Paper PO26

SAS/MACRO® NOTES: Lines and Columns in the Log

Paul D Sherman, San Jose, CA

ABSTRACT

Do you shy away from macros for lack of their source line number error reporting? Are those bogus (Line):(Column) values driving you nuts? If so, this article is for you. The SAS Macro Language is a good thing. However, locations of errors from code placed inside macros are reported in a slightly different way than from traditional open code. We show how to understand both macro and open code, and present a few clear and simple examples with some techniques for quickly debugging SAS Macros.
Skill Level: Intermediate to Advanced Base/SAS users, especially those timid of macros.

INTRODUCTION

How often have you encountered a note in the SAS Log like the following?

```
NOTE: Character values have been converted to numeric values at the places
      given by:
      (Line):(Column).
      1:27

NOTE: Numeric values have been converted to character values at the places
      given by:
      (Line):(Column).
      1:27
```

Perhaps you have seen a message like this:

```
NOTE: Missing values were generated as a result of performing an operation
      on missing values.
      Each place is given by: (Number of times) at (Line):(Column).
      143 at 1:54      2557 at 1:54      79874 at 1:54
```

Or like this:

```
NOTE: Invalid numeric data, ..., at Line 1 Column 31.
NOTE: Invalid character data, ..., at Line 1 Column 31.
```

Obviously, your code has more than one line, and the cause of the error generating these notes is usually *not* on the first line. These notes come from code which is buried within a macro. Unlike open code, where LINE refers to the actual line of source code, the macro "line" is actually a block of statements.
Line numbers advance across macro and non-macro statement boundaries. When your macro contains only a single data step, you will always report Line#1 as the error location, no matter how deep in your code that the macro is called. The line number takes on an altogether different meaning when dealing with macros.

DEFINITION

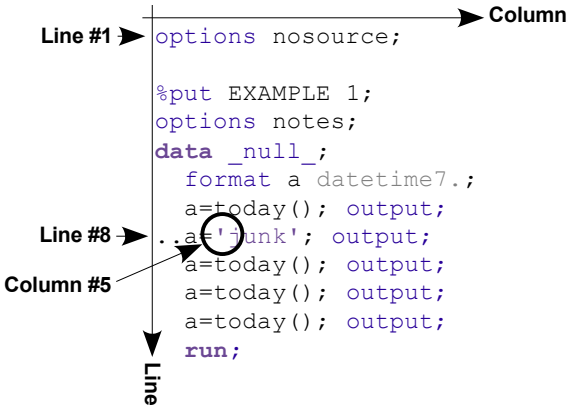
To summarize how the Log reports LINES and COLUMNS:

	Open Code	***Macro Code***
`NOTE:` reports	(Line):(Column)	(Block):(Character)
LINE represents ...	Submitted SAS line	Statement "block"
LINE is ...	Persistent as of start of session	Relative to each submit
COLUMN is ...	Relative to left margin	From end of last statement block

Presented below are examples of open and macro code, a description of how the three types of comments affect error position reporting, how this picture changes when viewed from the `Call Execute` environment, and a suggestion for "best practice" SAS program development using the Log Manager, which relaxes the need for knowing precise Line and Column positions of program errors.

OPEN-CODE EXAMPLE

In open code, LINE and COLUMN numbers are determined precisely for each statement in a `Data` step, and reported in an easy to read X-Y fashion. However, LINE numbers are persistent to a SAS session and advance each time source code is submitted as we will see later.



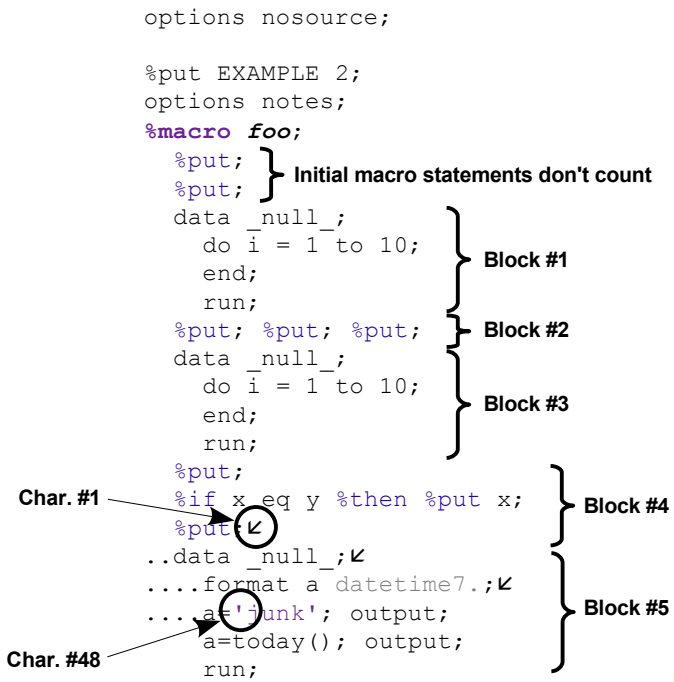

```
options nosource;

%put EXAMPLE 1;
options notes;
data _null_;
  format a datetime7.;
  a=today(); output;
..a='junk'; output;
  a=today(); output;
  a=today(); output;
  a=today(); output;
  run;
```

```
1    options nosource;
EXAMPLE 1

NOTE: Character values have been
converted to numeric values at the
places given by:
      (Line):(Column).
      8:5
NOTE: Invalid numeric data, 'junk' ,
at line 8 column 5.
a=01JAN60 _ERROR_=1 _N_=1
NOTE: DATA statement used:
      real time          0.00 seconds
      cpu time           0.00 seconds
```

MACRO-CODE EXAMPLE

Wrapped inside a macro, all code constitutes a single line with the LINE number being the number of macro and non-macro statement blocks before the bad code. The COLUMN number refers to the character position of the offending term in the bad statement block.

Column #1 starts immediately following the last known good statement block. Usually, column #1 is the newline character, depending on how you format your code. Macro and non-macro statement blocks are counted by the LINE number, which begins at Line #1 with the first non-macro statement, and source line separators (i.e., terminators) are newline characters.




```
options nosource;

%put EXAMPLE 2;
options notes;
%macro foo;
  %put;
  %put;
  data _null_;
    do i = 1 to 10;
    end;
    run;
  %put; %put; %put;
  data _null_;
    do i = 1 to 10;
    end;
    run;
  %put;
  %if x eq y %then %put x;
  %put;↙
..data _null_;↙
....format a datetime7.;↙
....a='junk'; output;
    a=today(); output;
    run;
  %put;
%mend;↙
%foo;
```

```
1    options nosource;
EXAMPLE 2

NOTE: DATA statement used:
      real time          0.00 seconds
      cpu time           0.00 seconds

NOTE: DATA statement used:
      real time          0.00 seconds
      cpu time           0.00 seconds

NOTE: Character values have been
converted to numeric values at the
places given by:
      (Line):(Column).
      5:48
NOTE: Invalid numeric data, 'junk' ,
at line 5 column 48.
a=01JAN60 _ERROR_=1 _N_=1
NOTE: DATA statement used:
      real time          0.00 seconds
      cpu time           0.00 seconds
```

Note: Dots (.) and newlines (↙) added to the macro declarations above for ease of character counting. See Appendix for more detail on the LINE and COLUMN reporting process.

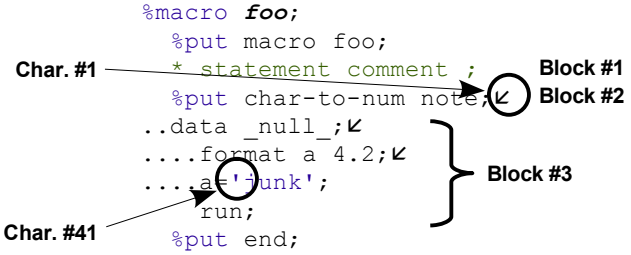
COMMENTS ABOUT COMMENTS

Every SAS programmer knows that there are at least three ways to comment code: The block comment, macro comment, and statement comment. The methods of commenting code differ by when their text is removed.

Block comment	`/* ... */`	Any characters, removed prior to macro resolution.
Macro comment	`%* ... ;`	Removed during macro resolution, just before the first pass.
Statement comment	`* ... ;`	Removed after macro resolution.

STATEMENT COMMENT

Because the statement comment survives macro resolution, it is considered a valid statement for the purposes of (Line):(Column) "block" counting. Our bad `Data` step is the third block.




```
%macro foo;
  %put macro foo;
  * statement comment ;
  %put char-to-num note;
..data _null_;
....format a 4.2;
....a='junk';
  run;
  %put end;
%mend;
%foo;
```

```
macro foo
char-to-num note
NOTE: Character values have been
converted to numeric values at the
places given by:
      (Line):(Column).
      3:41
NOTE: Invalid numeric data, 'junk' ,
at line 3 column 41.
a=. _ERROR_=1 _N_=1
NOTE: DATA statement used:
    real time           0.00 seconds
    cpu time            0.00 seconds
end
```

Unlike in open code when we "turn off" a statement as in `*%dont_run_me_now;` it is *not* appropriate to comment a macro statement using a statement comment from within a macro. Why? Because the macro processor resolves text before the statement comment takes the text out. Some examples illustrate this:

```
%macro foo; *%put this; data _null_; put 'that'; run; %mend; %foo;
%macro foo; *%let a=1; data _null_; put 'that'; run; %mend; %foo;
```

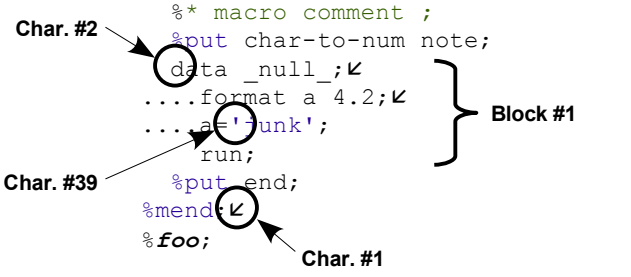
Both of these lines generate errors. The lines become

```
%macro foo; *          data _null_; put 'that'; run; %mend; %foo;
```

and it is easy to see why an error occurs. The underlined portion of the comment—including the semicolon—resolves away and performs what it says, either printing `this` on the log or assigning 1 to macro variable A. The star (`*`) remains in place, which subsequently comments out the `Data _null_` statement. As a result, an error occurs with the remaining out-of-place `Data` step statements.

BLOCK AND MACRO COMMENTS

Using either of the other two styes of comments means that text is taken away long before the macro interpreter processes the macro statements. The bad `Data _null_` step is the first non-macro statement block, and the log reflects this accordingly. Being the first block, we start counting characters with the newline character at the end of the macro declaration `%mend;` statement.




```
%macro foo;
  %put macro foo;
  /* block comment */
  %* macro comment ;
  %put char-to-num note;
  data _null_;
....format a 4.2;
....a='junk';
  run;
  %put end;
%mend;
%foo;
```

```
macro foo
char-to-num note
NOTE: Character values have been
converted to numeric values at the
places given by:
      (Line):(Column).
      1:39
NOTE: Invalid numeric data, 'junk' ,
at line 1 column 39.
a=. _ERROR_=1 _N_=1
NOTE: DATA statement used:
    real time           0.00 seconds
    cpu time            0.00 seconds
end
```

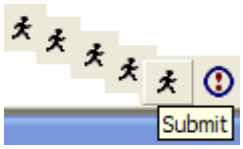
OPEN-CODE (LINE):(COLUMN) NOTES PERSIST

OR, WHY WE SHOULD USE MACROS

Within a SAS session, the LINE numbers reported in log notes persist and accumulate with each program submission. The only way to clear and reset these LINE numbers is to terminate SAS and start again.

Clicking Run and submitting the program line

```
1
 options nosource; data _null_; format a 4.2; a='junk'; run;
```



five times tells us, each time, that invalid data lies on

```
NOTE: Invalid numeric data, 'junk' , at line 1 column 48.
NOTE: Invalid numeric data, 'junk' , at line 2 column 48.
NOTE: Invalid numeric data, 'junk' , at line 3 column 48.
NOTE: Invalid numeric data, 'junk' , at line 4 column 48.
NOTE: Invalid numeric data, 'junk' , at line 5 column 48.
```

It is very awkward for the same line of code to report different line numbers. This is why we recommend a "best practice" of only one open code statement, the statement which starts your top-level main macro.

```
2                                                                1
%macro main; data _null_; format a 4.2; a='junk'; run; %mend; %main;
```

No matter how many times we submit this program line, we are told every time that invalid data is at

```
NOTE: Invalid numeric data, 'junk' , at line 1 column 31.
NOTE: Invalid numeric data, 'junk' , at line 1 column 31.
...
```

BEST PRACTICE EXAMPLE

Suppose you need to transpose and print a dataset. You might write a five-step program as follows.

```
data junk; set mydata; run;
proc sort data=junk nodupkey; by ptid testname; run;
proc transpose data=junk out=flat; by ptid; id testname; var testval; run;
proc print data=flat noobs; run;
proc datasets lib=work; delete junk flat; run; quit;
```

For reasons described above, if there is a problem with the data content, and you repeatedly submit this program during the course of debugging, you will get a different LINE number each time. Confusing.

Each step of your program is doing something different, so you "separate" them into their own macros. Use suitable and concise active verbs for the macro names to enhance human readability of the program. As a general rule functions, methods or macros perform actions so their names should be verbs; tables and datasets are objects (or, instances of classes or structures) and their names should be nouns.

The sorting step, however, is a prerequisite for transposing so these two steps belong together. Steps which are related should be "encapsulated" into the same macro operation.

```
%macro copydat(indat,outdat); data &outdat.; set &indat.; run; %mend;
%macro xpose(i, o);
  proc sort data=&i. nodupkey; by ptid testname; run;
  proc transpose data=&i. out=&o; by ptid; id testname; var testval; run;
%mend;
%macro showdat(indat); proc print data=&indat. noobs; run; %mend;
%macro remove(x); proc datasets lib=work; delete &x.; run; quit; %mend;
%macro main(thedata);
  %copydat(&thedata., junk);
  %xpose(junk, flat);
  %showdat(flat);
  %remove(junk flat);
%mend;
%main(mydata);
```

Separate

Encapulate

Open code: program entry point

SAS log LINE numbers never change with repeated program submissions. Later on we show how the Log Manager allows a program problem to nearly debug itself.

THE CALL EXECUTE() ENVIRONMENT

When macro programs are invoked dynamically in a `Data` step there are multiple phases of processing which generate (Line):(Column) notes. The first phase reports positions of errors in the outer-most macro, with (Block):(Character) described above.

During the `Call Execute` phase, all gray macro statements have been resolved and removed so that a "lean" macro with a single block remains. Notes report (Loop):(Char) to reflect the `Data` step loop count.

```
options nosource;

%macro mymac(n);
  data _null_; do i=1 to &n.; end; run;
  %put mymac(&n.);
..data _null_;
....put 'mymac data step';
....format a b 4.1;
....a=&n.;
....b='junk';
      put a;
      put 'mymac data done';
      run;
  %put mymac done;
%mend;

%macro iterate;
  * statement comment ;
  %put iterate;
..data _null_;
....put 'iterate data step';
....do i = 3, 10;
......call execute(
........'%mymac('||i||')'
      );
      end;
      put 'iterate data done';
      run;
  %put iterate done;
%mend;
%iterate;
```

Char. #2 → `&n.` in `%put mymac(&n.);`
Char. #22 → `....put 'mymac data step';`
Char. #114 (Char. #115) → `....b='junk';`
Char. #1, always → `%mend;`
Char. #1 → `* statement comment ;`
Char. #60 → `....do i = 3, 10;`
Char. #106 → `i` in `'%mymac('||i||')'`

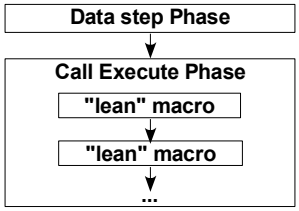
Block #1
Block #2
Block #3
Block #4

```
iterate

NOTE: Character values have been
converted to numeric values at the
places given by:
      (Line):(Column).
      3:60
NOTE: Numeric values have been
converted to character values at the
places given by:
      (Line):(Column).
      3:106
iterate data step
mymac(3)
mymac done
mymac(10)
mymac done
iterate data done

NOTE: Character values have been
converted to numeric values at the
places given by:
      (Line):(Column).
      1:22
NOTE: Character values have been
converted to numeric values at the
places given by:
      (Line):(Column).
      1:114
mymac data step
NOTE: Invalid numeric data, 'junk' ,
at line 1 column 114.
3.0
mymac data done

NOTE: Character values have been
converted to numeric values at the
places given by:
      (Line):(Column).
      2:22
NOTE: Character values have been
converted to numeric values at the
places given by:
      (Line):(Column).
      2:115
mymac data step
NOTE: Invalid numeric data, 'junk' ,
at line 2 column 115.
10.0
mymac data done

iterate done
```

Program scope
(Block):(Char)
Call Execute scope
(Loop):(Char)
Loop #1
Loop #2
10 is two chars wide

When all macro variables have been resolved, shown gray above, the reported character count depends on the width of the macro variable values. In this example, during the first loop `&n.` resolves to 3 and is a single width. The second loop has `&n.` resolving to 10 which is two characters wide.

Some things to remember: All statements of a `Data` step are executed *before* the target lean macro of `Call Execute` is invoked. The number of iterations of a `Data` step is precisely the number of copies of the non-macro portions of a `Call Executed` macro.



BEST PROGRAMMING PRACTICE WITH THE LOG MANAGER

Separate code which doesn't belong together, encapsulate code which does, and explicitly send status message notifications ❸ and ❹ at each critical step along the way.

```
options nosource nonotes;
%include "\\...\...\bloknote.sas";
%put EXAMPLE 3;
%macro init;
  %bloknote(OPEN,init());
  %bloknote(CLOSE);
%mend;
%macro dothis;
  %bloknote(OPEN,dothis());
  %bloknote(CLOSE);
%mend;
%macro dothat;
  %bloknote(OPEN,dothat());
  %bloknote(CLOSE);
%mend;
%macro remove(thedata, lib=work);
  %bloknote(OPEN,remove(&thedata.));
  proc datasets lib=&lib. nolist nowarn;
    delete &thedata.;
    run; quit;
  %bloknote(CLOSE);
%mend;
%macro finalize;
  %bloknote(OPEN,finalize());
  %bloknote(CLOSE);
%mend;
%macro make(thedata);
  %bloknote(OPEN,make(&thedata.));
❸ %putlog(creating dataset work.&thedata.);
  data work.&thedata.;
    format a datetime25.6;
    a=today(); output;
    a='junk'; output;
    run;
❹ %remove(&thedata.); 👍
  %bloknote(CLOSE);
%mend;
❷ %macro main;
  %bloknote(START,main());
  %init; 👍
  %dothis; 👍
  %dothat; 👍
❶ %make(foo); ☹
  %finalize; 👍
  %bloknote(END);
%mend;
%main;
```

Trouble is somewhere in here

```
1    options nosource;
EXAMPLE 3
❷ main() {
    init() { 👍
    }
    dothis() { 👍
    }
    dothat() { 👍
    }
❶ make(foo) { ☹
    ❸creating dataset work.foo
a=01JAN1960:04:42:52.000000
_ERROR_=1 _N_=1
    ❹remove(foo) { 👍
      }
    }
    finalize() { 👍
    }
}
```

"In the ❶ **%make** invocation of macro ❷ **%main**, there's an offending statement somewhere between ❸ *creating* and ❹ *removing* data set **work.foo**"

The Log Manager gets you very close to the problem very quickly without any NOTE: information. You still don't know exactly what source line, but you're very close.

CONCLUSION

There are three ways that SAS reports notes in the log on the positions of data errors. From open code we all know how to interpret (Line):(Column). Inside a macro the notes report (Block):(Character), while the `Call Execute` environment reports (Loop):(Character). Character position count is affected somewhat by the presence of comments, especially those which remain following macro resolution. Contrary to the effort of debugging those seemingly bogus (Line):(Column) values, it is recommended that the SAS programmer follow an object-oriented style of program development. Any program problem can be easily debugged without attention to `NOTE:` or `WARNING:` by separating, encapsulating and explicitly notifying your program macros.

REFERENCES

Berryhill, Tim. "Re: macro debugging using log" in *SAS(r) Discussion*, SAS-L, Item 11420, 10 Dec 1996. <http://listserv.uga.edu/cgi-bin/wa?A2=ind9612B&L=sas-l&P=R4007&D=0>

Delaney, Kevin P. and Art L. Carpenter. "SAS Macro: Symbols of Frustration? %Let us help! A Guide to Debugging Macros," in *Proceedings of the Twenty-ninth Annual SAS User Group International Conference*; 2004 May 9-12; Montreal, Quebec; Paper 128-29.

Diskin, Dennis. "Re: Wierd Line Numbers in Log" in *SAS(r) Discussion*, SAS-L, Item 118779, 24 Jan 2003. <http://listserv.uga.edu/cgi-bin/wa?A2=ind0301D&L=sas-l&D=0&m=107310&P=23834>

Maurer, Scott. "Re: macro debugging using log" in *SAS(r) Discussion*, SAS-L, Item 11447, 11 Dec 1996. <http://listserv.uga.edu/cgi-bin/wa?A2=ind9612B&L=sas-l&P=R5265&D=0>

SAS Technical Support. "Errors reported from DATA step view may not match SAS Log line numbers," SN-007051.

Sherman, Paul D. "Intelligent SAS Log Manager," in *Proceedings of the Twenty-sixth Annual SAS User Group International* Conference; 2001 April 22-25; Long Beach, California; Paper 108-26.

ACKNOWLEDGMENTS

I am very grateful to Rohit Dhanjal, Sunil Gupta, Dr. Mandyam Srirama, and the entire team at Quintiles for extremely useful discussions on techniques for debugging SAS macros which form the basis for this article. My sincere gratitude goes to Issam Elayle for testing example code on the SAS version 9.1 System. I thank Kirk Lafler, Patrick Thornton and Lauren Haworth for the exciting opportunity to present this work in their Data Presentation and Business Intelligence section at WUSS 2006.

RECOMMENDED READING

Carpenter, Art. *Carpenter's Complete Guide to the SAS® Macro Language*, Cary, NC: SAS Institute., 1998. 244pp.

Delwiche, Lora D. and Susan J. Slaughter. 2003. "Debugging your SAS Programs," Chapter 10 in *The Little SAS Book: A Primer, Third Edition*. Cary, NC: SAS Institute, Inc., pp252-280.

CONTACT INFORMATION

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

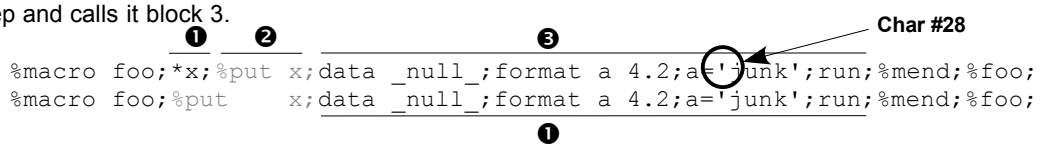
Paul D Sherman
Electrical Engineer
335 Elan Village Lane, Apt. 424
San Jose, CA 95134
Home Phone: (408) 383-0471
Email: sherman@idiom.com

Web site: `http://www.idiom.com/~sherman/paul/pubs/macnotes`

APPENDIX – WHAT REALLY HAPPENS BEHIND THE SCENES

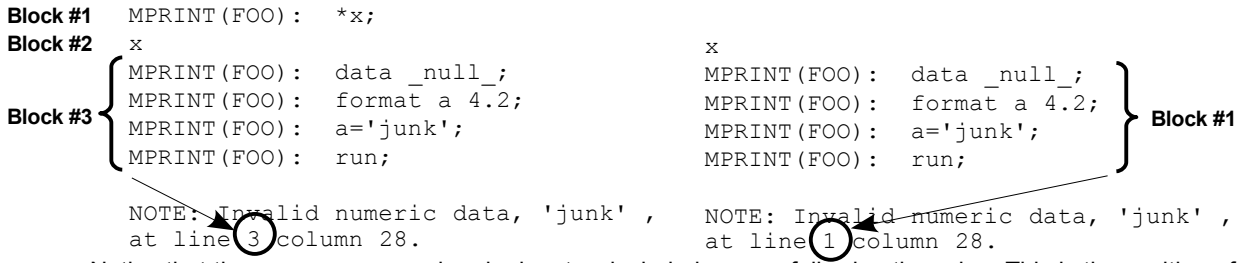
Although it is intuitive to count blocks and characters of the source program code, the real behavior of (Line):(Column) error reporting comes from looking at the macro processor itself with `options mprint;` turned on.

Consider two macros with an error at the same character position #28. Each time the macro processor switches from resolution to execution mode the "block" counter is incremented. In the first macro, the statement-comment is block 1, while the macro `%put` statement is executed immediately by the macro processor and serves to increment the block counter to 2. Subsequently the processor resolves the `Data` step and calls it block 3.



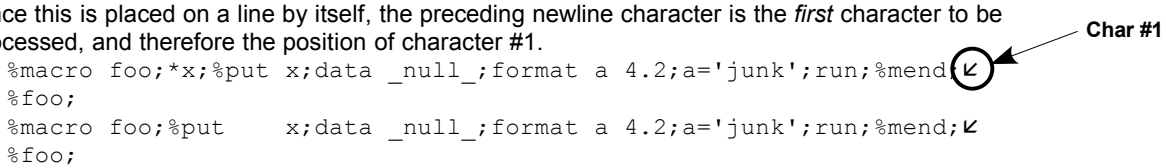
In the second macro, leading macro statements are executed immediately and therefore need no further macro processor resolution (i.e., `MPRINT` actions). Only when the `Data` step is encountered does resolution begin and this is at block #1.

The first macro log output is shown on the left, and the second macro log is on the right below.

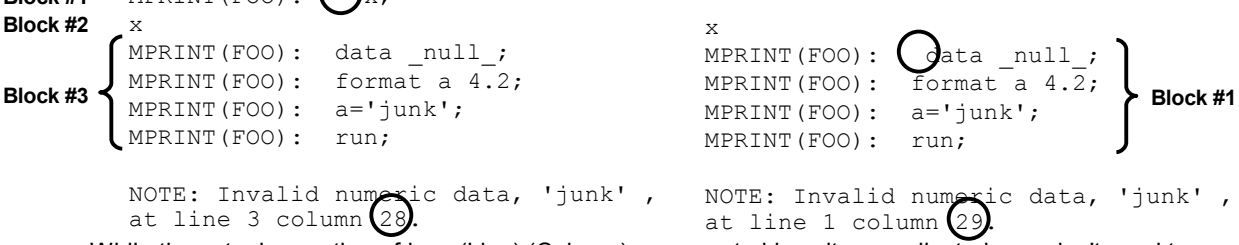


Notice that the `MPRINT(...):` header has two included spaces following the colon. This is the position of character #1.

Things get a little tricky when macro statement lines are broken up on multiple lines for good source code readability. At the end of each line there is a newline character (↙). The first macro statement to be processed by the macro processor is not the `%macro` definition itself but rather the invocation statement. Since this is placed on a line by itself, the preceding newline character is the *first* character to be processed, and therefore the position of character #1.



Block 1 of both macros reflect the newline character. In the first macro, it is block 3 which contains the error so its character count is seemingly unaffected by the new placement of the invocation statement. The second macro has only one code execution block and thus reflects the newline character. Additional lines preceding invocation statement will further increment the character count.



While the actual operation of how (Line):(Column) are reported is quite complicated, you don't need to stare at `MPRINT` logs to debug your code. Just remember a few simple rules and you will always be able to find the location of errors in your program source.

1. Block numbers start with non-macro code, and increment around macro code boundaries.
2. Character position #1 of block #1 starts immediately following the `%mend;` statement.
3. Statement comments count as non-macro code blocks and do not remove macro code.

EXAMPLE 1

```
options nosource;
/* Interpreting (Line):(Column) from
open code */
%put EXAMPLE 1;
options notes;
data _null_;
  format a datetime7.;
  a=today(); output;
  a='junk'; output;
  a=today(); output;
  a=today(); output;
  a=today(); output;
  run;
```

EXAMPLE 2

```
options nosource;
/* Interpreting (Line):(Column) from
macro code */
%put EXAMPLE 2;
options notes;
%macro foo;
  %put;
  %put;
  data _null_;
    do i = 1 to 10;
    end;
    run;
  %put; %put; %put;
  data _null_;
    do i = 1 to 10;
    end;
    run;
  %put;
  %if x eq y %then %put x;
  %put;
  data _null_;
    format a datetime7.;
    a='junk'; output;
    a=today(); output;
    run;
  %put;
%mend;
%foo;
```

EXAMPLE 3

```
options nosource;
/* Interpreting (Loop):(Char) */
/* from Call Execute macro code */
%put EXAMPLE 3;
%macro mymac(n);
  data _null_; do i=1,'2'; end; run;
  %put mymac(&n.);
  data _null_;
    put 'mymac data step';
    format a b 4.1;
    a=&n.;
    b='junk';
    put a;
    put 'mymac data done';
    run;
  %put mymac done;
%mend;

%macro iterate;
  * statement comment ;
  %put iterate;
  data _null_;
    put 'iterate data step';
    do i = 3, '10';
      call execute(
        '%mymac('|| i ||')'
      );
    end;
    put 'iterate data done';
    run;
  %put iterate done;
%mend;
%iterate;
```

EXAMPLE 4

```
options nosource;
/* Example courtesy of Delwiche & Slaughter, 3rd ed., p.264. */
data toaddata;
  input ToadName $ Jump1 Jump2 Jump3;
  cards;
Lucky  1.9   .   3.0
Spot   2.5  3.1  0.5
Tubs    .    .   3.8
Hop    3.2  1.9  2.6
Noisy  1.3  1.8  1.5
Winner  .    .    .
;
  run;

/* In open code, (Line):(Column) are reported as expected.            */
/*    Each place is given by: (Number of times) at (Line):(Column).   */
/*    3 at 20:24                                                      */
data _null_;
  set toaddata;
  AverageJump = (Jump1 + Jump2 + Jump3) / 3;
  run;

/* Within macros, (Line):(Column) become actually (Block):(Character) */
/*    Each place is given by: (Number of times) at (Line):(Column).   */
/*    3 at 1:51                                                       */
%macro toadjump;
data _null_;
  set toaddata;
  AverageJump=(Jump1+Jump2+Jump3)/3;
  run;
%mend;
%toadjump;

/*    Each place is given by: (Number of times) at (Line):(Column).   */
/*    3 at 3:102                                                      */
%macro toadjump;
data _null_; run;          /* Block #1 */
%put; %local x; %let x=1;  /* Block #2 */
data _null_;                          /* <----------+ */
  set toaddata;                       /*   Block #3 | */
  AverageJump=(Jump1+Jump2+Jump3)/3;  /*            | */
  run;                                /* <----------+ */
%mend;
%toadjump;
```

EXAMPLE 5

```
options nosource nonotes;
%include "\\...\...\bloknote.sas";  /* see References for this code */
/* Best practice for maintaining and debugging macro programs */
%put EXAMPLE 5;

%macro init; %bloknote(OPEN,init()); %bloknote(CLOSE); %mend;
%macro dothis; %bloknote(OPEN,dothis()); %bloknote(CLOSE); %mend;
%macro dothat; %bloknote(OPEN,dothat()); %bloknote(CLOSE); %mend;

%macro remove(thedata, lib=work);
  %bloknote(OPEN,remove(&thedata., lib=&lib.));
  proc datasets lib=&lib. nolist nowarn;
    delete &thedata.;
    run;
    quit;
  %bloknote(CLOSE);
%mend;

%macro finalize; %bloknote(OPEN,finalize()); %bloknote(CLOSE); %mend;

%macro make(thedata);
  %bloknote(OPEN,make(&thedata.));
  %putlog(creating dataset work.&thedata.);
  data work.&thedata.;
    format a datetime25.6;
    a=today(); output;
    a='junk'; output;
    a=today(); output;
    run;
  %remove(&thedata.);
  %bloknote(CLOSE);
%mend;

%macro main;
  %bloknote(START,main());
  %init;
  %dothis;
  %dothat;
  %make(foo);
  %finalize;
  %bloknote(END);
%mend;

%main;
```