ABSTRACT

Several years of experience supporting SAS/AF application developers have yielded some tried-and-true methods for writing good SCL code. This paper will highlight the techniques which are most useful for making SAS/AF applications easier to write, easier to debug, and easier to maintain. These techniques can be categorized as 1) good habits which contribute to clarity and ease of understanding, 2) handling common trouble spots and, 3) hints for debugging programs. Though geared toward novice application developers, the material in this paper should also be useful for more experienced developers.

INTRODUCTION

Many of the "tips" in this paper are just common-sense practices which apply to any programming language. Others are specific to SAS/AF. As is true with almost any application development language, the practices seem like an undue burden during the development phase of an application. It is only after some time has passed that their value becomes apparent. However, if you will make a concerted effort to follow some of these good practices, you will find that they become a matter of habit after a very short time.

GOOD HABITS

The majority of the tips and hints documented in this paper can be classified as good coding habits. They are further broken down into clarity, error avoidance, and performance. These are all things which, if done during the development phase of an application, will facilitate the support and maintenance of the application during its lifetime.

Clarity

The tips related to clarity are the most straightforward of all. They are things you can do to make your programs easier to read and follow. They become especially important in the future when you or someone else must make changes to a program.

- Use two level names for all data sets, including WORK data sets so it is easy to differentiate a data set NAME from a data set IDENTIFIER.

\[
\text{DO } \text{idxxx} = \text{open('WORK.XXX','I');}
\]

\[
\text{DON'T } \text{idxxx} = \text{open('XXX','I');}
\]

Also, on the subject of naming, you should use only 7 characters for SAS data set names. This is a requirement if you need to create an index for a SAS data set. You'll be better off by "playing it safe" and using 7 character names all the time.

- Code all arguments to functions so that someone looking at a program does not have to remember the default values. Also, function defaults have been known to change from one release of software to the next, so don't get caught - be specific.

\[
\text{DO } \text{idxxx} = \text{open('WORK.XXX','I');}
\]

\[
\text{DON'T } \text{idxxx} = \text{open('WORK.XXX');}
\]

\[
\text{DO } \text{rc} = \text{delete('WORK.XXX','DATA');}
\]

\[
\text{DON'T } \text{rc} = \text{delete('WORK.XXX');}
\]

- When using the WORD function to extract words entered on a SAS/AF screen (command line/PF key/PMENU command), always translate the string to upper case by specifying the second argument as 'U'. This will insure that you can always compare the command entered to an upper case literal.

\[
\text{DO } \text{word1} = \text{word(1,'U');}
\]

\[
\text{DON'T } \text{word1} = \text{word(1);}
\]

- Use some sort of case convention. A suggested method is:

  a) put SCL code in lower case:

  \[
  \text{if sclvar1 = 2 then do;}
  \]

  b) put SCL constants in upper case:

  \[
  \text{dsname = 'WORK.XXX';}
  \]

  c) put SAS statements in upper case:

  \[
  \text{OPTIONS LS = 80;}
  \]
A complete section of code might look like:

```sas
label:
  if flag_var = 'TRUE' then do;
    submit continue;
    DATA _NULL ;
    FILE PRINT NOTITLES;
    PUT // 'No data found.';
    RUN;
  endsubmit;
  flag_var = 'FALSE';
end;
return;
```

Of course any literal text should be in whatever case you prefer, either upper or mixed case.

NOTE: Some people like to put SCL labels and return statements in upper case so they are easier to spot. That's fine, either way, always leave a blank line after a RETURN; and before a LABEL.

If your terminal supports color, use colors to differentiate comments from labels from SCL statements, etc. You can use the COLOR command on the command line to change the color of the next text to be typed. For example, you might enter COLOR TEXT YELLOW before typing a comment line and then enter COLOR TEXT RED before typing a label. It would be wise to assign these COLOR commands to function keys!

Also, don't forget about the IGNORE option on the FIND command. When you want to be sure to find a string regardless of whether it is in upper or lower case, add the word IGNORE (abbreviated I) on the FIND command as in FIND 'string' I.

• INDENT and do it consistently!!! My personal preference is to use 3 spaces for each level. Using the same section of code from above:

```sas
label:
  if flag_var = 'TRUE' then do;
    submit continue;
    DATA _NULL ;
    FILE PRINT NOTITLES;
    PUT // 'No data found.';
    RUN;
  endsubmit;
  flag_var = 'FALSE';
end;
return;
```

If you find yourself more than 3 or 4 levels deep you probably ought to be breaking the code up into multiple blocks!

• Use the ASIS option on the CONTROL statement so that your submitted code will retain the indentation as you typed it. Otherwise, your submitted code gets run together as one long string and it's hard to follow in the SASLOG. This is especially important if you have submit blocks containing DATA steps in which you have embedded CARDS. The data following the CARDS; statement may not be properly interpreted causing errors in your program.

• Use comments to match up DO-END and SELECT-END groups. For example:

```sas
select (option);
  when ('1') do;
    bunch of stuff
  end; /* End for option 1 */
  when ('2') do;
    bunch of stuff
  end; /* End for option 2 */
  otherwise do;
    bunch of stuff
  end; /* End for no option*/
end; /* End for select (option) */
```

• Use LINK to a "label" when you must execute the same section of code within the same program (but that code doesn't appear in other programs). When you must execute the same section of code from more than one program, store it in an SCL entry and call it using either CALL DISPLAY or CALL METHOD. If used by more than one application, the SCL entry can be stored in a separate catalog which contains other common-code entries.

• You should also LINK to a "label" to make a complex program easier to follow.

```sas
DO main:
  word1 = word1, 'U';
  select (option);
    when ('REPORTS')
      link reports;
    when ('UPDATES')
      link updates;
    otherwise
      link msgs;
  end;
return;
reports:
  ... lots of lines ...
return;
updates:
  ... lots of lines ...
return;
msgs:
  ... lots of lines ...
return;
DON'T main:
  word1 = word1, 'U';
```
if word1 = 'REPORTS' then
  do;
  ... lots of lines ...
  end;
else if word1 = 'UPDATES' then
  do;
  ... lots of lines ...
  end;
else
  do;
  ... lots of lines ...
  end;
return;

- Use meaningful comments. Especially, at the start of a program, you should include information such as:
  - Program name
  - Name of calling program (if there is one)
  - Explanation of input parameters (if any)
  - Explanation of output parameters (if any)
  - Brief description of purpose of program
  - One-sentence change history notes, for example:

  /*
  03/13/93 Added DEPT variable to LENGTH stmt. JFD */
  /* 05/21/93 Modified error check for COMPANY. JFD */

  Put a general explanation of the purpose of any labeled blocks at the start of each block.

  Don't use comments to state the obvious, for example DON'T do something like:

  X = Y; /* Assign value of Y to X. */

Error Prevention/Avoidance

The ideas in this section are geared toward the prevention of messy errors once an application is in production. They deal with handling all those situations which might "never" occur. Some examples include "the data set which will ALWAYS exist", "the code variable which will ALWAYS have an associated description", and "the command or parameter which will ALWAYS be in upper case". The old saying "Never say 'Never'" is one which you can put your money on!

- Define any non-screen SCL variables in a LENGTH statement at the beginning of the program. This is better than taking chances on SAS correctly assuming the default attributes. Furthermore, it is a requirement if you are relying on the SET call routine to correctly establish relationships between like-named SCL variables and data set variables.

- When defining function keys, always define all 24 keys, even if you plan to use only 12. Make 13-24 the same as 1-12. Also, be sure to use the NUMFKEYS function to find out if you can assign values using the SETFKEY function. For instance, if someone is using a device which recognizes only 12 function keys and you attempt to define F13, the SETFKEY function will fail.

- When invoking your application, always use the PFKEY=ALT option. This will insure that, when a screen uses default function keys, they will be the ones desired (i.e. F3 will be END instead of UNMARK).

- Use data-driven techniques to avoid hard-coding values in programs. Just a few ways to do this are:

  Store selection list values in a table (SAS data set or DBMS file). Never use SAS/AF LIST or HELP entries.

  Store menu selections in a table (SAS data set or DBMS file) and use extended tables to present the menu instead of painting the selections on a screen.

  Store information about data values in a table (SAS data set or DBMS file) such as valid minimum/maximum values, etc.

- Design applications to be portable. NEVER hard-code names of external data sets. Instead, assign filerefs to them in the autoexec file which starts-up the application. Or, if you must reference an actual data set name, pass it as a parameter or a macro variable. For example, you could use the special macro variable &SYSSCP to find out which operating system is being used and then construct application parameters as required:

  %put Local operating system is: &SYSSCP ;
  %macro host;
  %global rptname ;
  %if &sysscp = OS then
    %do;
    /* running in TSO */
    libname system "S09999.SASPGMS.LIBRARY";
    libname library "S09999.FORMATS.LIBRARY";
    let rptname = REPORT LISTING &MTDISK;
    filename reports "&Iptname";
    %end;
  %else %if &sysscp = CMS then
    %do;
    /* running in CMS */
    libname system "&MDRIVE";
    libname library "SYSTEMS &MDRIVE";
    let rptname = REPORT LISTING &MTDISK;
    filename reports "&Iptname";
    %end;
  %else %if &sysscp = OIM then
    %do;
    /* running in OIM */
    libname system "&MDRIVE";
    %end;
  %endif;
  %end host;
  %host run;
  dm 'af c=system.afcat.startup.program' af;

- Use the SYSTEM SCL function to execute host commands (instead of using the EXECCMD call routine.
DO rc = system('ERASE file-spec');
DON'T call execmd('X ERASE file-spec');

- NEVER reference a macro variable in SCL code as &varname. As a general rule, you should never see references to ampersands (&'s) in SCL code. They are OK as long as they are in a SUBMIT block but should not appear in SCL statements. The reason for this is that macro variable references in SCL statements are resolved AT COMPILe TIME. This means that, when a program is compiled, the value of the macro variable becomes a constant to that program. Consider the situation where you need to assign the value of the currently logged on user to an SCL variable named USERID. You SHOULD use the value of the automatic macro variable &SYSJOBID to do this using the SYMGET function as shown:

DO length userid $ 8;
   userid = symget('SYSJOBID'); /* will be person who is */
   /* currently running program */
DON'T length userid $ 8;
   userid = "&SYSJOBID";
   /* will ALWAYS be person */
   /* who compiled program */

However, using the assignment statement (userid="&SYSJOBID"), if your userid is S123456 and you are the person who compiled the program, the value of USERID will ALWAYS be S123456, no matter who is actually running the program!

- Be sure to use the correct function to retrieve the value of a macro variable to a character (SYMGET) versus numeric (SYMGETN) SCL variable. This is slightly different from using the SYMGET function in a DATA step which applies to either character or numeric DATA step variables. For example:

DO: length custname $ 30 ordrqty 8,
   custname = symget('MCUSTNM') ;
   ordrqty = symgetn('MQTY') ;

- When you close a data set, you should also reset the value of the corresponding data set identifier to missing. This will help keep things balanced when you are opening/closing the same data set multiple times. For example:

dsid = open('WORK.TEMP', 'U');
if dsid > 0 then do;
   call set(dsid);
   apprc = append(dsid);
   /* Only close if open. */
   closerc = close(dsid);
   /* Explicitly reset id. */
dsid = .;
end;

Performance

The performance tips mentioned here are all ones which have worked for us at Eastman. Actual performance improvements will certainly vary from one operating system to another and from site to site. There are many excellent papers in past SUGI Proceedings which deal with performance of SQL statements, DATA steps, and other non-SAS/AF functions. Be sure to review them too.

- As a general rule, an SCL function will be more efficient than submitting SAS statements. For example, use the SCL sort function instead of submitting a PROC SORT step. Other tasks which can be performed without submit blocks are creating indexes and changing attributes (format, informat, etc.) of SAS data set variables.

- Close the log window while an application is running. This can make an enormous difference in the elapsed time to perform a task, especially if the task is writing several lines to the log. This can be done prior to invoking a SAS/AF application by using the Display Manager command 'LOG OFF'.

- Don't re-do work that you've already done. For example, if your application creates a temporary SAS data set from a permanent file (RDBMS, flat file, etc.) which will not be updated during the application session, be sure you only execute that code one time. Use the EXIST function to see if the data set has already been created and then skip that section of code if it has.

- Use the WHERE and FETCH functions instead of the LOCATEC/N function when working with SAS data sets which have been indexed. The LOCATEC/N function will always read a data set sequentially while the FETCH function will attempt to use an index if possible. If you are not using indexes on large SAS data sets, you may wish to consider doing so to improve performance when doing look-up type tasks.

DEBUGGING HINTS

Testing and debugging your programs can be a frustrating experience at best. However, a little advance planning during the programming phase can save you time and effort when testing your programs. It will also pay off later on when the users of the application encounter problems.

During Development

The tips in this section are all geared toward helping you
follow the flow of your programs as you develop and test them.

- Use PUT statements to help you follow the path the program(s) takes. One convention is to use:

Within each program block:

```put '### At start of BBBBBB block in XXXXX.PROGRAM ###',
put 'fI##
At end of BBBBBB block in XXXXX.PROGRAM ###',
```

Within a program block, indent the text of the PUT statement by another few spaces and use a different "indicator" character:

```put ' O@@ After fetch from data set.';
```

When putting out the value of a variable or variables, use "vamame=" so you see the variable name as well as its value.

```put ' ea@ After fetch from data sec;' rc z
varl'" var2=
```

- When you're not sure whether your SCL program is generating submitted statements correctly, use the PREVIEW window to "peek" at them as they are submitted. If you don't like what you see, you can delete what's in the preview buffer and nothing will get submitted. You will need to be sure to NOT use CONTINUE on your SUBMIT statements until AFTER you have previewed the code. For example:

```AF C-libtef.catalog.entry.entity.entity DEBUG=YES
```

Before the program begins executing, you will be placed in the DEBUG window. There will be a DEBUG> prompt where you can type debugger commands. The simplest command is STEP which tells the debugger to execute the next statement in the SCL program.

Typing the word "STEP" over and over can get rather tedious. You can assign the STEP command to be executed by the ENTER key by typing the command: ENTER STEP on the DEBUG> prompt line.

Once you have entered the ENTER STEP command, just press the ENTER key when you want to advance your program to the next statement.

To exit from the DEBUG window and stop your program, you can type the QUIT command at any time on the DEBUG> prompt line.

After Production

The ideas in this section are meant to help you identify and solve problems which may occur after an application is put into production. They are also meant to help the users of the application understand what action they are to take when errors occur.
Instead of using SQL on a SUBMIT or SUBMIT CONTINUE statement, use PROC SQL. This allows you to review the submitted statements in the SASLOG and makes debugging much easier.

```
DO submit continue;
   proc sql;
      select ...
      quit;
   endsubmit;

DON'T submit continue sql;
   select ... ;
      quit;
  .endsubmit;
```

Use an application parameter to turn debugging options like PUT statements and/or the PREVIEW window on or off. For example, if you had a macro variable named &TRACE which is either on (1) or off (0), you would have the following code at the start of each SCL program.

```
length trace 8;
trace = symgetn('TRACE');
```

Then, for any statement(s) you want to have executed only when the trace option is on (like PUT, PREVIEW, etc.), you would execute it based on the value of TRACE. For example:

```
if (trace) then
   put '@@@ After fetch:' roc;
```

or:

```
submit:
   proc print data = temp;
      where dept = &dept';
   run;
endsubmit;
if (trace) then
   rc = preview('EDIT');
submit continue;
endsubmit;
```

Use consistent methods for error notification. Be sure to inform the user when something goes wrong and instruct them on what action to take. Put the same information in the SASLOG. Be sure to always include the name of the executing program and a more specific program location such as the labelled section currently executing. It makes sense to match the hierarchy of ERROR, WARNING, and NOTE used by the SAS System.

CONCLUSION

There is no doubt that it will take a little extra discipline to use these techniques at first. Just keep in mind that, after a few programs, they will become quite natural.