SIMPLICITY THROUGH OBSCURITY: SOME TIPS TO SIMPLIFY YOUR PROGRAMMING LIFE

Frank DiIorio
CodeCrafters, Inc.
Chapel Hill NC

We are always looking for ways to simplify programs. Macro utility libraries, formats, templates, and the like are all good ways to accomplish this. Another approach to code reduction is harvesting the randomly acquired and randomly filed syntax minutia that most of us acquire over the years. With this in mind, this article addresses several typical and recurring needs: being able to identify the name of the currently executing program, and having a way to easily turn groups of statements on and off. The solutions utilize arcane items such as the EXTFILES dictionary table, the reserved FILEREF named #LN00006, and the RUN statement's CANCEL option. It also reminds us that the macro language can insert even the smallest piece of code to SAS for execution, and can do so *within* a program statement.

IDENTIFYING THE CURRENT PROGRAM

MOTIVATION

How familiar is this scenario? You're handed a page of output (even worse, one that's yellow with age) and are asked to rerun the program that produced it. The page, of course, has no clue in the title or footnote that reveals the name of the program that created it. A little sleuthing eventually identifies the correct program, but wouldn't it be nice if, say, the footnote had a reference? Wouldn't it be even better if the name of the program could be generated automatically? That way, you wouldn't have to change the hard-coded location if the program moved or was renamed.

In the VMS environment there is an automatic macro variable, VMSSASIN, that satisfies this need. Most, if not all, other systems have this need left unfulfilled. Fortunately, some straightforward SQL code accessing an obscure and undocumented FILEREF can accomplish this task in Windows environments.

SOURCE

The source for the CurrFile macro is shown in **Exhibit 1**. The macro is straightforward. It creates global macro variable `currfile`, then populates it from the EXTFILES dictionary table. Refer to "Source Comments," below, for an explanation of the highlighted items (those identified by white print in a black circle).

Exhibit 1: CurrFile Source

```
/*
          Name:  CurrFile
   Description:  Place the name of the currently executing program in a macro variable
         Input:  No parameters
        Output:  Global macro variable CURRFILE
   Usage Notes:  Works reliably in batch mode.  Interactive usage is "iffy" and      ❶
                 not completely predictable.
                 Uses dictionary table EXTFILES
   History:   Date        Init  Comments
              2004/05/30  FCD   Initial release to AutoCall library
*/
%macro CurrFile;
       %global CurrFile;
       proc sql noprint;
            select XPath
            into :CurrFile separated by ' '   ❷
            from dictionary.extfiles          ❸
            where fileref like "#LN00006"     ❹
            ;
       quit;
%mend;
```

SOURCE COMMENTS

❶ Although it isn't part of the immediate problem of interest, the program header is worth a look. One of the keys to effective use of generalized macros such as this is documentation. The comment block shown here is just one of *many* ways to implement this. It describes the input and output of the macro, along with a usage caveat.

❷ We use the macro language interface to write to `currfile`, taking care to preserve individual path names (the `separated by` clause) in the unlikely event of the program source being a concatenation of files.

❸ The path value is found in the SAS metadata table (reserved LIBNAME of `DICTIONARY`) named `EXTFILES`. The nature and contents of the dictionary tables is both important and beyond the scope of this article. Go to **www.CodeCraftersInc.com** for papers describing these useful resources. For now, just take it on faith that a row in `EXTFILES` is a FILEREF, and that `XPATH` contains the complete path name referenced by the `FILEREF`.

❹ "LN what?" This, almost literally, is the key. We want to capture the name of the file associated with the current program. Visual examination of the `EXTFILES` table – and *not* examination of any published documentation! – reveals that in Windows systems (and possibly others) the current file is always allocated with the reserved FILEREF of `#LN00006`. Armed with this arcane bit of knowledge, the rest is easy.

USING THE MACRO

A simple use of the macro is shown in **Exhibit 2**. We call the macro from a file named `UseCurrfile.sas`, in directory `C:\Publications\Articles\Newsletter Fall 2004`. Once the macro executes (as evidenced by the NOTEs about SQL execution

times), the %PUT statement displays macro variable CURRFILE. The variable could have also been used in TITLE or FOOTNOTE statements, thus scratching the "itch" described in the introduction to this section.

Exhibit 2: CurrFile Log

```
32         %currfile
NOTE: PROCEDURE SQL used (Total process time):
      real time           0.01 seconds
      cpu time            0.01 seconds

33         %put Global macro variable CURRFILE [&currfile.];
Global macro variable CURRFILE [C:\ Publications\Articles\Newsletter Fall 2004\UseCurrFile.sas]
```

EXTENSIONS

At least two modifications to CURRFILE come to mind:

1. Initialize the output, global macro variable to <not allocated> or something similar, so if we do not locate #LN00006 CURRFILE will not be simply null. This is more informative.
2. Create a parameter that would allow the user to name the output variable containing the path name. This is more flexible than the current coding, which hard-wires the name (CURRFILE).

CONTROLLING DEBUGGING OUTPUT

MOTIVATION

The next example is also motivated by a recurring need. Consider this scenario: while developing a somewhat complex program, you inserted numerious PUT statements in a DATA step and an assortment of PROCs to help shed light on when values changed and what shape their distributions took. Once the program was debugged, however, these diagnostics became irrelevant, and you spent time commenting out or deleting the PUTs and PROCs. You do this a bit ruefully, because you know that you'll eventually be uncommenting and/or rewriting these statements when you have to modify or enhance the program. Surely there must be an easier way to tell SAS to execute or bypass a group of related (debugging) statements.

This code "toggling" can be accomplished first by recognizing the compile-execution timing issues of SAS and the macro language. If we want to turn off some statements, we can do this by filtering them out with macro statements. That is, the compilation phase of the program will pass only the statements of interest to SAS for execution. The sample macro shown in this section is a simple but well-rounded collection of techniques for achieving this filtering. Once we are aware of the timing distinctions, the rest is "just coding."

SOURCE

An example of the output toggling technique is found in **Exhibit 3**. Refer to "Source Comments," below, for an explanation of the highlighted items (those identified by white print in a black circle).

Exhibit 3: Test Macro With Debug Parameter

```
%macro test(dbg=yes);
       %let dbg = %upcase(&dbg.);     ❶
       %if &dbg. = YES %then %do;     ❷
           %let star = ;  %let cancel =;
           %end;
           %else %do;         ❷
                 %let star = *; %let cancel = cancel;
                 %end;
       data part1;
            set datain.pressure;
            pressureDiff = new_bp - baseline_bp;
            change       = (pressureDiff / baseline_bp) * 100;
            ❸ &star. if abs(pressureDiff) < 1 then put "Pressure diff < +|-1 " subject= pressureDiff=;
            ❸ &star. if change > 0 then put 'BP increase! ' subject= baseline_bp= new_bp= change=;
       run;
       proc print data=part1(obs=10);
       run &cancel. ❹;
       %if &dbg. = YES %then %do;     ❺
           proc freq data=datain.pressure;
                tables drug;
           run;
           proc means min max;
                var age;
           run;
           %end;
%mend;
```

SOURCE COMMENTS

❶ Convert DBG to upper case. This makes subsequent references to it more consistent. We simply refer to YES rather than having to test for YES or yes.

❷ We create two variables, star and cancel, that will be used later to control the insertion / exclusion of code. If we want debugging output (DBG=YES), we assign null values to both variables. Otherwise, we assign an asterisk (*) and cancel, respectively.

❸ A reference to STAR precedes two IF statements. Recall that its value will be either null or *. Either way, the syntax of the resulting statement will be valid.

❹ The RUN statement is given &CANCEL as its optional parameter (yes, RUN has a parameter, and has had it for many years!). The result is either null or cancel. Here, as in the references to STAR, the statements passed to SAS for execution will be syntactically valid. There's no good reason why the entire PROC could be buried in the macro IF-THEN that immediately follows it. The RUN CANCEL technique is shown here simply to show an alternative solution.

❺ If DBG is turned on (is YES), the FREQ and MEANS procedures are executed. Otherwise, nothing is passed to SAS for execution.

USAGE

A few simple uses of the debugging toggle technique are shown in **Exhibits 4 and 6**. Let's first use the macro with debugging output on (DBG=YES), shown in **Exhibits 4 and 5**. Later, in **Exhibit 6** we turn debugging off. Output from all exhibits is modified for display purposes.

Exhibit 4: Log When Invoked with Debugging

```
%test(dbg=yes);

35         %test(dbg=yes);
MPRINT(TEST):   data part1;
MPRINT(TEST):   set datain.pressure;
MPRINT(TEST):   pressureDiff = new_bp - baseline_bp;
MPRINT(TEST):   change = (pressureDiff / baseline_bp) * 100;
MPRINT(TEST):   ❶ if abs(pressureDiff) < 1 then put "Pressure diff < +|-1 " subject= pressureDiff=;
MPRINT(TEST):   ❶ if change > 0 then put 'BP increase! ' subject= baseline_bp= new_bp= change=;
MPRINT(TEST):   run;

Pressure diff < +|-1 Subject=P1 pressureDiff=0.2
BP increase! Subject=P1 Baseline_BP=92.8 New_BP=93 change=0.2155172414
Pressure diff < +|-1 Subject=P2 pressureDiff=0.1
BP increase! Subject=P2 Baseline_BP=94.9 New_BP=95 change=0.105374078
Other diagnostic output omitted
Pressure diff < +|-1 Subject=A16 pressureDiff=-0.3
BP increase! Subject=A28 Baseline_BP=91.4 New_BP=93.6 change=2.4070021882
NOTE: There were 93 observations read from the data set DATAIN.PRESSURE.
NOTE: The data set WORK.PART1 has 93 observations and 7 variables.

MPRINT(TEST):   proc print data=part1(obs=10);
MPRINT(TEST):   run ❷;

NOTE: There were 10 observations read from the data set WORK.PART1.
NOTE: The PROCEDURE PRINT printed page 1.

MPRINT(TEST):   ❸ proc freq data=datain.pressure;
MPRINT(TEST):   tables drug;
MPRINT(TEST):   run;

NOTE: There were 93 observations read from the data set DATAIN.PRESSURE.
NOTE: The PROCEDURE FREQ printed page 2.

MPRINT(TEST):   ❸ proc means min max;
MPRINT(TEST):   var age;
MPRINT(TEST):   run;

NOTE: There were 93 observations read from the data set WORK.PART1.
NOTE: The PROCEDURE MEANS printed page 3.

NOTE: SAS Institute Inc., SAS Campus Drive, Cary, NC USA 27513-2414
NOTE: The SAS System used:
      real time           0.42 seconds
      cpu time            0.19 seconds
```

EXHIBIT 4 COMMENTS

❶ Since macro variable STAR was null, the two IF statements are passed to SAS for execution and are not preceded by an asterisk, (which would have commented them out).

❷ Since macro variable CANCEL was null, the statement passed to SAS for execution is RUN; The procedure code preceding the RUN statement will be executed.

❸ The macro's %IF statement evaluated DBG=YES as true, and thus passed the two procedure calls to SAS for execution.

Partial output from the macro is displayed in **Exhibit 5**, below.

Exhibit 5: Output When Invoked with Debugging

Obs	Subject	Age	Baseline_BP	New_BP	Drug	pressure Diff	change
1	N1	68	100.3	79.9	new	-20.4	-20.3390
2	N2	59	91.6	85.9	new	-5.7	-6.2227
3	N3	60	90	85.7	new	-4.3	-4.7778
4	N4	57	93.7	85.8	new	-7.9	-8.4312
5	N5	58	95.4	85.3	new	-10.1	-10.5870
6	N6	59	99.6	83.2	new	-16.4	-16.4659
7	N7	69	97.9	82.9	new	-15.0	-15.3218
8	N8	65	96	84.9	new	-11.1	-11.5625
9	N9	59	96.9	86.1	new	-10.8	-11.1455

```
    10     N10                41           95.4             90    new           -5.4      -5.6604
```

Page Break, Title

```
                                    The FREQ Procedure

                                                          Cumulative    Cumulative
                 Drug         Frequency     Percent     Frequency      Percent
                 ------------------------------------------------------------
                 approved          29       31.18            29        31.18
                 new               31       33.33            60        64.52
                 placebo           33       35.48            93       100.00
```

Page Break, Title

```
                                    The MEANS Procedure

                                  Analysis Variable : Age

                                     Minimum         Maximum
                                  ----------------------------
                                   41.0000000      75.0000000
                                  ----------------------------
```

Finally, let's see what changes when we turn off debugging. **Exhibit 6** tells the tale.

Exhibit 6: Log When Invoked without Debugging

```
35         %test(dbg=noThanks); ❶
MPRINT(TEST):   data part1;
MPRINT(TEST):   set datain.pressure;
MPRINT(TEST):   pressureDiff = new_bp - baseline_bp;
MPRINT(TEST):   change = (pressureDiff / baseline_bp) * 100;
MPRINT(TEST):   ❷ * if abs(pressureDiff) < 1 then put "Pressure diff < +|-1 " subject=
pressureDiff=;
MPRINT(TEST):   ❷ * if change > 0 then put 'BP increase! ' subject= baseline_bp= new_bp= change=;
MPRINT(TEST):   run;

NOTE: There were 93 observations read from the data set DATAIN.PRESSURE.
NOTE: The data set WORK.PART1 has 93 observations and 7 variables.

MPRINT(TEST):   proc print data=part1(obs=10);
MPRINT(TEST):   run ❸ cancel;

NOTE: The procedure was not executed at the user's request.

❹

NOTE: SAS Institute Inc., SAS Campus Drive, Cary, NC USA 27513-2414
NOTE: The SAS System used:
      real time           0.37 seconds
      cpu time            0.18 seconds
```

EXHIBIT 6 COMMENTS

❶ `YES` turns on debugging output. *Anything* else will bypass it.

❷ Since macro variable `STAR` was `*`, the two statements become comments – what were IF statements in the previous example are now passed to SAS as comments.

❸ Since macro variable `CANCEL` was given the value `CANCEL`, the statement passed to SAS for execution is `RUN CANCEL;` The procedure code preceding the RUN statement will not be executed. It *will*, however, be checked for valid syntax. If the syntax scan generates an error or a warning, it will print a message in the Log and, in most operating systems, set the value of automatic macro variable `SYSRC` to 4.

❹ The macro's %IF statement evaluated `DBG=YES` as false, and thus did not pass any statements to SAS for execution. That is, "compile time" produced nothing for "execution time."

EXTENSIONS

We can make a variety of tweaks, based on the demands of the program:

1. The `DBG` values YES and "not YES" are, in effect, values of 0 and 1. It's not too great a leap to consider a scale with more values. The expanded coding scheme would give more control over what, exactly, was produced. In this scenario, for example, 0 might suppress all output, 1 would just run the procedures, and 2 would be the "fully loaded" set of output, performing everything that level 1 did plus the DATA step's PUT statements. It's easy to code. The hardest aspect of this scheme would be deciding what gets done at each level (and, of course, documenting this behavior).
2. A new option, `DBGOBS`, could be added to control the number of observations that PRINT would produce. It could also be applied to the number of diagnostics that the IF statements would produce. Note that the latter case requires a counter in the DATA step. This adds some coding burden: not only does the program have some extra counting to do, but it also requires a DROP statement to ensure the counter isn't added to the data set. It's a small price to pay for flexibility.

COMMENTS

Sometimes the most basic of needs are satisfied with simple applications of obscure or overlooked and vaguely recalled knowledge. As these simple examples suggest, before you start complicated coding for what *seemed* to be a simple task take a minute to browse the dictionary tables, go through some old programs, and do whatever you can to jog your memory.

Your comments are welcome. Email Frank@CodeCraftersInc.com.