Paper 063-29

The Role of Consecutive Ampersands in Macro Variable Resolution and the Mathematical Patterns that Follow

Michael J. Molter, Howard Proskin and Associates, Rochester, New York

ABSTRACT

As the complexity of a macro application increases, the programmer may find the use of consecutive-ampersand macro variable references a convenient way of referring to combinations of macro variables. Advantages of these techniques include efficiency, brevity, program organization, and even elegance. Without an understanding of the rules behind this notation as well as the workings of the macro processor, these techniques get difficult to read and especially to write. Manuals provide the rules and some examples, but learning still requires extensive trial-and-error approaches. This paper takes a closer look at the consequences behind the rules of consecutive ampersands, observing regular patterns and stating them in simple mathematical terms. A brief review of macro variable initialization sources is discussed, followed by three simple rules for macro variable resolution and the associated concept of multiple reads of the variable reference by the macro processor, plus examples. Finally, the role of the period in macro variable resolution is discussed. This paper is suitable for programmers of all levels that have at least a basic knowledge of macro processing. It is intended for BASE SAS® users in any operating system.

INTRODUCTION

What purpose do consecutive ampersands serve in SAS® macro code? The technical answer to this is simple – they force multiple reads of a macro variable reference by the macro processor. The real question here is why a programmer would want this. Why would a programmer want a macro variable resolution by the macro processor to be followed by another resolution?

Mathematics teaches us that a variable is a quantity with the potential to hold different values or to change. Macro variables are not much different. Whether initialized by the user or the programmer, macro variables allow us to implement changes throughout an otherwise static piece of code in a simple and efficient way. Sometimes in building an application, the flexibility a programmer allows the user introduces additional, more subtle variability in coding other parts of the program. For example, maybe a reference to a macro variable is appropriate only under certain circumstances, while another macro variable is appropriate under other circumstances. According to our definition of a variable, the two circumstances point to another variable. By recognizing this and creating an additional macro variable that addresses each of the circumstances, consecutive ampersands can be used to first determine the appropriate circumstance, and then use a second read to resolve the appropriate macro variable. All that is left now is to understand the specifics of how the macro processor resolves such references.

In my own personal experience, I've discovered that when consecutive ampersands are used to resolve a combination of macro variables, the macro variables often come from separate sources. For example, a programmer initializes a macro variable transparent to the user to help manage macro parameters initialized by the user. For this reason, I will provide a brief review of macro variable sources.

I think that only a handful of different applications that make use of consecutive ampersands gives the appearance of a very complex topic within macro processing. This paper sets out to prove otherwise. Widespread exploitations of the rules behind processing these references give this appearance, but I will demonstrate that knowledge of only three very simple, general rules are required for complete understanding. After discussion of these rules, I will exploit them in two common ways. Mathematics also teaches us to look for patterns that result from controlled change. In this spirit, I will then expand upon each of these ways and generalize the patterns we discover in mathematical terms. Upon understanding of the rules and examples, the reader should be able to use consecutive ampersands in these two ways, discover other ways to exploit the three rules, and generalize patterns from them.

MACRO VARIABLE INITIALIZATION SOURCES

Macro variables can be initialized in a number of ways. The responsibility of assigning values to a macro variable depends on how it was initialized. Below is a table summarizing different initialization techniques and who is responsible for the values, followed by a brief description of each.

Initialization techniques	Responsibility of assignment
Macro Parameters	User
%WINDOW - %DISPLAY	User
%DO	Programmer
%LET	User / Programmer
CALL SYMPUT	Data
SELECT – INTO	Data

MACRO PARAMETERS

Macros are usually written to generate a piece of code under certain sets of circumstances. Macro parameters are macro variables whose values are the circumstances specified by the user for a given invocation of the macro. This is achieved by typing the name of the macro preceded by a percent sign (%), followed by a set of parentheses that encloses the user's choices for values. The programmer can create one or both of two types of parameters. Values for positional parameters are required and must be specified in a specific order. Keyword parameters are optional by nature, and the programmer can issue a default value to them.

%WINDOW - %DISPLAY

%WINDOW and %DISPLAY statements serve purposes similar to those of macro parameters. When executed, %WINDOW defines a prompt screen for the user, and the %DISPLAY statement displays the screen. Once the user has typed their answers into the screen and %DISPLAY is finished executing, the answers are assigned to macro variables defined in the %WINDOW statement.

%DO

Just as a DO loop in the DATA step needs a data set variable to keep track of iterations of the loop, the %DO loop used inside a macro needs a macro variable to keep track of iterations. This macro variable is assigned and used strictly by the programmer. We will see later how resolution of these macro variables can create references to other macro variables that were named for this purpose.

%LET

%LET is one of the more versatile of variable initialization techniques. A programmer may use it for his own purposes inside a macro and the user never knows about it. On the other hand, unlike the techniques discussed up to this point, %LET doesn't have to be used inside a macro. Rather than writing a macro, a programmer may just write a program with a few macro variable references scattered throughout that are initialized before the program is executed through %LET. Almost like filling in macro parameters, the user finishes the statement by typing the desired value after "%let var=".

CALL SYMPUT / SELECT INTO

CALL SYMPUT and its PROC SQL counterpart SELECT-INTO are different from other initialization techniques because neither the user nor the programmer explicitly assign values created by these statements. Rather, values are driven by the values of variables found in data sets. The macro variables are named either with text or with values of other variables in the data set. This becomes useful when information contained within a data set is needed after the data step is completed.

THREE RULES FOR PROCESSING CONSECUTIVE AMPERSANDS

With different sources of macro variables, we begin to see that a programmer can define and initialize macro variables to work with those that are under the control of the user or the data itself. With multiple reads, the processor is forced to first decide which circumstance is appropriate, and then resolve the macro variable that corresponds to that circumstance. In the beginning, I said that consecutive ampersands instruct the macro processor to make multiple reads. The following three rules provide specifics on how this is done.

Rule 1: Two ampersands always resolve to one ampersand. Immediately it is tempting to wonder why we would ever use consecutive ampersands. It would appear that if *n* is the number of ampersands, the first two would resolve to one, leaving *(n-1)* ampersands. Then this one pairs with the next ampersand, resolves to one, leaving *(n-2)* ampersands, etc, until you get down to one ampersand. The reason that is not the case is Rule 2.

Rule 2: The macro processor always continues reading left to right until the end of the reference is reached. In other words, the single ampersand that results from two from Rule 1 is left alone as the macro processor reads left to right, and will not be processed until a subsequent read. Again, with *n* ampersands, ampersands one and two resolve to one, then three and four resolve to one, and so on. The macro processor will then go back and re-process those results only after the entire reference has been processed initially.

Rule 3: When a reference contains consecutive ampersands, after resolution, at least one more read will follow. This is the only way to force more reads by the macro processor.

CONSEQUENCES AND PATTERNS

Though the rules are few and straightforward, it will soon become clear, if it is not already, that the use of consecutive ampersands must still be well thought out in order to achieve a specific purpose. There are infinitely many arrangements of ampersands we can use to precede macro variables. We will see that not all of them produce unique results (e.g. anything that can't be produced with smaller arrangements). With those that do produce unique results, we will look for relationships between arrangements and the additional information they provide. We will observe instances where the initial read (or reads) determines the appropriate circumstances - one of the purposes of having multiple reads that was stated earlier.

Before moving ahead, I will introduce some notation to be used throughout the rest of the paper. I will use the "=>" to denote the relationship "resolves to." For example, if *&a* resolves to *b,* then this relationship will be written as *&a* => *b,* with the understanding that the right side is the result of a macro processor read of the left side. Also, in an effort to avoid making the reader count ampersands, I will make use of notation that denotes the number of ampersands being used. *(n)a* will be used to denote the macro variable *a* preceded by *n* ampersands. For example, &&&*a* will sometimes be denoted by *(3)a*. Also, *n* will be referred to as the *coefficient* of the macro variable.

We're now ready to make some observations. For the sake of illustration, we will assume the following:

&*a* => *b*
&*b* => *c*
&*c* => *d*
&*d* => *e*
etc.

MACRO VARIABLE REFERENCES OF THE FORM *(n)a*
We will begin by examining the consequences of increasing *n* in the expression *(n)a.* With n=2, Rule 1 says that &&*a* =>&*a* => *b*. The only difference between &&*a* and &*a* is one more read, but the end result is the same. In this case, the extra ampersand adds no value. We can also say that n=2 is not a *useful* coefficient of *a.*

What about three ampersands? Rule 1 says that the first two will resolve to a single ampersand, and Rule 2 says to use the third ampersand to resolve *a.* After the first read, we are left with the single ampersand that came from the first two of the original reference, plus the resolution of *a,* which is *b.* A second read would then resolve &*b.* So &&&*a* => &*b* => *c*.

It appears that when the possible values of a macro variable are themselves macro variables, then three ampersands creates a "chain" effect of macro variable resolution. Consider the following example that contains a macro called SUGI.

```
   %macro sugi ;
   %let MONTREAL=29;
   %put Montreal held SUGI &MONTREAL ;
   %mend;
```

After a call of %sugi, the text string

Montreal held SUGI 29

is written to the log. Nothing much is exciting about this. However, suppose we allow the user to choose among selected cities rather than force information about Montreal. The new macro now contains the following statements.

```
   %let INDIANAPOLIS=25;
   %let LONG_BEACH=26;
   %let ORLANDO=27;
   %let SEATTLE=28;
   %let MONTREAL=29;
```

(Alternatively, we may have had a data set containing this information, and used a data step with CALL SYMPUT to create these macro variables. This would be more convenient when more choices are available.) In doing this, we are allowing the user more flexibility. How will the %PUT statement look now? &MONTREAL is only appropriate if the user chooses to receive information about this city. As described in the introduction, we have a situation where circumstances will dictate which macro variable to reference. We will therefore create an additional macro variable that addresses all possible circumstances. Since we are allowing the user to choose the city, we can achieve these two purposes at once by creating a macro parameter through which the user makes their choice. Not only does this parameter give the user a means to make their choice, but it also addresses all of the possible circumstances. Consider the new macro SUGI2.

```
   %macro sugi2(city) ;
   %let INDIANAPOLIS=25;
   %let LONG_BEACH=26;
   %let ORLANDO=27;
   %let SEATTLE=28;
   %let MONTREAL=29;
   %put &city held SUGI &&&city;
   %mend;
```

The user's choice becomes the value for &*city*. Because of that, *city* is the macro variable created to address all possible circumstances. Notice the &&&*city* reference in the %PUT statement. The first read by the macro processor reduces the first two ampersands to one, and resolves &city to the user's choice. Now the appropriate circumstance has been decided, and the second read produces the information of choice. So if `%sugi2(SEATTLE)` is executed, then &&&*city* => &SEATTLE => 28. The log contains the message

SEATTLE held SUGI 28.

In this example, we have seen that the use of three ampersands does serve a useful purpose. When values of a macro variable that is resolved with one ampersand are themselves macro variables, then the third of three ampersands resolves the first macro variable while the first two reduce to one. Left over is a single ampersand and the value of the first macro variable which is also the name of a second. A second read resolves this second variable.

Based on these two examples, we can make one general observation right away. Because pairs of ampersands each resolve to one ampersand, when a macro variable is preceded by an even number of ampersands, the macro variable does not get resolved on that read. With an odd number of ampersands, the pairs each reduce to one, but the one leftover ampersand resolves the macro variable. Can we conclude that an even number of ampersands never add value and an odd number do? When the initial reference (prior to any read) shows one macro variable and the number of ampersands is even, then eventually, after one or more reads, we will be left with an odd number of ampersands. If the even number is a power of 2 (e.g. 2, 4, 8, 16, 32, etc.), then after *n* reads, where the number of ampersands=2^n, the reference will be reduced to one ampersand. No value is added by having more than one ampersand. When the number of ampersands is even but not a power of two, one or more reads will resolve to an odd number of ampersands greater than one. So the even number adds no more value than the odd number it reduces to. For that reason, I will initially concentrate on odd ampersands, but we will not rule out the usefulness of even ampersands. I will return to that later.

Is an odd number of ampersands always useful? Up to this point, we have seen the following.
(3)*a* => (1)*b* => *c*

What about five ampersands? In this case, each of the first two pairs reduces to one and the fifth ampersand resolves *a.* This leaves us with &&*b* after the first read, which, as we know, is no better than &*b*, the result of a second read.
(5)*a* => (2)*b* => (1)*b* => *c*

We see here that five ampersands adds no value to having three. So while an even number of ampersands in front of a macro variable never adds value, certain odd numbers of ampersands do not add value either. The following table shows the results of other odd numbers of ampersands.

Number of ampersands	Resolution
1	(1)*a* => *b*
3	(3)*a* => (1)*b* => *c*
5	(5)*a* => (2)*b* => (1)*b* => *c*
7	(7)*a* => (3)*b* => (1)*c* => *d*
9	(9)*a* => (4)*b* => (2)*b* =>(1)*b* => *c*
11	(11)*a* => (5)*b* => (2)*c* => (1)*c* => *d*
13	(13)*a* => (6)*b* => (3)*b* => (1)*c* => *d*
15	(15)*a* => (7)*b* => (3)*c* => (1)*d* => *e*

From this table several patterns emerge. First, we see that seven ampersands resolves to *d* so this adds value to what we had with three. However, nine ampersands gets us no further than we were with three, and eleven and thirteen give us the same resolution as seven. So far, 1, 3, 7, and 15 ampersands add value. The question becomes, how do we know what odd numbers of ampersands will resolve to something unique?

The answer lies in a recursive sequence. A recursive sequence of numbers is a sequence in which the first term (or the first *k* terms) is explicitly initiated, and the *n*th term (n > k) is a function of prior terms. In this case, we have defined the beginning of a sequence whose terms each represent the number of ampersands that add value to a lesser number of ampersands. This sequence begins with the numbers 1, 3, 7, and 15. The challenge now is to define the *n*th term. We can do this by observing the relationship between the coefficients of *a* and the coefficients of *b*. Suppose *A* is a useful coefficient of *a* and that *(A)a => (B)b.* After an initial read, pairs of ampersands have each reduced to one, and the odd one has resolved *a* to *b.* So by subtracting the one odd ampersand from *A* and dividing by 2, we get *B.*
$B = (A - 1) / 2$ *(1)* or
$A = 2B + 1$ *(2)*

The key to finding the *n*th term of the recursive sequence is noticing that the coefficients of *b* that correspond to useful coefficients of *a* form the same sequence. Below is a table summarizing the resolution only of coefficients.

Number of ampersands	Resolution
1	(1)*a* => *b*
3	(3)*a* => (1)*b* => *c*
7	(7)*a* => (3)*b* => (1)*c* => *d*
15	(15)*a* => (7)*b* => (3)*c* => (1)*d* => *e*

According to the pattern, the next value of *A* or the next value in our recursive sequence will have a coefficient of 15 for *B.* By applying equation (2) from above, we can see that 31 is next in the sequence. As a general rule, knowing the *n*th term, S_n, we can calculate S_{n+1} by the following.

$S_{n+1} = 2S_n + 1$

If you do not like recursive formulas, the following exponential function of *n* is mathematically equivalent.

$S_{n+1} = 2^{n+1} - 1$

MACRO VARIABLE REFERENCES OF THE FORM *(n)a&b*

Up to this point, we've only discussed macro variable references of the form *(n)A*, or the name of a single macro variable preceded by one or more ampersands. Within this scope, we have seen that even values for *n* are no more useful than half that many ampersands. Even certain odd values for *n* proved not to be useful because somewhere before complete resolution, they reduced to an even coefficient. But is an even *n* always bad? Might there be forms of variable references where an even coefficient is beneficial?

Let us return to our macro %sugi2. Suppose we want to expand its capabilities by allowing the user to specify any number of cities in the parameter (e.g. %sugi3(seattle montreal)). For each city specified, the log will contain a message about which SUGI was held in that city. In other words, the %PUT statement that executed once in %sugi2 will now have to execute once per city listed. Also, unlike %sugi2 when a reference to the macro parameter was a reference to the city name, we now need to somehow extract each city name from the parameter and create a reference to it. This reference will be needed in the %PUT statement. So we see that each city listed requires some individual processing – identifying it and assigning it to a macro variable, and a %PUT statement to write the message to the log. However, in writing the macro, we don't know ahead of time how many individual cities will need this processing because it varies from one execution to the next. Luckily, the %DO - %UNTIL loop is equipped to handle this variability, executing the contents of the loop until there are no more cities listed in the parameter.

```
%macro sugi3(city) ;
%let INDIANAPOLIS=25;
%let LONG_BEACH=26;
%let ORLANDO=27;
%let SEATTLE=28;
%let MONTREAL=29;

%let i=1;
%do %until(%scan(&city,&i,%str( ))=%str());
  %let cty&i=%upcase(%scan(&city,&i,%str( )));
  %put /* CITY X */ held SUGI /* SUGI NUMBER Y */ ;
  %let i=%eval(&i+1);
%end;
%mend;
```

With the exception of the %PUT statement, %sugi3 is complete, but before we finish it off, let us make an observation. With the expanded capability of this macro, we have subtly introduced more variability – specifically, the number of cities the user can input. Because of that, we could not get away with a specific number of %LET statements, explicitly assigning city names to known variable names. The number of %PUT statements varies in the same way. The %DO-%UNTIL block allows us to manage this fact. Inside that block, *i* is created to keep track of the iteration of the loop. By processing one city per iteration, we can use *i* to systematically assign unique macro variables to each city listed. More generally, this macro variable also serves the purpose of addressing each of the possible circumstances. As before, the resolution of this macro variable will determine the appropriate macro variable to be referenced and resolved.

How do we replace /* CITY X */ in the %PUT statement? In sugi2, we only needed a simple reference to *&city*. However, such a reference in this case would produce all the cities requested. This is partly why *cty&i* was created – so that we could refer to an individual city. But when the name of the macro variable contains a reference to another macro variable, how do we refer to it? How many ampersands do we place in front of it to get the resolution we need?

It does not take long to see that one ampersand is not enough. In trying to resolve *&cty&i*, the macro processor first tries to resolve *&cty*. Since *cty* does not exist, the processor continues reading left to right and resolves *&i*. On the first iteration, after the first read, *&cty1* remains. However, according to Rule 3, the macro processor will not process a second read since no consecutive ampersands appeared in the original reference. But notice what happens with *&&cty&i*. We now have the consecutive ampersands to force a subsequent read, but because we have an even number of them, we know that the macro processor will not attempt to resolve anything before *&i*. Again, &CTY1 remains after the first read, but with a subsequent read, this resolves to the first city listed in the parameter.

So we see here that the effect of an even coefficient is that of delayed resolution. With an even coefficient, *n,* we are asking the macro processor to wait to resolve the entire reference *(n)a&i* until a part of the reference, *&i,* is resolved. Once *&i* is resolved and *n* is cut in half, we are back to the form *(n)a.*

Now that the reference form has changed to *(n)a&i,* we have a whole new set of useful value for *n* that includes 2. We could take an approach similar to the one we took when we discovered the pattern of useful coefficients for *(n)a,* but we have a shortcut sitting right in front of us. We just noticed that after the first read, *&i* is resolved, the coefficient of *a* is cut in half, and we are left with the form *(n)a.* If the first read is to yield a useful coefficient of *a,* and we know what those are, then the useful coefficients of *a* in the form *(n)a&i* are twice the useful coefficients of *a* in the form *(n)a.* To illustrate, let's complete the %PUT statement in SUGI3.

We know that /* CITY X */ can now be replaced with *&&cty&i*. We know that n=3 is useful for *(n)a,* so let us see what six ampersands will do for *(n)a&i.* Suppose the user executes %sugi3(INDIANAPOLIS SEATTLE). On the first iteration of the %DO loop,
&&&&&&cty&i => *&&&cty1* => *&INDIANAPOLIS* => 25.

The second iteration yields
&&&&&&cty&i => *&&&cty2* => *&seattle* => 28.

The final %PUT statement reads as follows:

```
   %put &&cty&i held SUGI &&&&&&cty&i ;
```

In sugi2, the macro parameter itself was the macro variable used to determine which macro variable would provide the desired information. Its resolution was the city specified by the user – a macro variable itself. Because of the nature of sugi3, a little more processing was needed to create such a macro variable. In the end, it was *i* that played this role. Because *i* tracked the iteration number, its resolution determined which part of the parameter to process and ultimately, which city to process.

In this paper, we've only looked at two forms of macro variable references – *(n)a* and *(n)a&b.* We've seen the need for forms like these and have examined in great detail the consequences of the rules of consecutive ampersands to determine what values of *n* make these forms useful and which do not. Of course, countless other forms exist that serve their own purposes with their own sets of useful values. For example, we just studied *(n)a&b* where *a* was <u>not</u> a macro variable. Things change if *a* <u>is</u> defined in this form. How would *&&&cty&i* resolve if *cty* did exist? What about *&&&&&&&cty&i*? Other forms include *(n)a(m)b* where *m>1,* and *(n)a(m)b(q)c.* Experimenting with different combinations of these and other forms can show you how much the macro processor is capable of and ultimately give you an idea as to how to set up and name macro variables in your own applications to make use of any of these forms.

ANOTHER SPECIAL CHARACTER

Ampersands are not the only characters that mean something to the macro processor. The macro processor does not need a macro variable reference to stand alone. Wherever an ampersand shows up, the macro processor will try to resolve anything that follows, including within a string of text not meant to be resolved. Consider the following code.

```
   %let state=CHIGAN;
   %put MY FAVORITE STATE THAT STARTS WITH MI IS MI&state ;
```

Upon execution, the macro processor finds the ampersand that follows the text MI, resolves all that follows, and writes to the log

MY FAVORITE STATE THAT STARTS WITH MI IS MICHIGAN.

With this much flexibility, one might ask, when a macro variable reference is found at the <u>beginning</u> of a text string, how does the macro processor know when the reference ends and the text begin? Consider a macro which generates a PROC SUMMARY and provides statistical summaries (e.g. sum, std, max, min, mean) requested by the user through a macro parameter called STAT. The summary statistics in the output data set are to be named by the statistic followed by the text string "summary." For example, if the max of the analysis variable is requested, the output data set will contain a variable called MAXSUMMARY.

With this example, we see that the text string is preceded by the value of a macro variable. In trying to name this variable in the OUTPUT statement, it might be tempting to try to use &statsummary, the macro variable followed by the text string. The problem is that the macro processor thinks you are trying to refer to a macro variable called STATSUMMARY which does not exist, and so an error is issued in the log. For situations like this, the macro processor recognizes the period (.) as a delimiter to signify the end of a macro variable reference.

```
   %macro summary(stat);
   proc summary;
   class var1;
   var var2;
   output out=ds1 &stat(var2)=&stat.summary ;
   run;
   %mend;
```

Note the period between the reference to *stat* and the text string "summary." One other observation is noteworthy. Notice that a period was not used between &stat and the opening parenthesis. That is because macro variable names may not contain parentheses so the macro processor knows that the open parenthesis is not part of a macro variable name. Therefore a delimiter was not necessary.

Suppose now that we expand the capability of the macro to allow for more than one statistic to be requested. In a way similar to the way we did with %sugi3, assume the macro begins with code that assigns unique macro variables *&st1*, *&st2*, …, to each of the requested statistics. We know we can make use of the form *&&a&b* inside a %DO loop to correctly generate the variable names we want to use. However, notice what happens when the macro processor tries to resolve *&&stat&b.summary* on the 1st iteration.

&&st&b.summary => *&st1summary* => ?

The problem is that two reads were required. *&b* resolved without any confusion with the help of the period, but on the second read, no delimiters were around to clear up any confusion. A second period clears this up. Below is the macro code for the OUTPUT statement.

```
   OUTPUT OUT=DS1

   %do b=1 %to &a;   /* where a is the number of statistics requested */
          &&st&b(var2)=&&st&b..summary
   %end;
   %str(;)
```

So if %summary(max) is called, then *&st1*=>max, and *&&st&b..summary* => *&st1.summary* => *maxsummary*. The key here is to know how many reads will be necessary to resolve the macro variable immediately before the text string and use that many periods. If that macro variable is preceded by another macro variable, be sure enough periods are used to delimit those references too. So what happens if too many periods are present?

Maybe we want periods to be present after final resolution is compete. Suppose we want to create a text file from the output data set, naming it only with the first statistic named in the parameter. Consider the following FILENAME statement.

```
FILENAME STATS "C:\SUMMARY&ST1..TXT" ;
```

With only one read by the macro processor, why do we use two periods? Keep in mind that a period that immediately follows a macro variable reference is used strictly as a delimiter and is not part of the final resolution. The first period serves this purpose. However, we want *.txt,* with the period, to be part of the final resolution. The second period is not trying to serve as a delimiter, but rather, to be part of the end result.

The macro processor does not need a macro to process periods this way. Consider the following sample code found in the beginning of a program.

```
%let libref=work;
DATA &libref..ds1 ;
more statements
```

Again, the first period serves as a delimiter, leaving one period to separate the two levels of the data set name.

CONCLUSION

In this paper, I have set out to illustrate that the otherwise difficult topic of macro variable resolution in the presence of consecutive ampersands can be made easy to understand by knowing three simple rules the macro processor follows. After stating these rules, I observed consequences of them based on two particular forms of references. With these two forms, we discovered the presence of patterns in the number of ampersands that could be added to yield unique resolutions. We then stated those patterns in mathematical terms. It is important to understand that, just as no math text book or entire curriculum can possibly provide exercises in all the mathematical patterns that occur in nature, the forms of macro variable resolution examined in this paper in no way represent all possible ways consecutive ampersands can be useful. However, just as a good text book explains concepts, clearly states rules in an understandable way, provides examples, and gives the student the tools needed to discover other patterns on his/her own, this paper is intended to provide the reader with an understanding of how the macro processor processes such references. After reading through the examples provided here and seeing that patterns do exist, the reader should now have the tools to discover other uses for consecutive ampersands that apply to their own applications, and associated mathematical patterns that follow.

CONTACT INFORMATION

I am happy to answer any questions you may have regarding this subject. Please feel free to contact me in any of the following ways.

Mike Molter
Howard Proskin and Associates
300 Red Creek Drive
Rochester, New York
Phone: (585) 359-2420
Fax:
E-Mail: