Paper 4958-2020

A Close Look at How DOSUBL Handles Macro Variable Scope

Quentin McMullen, Siemens Healthineers

ABSTRACT

The macro variable scoping rules of the SAS® macro language are complex, and well-documented. The DOSUBL function, introduced in SAS 9.3M2, adds an additional layer of complexity to these scoping rules, as the macro programmer needs to understand how code executing in the DOSUBL side-session will create or update macro variables, and what impact this will have on macro variables stored in the main session symbol tables. Unfortunately, the current SAS documentation does not provide a clear definition of the DOSUBL scoping rules. This paper presents a series of test cases designed to illustrate DOSUBL's handling of macro variable scopes, and infers a set of DOSUBL macro variable scoping rules. The intended audience is experienced macro programmers interested in learning how DOSUBL manages macro variable scopes.

INTRODUCTION

The DOSUBL function, introduced in SAS 9.3M2, brings exciting new possibilities to the world of SAS programming. It changes many of the basic rules of SAS programming, such as "you cannot execute a DATA step inside of another DATA step," and "function-style macros cannot execute SAS statements." Not surprisingly, such power also brings with it additional complexity. Even without DOSUBL, the macro language has important rules for managing the scope of macro variables. DOSUBL introduces a new level of macro scopes, and new scoping rules. Unfortunately, the current SAS documentation does not provide a clear definition of the DOSUBL scoping rules. This paper presents a series of test cases designed to illustrate DOSUBL's handling of macro variable scopes, and infers a set of DOSUBL macro variable scoping rules.

DISCLAIMER

Most SAS Global Forum papers are written by an expert who thoroughly understands the subject matter at hand, and seeks to communicate that understanding to the reader by illustrating and explaining known features. This is not one of those papers. I am not an expert in DOSUBL, and I do not profess to have a complete understanding of how DOSUBL works. This paper presents an *investigation* into the manner in which DOSUBL handles issues of macro variable scope, with the hope of inferring DOSUBL macro scoping rules. The reader is forewarned that some of my inferences may be incorrect. I hope this paper will encourage others to explore DOSUBL, and share the results of their explorations with the SAS community.

MOTIVATING EXAMPLE

Prior to the introduction of DOSUBL in SAS 9.3M2, there was a golden rule of writing function-style macros: A function-style macro can contain only macro language statements; it cannot contain complete SAS language statements.[1] Function-style macros are designed to be executed within a SAS statement, and thus cannot execute SAS statements. DOSUBL

[1] This is not strictly accurate. Mike Rhoads (2012) presented an ingenious approach for writing function-style macros that contain SAS statements in SAS 9.2, which used the FCMP procedure and RUN_MACRO function.

allows the creation of macros which contradict that rule. The seminal DOSUBL paper by Rick Langston (2013) presents a function-style macro that returns a list of variables in a data set.[2] Below is a modified version of that macro:

```
%macro ExpandVarList(data=, var=_ALL_) ;
  %local rc temp_varnames ;
  %let rc = %sysfunc(dosubl(%nrstr(
    data __MyData ;
      set &data (obs=0) ;
    run ;
    proc transpose data=__MyData out=__ExpandVarList ;
      var &VAR ;
    run ;
    proc sql noprint ;
      select _name_ into :temp_varnames separated by ' '
      from __ExpandVarList
      ;
      drop table __ExpandVarList ;
      drop table __MyData ;
    quit ;
  )));
&temp_varnames
%mend ExpandVarList;
```

Because it is a function-style macro, it can be called as part of a SAS statement, e.g.:

```
data want ;
  varlist="%ExpandVarList(data=sashelp.shoes,var=_Numeric_)" ;
  put varlist= ;
run ;
```

returns:

```
1    data want ;
2      varlist="%ExpandVarList(data=sashelp.shoes,var=_Numeric_)" ;
MPRINT(EXPANDVARLIST):  "Stores Sales Inventory Returns"
3      put varlist= ;
4    run ;

varlist=Stores Sales Inventory Returns
```

As a function-style macro, the job of %ExpandVarList is to generate a list of variables and return that list. The magic of %ExpandVarList lies in the fact that it generates the list by somehow executing a DATA step and two PROC steps, *while the DATA WANT step is in the middle of compiling*. Consider the timing of the DATA WANT step. While it is compiling the assignment statement, the compiler sees the macro call. The macro processor executes %ExpandVarList. Thanks to the magic of DOSUBL, SAS immediately executes the DATA __MYDATA step, the PROC TRANSPOSE step, and PROC SQL step, all without interrupting the compilation of the DATA WANT step. When the macro execution completes it returns the list of variables, and the DATA WANT step completes compilation and executes.

When I first read Langston (2013) I was in shocked disbelief. How could DOSUBL, a function, be capable of compiling and executing multiple steps of code while the step that

[2] Interestingly, the DOSUBL ExpandVarList macro presented in Langston(2013) was inspired by the PROC FCMP ExpandVarList macro presented in Rhoads (2012). The PROC TRANSPOSE approach used in both macros was developed by John King (2012), and King (2017) presents a fully developed ExpandVarList macro that uses DOSUBL.

called the function is still compiling? The paper explained that it does this by submitting SAS code to run 'on the side.' But what does that mean?

DOSUBL AND THE "SIDE SESSION"

Consider the simple example:

```
data Main ;
  rc=dosubl('data Side;run;') ;
run ;
```

When the code is submitted, the SAS session executes the DATA MAIN step. This paper will refer to the usual SAS session as the 'main session.' When the DOSUBL function executes, I believe it creates a 'side session' for the purpose of executing the code passed as an argument to DOSUBL (i.e. the DATA SIDE step). I envision the side session as a separate, almost independent SAS session. The side session has its own system options (some of which are inherited from the main session), its own automatic macro variables (some of which are inherited), and its own global symbol table for global macro variables.[3] The side session uses the same WORK library as the main session. Each time DOSUBL executes, it creates a fresh side session and uses it to execute the code passed as an argument. Any data sets created by the side session are written to the WORK library shared with the main session. When DOSUBL completes execution, any global macro variables that were created in the side session are returned to the main session.

Like the program data vector (PDV), the side session is a logical construct. Understanding how the side session works is important to understanding how DOSUBL works, in the same way that understanding the PDV is critical to understanding how the DATA step works. As a SAS programmer, you do not need to know how the side-session in implemented in the machine code behind DOSUBL (is it *really* a separate session?), but you do need a logical model to understand how code executed in the side session behaves and interacts with the main session, so that you can use DOSUBL effectively and safely.

The DOSUBL documentation does not use the terms "main session" or "side session," but it is consistent with these constructs. The documentation states simply:

> The DOSUBL function enables the immediate execution of SAS code after a text string is passed. Macro variables that are created or updated during the execution of the submitted code are exported back to the calling environment.

If macro variables that are created by the DOSUBL function are returned "*back* to the calling environment" (i.e. "main session" which called DOSUBL), that suggests they were returned from some other environment where DOSUBL created them in the first place. Langston (2013) describes the DOSUBL function as being designed to "submit SAS code to run 'on the side' while your DATA step is still running." Thus, it seems reasonable to consider the "calling environment" as (typically) the "main session", and the environment where DOSUBL code runs as the "side session."[4]

---

[3] Proving the assertion that the side session inherits some system options and some automatic macro variables from the main session is outside the scope of this paper. The reader is encouraged to investigate by comparing the results of submitting PROC OPTIONS and %PUT _AUTOMATIC_ in both the main session and as an argument to DOSUBL.

[4] Note that one benefit of the documentation's use of the broader term "calling environment" and implied "DOSUBL environment" is that it correctly suggests the calling environment is not always the main session. It is possible to have a DATA step run in the main session, which calls DOSUBL, and the side session created by DOSUBL can call DOSUBL. Thus there can be nested side sessions, i.e. a side session can also be a calling environment. This paper will not address nested DOSUBL environments, thus "main session" and "side session" are clear terms.

A critical timing feature of DOSUBL is that the code passed as an argument is executed *immediately*. This distinguishes DOSUBL from CALL EXECUTE, in which the generated code cannot be executed until after that DATA step that invoked CALL EXECUTE has completed. The following code uses DOSUBL to create several output data sets from a single data set. SASHELP.PRDSALE has data for three countries (Canada, Germany, USA) and DOSUBL is called three times, executing one DATA step for each country-specific data set:

```
data _null_ ;
  set sashelp.prdsale ;
  by country ;
  if first.country then do ;
    put _n_= ;
    rc = dosubl('
      data '|| compress(country,".") || ' ;
        set sashelp.prdsale ;
        where country="' || trim(country) ||'" ;
      run ;
    ') ;
    MadeData=cats(exist("work.canada")
                 ,exist("work.germany")
                 ,exist("work.usa")
                  ) ;
    put MadeData= ;
  end ;
run ;
```

The log shows:

```
_N_=1
NOTE: There were 480 observations read from the data set SASHELP.PRDSALE.
      WHERE country='CANADA';
NOTE: The data set WORK.CANADA has 480 observations and 10 variables.

MadeData=100


_N_=481
NOTE: There were 480 observations read from the data set SASHELP.PRDSALE.
      WHERE country='GERMANY';
NOTE: The data set WORK.GERMANY has 480 observations and 10 variables.

MadeData=110


_N_=961
NOTE: There were 480 observations read from the data set SASHELP.PRDSALE.
      WHERE country='U.S.A.';
NOTE: The data set WORK.USA has 480 observations and 10 variables.

MadeData=111
NOTE: There were 1440 observations read from the data set SASHELP.PRDSALE.
```

The log shows that *while* the DATA _NULL_ step was executing in the main session, DOSUBL executed three DATA steps in the side session. On the first iteration of the DATA _NULL_ step, it read the first record with country='CANADA', and the DOSUBL function created a side session and executed the DATA CANADA step. On the 481st iteration of the DATA _NULL_ step, it read the first record with country='GERMANY', and the DOSUBL function created a side session and executed the DATA GERMANY step. On the 961st iteration it read

the first record with country='U.S.A.', and the DOSUBL function created a side session and executed the DATA USA step. On the 1441st iteration the DATA _NULL_ step completed.

When the same step is run using CALL EXECUTE instead of DOSUBL, the log shows that the timing is very different. The DATA _NULL_ step completes execution before any of the generated DATA steps have been executed (or even compiled).

```
_N_=1
MadeData=000
_N_=481
MadeData=000
_N_=961
MadeData=000
NOTE: There were 1440 observations read from the data set SASHELP.PRDSALE.

NOTE: CALL EXECUTE generated line.
1   + data CANADA ; set sashelp.prdsale ; where country="CANADA" ; run ;

NOTE: There were 480 observations read from the data set SASHELP.PRDSALE.
      WHERE country='CANADA';
NOTE: The data set WORK.CANADA has 480 observations and 10 variables.

2   + data GERMANY ; set sashelp.prdsale ; where country="GERMANY" ;run ;

NOTE: There were 480 observations read from the data set SASHELP.PRDSALE.
      WHERE country='GERMANY';
NOTE: The data set WORK.GERMANY has 480 observations and 10 variables.

3   + data USA ; set sashelp.prdsale ; where country="U.S.A." ;run ;

NOTE: There were 480 observations read from the data set SASHELP.PRDSALE.
      WHERE country='U.S.A.';
NOTE: The data set WORK.USA has 480 observations and 10 variables.
```

Look again at the definition of %ExpandVarList, and note that the code in bold is executed within the side session:

```
%macro ExpandVarList(data=, var=_ALL_) ;
  %local rc temp_varnames ;
  %let rc = %sysfunc(dosubl(%nrstr(
    data __MyData ;
      set &data (obs=0) ;
    run ;
    proc transpose data=__MyData out=__ExpandVarList ;
      var &VAR ;
    run;
    proc sql noprint ;
      select _name_ into :temp_varnames separated by ' '
      from __ExpandVarList
      ;
      drop table __ExpandVarList ;
      drop table __MyData ;
   quit ;
  )));
&temp_varnames
%mend ExpandVarList;
```

When the macro is invoked in the main session, local macro variables DATA, VAR, RC and TEMP_VARNAMES are created. When the DOSUBL function executes, it creates a side session which executes the code in bold. The side session code references the macro variables DATA and VAR which exist in the main session. This means somehow, code executing in the side session is able to resolve macro variables that exist in the main session symbol tables. The side session code determines the list of variables and writes the list to the macro variable TEMP_VARNAMES, which is later resolved in the main session. This means code executing in the side session can update macro variables in the main session. When the DOSUBL side session code references, updates, or creates macro variables, it is critical to understand the scoping rules which determine how references resolve in the side session, and how the main session symbol tables are updated.

OVERVIEW OF TEST CASES

The following test cases investigate DOSUBL handling of macro variable scope. Each code block is designed to be run in a "clean" SAS session, with no preexisting user-created macro variables. The test cases were run on SAS 9.4M4 (Windows) and 9.4M6 (Linux).

The examples call DOSUBL in a DATA step:

```
data _null_ ;
  rc=dosubl('/*side session code*/') ;
run ;
```

rather than calling DOSUBL via the %SYSFUNC macro function:

```
%let rc= %sysfunc(dosubl(%nrstr(/*side session code*/)));
```

The choice of how DOSUBL is called does not impact how the side session code is executed, or the associated scoping rules. Note that when DOSUBL is called from a DATA step, the side session code is nested in single quotes. This prevents any macro triggers from being resolved prematurely, while the DATA step is compiling. When DOSUBL is called by %SYSFUNC, the side session code is nested in the %NRSTR macro quoting function for the same reason.

SIDE SESSION RESOLUTION OF MAIN SESSION MACRO VARIABLES

Macro variables that exist in the main session symbol tables (global symbol table and any local symbol tables that exist) will resolve in the DOSUBL side session code. Assuming there were no macro variables created in the side session code, the resolution follows the usual macro variable scoping rules. When the macro processor resolves a macro variable reference it starts by looking in the inner most local symbol table that exists, and then looks consecutively in outer scopes until the symbol is found.

The following code creates a global macro variable in the main session global symbol table, and then in the side session code uses a %PUT statement to resolve the global macro variable, and a %PUT _USER_ statement to show the user-created macro variables that the side session can see:

```
%let x=1 ;

data _null_ ;
  rc = dosubl('
    %put --Side Session-- ;
    %put &=x ;
    %put _user_ ;
  ') ;
run ;
```

The log shows that X is resolved appropriately, and that %PUT _USER_ shows the content of main session global symbol table:

```
--Side Session--
X=1
GLOBAL X 1        ← stored in main session global symbol table
```

The following code creates a global macro variable X in the main session global symbol table, and then executes a macro which creates a local variable X, and calls DOSUBL to execute the same %PUT statements in the side session:

```
%let x=global ;

%macro try() ;
  %local x ;
  %let x=local ;

  data _null_ ;
    rc = dosubl('
      %put --Side Session-- ;
      %put &=x ;
      %put _user_ ;
    ') ;
  run ;

%mend ;

%try()
```

If code executing in the side session follows the familiar macro resolution rules, the reference to macro variable X should resolve to the value of the local macro variable X defined in the macro TRY, and the %PUT _USER_ statement should see both the local macro variable X and the global macro variable X. Happily, this is the case:

```
--Side Session--
X=local
TRY X local           ← stored in main session local symbol table
GLOBAL X global       ← stored in main session global symbol table
```

Inferred Rules

When there are no macro variables created in the side session, macro references are resolved following the same familiar rules as the main session. The picture becomes more complex when the side session code updates existing macro variables or creates new macro variables, which will be discussed below.

SIDE SESSION CREATION OF MACRO VARIABLES

DOSUBL CALLED IN OPEN CODE

When DOSUBL is called in open code (i.e. outside of a macro), the side session code can create new main session global macro variables, and can also update existing main session global macro variables.

The following code creates a global macro variable in the side session global symbol table. When DOSUBL completes, it returns the global macro variable to the main session global symbol table.

```
data _null_ ;
  rc = dosubl('
    %put --Side Session--  ;
    %let x=1 ;
    %put &=x ;
    %put _user_ ;
  ') ;
run ;

%put --Main Session--  ;
%put &=x ;
%put _user_ ;
```

The log shows that the global macro variable X created in the side session exists in the main session after DOSUBL() completes:

```
1    data _null_ ;
2      rc = dosubl('
3        %put --Side Session--  ;
4        %let x=1 ;
5        %put &=x ;
6        %put _user_ ;
7      ') ;
8    run ;

--Side Session--
X=1
GLOBAL X 1       ← stored in side session global symbol table

9
10   %put --Main Session--  ;
--Main Session--
11   %put &=x ;
X=1
12   %put _user_ ;
GLOBAL X 1       ← stored in main session global symbol table
```

The log above shows that the macro variable X created in the side session exists in the main session after DOSUBL has completed. It does not, however, provide evidence of my claim that there is a side session global symbol table, separate from the main session global symbol table. I came to that conclusion when I noticed an oddity in the following results.

The following code creates two global macro variables A and B in the main session, and then the side session creates a global macro variable C:

```
%let B=B_Main ;
%let A=A_Main ;

data _null_ ;
  rc = dosubl('
    %put --Side Session--  ;
    %let C=C_Side ;
    %put _user_ ;
  ') ;
run ;

%put --Main Session--  ;
%put _user_ ;
```

The log shows that the macro variable C is created in the side session global table. Within the side session, the %PUT _USER_ statement shows all three macro variables are

available. When DOSUBL completes execution, the macro variable C is returned to the main session global symbol table. The main session %PUT _USER_ also shows all three variables are available:

```
1    %let B=B_Main ;
2    %let A=A_Main ;
3
4    data _null_ ;
5      rc = dosubl('
6        %put --Side Session--  ;
7        %let C=C_Side ;
8        %put _user_ ;
9      ') ;
10   run ;

--Side Session--
GLOBAL C C_Side       ← stored in side session global symbol table
GLOBAL A A_Main       ← stored in main session global symbol table
GLOBAL B B_Main       ← stored in main session global symbol table

11
12   %put --Main Session--  ;
--Main Session--
13   %put _user_ ;
GLOBAL A A_Main       ← stored in main session global symbol table
GLOBAL B B_Main       ← stored in main session global symbol table
GLOBAL C C_Side       ← stored in main session global symbol table
```

What evidence is there that the macro variable C was created in a side session global table which is separate from the main session symbol table? Note that in the log from the side session %PUT _USER_ statement, the three global macro variables are NOT listed in alphabetical order. The %PUT _USER_ statement orders macro variables by scope (starting with the inner most scope that exists), and within each scope orders macro variables alphabetically. In the side session, the macro variable C is printed first because it is stored in the side session global symbol table. The macro variables A and B are printed after C because they are stored in a different scope, the main session global symbol table. After DOSUBL completes, it returns macro variable C from the side session global symbol table to the main session symbol table. When the main session %PUT _USER_ statement executes, all three macro variables are stored in the same scope, and it orders them alphabetically.

This begs the question of what would happen if the side session assigned a value to macro variable B when a main session macro variable B already exists. Would it be written to the side session global symbol table or would it directly update the main session global symbol table? It turns out that when the side session %LET executes, it writes directly to the main session global symbol table:

```
1    %let B=B_Main ;
2    %let A=A_Main ;
3
4    data _null_ ;
5      rc = dosubl('
6        %put --Side Session--  ;
7        %let B=B_Side ;
8        %let C=C_Side ;
9        %put _user_ ;
10     ') ;
11   run ;
```

```
--Side Session--
GLOBAL C C_Side        ← stored in side session global symbol table
GLOBAL A A_Main        ← stored in main session global symbol table
GLOBAL B B_Side        ← stored in main session global symbol table

12
13   %put --Main Session--  ;
--Main Session--
14   %put _user_ ;
GLOBAL A A_Main        ← stored in main session global symbol table
GLOBAL B B_Side        ← stored in main session global symbol table
GLOBAL C C_Side        ← stored in main session global symbol table
```

Looking again at the order of macro variables listed in the side session %PUT _USER_ statement, the macro variable B is still listed after the macro variable C. This tells us that B was written to the main session global symbol table. If B had been written to the side session global table, we would have seen B listed before C. In fact, we would have seen B listed twice, because it would have shown the macro variable B in the side session global table and the separate macro variable B in main session global symbol table.

Automatic Macro Variables

While this section has showed that user-created global macro variables created in the side session are returned to the main session, it is important to know that automatic macro variables created in the side session are NOT returned to the main session. For example, consider the automatic macro variable SYSCC, which stores a job condition code (0=success). The following code uses DOSUBL to execute a DATA step which errors.

```
data _null_ ;
  rc = dosubl('
    data foo ;
      Invalid Code ;
    run ;
    %put --Side Session--  ;
    %put &=SYSCC ;
    %put ;
  ') ;

  put "Main Session DOSUBL return code: " rc= ;
run ;

%put --Main Session--  ;
%put &=SYSCC ;
```

The log shows an error message, and the value of SYSCC is appropriately set to non-zero in the side session. But the value of the side session automatic variable SYSCC is not returned to the main session automatic variable SYSCC. After DOSUBL completes, the main session SYSCC remains as zero. Note also that the value returned by DOSUBL (stored in the data step variable RC) is also zero. DOSUBL returns zero when it is able to execute code, regardless of whether or not the code executes successfully:

```
ERROR 180-322: Statement is not valid or it is used out of proper order.
NOTE: The SAS System stopped processing this step because of errors.
WARNING: The data set WORK.FOO may be incomplete.  When this step was
stopped there were 0 observations and 0 variables.

--Side Session--
SYSCC=3000

Main Session DOSUBL return code: rc=0
```

```
13
14   %put --Main Session--  ;
--Main Session--
15   %put &=SYSCC ;
SYSCC=0
```

How can we capture a return code from DOSUBL? The above shows that the automatic macro variable SYSCC is not returned from the side session to the main session. We have already seen that user-created global macro variables in the side session are returned to the main session. Thus, one way to capture a return code from DOSUBL is to create a global macro variable in the side session, storing the value of SYSCC:

```
data _null_ ;
  rc = dosubl('
    data foo ;
      Invalid Code ;
    run ;
    %let MySYSCC=&SYSCC ;
    %put --Side Session--  ;
    %put &=MySYSCC ;
    %put ;
  ') ;

  put "Main Session DOSUBL return code: " rc= ;
run ;

%put --Main Session--  ;
%put &=MySYSCC ;
```

```
ERROR 180-322: Statement is not valid or it is used out of proper order.
NOTE: The SAS System stopped processing this step because of errors.
WARNING: The data set WORK.FOO may be incomplete.  When this step was
stopped there were 0 observations and 0 variables.

--Side Session--
MYSYSCC=3000

Main Session DOSUBL return code: rc=0

14
15   %put --Main Session--  ;
--Main Session--
16   &=MySYSCC ;
MYSYSCC=3000
```

Inferred Rules for DOSUBL Called In Open Code

When DOSUBL is called in open code and the side session assigns a value to a user-created global macro variable, the following rules are applied:

1. If the main session global symbol table has a macro variable with the same name, that macro variable is updated.
2. Otherwise, the macro variable is created in the side session global symbol table, and is returned to the main session symbol table when DOSUBL completes.

DOSUBL CALLED INSIDE A MACRO

When DOSUBL is called inside a macro, the scoping rules are more complex, because the side session code may return macro variables to the main session global symbol table or a macro's local symbol table. The DOSUBL practitioner must understand the rules that dictate how macro variables are returned.

In the following code DOSUBL is called inside a macro. When DOSUBL is called, the main session has macro variables A and B in the main session global symbol table, and macro variable B in the local symbol table of the macro TRY. The DOSUBL side session code has three %LET statements that assign values to A, B, and C.

```
%let A=A_Main ;
%let B=B_Main ;

%macro try() ;
  %local B ;
  data _null_ ;
    rc = dosubl('
      %put --Side Session--  ;
      %let A=A_Side ;
      %let B=B_Side ;
      %let C=C_Side ;
      %put _user_ ;
      %put ;
      %put In side session %nrstr(&)A resolves to: &A ;
      %put In side session %nrstr(&)B resolves to: &B ;
    ') ;
  run ;

  %put --Main Session--  ;
  %put _user_ ;
%mend try ;

%try()
```

The log shows:

```
21   %try()

--Side Session--
GLOBAL A A_Side       ← stored in side session global symbol table
GLOBAL B B_Side       ← stored in side session global symbol table
GLOBAL C C_Side       ← stored in side session global symbol table
TRY B                 ← stored in main session local symbol table
GLOBAL A A_Main       ← stored in main session global symbol table
GLOBAL B B_Main       ← stored in main session global symbol table

In side session &A resolves to: A_Side
In side session &B resolves to: B_Side

--Main Session--
TRY B B_Side          ← stored in main session local symbol table
GLOBAL A A_Side       ← stored in main session global symbol table
GLOBAL B B_Main       ← stored in main session global symbol table
GLOBAL C C_Side       ← stored in main session global symbol table
```

Note then when the %LET statements execute in the side session, they create three new global macro variables in the side session global symbol table. Even though A existed in the

main session global symbol table, and B existed in both the main session global symbol table and the local symbol table of TRY, the %LET statement does NOT update them. The side session %PUT _USER_ shows A, B, and C created in the side session global symbol table, then B in the local symbol table of TRY, then A and B in the main session global symbol table. References to A or B in the side session resolve to the value stored in the side session global symbol table.

When DOSUBL completes it returns the three global macro variables created in the side session to the main session. When returning a macro variable, *DOSUBL looks at the existing main session symbol tables to determine which scope to return it to*. If the macro variable exists in a local symbol table, it is returned to the inner most local symbol table. If the macro variable does not exist in a local symbol table, it is returned to the global symbol table. The log shows that when A is returned from the side session it updates the existing macro variable A in the main session global symbol table. When B is returned it updates the existing macro variable B in the local symbol table of TRY. When C is returned it creates a new macro variable C in the global symbol table, because it did not exist in any scope previously.

I had hoped that the rule might be that DOSUBL would return a macro variable to the inner most scope where it exists, but that is *not* the rule. If the macro variable exists in *any* local scope, it is returned to the inner most local scope. The below code has a macro OUTER which creates a local macro variable A, and then invokes a macro INNER. INNER calls DOSUBL which creates a global macro variable A in the side session:

```
%macro outer() ;
  %local A ;
  %let A=A_Main ;
  %inner()
%mend outer ;

%macro inner() ;
  data _null_ ;
    rc = dosubl('
      %put --Side Session--  ;
      %let A=A_Side ;
      %put _user_ ;
    ') ;
  run ;
  %put --Main Session--  ;
  %put _user_ ;
%mend inner ;

%outer()
```

I had hoped that when DOSUBL completed, it would return A to the local symbol table of OUTER, because that is the first symbol table where A exists. In fact, the log shows that it returned A to the local symbol table of INNER:

```
19   %outer()

--Side Session--
GLOBAL A A_Side       ← stored in side session global symbol table
OUTER A A_Main        ← stored in main session local symbol table

--Main Session--
INNER A A_Side        ← stored in main session local symbol table
OUTER A A_Main        ← stored in main session local symbol table
```

As shown above, when DOSUBL is called in a macro, the side session open %LET statement will always create a global macro variable in the side session global symbol table. Surprisingly, when CALL SYMPUTX is called with the "G" parameter, and there is a pre-existing macro variable with the same name in the main session in any scope, CALL SYMPUTX will write directly to the main session global symbol table. The below code invokes a macro TRY with two local macro variables A and B. The DOSUBL side session code uses %LET to create a macro variable A, and CALL SYMPUTX to create a macro variable B, in hopes of having both macro variables returned to the local symbol table of TRY:

```
%macro try() ;
  %local A B ;
  data _null_ ;
    rc = dosubl('
      %put --Side Session--  ;
      %let A=A_Side ;
      data _null_ ;
        call symputx("B","B_Side","G") ;
      run ;
      %put _user_ ;
    ') ;
  run ;

  %put --Main Session--  ;
  %put _user_ ;
%mend try ;

%try()
```

The log shows that the %LET statement writes A to the side session global symbol table, and then DOSUBL returns A to the local scope of TRY, as expected. CALL SYMPUTX, however, writes B directly to the main session global symbol table. After DOSUBL completes, the local macro variable B is still null:

```
18   %try()

--Side Session--
GLOBAL A A_Side        ← stored in side session global symbol table
TRY A                  ← stored in main session local symbol table
TRY B                  ← stored in main session local symbol table
GLOBAL B B_Side        ← stored in main session global symbol table

--Main Session--
TRY A A_Side           ← stored in main session local symbol table
TRY B                  ← stored in main session local symbol table
GLOBAL B B_Side        ← stored in main session global symbol table
```

I find this behavior surprising. Happily, if the call to the SYMPUTX routine does not specify the third parameter as "G", the macro variable B will be written to the side session global symbol table, and returned to the local symbo table of TRY.

Inferred Rules for DOSUBL Called Inside A Macro

When DOSUBL is called inside of a macro and the side session assigns a value to a user-created global macro variable, the following rules are applied:

1. The macro variable is created in the side session global symbol table.[5]
2. When returning the side session macro variable to the main session, DOSUBL looks at the existing main session symbol tables to determine which scope to return it to.
    a. If the macro variable exists in any local symbol table, it is returned to the inner most local symbol table.
    b. If the macro variable does not exist in a local symbol table, it is returned to the global symbol table.

DOSUBL INVOKING MACROS

DOSUBL can execute code which invokes a macro. The macro may create local macro variables, and it may assign values to global macro variables. When a macro assigns a value to a global macro variable, the "Inferred Rules" defined in the two previous sections are applied. Only global macro variables are returned to the main session when DOSUBL completes. Local macro variables created in the side session are not returned to the main session, because the local macro variables are deleted when the macro completes execution, before DOSUBL completes.

The below code creates global macro variables A and B in the main session, and then calls DOSUBL to invoke the macro MakeVars in the side session. MakeVars assigns values to global macro variable A, local macro variable B, and global macro variable C:

```
%let A=A_Main ;
%let B=B_Main ;

%macro MakeVars() ;
  %global A C ;
  %local B ;
  %let A=A_side ;
  %let B=B_side ;
  %let C=C_side ;
  %put _user_ ;
%mend ;

data _null_ ;
  rc = dosubl('
    %put --Side Session--  ;
    %MakeVars()
  ') ;
run ;
%put --Main Session--  ;
%put _user_ ;
```

The log shows that when the macro MakeVars executes in the side session, it updates the existing main session global macro variable A, and creates a new side session global macro variable C. It also creates a side session local macro variable B. After DOSUBL completes execution, it returns the side session global macro variable C to the main session global symbol table. That is consistent with the previously inferred rules for how DOSUBL handles macro variable scope when it is called in open code:

[5] The example of CALL SYMPUTX with the "G" argument and a preexisting macro variable in the main session is an exception to this rule.

```
--Side Session--
MAKEVARS B B_side       ← stored in side session local symbol table
GLOBAL C C_side         ← stored in side session global symbol table
GLOBAL A A_side         ← stored in main session global symbol table
GLOBAL B B_Main         ← stored in main session global symbol table

19   %put --Main Session--  ;
--Main Session--
20   %put _user_ ;
GLOBAL A A_side         ← stored in main session global symbol table
GLOBAL B B_Main         ← stored in main session global symbol table
GLOBAL C C_side         ← stored in main session global symbol table
```

The above also demonstrates an important point: macros compiled in the main session can be invoked in the side session (but the converse is not true).

The prior example showed that a macro invoked in the side session can create a global macro variable and return it to the main session global symbol table. It should also be possible to have a macro invoked in the side session create a global macro variable and return it to a main session local symbol table. Below is my first attempt, but it fails. It invokes a macro TRY which has a local macro variable A, and calls DOSUBL which invokes the macro MakeVars in hopes of making a side session global macro variable A, which will be returned to the local symbol table of TRY:

```
%let A=A_Main ;

%macro try() ;
  %local A ;
  data _null_ ;
    rc = dosubl('
      %put --Side Session--  ;
      %MakeVars()
    ') ;
  run ;
  %put --Main Session--  ;
  %put _user_ ;
%mend try ;

%macro MakeVars() ;
  %global A ;
  %local B ;
  %let A=A_side ;
  %let B=B_side ;
  %put _user_ ;
%mend ;

%try()
```

Unfortunately, the code errors. The log shows:

```
ERROR: Attempt to %GLOBAL a name (A) which exists in a local environment.
```

When the %GLOBAL A; statement executed in the side session, it errored because when the %GLOBAL statement executes, it checks to see if there are any local macro variables with the same name. Unfortunately, even though this %GLOBAL statement is executed in the side session, it apparenty still checks the main session local symbol tables, and throws an error when it finds the macro variable A in the symbol table of TRY.

In order for DOSUBL to return the macro variable A to the local symbol table of TRY, it needs to create a global macro variable A in the side session symbol table. But MakeVars

cannot use the %GLOBAL statement to create the side session global macro variable, because it errors. One work-around for creating the global macro variable A in the side session is to use an open %LET statement in the side session before calling MakeVars. The %LET statement will create the global macro variable A in the side session, and then MakeVars can update its value:

```
%let A=A_Main ;

%macro try() ;
  %local A ;
  data _null_ ;
    rc = dosubl('
      %put --Side Session--  ;
      %let A= ; %*Create global macro var A in side session ;
      %MakeVars2()
    ') ;
  run ;
  %put --Main Session--  ;
  %put _user_ ;
%mend try ;

%macro MakeVars2() ;
  %* Do not declare scope of A ;
  %local B ;
  %let A=A_side ;
  %let B=B_side ;
  %put _user_ ;
%mend ;

%try()
```

Note that the macro MakeVars2 does not declare the macro variable A as global (or local). When the statement `%let A=A_side ;` executes it sees that A exists in the side session global symbol table and updates the value there. When DOSUBL completes, it returns the side session global macro variable A to the local symbol table of TRY. That is consistent with the previously inferred rules for how DOSUBL handles macro variable scope when it is called inside a macro:

```
24   %try()

--Side Session--
MAKEVARS2 B B_side      ← stored in side session local symbol table
GLOBAL A A_side         ← stored in side session global symbol table
TRY A                   ← stored in main session local symbol table
GLOBAL A A_Main         ← stored in main session global symbol table

--Main Session--
TRY A A_side            ← stored in main session local symbol table
GLOBAL A A_Main         ← stored in main session global symbol table
```

Note that this is the first time that the side session %PUT _USER_ shows macro variables from all four different scopes: The macro variable B is stored in the side session local symbol table of MAKEVARS2. The macro variable A with the value A_Side is stored in the side session global symbol table. The macro variable A with the value null is stored in the main session local symbol table of TRY. The macro variable A with the value A_Main is stored in the main session global symbol table.

Inferred Rules

When DOSUBL invokes a macro, the macro executing in the side session can create local macro variables and it can assign values to global macro variables. When the macro assigns a value to a global macro variable, it may update an existing main session global macro variable, or create a new side session global macro variable, according to the previously identified inferred rules. Only side-session global macro variables are returned to the main session, using the previously identified inferred rules to determine the main session macro variable scope. If the main session has a local macro variable defined and DOSUBL executes a %GLOBAL statement which refers to a macro variable with the same name, the %GLOBAL statement will error.

CONCLUSION

When DOSUBL executes, it creates a "side session" to execute code. Code executed in the side session may directly update the main session global symbol table, or create macro variables in side session symbol tables. When DOSUBL completes execution, any side session global macro variables are returned to the main session. Following is a summary of the rules DOSUBL seems to use for deciding the scope of macro variables.

When code executed by DOSUBL assigns a value to a user-created global macro variable:

1. IF (the macro variable exists in the main session global symbol table) AND (DOSUBL was called in open code OR the macro variable was assigned a value by CALL SYMPUTX specifying the G symbol table) THEN update the main session global macro variable.
2. ELSE create a new side session global macro variable.

When code executed by DOSUBL references a macro variable, the macro processor searches the symbol tables in the following order:

1. Side session local symbol tables (starting with the inner-most local symbol table then moving outward)
2. Side session global symbol table
3. Main session local symbol tables (starting with the inner-most local symbol table then moving outward)
4. Main session global symbol table

When DOSUBL completes execution, any side session user-created global macro variables are returned to the main session:

1. IF a same-named macro variable exists in any main session local symbol table THEN create (or update) the macro variable in the most local main session local symbol table that exists.
2. ELSE create (or update) the macro variable in the main session global symbol table.

REFERENCES

Langston, Rick. 2013. "Submitting SAS Code On The Side". SAS Global Forum 2013, San Francisco, California. Available at https://support.sas.com/resources/papers/proceedings13/032-2013.pdf.

King, John. 2012. "Atypical Applications of Proc Transpose". PharmaSUG 2012, San Francisco, California. Available at http://www.lexjansen.com/pharmasug/2012/TF/PharmaSUG-2012-TF12.pdf.

King, John. 2017. "DOSUBL and the Function Style Macro". MidWest SAS Users Group 2017, St. Louis, Missouri. Available at https://www.mwsug.org/proceedings/2017/BB/MWSUG-2017-BB142.pdf.

Rhoads, Mike. 2012. "Use the Full Power of SAS in Your Function-Style Macros." SAS Global Forum 2012, Orlando, Florida. Available at https://support.sas.com/resources/papers/proceedings12/004-2012.pdf.

ACKNOWLEDGMENTS

I am grateful for the input of several people kind enough to review a draft of this paper and provide helpful feedback: Roger DeAngelis; Richard DeVenezia; Bartosz Jabłoński; Rick Langston and Russ Tyndall. Countless others have contributed to my understanding of DOSUBL by participating in online conversations on SAS-L (https://listserv.uga.edu/cgi-bin/wa?A0=SAS-L) and communities.sas.com (https://communities.sas.com/). Any remaining errors or misconceptions in this paper are of course my own.

CONTACT INFORMATION

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

Quentin McMullen
Siemens Healthineers
qmcmullen@gmail.com