Build a Production System in SAS MACRO

Robert P. Beeman
GPU Nuclear Corporation
TMI Nuclear Generating Station
Middletown, PA 17057
(717) 948-8181

ABSTRACT

The many nuances available to the SAS programmer can make macros seem daunting and hard to master. But in truth, a very straightforward path leads through the labyrinth of percent signs and ampersands. At its end is a simple program which will support the three basic parts shared by all production computer systems — The In, The Crunch, and The Out. Four uncomplicated macro modules tie together these three processes.

In addition to simplicity, this program maximizes the use of SAS subsystems (for instance, FSP), to shorten code and to ease later maintenance chores.

My emphasis will be upon code that the audience member can take away and use directly in his/her particular application.

Build a Production System in SAS MACRO

If you have programmed in SAS for several years, you may remember when linear code was all that SAS offered. We both got used to one-way, from-the-top-down code construction. Build a data step to INPUT from cards or SAS direct access storage. Perhaps do some mild data crunching in the input phase; exclude observations, or set program variables IF certain criteria are met. Then, SORT the data in some way, or MERGE or UPDATE it with other stored observations. Then PROC PRINT or FILE PRINT it as a report, or simply OUTPUT it in its new form back to storage again.

We have just identified all the components of any business computer system. In all of them, some way must be found to a) access data (The In), to b) operate on it (The Crunch), and to c) dispose of what remains (The Out). This is not a particularly startling revelation, but I ask you to bear it in mind for a moment while I suggest that programming in SAS MACRO is absolutely no different from linear SAS in its object. The object? Some variation of these three processes.

Programming in SAS MACRO is like learning to ride a motorcycle after driving a car for years. The basic object is the same, but the techniques vary a bit.

Enough theory. Let us build a system that you can take away with you and put to immediate use in your own business.

In one TSO dataset we will place four SAS MACRO modules, each of which will contain pieces of linear SAS code. Please remember that we can be as fancy as we like later, but for now, all we need are macros to do each of the three basic operations (In, Crunch, Out), and one more to keep an eye on everything.

Let's start by coding the module for The In. We'll want this macro to offer a choice of valid options to a system operator, to accept input from outside, and to go for help to interpret what it gets from the CRT.

%MACRO MAINSCR;
CLEAR;
%PUT %STR(MENU);
%PUT %STR;
%PUT %STR(ENTER TILLS TO);
%INPUT REPLY;
%MAINMAC;
%MEND MAINSCR;

Take a close look at the code. MACRO MAINSCR and MEND MAINSCR seem to suggest the macro's limits. Anything set within a MACRO and MEND statement will not be acted upon by the SAS system processor. To be used, the code inside must be called. This would imply that the above macro would never function if these lines were the only ones in the dataset, and this is quite true.

The %PUTs seem straightforward enough. Obviously, they write something to the screen, in this case a very rudimentary menu with one choice. Below them the %INPUT asks the operator to choose from a list of one.

So far, nothing special. But what is %MAINMAC? Here is where SAS MACRO begins to pay for itself. %MAINMAC is a macro call to a macro which we have mentioned, but haven't written yet—the system supervisor.

%MACRO MAINMAC;
%IF &REPLY=Q %THEN %00; %QUIT; RUN; %END;
%IF &REPLY=QUIT %THEN %QUIT; RUN; %END;
%IF &REPLY=SESAME %THEN %UTILITY; RUN; %END;
%MEND MAINMAC;

As before, we set the macro boundaries by a paired %MACRO MAINMAC/%MEND MAINMAC statement. Inside, we place a series of IF tests whose sole function (at this stage) is to examine the value of &REPLY, which holds the string that the operator has just input from the console. If the operator has chosen to QUIT the system, or if he/she has entered the alias Q, then this macro calls another macro, %MACRO QUIT, which, as you may suspect, contains an ENDSAS statement. It may, of course, leave the operator with as elaborate a farewell as your imagination can devise.
But for our purposes here, we might want to be satisfied with a simple and humble ENDSAS:

%MACRO QUIT:
ENDSAS;
%MEND QUIT;

Notice that either the explicit macro call (%IF &REPLY=Q %THEN %QUIT) or the longer, more formal do statement (%IF &REPLY=QUIT %THEN %DO; %QUIT; RUN; %END;) will work nicely. As in most of SAS, the shorter version saves a few code lines, but the longer version may be more self-explanatory to the system maintenance person.

Notice also that all of the valid choices are not on the screen. The programmer may wish to leave a door into a maintenance environment through the command SESAME, as above. You might use SESAME to experiment with your own macros without changing the rest of this code.

Moreover, I have purposely used commands which were the same as macro names, and those which weren't, to point out that MACRO MAINMAC frees the programmer from any regard for SAS MACRO naming conventions (8 characters, beginning with a letter, and so forth). Literally anything can be used from the screen to call a macro (including text strings with embedded blanks, but that technique requires some tinkering with %INPUT which is beyond us at the moment).

But suppose &REPLY contains something else, a value which falls right through the If statements? Using only the code I've provided so far, the system would just stop. We need one more line to close the loop.

Remember that we said that the SAS system processor will not act on any code within a %MACRO/%MEND statement until the macro is called within the program. There is one case, however, in which we must place a statement outside all macros, and this is the initial macro call to get the entire system started. Let's return to our first macro which writes the menu to the terminal.

At the end of this macro, but outside of its boundaries, we'll place the call to the menu writer:

%MACRO MAINSCR;
%PUT %STR( MENU);
%PUT %STR();
%PUT %STR( ENTER THIS TO);
%PUT %STR();
%PUT %STR( QUIT, Q LOG OUT OF RIGHT AWAY);
%INPUT REPLY;
%MEND MAINSCR;
%MEND MAINMAC;
%MEND QUIT;

We already know that MACRO MAINSCR prints a menu, accepts input, and then calls MACRO MAINMAC to interpret what it gets from the operator. Suppose MACRO MAINMAC can make no sense of the input, that is, suppose the value of &REPLY triggers none of the %IF statements. What happens next? The only statement available to be processed is %MAINSCR. So, failing to find a valid input, the system simply refreshes the screen and waits for better times.

This method of gentle persuasion avoids a sizeable burden of coding, and maintaining, error messages. Moreover, the most enlightened and well-worded error message is sure to be misinterpreted eventually, while a simple screen refreshment and the reprinting of the menu cannot be mistaken even by the novice.

For our purposes here, the actual location of the %MAINSCR call makes no difference in the same way that these macros can be held in any order in the dataset. I placed %MAINSCR just after the MAINSCR macro for the convenience of our discussion, but it could be located as the first line, or as the last if you wish.

Since it is possible that you may wish to do other things with your system than merely bail out of it, we should write at least two more macros to address the remaining two basic parts of any business system — The Crunch and The Out.

So far we have provided a way to start up the system and keep it going in the face of garbage from the terminal (%MAINSCR). We have written a menu and a filter to check commands, as well as a simple logoff routine. Each of these sub-areas of The In can be elaborated almost indefinitely. For instance, you may wish to have your beginning macro check for a USERID, and then TSO ALLOCATE only certain datasets. Or, you might call %MACRO BACKUP to back up your SAS direct-access storage through several iterations. Or, your MACRO MAINMAC might make some conditional tests on input from the screen. If that value is less than or greater than some other value, call a certain macro to do, or prevent, some specific thing. Or, you might test for the first pass through MACRO MAINSCR by setting a counter. Then, globalize this variable for use by other macros. This is particularly valuable when backing up system datasets at the beginning of a session; it prevents the backup from running every time %MACRO MAINSCR is called.
If time permits, we might discuss some of these specific applications. For now, let us write a macro to access your SAS storage through an FSP screen:

```sas
%MACRO _UPDATE;
PROC FSEDIT DATA=SAS.MASTER
SCREEN=SAS.SCREENV1;RUN;
%MEND _UPDATE;
```

At this point we had better revisit %MAINMAC and %MAINSCR. We need to instruct MAINMAC to recognize UPDATE as a valid command (and perhaps also its alias, UP or U) and we must add a line to our menu to offer this choice to our operator:

```sas
%MACRO MAINSCR;
CLEAR;
%PUT %STR( MENU);
%PUT %STR( ENTER THIS TO);
%PUT %STR( QUIT, Q LOG OUT RIGHT AWAY);
%INPUT REPLY;
%MEND MAINSCR;
%MEND MAINMAC;
```

A neat way to do this in test is by inserting %PUT statements into your linear SAS code within your macros. For example, in MACRO _UPDATE, a %PUT can inform the programmer that the FSP call is about to be made. In this case, it will be fairly obvious when FSP invokes; but suppose you wished to know when a specific dataset was allocated. A %PUT with some phrase in it just after the TSO ALLOC statement would suffice. Moreover, you can trace the actual execution of the macros themselves by making the first statement in each on a %PUT:

```sas
%MACRO MAINMAC;
%PUT %STR(MACROMAINMAC RUNS NOW);
%MEND MAINMAC;
```

If we go on we should consider the type of code we will write into our macro system. Every SAS programmer seems to have his or her own idea of what constitutes "elegant" code structure. Some prefer elaborately nested do loops, while others write simpler lookup tables. In many cases, the more a piece of code relies upon its implicit mainstream function, the more elegant it is considered.

When writing in a macro context, I urge you be as explicit as possible. Make your macro calls explicit, and not the implicit result of a failed test. You may notice that I have violated my own suggestion by causing %MAINSCR to activate after the implicit failure of the pass through MAINMAC. However, this is unavoidable if auto-refreshment is to be used in place of error message calls. In general, then, and mindful that no rule is unbendable, try to explicate calls, and to leave a clear trail of action through your modules.

```sas
%MACRO MAINMAC;
%PUT %STR(REPORTS, R SHOW REPORTS);
%MEND MAINMAC;
```

It may seem a little silly to insert %PUTs in a simple system such as this one. However, they can be valuable as your system grows more complex.

Our final macro will demonstrate a secondary screen call. First, we need another choice for our menu:

```sas
%MACRO MAINSCR;
CLEAR;
%PUT %STR( MENU);
%PUT %STR( ENTER THIS TO);
%INPUT REPLY;
%MEND MAINSCR;
%MEND MAINMAC;
```

Before we go on we should consider the type of code we will write into our macro system. Every SAS programmer seems to have her or his own idea of what constitutes "elegant" code structure. Some prefer elaborately nested do loops, while others write simpler lookup tables. In many cases, the more a piece of code relies upon its implicit mainstream function, the more elegant it is considered.

When writing in a macro context, I urge you be as explicit as possible. Make your macro calls explicit, and not the implicit result of a failed test. You may notice that I have violated my own suggestion by causing %MAINSCR to activate after the implicit failure of the pass through MAINMAC. However, this is unavoidable if auto-refreshment is to be used in place of error message calls. In general, then, and mindful that no rule is unbendable, try to explicate calls, and to leave a clear trail of action through your modules.

Notice that in MAINMAC I have used a third, and equally valid, construction of the do loop. The OR does not receive a % prefix; nor does AND when used this way. Generally speaking, the order of the IF statements is immaterial unless you decide to make it significant within MAINMAC for your own purposes. I would suggest, however, that it be left nonsignificant for ease of understanding later on.

Before we go on we should consider the type of code we will write into our macro system. Every SAS programmer seems to have her or his own idea of what constitutes "elegant" code structure. Some prefer elaborately nested do loops, while others write simpler lookup tables. In many cases, the more a piece of code relies upon its implicit mainstream function, the more elegant it is considered.

When writing in a macro context, I urge you be as explicit as possible. Make your macro calls explicit, and not the implicit result of a failed test. You may notice that I have violated my own suggestion by causing %MAINSCR to activate after the implicit failure of the pass through MAINMAC. However, this is unavoidable if auto-refreshment is to be used in place of error message calls. In general, then, and mindful that no rule is unbendable, try to explicate calls, and to leave a clear trail of action through your modules.
%MACRO RPRTSCR;
  (use the same techniques as in MAINSCR to display
   a report menu. You might offer a warehouse report
   by PART NUMBER, by BIN LOCATION, and by COUNT).
%INPUT &REPLY;
%IF &REPLY=NUMBER %THEN %PARTNO; RUN; %END;
%IF &REPLY=LOC %THEN %BINLOC; RUN; %END;
%IF &REPLY=COUNT %THEN %COUNT; RUN; %END;
%IF &REPLY=MENU %THEN %MAINSCR; RUN; %END;
%RPRTSCR;
%MEND RPRTSCR;

Notice how this secondary screen contains its own
input and "mini-mainmac" input processor. You
could just as easily keep all the IF statements in MAIN-
MAC itself, and simply call MAINMAC as you do in
MAINSCR. Either technique produces valid results. I
sometimes prefer to keep all my validation logic
together in one place (in MAINMAC), but either
method suffices. In either case, this secondary screen
continues to call itself (to refresh itself) until it gets a
valid input.

And that is all there is to constructing a modular SAS
MACRO production system which can be put to any
business use. May I suggest that you take this code
away to your own sphere of interest, plug it in as
written, and then begin experimenting. What you
see... works, albeit a bit crudely and without much
flair.

It has been adapted from a body of parent code of
about four thousand lines of my own which is in daily
use. However, as a basic framework upon which to
hang your own variations and elegancies, and around
which to construct your own answers to your own
particular needs, I hope it may be of some help.