, Run, Set, Retain, etc.) have meaning to SAS. It is the programmer's responsibility to tell SAS what not to try to interpret and to instead treat as just plain text. In the macro language this is reversed as everything is considered plain text and it is the programmer's responsibility to tell SAS what to interpret as having some significant meaning by prefixing it with either a % or &. This reversal of philosophies between the two languages can create confusion: if everything in the macro language is text, how does it handle numeric values? How does it increment the index value of a %DO-loop?

%DO-loops in the macro facility have a few differences than those in the DATA step. Chiefly, they can only be used within %Macro code. In a previous section we discussed that the index variable in a DATA step becomes a variable

in the data set. In the macro language the index variable becomes a macro variable, instead of a data set variable. So remember to declare your index variables a %Local to that %Macro in which it is used.

%DO Iterative

%Do <Index Variable> = <Start Value> %To <Stop Value> %By <Increment Value> ;
  <Code>
%End ;

There are a couple of major differences in the %DO Iterative versus the DO Iterative. First the key words DO, TO, BY, and END must be prefixed with a %. Since we want these to be interpreted as having meaning we need to identify them for SAS. Second, one can only specify starting, ending, and BY values. This means you cannot use a list of values to iterate over, use a WHILE or UNTIL, nor can you mix any of these. You have only one option: to specify a starting value, an ending value, and a value which you want SAS to increment by from the start to end values. The macro language is rather crude when compared to the DATA step language so its functionality is limited.

```
%Macro Test ;
       %Do I = 1 %To 3 ;
              %Put I = &I ;
       %End ;
%Mend ;
```

```
%Test
```

```
I = 1
I = 2
I = 3
```

In the above example we see that the %DO-loop iterates three times. The %PUT statement puts the values in the log. Since the index variable is a macro variable we have to prefix it with a &. Since we did not give the %DO-loop a %BY value, it assumes it to be a 1. Since the macro language assumes everything is text, how is the %DO-loop able to interpret the values to loop over to increment or decrement them? %Do –loops use an implied call to the %EVAL statement. So long as one uses integers this does not pose much of a problem. However, when one wants their BY values to be a non-integer, SAS starts generating error messages.

```
%Macro Test  ;
       %Do I = 1 %To 3 %By .5  ;
              %Put I = &I  ;
       %End  ;
%Mend  ;
```

```
%Test
```

```
ERROR: A character operand was found in the %EVAL function or %IF condition where a
numeric operand
       is required. The condition was: .5
ERROR: The %BY value of the %DO I loop is invalid.
ERROR: The macro TEST will stop executing.
```

In the above code we see that using a .5 in the %BY value creates some error messages. The reason is the %EVAL can only handle integers. When it sees the .5 the decimal gets treated as a character while the %EVAL function is trying to do a mathematical operation. Thus, the first message about a character operand being found in the %EVAL function is generated. The easiest way to overcome this is to use the %SysEvalF function which can handle non-integers. However, we have to be creative about this since we cannot tell the %DO-Loop to use %SysEvalF instead of %EVAL.

```
%Macro Test  ;
      %Do I = %SysEvalF( 1 * 10 ) %To %SysEvalF( 2 * 10 )  ;
            %Put I = %sysEvalF( &I / 10 )  ;
      %End  ;
%Mend  ;

%Test

I = 1
I = 1.1
I = 1.2
I = 1.3
I = 1.4
I = 1.5
I = 1.6
I = 1.7
I = 1.8
I = 1.9
I = 2
```

In the above code we have used the %SysEvalF for the Start, Stop and in the %PUT to increment the values from 1 to 2 by 0.1. This works solely because the math in the Start and Stop values will create the needed number of iterations and the %SysEvalF in the %PUT will manipulate the values to create the desired output. This method works only when you intervals to be calculated that are a factor of 10; for .1 divide by 10, for .2 divide by 5 and for .5 divide by 2. It fails when you want to increment by something else like .3. For any units other than the three mentioned here, you should use macro list processing.

In a similar manner we can have decimal values as our Start and Stop values.

```
%Macro Test ;
      %Do I = %SysEvalF( 1 * 10 ) %To %SysEvalF( 2 * 10 ) ;
            %Put I = %SysEvalF( &I / 100 ) ;
      %End ;
%Mend ;

%Test

I = 0.1
I = 0.11
I = 0.12
I = 0.13
I = 0.14
I = 0.15
I = 0.16
I = 0.17
I = 0.18
I = 0.19
I = 0.2
```

%DO %While and %Until

```
%Do %While/%Until( Exiting Criteria ) ;
  <Code>
%End ;
```

The %DO %While or %Until loops act in much the same manner as their corresponding DATA step versions. The %While will test at the top of the loop and the %Until will test at the bottom of the loop. Two factors are noteworthy: you can only have exit criteria so there can be no mixing of the code (i.e. %Do I = 1 To 3 Until(&I < 2) and you will have to prime the loop as both will require a macro variable to test the exiting criteria.

```
%Macro Test  ;
       %Local I  ;
              %Do %While( &I < 3 )  ;
              %Let I = %Eval( &I + 1 )  ;
       %Put I = &I  ;
       %End  ;
%Mend  ;

%Test

I = 1
I = 2
I = 3
```

Notice the %Local I in the above code. Technically this is not exactly what one would think of when you say "priming the loop" as it does not give it a value. However, since the %EVAL will do the math and not give an error it is acceptable. If one wanted to, you could explicitly define the macro variable I in a %LET statement and give it a value of 0 .

```
%Macro Test ;
       %Local I ;
              %Do %Until( &I > 2 ) ;
              %Let I = %Eval( &I + 1 ) ;
       %Put I = &I ;
       %End ;
%Mend ;

%Test

I = 1
I = 2
I = 3
```

One of the most common problems people have with %DO-loops is iterating over a known set of values, especially if that set of values is character. One method would be to use macro arrays:

```
%Let A1 = Alpha ;
%Let A2 = Beta  ;
%Let A3 = Theta ;

%Macro Test( Prefix = , Start= , Stop= ) ;
       %Local I ;
              %Do I = &Start %To &Stop ;
              %Put Array Value = &&&Prefix&I ;
       %End ;
%Mend ;
```

```
%Test( Prefix = A , Start = 1 , Stop = 3 )

Array Value = Alpha
Array Value = Beta
Array Value = Theta
```

Notice that the macro code became more complicated by having to specify the prefix for the macro variable names, the starting and stopping values you wish to iterate over as well as the fact that the reference call in the %PUT statement is now having to use multiple ampersands. The experienced SAS programmer will try to avoid code like this whenever possible. Macro arrays are deprecated and should only be used when sorting a macro list; as of the writing of this paper, there is no pure macro code way to do that sorting without them.

The better solution is to use macro list processing. There are two methods to do this: either use a %Until loop or a %DO Iterative and let the code discover the number of iterations to make. The authors prefer the simplicity of the latter.

```
%Macro Test( List = )  ;
      %Local I Word  ;
            %Do %Until( %Length( %SuperQ(Word) ) = 0 )  ;
      %Let I = %Eval( &I + 1 ) ;
      %Let Word = %Scan( &List ,&I , %Str( ) )  ;
            %Put Array Value = &Word  ;
      %Let Word = %Scan( &List ,%Eval(&I+1) , %Str( ) )  ;
      %End  ;
%Mend  ;

%Test( List = Alpha Beta Theta )

Array Value = Alpha
Array Value = Beta
Array Value = Theta
```

In the above code a %Until Loop is used iterate the code that will parse the macro list. The problem we see with this is two-fold. First you have to write a unnecessary code to make this happen which means there is more code to debug and maintain. Second, you have to artificially create the exit condition when it can be discovered easily. If you remember from the DATA step section, we mentioned that when it is at all reasonably possible to determine the number of iterations a loop needs to make that it should be done using a DO Iterative.

```
%Macro Test( List = )  ;
      %Local I  ;
            %Do I = 1 %To %SysFunc( CountW( &List ) )  ;
            %Put Array Value = %Scan( &List ,&I , %Str( ) )  ;
      %End  ;
%Mend  ;

%Test( List = Alpha Beta Theta )

Array Value = Alpha
Array Value = Beta
Array Value = Theta
```

Now the code accomplishes the same task as in the previous example while being cleaner and easier to understand. In this case, less code provides some much needed clarity.

Conclusion

Our goal in this paper was to describe the principles underlying DO-loops, the various types available in SAS and to describe more consistent coding practices when writing them. By delving into the possibilities of iterative DO-loops, DO Until and DO While, DoW and avoiding infinite loops, we hope to have broadened your understanding of how they work and made you think more in depth about how you use them in your work or could do so in the future. By carefully exploiting the logic of these SAS tools, you can increase your productivity and code efficiency.

References

Dorfman, Paul "The Magnificent Do"
http://devenezia.com/papers/other-authors/sesug-2002/TheMagnificentDO.pdf

SAS OnlineDoc 9.1.3 for the Web
SAS Institute Inc., Cary, NC

Contact Information

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

Sarah Woodruff
Westat
1600 Research Boulevard
WB 401
Rockville, MD 20850
(240) 314-7562
SarahWoodruff@Westat.com

Toby Dunn
AMEDDC&S (CASS)
San Antonio, TX
Toby.Dunn@amedd.army.mil