Paper 220-26

Structuring Base SAS® for Easy Maintenance

Gary E. Schlegelmilch, U.S. Dept. of Commerce, Bureau of the Census, Suitland MD

ABSTRACT

Computer programs, by their very nature, are built to be flexible. A program is no more than a series of versatile building blocks, stacked in such a manner to produce a desired result. However, the user requirements change over time, and so must the program change to reflect those new requirements. There are a number of ways to lay out a program, so as to make it easy to find the places where change is required. And as always; it is far easier to start with a solid foundation than to try and retool after the fact.

INTRODUCTION

There is no single "perfect" set of rules for structuring program code. In this paper, I have made a series of suggestions and offered a number of observations as to habits I have tried to form over the years. I have found them to be effective and helpful in creating code that is easy to read, maintain, and understand.

SAMPLE CODE: LOGGING INTO A REMOTE SITE

Imagine, if you will. A new program, named MERGERMT.SAS is to be built on a VAX Alpha platform. The program is to accept a file from a remote site to update a local master. Space is at a premium on the local site, so the decision is made to log into the remote site needed, and update from the remote transaction file without copying it to the local site.

The files for interface are all in a uniform format. The current requirement calls for the transactions to be coming in from a UNIX platform, but could be coming from a number of different platforms in the future. So, I started with a simple piece of code that could log into one remote site, and establish the SAS library containing the data I needed to access.

```
/*  Log into the UNIX
    platform            */

filename RLINK
         'LIB:TCPUNIX.SCR';
%let UNIX01=184.131.137.13;

/* IP address for the node */

data _null_;
  options comamid=TCP
          remote=UNIX01;
  signon;
run;

libname REMOTE
        '/sys/update'
        server=UNIX01;
```

Note: TCPUNIX.SCR is a script that contains the SAS/CONNECT SIGNON/SIGNOFF script for connecting to any UNIX host via the TCP access method. The script is copyright ©1990 by SAS Institute Inc. The script is associated with filename RLINK to connect with the SIGNON process.

The `%let` is a Macro command, which sets a macro variable to the value specified.

Tip: To determine the IP address of a given node or platform, enter `multinet nslookup <node>` at the $ prompt on VAX Alpha, `grep <node> /etc/hosts` on UNIX, or `PING <node name>` at a DOS prompt on a Windows platform. These commands will return the IP address for the requested node name, if it is connected to the network on which you are currently working.

Notice that the few lines of the module already reflect a few basic structuring items. The lines within a DATA step are indented to reflect their subordinate status to the DATA step. Comments are included, both to indicate the function of the block, but also to point out uncommon features in the code. Blank lines are added to separate the code into small, readable blocks.

One of the main uses of indentation is to show a more detailed level of program flow. In the above example, it's easy to see at a glance that the OPTIONS and SIGNON statements are performed within the DATA step. The positive benefits become more noticeable in later steps with the `%macro` and `if` statements.

I also use both upper- and lower-case when writing my code. This, of course, is a matter of personal preference; SAS does not distinguish between upper- and lower-case lettering. However, you will note that I put all SAS constructs in lower case, and data items in upper-case. In Interspeak, the common dialect for Internet communication, common text is in lower-case, and items to be accentuated or "shouted" are in upper-case. To my way of thinking, the SAS code is set, checked by the interpreter, and as such does not need to be accented; the interpreter will flag any errors. On the other hand, user-defined items like data names and libraries are frequently things that deserve extra notice during debug and maintenance. So, I accent them by putting them in upper-case. You can see even from the small example here what stands out.

Another way you can help yourself in maintaining the program is to use field names that are as clear as possible. If a field is used only as a macro to carry a text message to the log in the event of an error, don't call it TXT or X2; call it ERRTEXT. That way, the program becomes more self-documenting.

In Version 6.12 and earlier, it could become difficult to clearly document data items solely by name. So, take the time to comment the field as soon as possible. An old documentation standard is to define a term as soon as it is used; it is helpful to do the same in your program. For instance, if you needed to write an inventory program for a retail store:

```
length NBRONHND 6.
  /*  Number of items
      on hand          */
       NBRONORD 6.;
  /*  Number of items
      on order         */
```

In Version 8 on, the programmer can make use of the 32-character field names, including underscores. So the same names in the example could be named NBR_ON_HAND and NBR_ON_BACK_ORDER, respectively.

Back to MERGERMT.SAS. Now, once this small login module is functional, let's take it to the next level; getting set up for different nodes. For the sake of brevity, we'll just use two nodes here. The Macro versions of `%if, %then, %do, %else` and `%end` are used here, because the commands are being executed outside a DATA Step.

```
%let NODE=SYSPARM();
%if &NODE=UNIX01 %then %do;
  %let UNIX01=
       184.131.137.13;
  libname REMOTE
          '\sys\update'
          server=&SYSPARM;
  data _null_;
    options comamid=TCP
            remote=UNIX01;
    signon;
  run;
%end;
%else
```

```
  %if &NODE=UNIX02 %then %do;
    %let UNIX02=
         184.131.137.18;
    libname REMOTE
            '\sys\update'
            server=&SYSPARM;
    data _null_;
      options comamid=TCP
              remote=UNIX02;
      signon;
    run;
  %end;
  %else %do;
    %let ERRTEXT=INVALID NODE
NAME ENTERED.  LOGIN TO REMOTE
NODE ABORTED.;
    %put &ERRTEXT;
    %put &NODE;
    data _null_;
      abort;
    run;
  %end;
%end;
```

The `libname` statement is placed here in the event that a different platform is added later. The initial requirement calls for the remote platform to be UNIX. However, if the requirement changes later to include a different platform, the `libname` statement will have to reflect the directory structure of the remote platform.

Notice that I always put each command on a separate line. This not only makes each line easier to read, by making it more clearly defined; but it lends itself more easily to future expansion of the code.

If you use the Data Step Debugger to find flaws in your code, it becomes even more significant. If multiple commands are on a single line, it is more difficult to follow the flow of the code. With a separate line for each command, you can readily see which line is being processed by the Debugger, and hence more easily find the flawed line.

Another time-saver; when an edit hinges on the value of a variable, display the value of the variable as a part of the abort message. It saves time when trying to find out why a process failed.

The code above was the second version of the program. As I was always running the program interactively, it was no problem to set the NODE variable manually at runtime. And an error routine was added in, just in case of a typo in entering the node I wished to use.

The `%put` is a Macro command, which can exist inside or out of a DATA step, and writes the contents of the macro variables into the log.

This fulfills the immediate requirement; it permits the program to log into either node, but requires a program change if login to a different node is desired. The next step is to make it more versatile, and not require the program change each time. Because the comparison statements are used within a macro, and not inside a DATA step, we must use `%if`, `%do`, and `%end` rather than `if`, `do`, and `end`.

```
filename RLINK
'LIB:TCPUNIX.SCR';

%macro LOGNODE(NODE);
  %if &NODE=UNIX01 %then %do;
    %let UNIX01=
         184.131.137.13;
      ::
      ::
      ::
  %end;
%mend LOGNODE;

%LOGNODE(&SYSPARM);
```

Note: the :: here represents the code from the previous example, so as not to take up the space in this paper with repetitive code. Again, the code is indented one level, to show that it is subordinate to the `%macro` command. To further document, the `%mend` statement includes the name of the Macro being used; it's not necessary, but it does clearly show where the macro ended.

Using a Macro to define repetitive code has a number of advantages. Compile time is reduced slightly, as the code within the Macro is only defined once. Your program will be smaller, for the same reason. Most importantly, the overall maintenance is reduced; in the event that future enhancements are required, there will only be one change to make, instead of searching the program to ensure they are all made in the same way in each place.

So, how does the name of the node get into the program? By adding a parameter to the runstream;

```
$ SAS/SYSPARM="UNIX01" -
    PROGRAM1.SAS
```

SYSPARM is an automatic macro variable; that is, it is already defined by SAS. A parameter passed in this manner is automatically passed into the system-defined macro variable SYSPARM, and requires no further definition by the user.

Notice here that the indentation becomes more significant. In each of the `%if...%do` blocks, the terminating `%end` is aligned with the `%if`. What happens if the statement is TRUE is quite evident. Moreover, in the case of debugging, it narrows down the focus of what is or is not happening in the program. As an example; if the "INVALID UNIX ID" prints in the logs, you can be certain that the contents of the variable &NODE is neither UNIX01 nor UNIX02.

Important note to all programmers; no matter how many times you say "It should (or shouldn't) be", it rarely changes the program code or its functionality. It's much better to start looking at blocks of code and following the flow of data. That is a great deal easier with a little structure.

So, now we have a program that successfully allows the program to establish login and a remote site of UNIX01 or UNIX02 to your session. It can be executed with different parameters without changing the regular version of the program. Then, it occurs to you; other people in your section are also creating remote sessions in their programs, plus four other programs on your development schedule will require the same type of code. You could cut-and-paste it into other programs, and e-mail the code to everyone; or create a reusable module.

Take the macro code, without the invocation, and place it in a separate file called LOGNODE.MAC. Add a block of comments to the top, perhaps something like this:

```
/*------------------------
   LOGNODE.MAC

   This routine will permit
   the user to pass a node
   name as an argument and
   log into that node as a
   remote host.
------------------------ */
```

For purposes of this example, we'll assume that all reusable code modules are stored in a centralized directory, and they are accessed via a logical called LIBRARY:. The naming convention will of course change depending on the platform; it might as easily be /office/common/lib on UNIX, or C:\lib on the PC.

So now the program might look something like this:

```
/* ------------------------
   MERGERMT.SAS

   This program will merge
   a file on a remote node
   into the master dataset
   on the VAX Alpha.
------------------------ */
```

```
/*  use options that place
    the value of macro
    variables, and the
    macro code as it
    executes, into your
    LOG file.           */

options mlogic mprint
        symbolgen;

/*  incorporate the macro
    file into your program. */

%include "LIBRARY:LOGNODE.MAC";

%LOGNODE(&SYSPARM);

/*  update the existing
    dataset with the one
    from your remote host.
-------------------------*/

libname FILES
        'FILE_DEV:[MASTER]';

data FILES.MASTER;
  update FILES.MASTER
         REMOTE.TRANS;
  by ID CATEGORY;
run;
```

The final product takes advantage of the modular nature of SAS, laying out each function in a separate, testable module. The program is easy to read, enhance, and maintain. Moreover, if someone in your area needs to log into UNIX99, VAX045, or UNISYS37, those can easily be added to the LOGNODE.MAC macro at no impact to the calling program. Everyone wins.

For example, it could easily be enhanced to read a file or dataset containing all available nodes and their corresponding IP addresses, and reduce the code overhead further.

More importantly for the immediate process, we can now expand on what was started by putting structured code in the macro. Now space and time is reduced for multiple programs, not just the originating one. Also, maintenance done in the .MAC routine is immediately accessible to any program using the module, without the compile or relink required by other languages.

Now, let's look at the same code without the benefit of the lines we added for readability and maintainability:

```
OPTIONS MLOGIC MPRINT SYMBOLGEN;

%LET NODE=SYSPARM();
%IF &ND=UNIX01 %THEN %DO;
%LET UNIX01=184.131.137.13;
DATA _NULL_;
OPTIONS COMAMID=TCP
REMOTE=UNIX01; SIGNON; RUN;
%END;
%ELSE %IF &ND=UNIX02 %THEN %DO;
%LET UNIX02=184.131.137.18;
DATA _NULL_;
OPTIONS COMAMID=TCP
REMOTE=UNIX02; SIGNON; RUN;
%END;
%ELSE %DO;
%LET TEXT=INVALID UNIX ID
ENTERED.  LOGIN TO UNIX
ABORTED.;
%PUT &TEXT; %PUT &ND;
DATA _NULL_; ABORT; RUN;
%END;
LIBNAME REMOTE '\SYS\UPDATE'
SERVER=&SYSPARM;
LIBNAME FILES
'FILE_DEV:[MASTER]';
DATA FILES.MASTER;
UPDATE FILES.MASTER
REMOTE.TRANS;
BY ID CATEGORY; RUN;
```

This program is precisely the same, functionally, as the structured one above.

It's fairly easy to see which would be the easier one to debug and maintain. Or, to put in another light; say you write this program, and it runs fine for a year. Then, the decision is made to add some new features. A programmer could tell at a glance where the changes might be required in the first program – and have to re-analyze the second.

In fact, that alone is a good test of a

program's structure. Minor requirements changes should translate to minor changes to the program. If a minor requirements change means an overhaul of the program; then the program's basic structure should be examined. And there's no better time to do that, than when it's first laid out.

STRUCTURING FOR EFFICIENCY – WITHOUT LOSING READABILITY

Sometimes, you can make existing code clearer to read, and more effective at the same time.

Take this example. One of my programs was using a DATA step to reduce the size of a dataset prior to sorting. It seemed sensible to sort only the data I actually needed. So, I wrote a very simple DATA step prior to my sort:

```
data WORK.TEMP;
  set DATA00.TRANSACT;
  if TYPE='1' or TYPE='2';
run;

proc sort data=WORK.TEMP
          out=WORK.SORTED;
  by ID TYPE;
run;
```

Nice and simple. However, another member of my group suggested something even cleaner;

```
proc sort
     data=DATA00.TRANSACT
     out=WORK.SORTED;
  by ID TYPE;
  where TYPE in ('1','2');
run;
```

Much better. The new version of the code (a) eliminated the storage of one temporary dataset (b) reduced the physical I/O of reading and writing from the input dataset twice, and (c) made more clear the intention of the routine. It also makes the routine easier to update. One or two IF statements aren't hard to read or follow, but if the new requirements call for 15 different TYPE's, the IN statement makes the intent of the WHERE clause very clear.

CONCLUSIONS

Any code, regardless of the language used, can be written for greater understanding. SAS tools lend themselves well to modular programming, permitting an ease in adding and changing functions of the programs in the future. Developing reusable modules saves coding time at the start of a project, and maintenance time in the future. Most importantly, building well-structured programs is a source of justifiable pride for the developer, because well-written code endures – while poorly written code that is difficult to maintain and use is inevitably discarded.

REFERENCES

Aster, Rick, Professional SAS Programmer's Pocket Reference, 2nd Edition, 1998, Breakfast Books.
SAS Companion for Microsoft Windows Environment, Version 6, 2nd Edition, SAS Institute Inc.

CONTACT INFORMATION

Gary E. Schlegelmilch
U.S. Dept. of Commerce, Bureau of the Census, ESMPD/MCDIB
Suitland Federal Center, Rm. 1200-4
4700 Silver Hill Road
Suitland MD 20746
Email
Gary.E.Schlegelmilch@ccmail.census.gov