Building Macros and Tracking Their Use

Richard Koopmann, Jr., Capella University, Minneapolis, MN

ABSTRACT

Many of the routine tasks we do in SAS can be converted into parameterized macros. Whether you are processing data sets for regular reports, automating infrequent and mundane tasks, or documenting SAS libraries, macros can help make repetitious tasks more tolerable and less error-prone. Developing a common macro script structure and committing to it are just the first steps toward having a more concise code base.

One immediate benefit of using macros is the ease of implementing code changes. Typically, estimating the impact of impending source changes (e.g., adding a column, renaming a schema, or deleting a table in a SQL database) is a distributed task—end-users are asked to identify potential breaks in their code base. If the end-users tend to work with different versions of their code for different projects or requests, the number of SAS files to evaluate quickly multiplies. With macros, however, there will be a smaller fraction of SAS files to evaluate. Further, when it comes time to implement the changes, working with a smaller number of macros will always be easier than working with a larger number of files.

Macros, however, have a tendency to proliferate in their own right, though this is hopefully the result of identifying new opportunities for macros to replace longer pieces of code (or at least make it more succinct). With many macros being used throughout an organization, being able to track their use becomes equally important.

Simply asking SAS users which macros they use is an option, but this too will likely be inconsistent and incomplete. Adding a few lines of code to individual macros can make tracking a trivial task.

Tracking *what* macros are called *when* and by *whom* provides those responsible for maintaining the macros with vital information which can be leveraged to gather targeted feedback regarding upgrades or help identify macros to be discontinued.

This paper outlines framework for developing component macros that report back macro usage. While the framework was developed in a SAS 9.1/9.2 client-server environment (running under Windows XP, 32-bit), it should work under any client-server environment and could be modified to run under disparate clients sharing a common networked drive.

BUILDING MACROS

TURNING CODE INTO MACRO CODE

Removing static values from a piece of code is the first step toward making that code generalizable and reusable in similar contexts. On its surface, the process of converting a piece of working code into a functioning, parameterized macro is straightforward:

1. Identify the parameters in the working code.
2. Convert these parameters to macro variables.
3. Wrap code in a macro definition and add corresponding named or positional parameters.

Simply stopping at that point, the macro should perform as expected and the end-users will experience the benefit of concise code. However, adding some in-line help documentation to the macro will help new users of the macros become acquainted with the different parameters and the options.

A MACRO TEMPLATE

The following macro code outline is the result of many years of refinements to macros developed for use by a small number of analysts:

- Documentation header
  - Project – what project was the code developed for?
  - The purpose of the code – what is it supposed to do?
  - Inputs – any input data sets? external files?
  - Outputs – what are the resulting data sets? log output? external files?
  - Notes – anything that you'd like to share with others?
  - Change log – major changes are summarized, dated, and attributed to an individual
- Macro definition statement
- Set sensible defaults
- In-line help

- Use Tracker
- The functional portion of the macro code
- Clean up if needed

DOCUMENTATION HEADER

The initial portion of every SAS script should be the program documentation header. This identifies the project that the code was built to support, a quick overview of the purpose of the code, input and output data sets and sources, along with some overall use notes and a change log. Doing so helps set the expectation that new code produced should be documented. Additionally, Exploiting Consistent Program Documentation is considerably easier with consistently documented scripts.

SENSIBLE DEFAULTS

In the event that a user does not specify a value for a positional parameter where a default value makes sense, the macro developer should set these values with `%let` statements or via a `PROC SQL` or `DATA` step.

IN-LINE HELP

Many DOS (and UNIX) commands allow users to get high-level help by executing the command with a help parameter (`dir /?` or `man` pages). This convention is often comforting for those SAS programmers who are accustomed to command line computing. One issue to note is that when providing example macros calls in the in-line macro help, you need to make the example calls to not execute as regular calls. For simplicity, the % symbol is replaced with the # symbol.

For example, submitting `%d(HELP);` will generate the following in the log window and bring the log to the foreground:

```
*********************************************************************************
* D Returns a date value in the specified format.                               *
*********************************************************************************
* Positional Parameters (in this order):                                        *
*  FORMAT    The format to return DATE in. Defaults to date9 (drop '.').        *
*                                                                               *
* Optional Keyword Parameters (in any order):                                   *
*  D    Date value to use entered in date9. format. Default: value of SYSDATE9. *
*       This can be assigned with an additional, nested #d macro call.          *
*  INTERVAL    Interval to shift the DATE parameter by. Defaults to day.        *
*  INCREMENT    Amount to shift the DATE parameter by. Defaults to 0.           *
*  ALIGNMENT    Sets the position of the result. Defaults to b(eginning).       *
*********************************************************************************
* Example macro call (replace # with percent symbol)                            *
*  #d(worddate, d=07DEC1942, interval=month, increment=5, alignment=s)          *
*  #d(increment=3, d=#d(d=19NOV1863, interval=year.7, increment=-87))           *
*********************************************************************************
```

TRACKING USE

Some concerns with releasing a macro into the wild is that you generally don't know where it goes (who is using it), how popular it becomes (how frequently it gets called), or when it is time to retire it (when people stop using it). Additionally, the author of the macro often becomes its single source of support—though this burden can be lessened by providing in-line documentation, by naming parameters consistent with source variables, by using sensible default parameter values, and by sharing the macro source code for the curious.

These concerns are most easily resolved by tracking each time a macro is called. Since SAS compiles a macro from source only once (subsequent calls in the same session rely on the compiled version), the tracking cannot occur at the system level. One integrated tracking approach is accomplished by inserting a timestamped record into a permanent data set. For example,

```
proc sql noprint;

insert into sasuser.tracker

  set sysuserid=upcase("&SYSUSERID")

, Action=upcase("A brief description")

, Description=upcase("A detailed description of what is being tracked")

, datetime=dhms(today(),0,0,time());

quit;
```

The idea of tracking macro use *without* using a macro is illogical, so the static code was converted into a macro called `%TRACKER` accepting 2 parameters:

1. Action – a brief description of what type of action is being logged (e.g., *Login*, *Macro*).
2. Description – a more detailed description of the action (e.g., *Connect to server*, Name of macro being called)

Once the to-be-tracked macro has been built, adding the code to trigger the `%TRACKER` macro is a simple matter of adding three statements. To prepare, add these two statements near the top of the to-be-tracked macro (shortly after the `%macro` *`xxx`*`(...);` statement):

```
%local macro;

%let macro=&SYSMACRONAME;
```

After any in-line help that may be in the to-be-tracked macro, add the following statement to trigger the %TRACKER macro:

```
%tracker(Macro, &MACRO);
```

Client-Server Environments

At this point, in a single client environment, the tracking can stop. However, in a client-server environment, each client will need to push their accumulated tracker data up to a shared data set and then clear their local data set. Adding the following code to the autoexec will keep the centralized tracking data set up-to-date.

```
libname sys 'path-to-common-directory';

*** push local data ***;

proc append base=sys.tracker data=sasuser.tracker;

run;

*** clear local data ***;

data sasuser.tracker;

    set sys.tracker;

    stop;

run;

libname sys clear;

dm log 'clear;';
```

Others may want to add some additional code to confirm the local data set has been appended prior to clearing. The last line tells the display manager to clear the log and may be excluded according to user preference.

SIMPLE QUERIES

Returning the number of macro transactions logged in the prior 6 weeks is a trivial task (the %d macro returns a relative date value and is described below).

```
proc freq data=sys.tracker order=freq;

    where action eq 'MACRO'
    and datepart(datetime)
    between "%d(interval=week, increment=-6)"d and "%d(interval=week)"d ;
    table Description / nocum;
run;
```

And the (truncated) results:

```
Description                                                  Frequency     Percent
-----------------------------------------------------------------------------------
EDWREGLITE                                                        141       28.89
EXPEXCEL                                                           92       18.85
RLIBNAME                                                           65       13.32
EDWGRADLITE                                                        38        7.79
EDWREGLITE_SAS                                                     38        7.79
```

This tracking system was first implemented in JAN2008 and has logged over 75,000 transactions from 70 different users and 75 different macros.

RELEVANT EXAMPLE MACROS

%TRACKER

As described above, the `%TRACKER` macro takes only two arguments–an action and a description.

```
%macro tracker(action, description);
option nonotes;
proc sql;
insert into sasuser.tracker
  set sysuserid=upcase("&SYSUSERID")
, Action=upcase("&ACTION")
, Description=upcase("&DESCRIPTION")
, datetime=dhms(today(),0,0,time());
quit;
option notes;
%ByeBye:;
%mend;
```

To avoid cluttering the log with notes irrelevant to the macro, the `nonotes`/`notes` option is toggled when inserting the record into the tracker data set.

Since the `&SYSMACRONAME` macro variable is local to the macro evaluating it, we need to send the value of this automatic macro variable to a new macro variable. The value of this new macro variable is then sent to the `%TRACKER` as a parameter.

%PUSHLOCAL

Within macros, there are local and global variables–Local variables exist only within the scope of the containing macro. In client-server environments where macros are called and executed locally with portions of the code being `rsubmit`ted, it is vital to push local (client) macro variables up to the server. This tends to involve a series of `%SYSLPUT` statements. To facilitate this, the `%PUSHLOCAL` macro grabs the values of macro variables local to the calling macro and generates a series of `%SYSLPUT` statements which are subsequently `CALL EXECUTE`d.

```
%macro pushlocal(MACRO, debug=FALSE);
proc sql;
    create table _local_(label=Local variables used by &MACRO. macro) as
    select distinct name, value
    from dictionary.macros
    where scope eq "&MACRO";
quit;
data _null_;
    set _local_ end=last;
    calltext=catx(' ','%syslput',catx('=',name,value),';');
    call execute(calltext);
    put name @20 value;
run;
%if %upcase("&DEBUG") ne "TRUE" %then %do;
proc datasets library=work nodetails nolist;
    delete _local_;
run;
quit;
%end;
%mend;
```

%D

This is a simple utility macro which returns a formatted datevalue based on user specifications (defaults to `date9.`) format relative to a specified date (defaults to value of `&SYSDATE9.`). In this example, the bulk of the macro code devoted to providing in-line help–the functional portion of the macro is a single line:

```
%unquote(%left(%quote(%sysfunc(intnx(&INTERVAL, "&D"d, &INCREMENT, &ALIGNMENT),
&FORMAT..))))
```

Despite the lopsidedness of this macro, it has proven to be a widely used code for ongoing reporting needs.

```
%macro d(
  format
, d=&SYSDATE9
, interval=day
, increment=0
, alignment=b
);
%local macro; %let macro=&SYSMACRONAME;

%if &FORMAT eq %then %let format = date9;

%else %if %upcase(&FORMAT) eq HELP %then %do;
  %put ********************************************************************************;
  %put * D Returns a date value in the specified format.                              *;
  %put ********************************************************************************;
  %put * Positional Parameters (in this order):                                        *;
  %put *  FORMAT    The format to return DATE in. Defaults to date9 (drop '.').          *;
  %put *                                                                               *;
  %put * Optional Keyword Parameters (in any order):                                   *;
  %put *  D         Date value to use entered in date9 format. Defaults to SYSDATE9.     *;
  %put *  INTERVAL  Interval to shift the DATE parameter by. Defaults to day.            *;
  %put *  INCREMENT Amount to shift the DATE parameter by. Defaults to 0.                *;
  %put *  ALIGNMENT Sets the position of the result. Defaults to b(eginning).            *;
  %put ********************************************************************************;
  %put * Example macro call (replace # with standard percent symbol)                   *;
  %put *  #d(worddate, d=07DEC1942, interval=month, increment=5, alignment=s)          *;
  %put *  #d(increment=3, d=#d(d=19NOV1863, interval=year.7, increment=-87))           *;
  %put ********************************************************************************;
  dm log 'show';
  %goto ByeBye;
%end;

%unquote(%left(%quote(%sysfunc(intnx(&INTERVAL, "&D"d, &INCREMENT, &ALIGNMENT),
&FORMAT..))))

%ByeBye:
%mend  d;
```

CONCLUSION

Designing macros to be easy to learn with readily available help will encourage adoption. Designing macros to be robust and flexible will encourage continued use. Building in a tracking method will help macro designers know where to focus their limited attention and which end-users are using their macros.

FULL SOURCE CODE

The most current version of code contained within this paper will be maintained on the project page on GitHub. Additional relevant materials (e.g., this paper) will also maintained there.

CONTACT INFORMATION

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

Name: Richard Koopmann, Jr.
Company: Capella University
E-mail: richard.koopmann@capella.edu
Project: https://github.com/rkoopmann/build-and-track-sas-macros
Issues: https://github.com/rkoopmann/build-and-track-sas-macros/issues