

A Use of External Files in Dynamic Code Generation

Greg M. Woolridge, Abbott Laboratories, Abbott Park, IL
Henry B. Winsor, Taylor Management Systems, Des Plaines, IL

ABSTRACT

This paper illustrates the technique of having the program write executable code to an external system file, then using a %INCLUDE command to insert the contents of this file back into the program for compilation and execution. The program can use input data to decide which lines of code need to be written to the external file, allowing for dramatic simplification of program code and often enhanced performance. An example that demonstrates the specification of exact formats for printing numeric data is presented.

INTRODUCTION

Every now and then, it would be nice to have your program determine what code is required by the data and create that code for later use in the program. This is possible by using PUT statements in a DATA _NULL_ step to write the code to an external file, then using a %INCLUDE statement to bring this code back into the program at the appropriate place. Writing to an external file allows a program to create, but not immediately execute, legitimate SAS® code. In the data step where the code is being written, the compiler is merely putting characters to an external target. If any executable statements are to be written, just enclose them within quotation marks. The compiler will consider them literal character strings and make no effort to evaluate them. This allows for the creation of elaborate, data-specific macros, without worrying about errors in macro variable resolution or missing parameter values. This technique is more easily debugged than it may first appear. If the external file is examined, it should contain only the statements that perform the desired action. If not, the offending statement is easily identified and corrected.

THE PROBLEM

At Abbott Laboratories, this technique is necessary to implement one particular user request; to print laboratory test data so that the precision of the data (the number of significant digits) is apparent, yet consistent in format. This request necessitates using a W.D rather than a BEST. format. Lab data are reported with different levels of precision by different sites, and displaying that data with the correct precision requires that the formats used for output purposes be the formats in which the data were reported. But, the SAS System does not store native format when data are input, so the precision is not directly recoverable.

Instead, it is possible to examine the data and determine the format. You can safely assume that if a decimal place is a significant digit, a non-zero value will be present in even a small number of observations. At worst, the precision will be understated, which is greatly preferable to overstating it. So, let the data indicate its native format. This requires grouping the data into unique subsets (normally by lab site and test), then checking each subset to find the largest format needed to print each numerical value. These unique subsets should also include the normal range values for each test, since the high or low normal range value is also printed when the value falls outside the normal range.

The calculation of the appropriate W.D value for the format is straightforward, but the insertion of the format into the output code is not. The compiler currently expects a format expression to be literal; i.e., you cannot store the appropriate format in a data step variable and have the compiler evaluate the variable for the format. This means that you cannot change the output format expression from one observation to the next in a

direct fashion.

SOLUTIONS

There are probably many methods for correctly specifying the format and changing it as needed. You may classify these methods by two criteria; whether the program code is fixed or dynamic, and whether the tender efforts of a human programmer are needed to alter the code. The cross-product of these classifications, with utility evaluation, is below:

	data-driven	human input
static	complex and cumbersome	tedious, prone to error
dyna- mic	initially requires more coding	exists only in science fiction

An example of a static, human input solution is to write mutually exclusive blocks of code that explicitly state the output format. After running a utility program that identifies and lists the correct format, the programmer transfers the format to the appropriate lines in a program. This process must be repeated whenever an updated output listing is requested. Also, with 12 lab tests and 5 lab sites, the programmer must verify or alter at least 60 formats in 60 blocks of very similar-appearing, contiguous code. Any method that will require human intervention is certain to cause frequent and innumerable headaches for the responsible programmer.

An example of a static, data-driven solution is to maintain a master list of lab tests and their respective formats. Then, the program can parse through this list; when it finds the correct lab site and test, it can insert the appropriate format into the code at the proper place. However, a lab may change its precision over time. This necessitates an additional parameter for evaluation, such as the effective date of the change. There may be other factors that require differentiation, and recall that

adding additional factors will have a multiplicative rather than an additive effect on the number of entries in this master list. If your company uses 50 lab tests and 50 different lab sites, the master list must contain 2500 entries for completeness. If the output is from a study with only 1 lab site and 5 lab tests, it is easy to see that a program will waste significant resources in perusing and inserting formats from a list of 2500. It should be apparent that any static solution is not going to be as effective as a dynamic solution.

An example of a dynamic, data-driven solution is to have the program take the numeric lab data, and evaluate this data for the correct format. Then, it constructs a macro in the form of a function, so that it returns the appropriate format value when called with the lab test and site as macro parameters. Later, when supplied with the list of tests to be printed, it builds in an external file the necessary code to print the formatted output, including a call to the above created macro in each place that a format is needed. Finally, the program inserts the code (via %INCLUDE) into the DATA _NULL_ step that actually prints the output.

As implemented at Abbott Laboratories, this solution depends on the programmer to place two macro calls into the program, decide on the column placement for each lab test, and put a %INCLUDE statement in the DATA _NULL_ step. Everything else is determined by the program itself, each time the program is executed. The hidden portion of this process is done by three macros, one of which is created every time the program is executed.

MACROS

The first macro of the three is called TSTFRMAT. This macro actually does the data evaluation and format calculation, then builds the second macro, called FRMATS, in an external file. It immediately places the macro FRMATS into the program with a %INCLUDE statement, then deletes the external file.

The FRMATS macro behaves like a function; it returns the appropriate format when called with

lab test and site as parameters. Since this macro is created each time the program is executed, the formats reflect the correct precision without any programmer action. A sample copy of the macro is below, in exhibit A.

EXHIBIT A

```
%macro frmats(labtest,labsite);
  %if &labtest=TESTA %then %do;
    %if &labsite= 10 %then 2.0 ;
    %else 10.3;
  %end;
%mend frmats;
```

The code presented in exhibit B is the third macro, TSTWRITE. This macro writes the PUT statements to the external file that actually print the formatted values. Because the methods used in TSTFRMAT and TSTWRITE are so similar, only TSTWRITE was chosen for complete exposition.

The macro has five parameters: COL, the column where the test observation is to be printed, LAB, the variable containing the actual lab value, FL, the flag that indicates if the lab value is out of the normal range (H=High, L=Low), LO, the variable containing the low normal range value, and HI, the variable containing the high normal range value. For each lab site, this macro creates a DO loop of IF statements that will execute the formatted printing when inserted into the DATA _NULL_ step.

Within the DATA _NULL_ step, the literal character strings containing lower case code are the executable statements that are written to the external file. Care must be taken to match the quotation marks properly since there is constant switching between writing text and variable values. Also, both types of quotation marks serve a purpose here. The double quotation marks are used everywhere except around the literal containing the calls to FRMATS. If these calls were contained within double quotation marks, the compiler will evaluate this macro call immediately. Since the call is lacking parameters at this time, the macro will return a blank instead of a usable format. These calls are the only items requiring enclosure within single quotation marks. In every other instance, the double quotation marks are

used to allow the resolution of the parameters within the confines of the macro. Otherwise, these macro variables will resolve at the same time as the FRMATS calls (when the code is inserted in the DATA _NULL_ step by the %INCLUDE statement), at which time these macro variables may well have different values, if any.

EXHIBIT B

```
%MACRO TSTWRITE(COL=,LAB=,FL=,LO=,HI=);
DATA LABSRC; SET _LAST_(KEEP=SITE);
  IF SITE>.;
RUN;

PROC SORT NODUPS;
  BY SITE;
RUN;

FILENAME TSTWRTER DISK "TSTWRTER SAS A";

DATA _NULL_; SET LABSRC END=EOF;
  FILE TSTWRTER NOTITLES;
  IF _N_=1 THEN
    PUT @1 "if SITE=" SITE 2. " then do;";
  ELSE PUT @1 "else if SITE=" SITE 2.
    " then do;";
  PUT @3 "if &LAB >. then do;";
  PUT @5 "put @&COL &LAB " '%frmats('
    "&LAB," SITE 2. " ) @;";
  PUT @5 "if &FL in ('L','H') "
    "then put &FL $1. @;";
  PUT @5 "if &FL='L' " /
    @5 "then put '(' &LO "
    '%frmats(' "&LAB," SITE 2.
    ") ' ' @;"; /
    @5 "else if &FL='H' " /
    @5 "then put '(' &HI "
    '%frmats(' "&LAB," SITE 2.
    ") ' ' @;";
  PUT @3 "end;" /
    @1 "end;";
RUN;
%MEND TSTWRITE;
```

If you look closely, you will see that every PUT statement written to the external file contains a trailing @ sign. This allows the programmer the flexibility of printing other data on the same line; this includes the possibility of executing TSTWRITE several times, renaming the external files to different names, and putting several %INCLUDE statements in the same DATA _NULL_ step. In any case, you must be certain the line is released after the observation is printed, if necessary by using a null PUT statement.

EXAMPLE

These macros have been executed in a program with hypothetical data. Here are the results of two passes of the program, the first pass using 4 data records and the second pass adding 2 records.

PASS ONE

DATA

SITE	TESTA	TESTAHI	TESTALO	TESTAFL
10	14.000	12.000	6.000	H
10	4.000	12.000	6.000	L
12	14.000	12.000	6.000	H
12	5.100	12.000	6.000	L

Note that the data come from 2 different lab sites (10,12) and are reported with different levels of precision. The data from lab site 10 are reported with no significant digits beyond the decimal point while the data from lab site 12 are reported with 1 significant digit (zeros are not significant digits unless followed by a non-zero digit).

TSTWRITE OUTPUT

```
if SITE=10 then do;
  if TESTA >. then do;
    put @20 TESTA %formats(TESTA,10) @;
    if TESTAFL in ('L','H')
      then put TESTAFL $1. @;
    if TESTAFL='L' then put '('
      TESTALO %formats(TESTA,10) ')' @;
    else if TESTAFL='H' then put '('
      TESTAHI %formats(TESTA,10) ')' @;
  end;
end;
else if SITE=12 then do;
  if TESTA >. then do;
    put @20 TESTA %formats(TESTA,12) @;
    if TESTAFL in ('L','H')
      then put TESTAFL $1. @;
    if TESTAFL='L' then put '('
      TESTALO %formats(TESTA,12) ')' @;
    else if TESTAFL='H' then put '('
      TESTAHI %formats(TESTA,12) ')' @;
  end;
end;
```

If you examine the above code, you can see that the generated code is merely a series of PUT statements grouped within mutually exclusive DO

loops. These conditional PUT statements first verify that there is data in the observation to print. If not, this code does nothing. Otherwise, it prints the test value(TESTA), and if the value of the flag variable (TESTFL) is either L or H, prints the flag. Finally, if the flag has been printed, it will print either the corresponding low (TESTLO) or high normal value (TESTHI). None of these statements are hard-coded into the program, but are all produced by the macro TSTWRITE. The FRMATS macro calls are the only unusual code present, appearing where you would normally expect to see a numeric format.

After inclusion and resolution, the compiler evaluates:

```
if SITE=10 then do;
  if TESTA >. then do;
    put @20 TESTA 2.0 @;
    if TESTAFL in ('L','H')
      then put TESTAFL $1. @;
    if TESTAFL='L' then put '('
      TESTALO 2.0 ')' @;
    else if TESTAFL='H' then put '('
      TESTAHI 2.0 ')' @;
  end;
end;
else if SITE=12 then do;
  if TESTA >. then do;
    put @20 TESTA 4.1 @;
    if TESTAFL in ('L','H')
      then put TESTAFL $1. @;
    if TESTAFL='L' then put '('
      TESTALO 4.1 ')' @;
    else if TESTAFL='H' then put '('
      TESTAHI 4.1 ')' @;
  end;
end;
```

You can see that the appropriate formats have been substituted for the macro calls, the rest of the code is identical to the pre-inclusion code.

The output from the first run is below:

Lab Site	Lab Value
10	14H(12)
10	4L(6)
12	14.0H(12.0)
12	5.1L(6.0)

You will note that the lab values from the two sites are printed with different formats, yet consistent within the lab site. The different formats in the same column will indicate to the reader that the

printed values are not necessarily comparable.

PASS TWO

DATA

SITE	TESTA	TESTAHI	TESTALO	TESTAFL
10	14.000	12.000	6.000	H
10	4.000	12.000	6.000	L
10	14.000	12.000	6.000	H
10	5.100	12.000	6.000	L
10	214.100	12.000	6.000	H
10	6.154	12.000	6.000	-

All the lab sites are now the same. The original entry of lab site=12 was apparently an error, corrected between pass one and two. Two additional records have also been added, both containing format-altering extreme values.

TSTWRITE OUTPUT

```
if SITE=10 then do;
  if TESTA >. then do;
    put @20 TESTA %formats(TESTA,10) @;
    if TESTAFL in ('L','H')
      then put TESTAFL $1. @;
    if TESTAFL='L' then put '('
      TESTALO %formats(TESTA,10) ')' @;
    else if TESTAFL='H' then put '('
      TESTAHI %formats(TESTA,10) ')' @;
  end;
end;
```

This block of output is much shorter than the previous one, because the macro did not need to create code for lab site 12. Again, since none of this was hard-coded into the program, this effective elimination of extraneous code takes place automatically with no intervention needed.

After inclusion and resolution, the compiler evaluates:

```
if SITE=10 then do;
  if TESTA >. then do;
    put @20 TESTA 7.3 @;
    if TESTAFL in ('L','H')
      then put TESTAFL $1. @;
    if TESTAFL='L' then put '('
      TESTALO 7.3 ')' @;
    else if TESTAFL='H' then put '('
      TESTAHI 7.3 ')' @;
  end;
end;
```

Note that the formats have changed from 2.0 to

7.3 for lab site 10. Again, this change happens automatically without programmer intervention.

Here is the output from the second pass:

Lab Site	Lab Test
10	14.000H(12.000)
10	4.000L(6.000)
10	14.000H(12.000)
10	5.100L(6.000)
10	214.100H(12.000)
10	6.154

In pass two, the data are all aligned, which further reinforces that they are comparable values.

CONCLUSION

This paper demonstrates one of the possibilities of using external files to contain data-determined blocks of code, and later bring the contents into the program for execution. The macro TSTWRITE is actually a very simplified version of the macro code currently in use at Abbott Laboratories. Among the additional capabilities are: the printing of multiple variables with one macro call, overwrite checking, and conditionally printing data. However, without this technique, these macros would not exist, and the lives of some of your fellow programmers would be much busier, and infinitely more tedious. You are urged to consider using this technique whenever you have need to modify programs in a way that can be determined by the data.

ACKNOWLEDGEMENTS

The authors would like to give special thanks to Jim Pecho for originating the concepts that underlie this paper. We also wish to thank Robert (Tad) Braden, Jim Pecho, Helen Bruce Winsor, and Barbara J. Worth for their kind comments and help in preparing this paper. However, the authors take full responsibility for all omissions and errors contained herein.

SAS is a registered trademark or trademark of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.