Abstract

Getting a program working correctly is sometimes only half the problem. Good programs are easy to understand and maintain by both the original programmer or others. This is especially true in a production environment or where someone is contracting to write SAS code. Users are paying large amounts for SAS systems and they deserve efficient, maintainable code when the project is finished. This paper will pull elements from several large SAS implementations and examine the techniques used to create tight, efficient data structures from within programs that are easy to write, understand, and maintain.

Introduction

This paper is the result of a staff meeting where we were performing a code review of a SAS system. While our organization does not have extremely rigid coding standards we do try to deliver a product that not only works, but is easy to understand and maintain from that point on. The techniques described below are some of the things that we work into our contracted systems.

What is a good maintainable SAS program, and when is it finished?

In a production environment there are at a minimum a few requirements for a program to be finished:

- it works
- it has accounted for all (most) of the data, or other unusual circumstances it might encounter
- it is commented
- it can be understood and maintained by a reasonably competent programmer.

Beyond the minimums above, a few desirable features are:

- good naming conventions
- clear, simple, modular code
- explicit statements and options
- reasonable efficiency
- the source is easy to find, print, and change.

A large numbers of programs that I have maintained, appeared to have never gone beyond the first point (it works), when originally written. The program appears to work, and may very well have worked at some point in time. The problem of course is that requirements change over time, and programs need changes and maintenance.

Efficient Programs

Efficiency is a very interesting topic and much has been written about it.

Some program efficiency components are:

- CPU time
- wall clock time
- I/O time
- data storage consumed
- human efficiency.

This paper is most concerned with human efficiency consisting of programming time to:

- design the program
- initially code the program
- test, debug, document
- maintain after production.

Good Naming Conventions

One of the simplest things, but also one of the most overlooked is to establish good naming conventions before any coding is started. If this is put off either a lot of renaming will be done later, or at a minimum there will be confusion. Most sites have standards for naming source that may or may not have meaning, and you certainly must follow those rules if in place. If you have the choice, you can either try and build in some meaning to the programs, or you can just name them and not try to introduce any meaning.

For example: A driver program calling four other programs might be named like this:

SDRIVER
SOPTIONS
SFORMATS
SINPUT
SREPORT
It really isn’t that important which style is used, but my experience is that deciding on names before coding saves the most work later.

Managing the Source

One function that a large number of computer sites does not do very well is to make sure that:

- source programs match what is running in production
- they are easy to find, print, change and move into production
- they are backed up and protected.

Many sites have good procedures, but even then they may apply to traditional compiled languages, and SAS code may be difficult or unusual for those systems to handle. SAS allows you to store both uncompiled source and several types of compiled code. To complicate things source can be stored in native text files, in special library files such as Panvalet or Librarian, and more recently in SAS catalogs. Each of the combinations above have their own idiosyncrasies.

Uncompiled Source Code

Until recently, this was the only way SAS code could be stored. Source statements are edited and are usually stored in native text files, but could also use the library systems or SAS catalogs mentioned above.

Native Files

The SAS DMS FILE and INCLUDE commands as well as the %INCLUDE statement imports the source to the SAS session. Batch jobs point to those files with system control statements or startup options.

Advantages:

- a straight forward place to store programs
- there is only one copy of the program to manage
- it must be the same program that SAS is executing
- normal editors, utilities, backup, security can be used on these programs.

Disadvantages:

- other utilities and editors may need to be used
- the programs are not SAS files, so additional libraries are needed.

Source Management Systems

Systems such as Panvalet and Librarian store the same basic source as above but do it in proprietary libraries.

Advantages:

- good control over and auditability
- fairly good storage and reporting capabilities.

Disadvantages:

- systems are proprietary and may be difficult to use
- special programs and commands are needed to edit, extract for SAS to execute or include.

SAS Catalogs

Also recently the ability to store SAS source as program entries in a SAS catalog became available. The DMS SAVE and COPY commands export and import programs between DMS and the catalog.

Advantages:

- one library can be used to store programs along with all the other things SAS stores.

Disadvantages:

- SAS catalogs are foreign to many users
- naming conventions are difficult but important to understand
- to my knowledge only the SAVE and COPY commands can reference the source
- batch program cannot reference this source
- %INCLUDE cannot reference this source.

SAS Compiled Code

Version 6 brought us compile DATA steps, compiled SAS Macros, as well as compiled programs for SAS/FSP and SAS/AF. These complicate managing the source because there are now two things to manage for each program. The original source and the compiled version. The source is usually stored
in one of the files above, and the compiled version must be stored in a SAS catalog. There is no way to reconstruct the original source from the compiled code and no cross referencing between the two. This is exactly the problem faced by many COBOL shops and SAS may or may not fit well with source management procedures in place.

Advantages:
- compiling is only done once

Disadvantages:
- both source and compile modules must be managed
- source cannot be reconstructed from compile code
- macro facilities may behave differently because of the time delay between compiling and execution.

To summarize source management, there are a variety of places SAS can store source. Since source is an extremely valuable resource, it should be managed carefully.

Modular Techniques

Most people would agree that modular programming is desirable and SAS fits with modular coding in many ways by its basic design. At the highest level the use of DATA steps and PROC steps force a modular approach.

Use the PROC Step

Certainly one of the most unique features of SAS is the PROC step, and in many cases, especially with programmers proficient in other languages, the PROC step is under utilized. When learning SAS in my own case, I liked to program, I was used to handling all phases of programming such as initialization, looping, output etc. Upon finding a rich programming language like the DATA step I wrote a lot of extensive and complex DATA step programs. While I used the basic PROC PRINT and PROC SORT steps that I needed, I didn’t code many more PROC steps. Examples of PROCs being much simpler that DATA step coding are numerous:
- PROC FREQ for counting, and calculating percentages
- PROC MEANS/SUMMARY for summing at several levels and views simultaneously
- PROC RANK for ordering and assigning rankings and percentiles
- PROC TRANSPOSE for flipping data around
- PROC TABULATE for summaries, percentages and printed tables
- PROC FORMS for repetitive forms
- PROC FSLetter for form letter generation
- PROC SQL for virtually complete systems in a single step

I have seen DATA steps that did all of the above, and in most cases using at least some PROC step coding would have made those programs much simpler.

Use of %INCLUDEd Code

The %INCLUDE facility certainly provides a vehicle for a driving program to reference subprograms in a modular style. Besides the use of good modular structures in a single program, %INCLUDE can be used in a more general way to simplify:
- standard startup or cleanup code in a program
- macro code to be shared
- formats that are shared
- complex device or graphics settings
- common INPUT statement definitions
- utility routines
- including generated code from a previous step
- SEC (Somebody Else’s Code).

%INCLUDE with Version 6 allows more flexibility than ever before. The key to successful implementation of the above is to document common code so that users know where to look for it.

Explicit Coding vs. SAS Defaults

Certainly SAS has many useful default that can minimize the amount of coding required for a SAS program.

SAS can:
- provide SAS data set names when created
- assume all fields are wanted on the output data set
- assume field input columns, types, input lengths, internal lengths
- provide default formats and labels
- assume end of steps
- assume end of jobs
- assume the last data set created for input to another step
- assume widths for SAS formats and informats
- automatically initialize fields
automatically control looping in the DATA step
- automatic output in the DATA step
- automatic end of file checking.

Defaults such as these can be the most difficult thing for new SAS users to understand because nothing in the program indicates their presence, even though they may have great effect on the processing.

Recommendations:
- always provide a name on DATA steps and PROCs that create datasets
- always provide a name when referencing a SAS data set
- use the KEEP and DROP options on statements that create and read SAS datasets
- use one style of INPUT statement and be explicit
- ALWAYS CODE RUN STATEMENTS
- consider using ATTRIB, or LENGTH, FORMAT, INFORMAT, LABEL statements
- comment any unusual circumstances.

Example:
```
data softsale;
  input name $1-10 Age 14-15;
run;
proc print data=softsale;
title 'softsale';
run;
```

Obviously a program can be extremely implicit or can instead specify every possible option. A good compromise is probably the best way to go.

Of the list mentioned above the two that I would encourage the most are always coding the dataset name, and always providing RUN statements.

Coding dataset names on PROC and reading statements will eliminate accidentally using the wrong dataset and generally makes the program easier to understand.

Coding RUN explicitly stops the compile process and starts execution at that point. It will make programs behave exactly the way we want them to, it helps with some macro features, and it makes errors easier to spot.

Input Statements

One of the most critical, but also time consuming and tedious jobs is creating INPUT statements. Things that can help:

- use formatted style input statements
- code only one field per line
- add comments to the right of the field
- code all fields on the file if possible, comment out as needed
- generate INPUT statements rather than coding them
- share them with other users.

Example:
```
data MASTREC;
  infile d01;
  input &0001 RECYPE $CHAR01. /*RECORD TYPE */
    &0002 COMPNM $CHAR01. /*COMPANY NAME */
    &0003 OFFICE $CHAR06. /*OFFICE NUMBER (KEY) */
    &0009 MERCHANT $CHAR03. /*MERCHANT NUMBER (KEY) */
    &0012 STORENUM $CHAR02. /*STORE NUMBER (KEY) */
    &0014 CUSTNUM $CHAR09. /*CUSTOMER NUMBER (KEY) */
    &0023 CHECKT $CHAR01. /*CHECK DIGIT (KEY) */
    &0024 FILLER $CHAR01. /*FILLER */
    &0025 CONTRACT $CHAR01. /*contract date */
run;
```

There are several things you can do as you are coding your programs to make them more maintainable.

Clear concise programs are easier to write, maintain, transport, and just work better.

Coding Styles

Consistent spacing, indenting, and blocking of SAS code can greatly clarify programs.

Possibilities:
- code global statements, DATA, PROC and RUN statements in column 1, indent the rest
- indent IF, DO, SELECT logic
- consistently use upper and lower case characters.

Commenting Styles

SAS provides the * ; comment statements and the /* */ comment clause. The first style is a complete statement and as such can be used between other SAS statements as needed. The */ */ clause can be coded anywhere a space could appear including in the middle of or across several SAS statements. Special care must be taken in the MVS batch environment to avoid /* in columns 1 and 2 because that is also a JCL statement meaning end of program. In either case, comments are always a good idea if they are correct and maintained.
A mixture of "flower boxes" before major routines and comments on the same line as the SAS code works well in most cases. The hard part is getting programmers to code them. In addition a program header listing program name, purpose, inputs and outputs, author and contact information, as well as a change log are always a good idea. A quick line marking the end of each module is also useful when including them with other modules.

Example:

```
/****************************************************/
/* PROGRAM: TESTPGM1 */
/* AUTHOR : STEVEN FIRST */
/* SYSTEMS SEMINAR CONSULTANTS 608 222-7081 */
/* DATE : 3/1/94 */
/* DESCRIPTION: */
/* INITIALIZE SAS OPTIONS AND BUILD TEMPLATES */
/* **************************************************
GOPTIONS RESET=ALL;
TITLE1; FOOTNOTE;
RUN;

***************************************************/
/* INIT OPTIONS, TITLES, FOOTNOTES */
/***************************************************/
GOPTIONS ROTATE=PORTRAIT HPOS=80 VPOS=60 NOCELL
NOCHARACTERS NOBORDER;
/* ***************************************************/
PROC GREPLAY
TC=TEMPCAT
NOFS;
/* DEFINE TEMPLATE */
TDEF FOURPER
/* DEFINE PANEL 1 */
1/LLY=50 LLX=0
URY=50 URX=50
COLOR=BLACK
/* DEFINE PANEL 2 */
2/LLY=50 LLX=50
URY=50 URX=100
COLOR=BLACK
/* DEFINE PANEL 3 */
3/LLY=0 LLX=0
URY=0 URX=50
COLOR=BLACK
/* DEFINE PANEL 4 */
4/LLY=0 LLX=50
URY=0 URX=100
COLOR=BLACK
RUN;QUIT;
PROC CONTENTS
One of the simplest and best ways to document the data structures and output files is to include a PROC CONTENTS in the SAS job. In some cases it may not be necessary after testing is complete, but whenever data files are being moved to another site or system, a CONTENTS listing will be a welcome addition for the programmer on the other end.

Meaningful Structures and Names

Within SAS constraints of 8 character maximum names, it certainly makes sense to try and imbued as much inherent meaning as possible into the names themselves. There is a good chance that the Institute will increase the naming lengths, but in the meantime we must live with 8 characters. We can however assign variable labels of up to 40 characters to the variables. Those labels are used by most procs and are displayed via PROC CONTENTS, and as such serve as documentation.

Simplifying Complicated Code

There are times when complicated statements could be broken into several small clear statements for simplicity. Even though machine time may suffer clarity is usually worth it.

Example: Convert a packed decimal date to a SAS date.

Adequate:

```
DATA PACKDATE;
  INFILE RAWIN;
  INPUT NAME $1-6
  BIRTH PD4.0;
  SASBIRTH=INPUT(PUT(BIRTH,Z6.0),MMDDYY6.);
  FORMAT SASBIRTH DATE7.;
  RUN;
PROC CONTENTS DATA=PACKDATE;
  TITLE 'PACKED DATE EXAMPLE';
  RUN;
```

Meaningful Structures and Names

Within SAS constraints of 8 character maximum names, it certainly makes sense to try and imbued as much inherent meaning as possible into the names themselves. There is a good chance that the Institute will increase the naming lengths, but in the meantime we must live with 8 characters. We can however assign variable labels of up to 40 characters to the variables. Those labels are used by most procs and are displayed via PROC CONTENTS, and as such serve as documentation.
PROC PRINT DATA=PACKDATE;
RUN;

Better:

DATA PACKDATE;
INFILE RAWIN;
INPUT NAME $1-6 BIRTH PD4.0; /* PACKED DECIMAL BIRTHDAY */
    BIRTHC=PUT(BIRTH,MDMDY6.); /* CONVERT IT TO CHARACTER */
    SASBIRTH=INPUT(BIRTHC,MMDDYY6.); /* CONVERT IT TO SAS DATE */
    FORMAT SASBIRTH DATE7. ;
RUN;

PROC CONTENTS DATA=PACKDATE;
TITLE 'PACKED DATE EXAMPLE';
RUN;

Declarative vs Executable Statements

Declarative statements such as ATTRIB, LENGTH, RETAIN, ARRAY do their job during the SAS compilation process, where active statements such as INPUT, IF, DO etc. affect step execution. It may be advisable to separate the two kinds of statements to eliminate the clutter in the logic portion of the program.

Explicitly Coding ATTRIB and LENGTH Statements

A good argument can be made to always coded complete ATTRIB statements for all variables used in a DATA step. Doing so will insure that there is no question with the lengths, formats etc. that are assigned. For large files the space savings gained can be enormous.

Example:

DATA MASTREC;
    ATTRIB RECYPE LENGTH=7  FORMAT=$CHAR01.
LABEL='RECORD TYPE';
    ATTRIB COMPNM LENGTH=10 FORMAT=$CHAR01.
LABEL='COMPANY Name';
    ATTRIB OFFNUN LENGTH=3 FORMAT=$CHAR01.
LABEL='OFFICE NUMBER (KEY)';
    ATTRIB MERCHNUM LENGTH=3 FORMAT=$CHAR01.
LABEL='MERCHANT NUMBER (KEY)';
    ATTRIB STORENUM LENGTH=2 FORMAT=$CHAR01.
LABEL='STORE NUMBER (KEY)';
    ATTRIB CUSTNUM LENGTH=9 FORMAT=$CHAR01.
LABEL='CUSTOMER NUMBER (KEY)';
    ATTRIB CKOIGIT LENGTH=1 FORMAT=$CHAR01.
LABEL='CHECK DIGIT (KEY)';
/*ATTRIB FILLER1 LENGTH=1 FORMAT=$CHAR01.
 LABEL='FILLER 1';
/*
    ATTRIB CONTO LENGTH=3 INFORMAY=YYMMDD6.
FORMAT=YYMMDD;
LABEL='CONTRACT DATE YYMMDD';
INFILE D01;
INPUT
    0001 RECYPE $CHAR01. /*RECORD TYPE */
    0002 COMPNM $CHAR01. /*COMPANY NAME */
    0003 OFFNUN $CHAR01. /*OFFICE NUMBER (KEY) */
    00003 MERCHNUM $CHAR01. /*MERCHANT NUMBER (KEY) */

SAS Formats

SAS formats along with the PUT function can be used as one-way table lookups to transform data and eliminate multiple IF statements. This technique utilizes a binary lookup algorithm which is very efficient even for very large files. The PUT function is not understood by all programmers, thus a comment is certainly in order.

SAS Arrays

SAS has a very powerful array processor that can greatly shorten programs. The use of arrays, especially when they have multiple dimensions can be complex and imply much that is not shown in the program. Again, good commenting can help enormously.

Code for the Unexpected

One of the most common and easy mistakes to make, is to not allow for all possible data values that might be encountered. A recent project required summarized data at the sales region level, and so I asked for a complete list of valid regions, and was told regions were values 1 through 8. I never believe such a list without further checking. A simple PROC FREQ table of region showed values 1 through 8, but also a large number of zeros, and also values like 77 and 88. The zeros were the unassigned regions and 77 and 88 specified foreign business. Inevitably this type of data exists, and PROC FREQ can uncover it, before much coding takes place.

Another system runs monthly and all transactions should be dated in the prior month, but from time to time the wrong dataset was read into the system. A PROC FREQ table of DATE was inserted in the job, and if numbers don’t tie out, that table is the first thing checked.

Other SAS Products

The SAS Macro Processor

The macro processor certainly can generate huge amounts of SAS code very easily. This again can be
used or abused, because so much that is happening is implied. Macros certainly have their place, but coding them is more difficult than base SAS code, so care must be taken.

Some possible approaches might be:

Minimize Macros

There are some cases where coding macros greatly complicates the SAS job, and the may be other ways or options that yield the same result. The most common use of macros in my systems was to include by-variables and their values in titles and footnotes of graphs and reports produced with the SAS BY statement. For example producing a bar chart for multiple regions required a macro with a DATA step to create macro variables, a %DO loop, and appropriate WHERE statements. With SAS 6.07 the availability of the #byvar and #byval variables for titles and footnotes, this need completely disappears. We have found this simple addition to be one of the most valuable features of release 6.07.

Another very common use of macros was to generate dynamic PROC FORMAT value clauses. PROC FORMAT’s CNTLIN option allows input to come from a SAS dataset instead of though macro generation again.

Generated Code

With macros generating many lines of SAS code, the MPRINT, MLOGIC, and SYMBOLGEN options can be turned on and off to show the various levels of code and logic generated. We generally leave the MPRINT option on even when the job is moved to production. Our rationale is that the SAS log is a message dataset that is really needed when things go wrong, and including all generated code makes sense in most cases. Of course one of the attractions of invoking a macro is that with a single line many lines are generated. If the macros are debugged completely and the calls are documented leaving MPRINT off may be better.

AUTOCALL Macros

The macro autocall facility allows you to store macros in a location where invoking jobs can find them automatically. This is a great option for shared macros and if users know about the available macros they can take advantage of them with very little work. Documentation can let users know what is available.

Statement Style Macros

Statement style macros allow you to create what look like new SAS statements. It can be mind boggling to encounter a new statement in a SAS job that looks just like other SAS statements, but you have never seen it anywhere. We encountered a statement style macro that looked like this:

INTERLEAVE DS1 DS2;

It turned out to be a statement style macro that performed SET statements, but MPRINT was off and nothing else appeared in the log. I suppose that there are places where this makes sense, but they should be documented or avoided if not needed.

Compiled Macros

The same rules for keeping track of the source and compiled versions applies to macros much as it did above. We have not used compiled macros much at this time.

SAS/AF and SAS/FSP

SCL programs for SAS/AF and SAS/FSP have both source and the compiled versions stored in SAS catalogs. Some considerations might be

- establish a good module naming convention
- keep a printed version of the source programs filed safely away
- display module names somewhere on the screen
- display common function keys setting near the bottom of the screen.

SAS/GRAPH

The complexities of SAS/GRAPH offer many opportunities for simplification.

- storing device setting where many users can access them
- creating custom devices that are optimized to a shop and that allow quick switching to different devices by name alone

Summary

In summary, like most languages SAS programs are best when they are well planned, documented, and
maintained. There are however many implicit features of SAS that are not visible at first that are great for coding jobs quickly, but can cause problems if not understood or documented.

The author will be glad to answer questions and accept suggestions at:

Steven First  
Systems Seminar Consultants  
6014 Gateway Green  
Madison, WI 53716  
(608) 222-7081

SAS, SAS/CONNECT, SAS/AF and SAS/GRAPH are registered trademarks of SAS Institute Inc.