SOME GUIDELINES FOR DOCUMENTING SAS SOURCE CODE
Scott L. McGregor and Mary Nelson, Hewlett-Packard Company

Abstract

Programmers use a variety of techniques to document their programs. The styles, idioms and formats are frequently so diversified that it is often difficult to maintain programs developed by several different individuals. Additionally, good documentation practices can ensure good programming practices.

This paper presents some guidelines for documenting SAS programs for easy maintenance and comprehension. Guidelines are based on principles from diversified fields from systems design to graphics design. The guidelines discussed have been used by the author and several others at the Hewlett-Packard Corporate Data Center with great success.

Guidelines for Writing Easy to Read SAS Code and Documentation

This paper is designed to provide SAS programmers with a guide to writing easy-to-read SAS code. It covers how to format raw code, choose names, describe fields and provide additional documentation text. SAS’s step orientation lends itself to structured documentation practices, so be sure to take advantage of this feature. The guidelines described below can be easily extrapolated to other programming languages. In addition, the comments given here include some hints on machine efficiency, to demonstrate how easy to read code can peacefully coexist with efficient code.

Of course, these are guidelines, not commandments carved in silicon dioxide; they represent some of the experience and prejudices of the author. However, these guidelines all have good reasons behind them, and if you follow them closely they should serve you well. Many of these guidelines are rooted in time proven graphic design principles; ignore them if you wish, but be prepared to suffer the consequences!

Guidelines:

Use lots of white space. White space, the empty space around your text and code, should be your best friend. By using lots of white space you can escape from having your programs suffer from the “cluttered” look. A good program should sit on a page like an outline, not like a text book.

Specifically:

Use vertical white space to delineate separate DATA steps and PROC steps.

Use horizontal white space to separate keywords, variables and comments. If you have large sums to calculate, or any other large string of variables, be sure to set off the names by surrounding spaces. Consider putting one variable name on each line for PUT and INPUT statements as well as for complex sums or mathematical calculations. If you have complex boolean conditions (ANDs and ORs) consider separating logical conditions by placing each condition on a separate line. If you have a comment on the same line as some source code, be sure to separate it by some horizontal white space.

Use indenting. Whenever you are inside either a DO-END compound statement, or some other nested structure, indent each line in the loop. We find that indenting four spaces for each level of nesting works well for us. Also indent any portion of code called by a label, so that the label stands out clearly in the left margin. If you must continue a statement from a previous line, indent a little more; if the continuation is part of a list, start the second line under the first member of the list in the previous line, else indent four characters.

A helpful hint: Start DATA, PROC, TITLE and OPTIONS statements in column one. Start all other statements indented at least two characters. Some people also prefer to indent additionally for BY statements referring to MERGE, SET or UPDATE statements.

Make vertical lists: Keep matching DOs and ENDS, IF and THENs (and ELSEs), starting in the same vertical columns. Additionally, put lists of variables in vertical orientations so...
that all variable names start in the same column, all formats in the same column, etc. Use at least two spaces to separate elements on the same line so that SPF won’t realign the vertical list if you use the SPF change command.

One statement to a line. Never put more than one statement on a line. If you have a long line, continue on another line at a logical breaking point rather than compacting it in. Use the area to the right for adding pertinent comments about the line if necessary.

Use comments heavily. You should use two kinds of comments:

Indexing comments. This comment is used to delineate and describe DATA steps, PROC steps, and compound statements. Its purpose is to describe the task to be done at a relatively high level. It will help you and other programmers to isolate areas of the code which are causing problems or which need to be modified without it being necessary to wade through all the other code. We usually write indexing comments like this (separated from the code by blank lines):

/*--------This is an example of an indexing comment--------*/

If we have several DATA steps or PROC steps relating to the same high level task we usually separate the tasks like this:

/*========This is a major task index comment==========*/

Comments that contain long strings of asterisks, (*), tend to demand a large amount of eye contact, because they cause are mostly dark space and large dark patches attract the eye. This usually isn’t good for comments, unless they are for vital caveats which could lead to major errors. For this reason we use asterisks rather sparingly; we use them mainly in error messages where the extra attention is desirable.

Explanational comments. These are comments which explain the purpose of a statement. They can be put in the white area to the right of the statement involved. They should be mandatory in the case of most macros, links and labels. They should describe what the macro or labeled code does. Be sure to comment any special JCL requirements of the job at the beginning.

Note: When commenting avoid using a /* in the first two columns, as this could cause the computer to read a premature end of file.

Use features that are self-documenting and standardized. Order your modules in a logical manner: Code OPTIONS first, any macros second, TITLE statements next, and then DATA or PROC statements. Within DATA steps place declarations first, then procedural code, like this:

DATA dataset1 (KEEP= variable ...)
   dataset2 (KEEP= variable ...);
/* comment to describe the data step */
FILE statement;
INFILE, SET, MERGE or UPDATE statement;
BY statement;
FORMAT, LENGTH, and RETAIN statements;
Procedural code(input, output, logical, and action statements);

When you are selecting and sorting a dataset, select first, then sort. This will reduce the number of elements to be sorted and increase the efficiency of your program.

Identify yourself. Be sure to identify yourself in a comment at the front of the code, or near where you modified it. Take pride in your work.

Be explicit about defaults. Define things beyond question by specifying defaults explicitly. Where the logic is complicated, explicitly specify OUTPUT, DROPS and KEEPS. Define the starting locations and the formats of all INPUT and PUT variables explicitly. Use columnar specifications wherever possible. We find that descriptions such as:

INPUT @25 RATE 12 ;
are usually more readable, and more useful in checking against the actual file than are descriptions like this.

INPUT RATE 25-37 ;

Use boolean symbols sparingly. Use ORs and ANDs over "I" and "& II where possible. However avoid using too many NOTs. Double negatives can be confusing.

Choose meaningful and explicit variable names. Avoid using names like R or T where you could use RATE or TIME. Use the underscore to replace a space as in NEW_RATE. Choose names for your SAS data-sets which are suggestive of what the DATA step does. Choose names for your files which are suggestive of what the file contains, not just an alphanumeric identifier; that is choose MONTHLY over RM2000.

Don’t define unneeded variables. Don’t
mention variables that you won’t use.

Unlike COBOL, SAS has no need for fillers. In fact, use of unnecessary variables causes considerable overhead in SAS so your program will be more efficient if you don’t include them. It is a good idea to explicitly define what variables you wish to keep in DATA statements by using the KEEP option. This will help people who read your code know what you are keeping and still enable SAS to act more efficiently by minimizing its data storage and transfer. Try to handle and store as little data as possible; do this by only KEEPing variables you want and only OUTPUTing the observations you want.

INPUT only the data that you need to decide if you will use the observation. If you then need to retrieve some additional data, use the trailing @ to hold the record for a second INPUT. When using PROCs which create output datasets, specify only the variables that you will need, rather than getting all of the statistics. If you won’t need the data in a later step, then use DATA _NULL_; to prevent the creation of an unnecessary SAS temporary dataset.

Keep things simple. Modularize your code. Keep the number of statements in a DATA step or compound statement down to a minimum. Steps of more than 50 statements become hard to follow.

Write each program as if you were going to publish it. If you take the time to format and document your program for maximum readability you will be justly rewarded over and over again. The rewards for minimizing keystrokes are few so don’t do it.

Here are some examples:

Bad:

```
DATA ESCPU (KEEP=RMS_ALL) REGCPU (KEEP=RMS_ALL) TSOCPU (KEEP=RMS_ALL) TSOCONN
 (KEEP=RMS_ALL) SETUP (KEEP=RMS_ALL) CARDSIN (KEEP=RMS_ALL) REMOTIN (KEEP=RMS_ALL)
 DISKIO (KEEP=RMS_ALL) TAPEIO (KEEP=RMS_ALL); SET DAySMF.JOBS; RMS FRMT IF JOB TSO='J'
THEN DO; LINK ESCPU; LINK REGCPU; LINK SETUP; LINK CARDSIN; LINK REMOTIN; END;
ELSE DO; LINK TSOCPU; LINK TSOCONN; END; LINK DISKIO; LINK TAPEIO; RETURN;
```

Better:

```
/*-----------------------------------------*/
/* COPYRIGHT (C) 1982 BY SCOTT MCGREGOR, */
/* HEWLETT PACKARD, PALO ALTO, CA */
/*-----------------------------------------*/

/*==== SET UP BILLING INFORMATION BASED ON JOBS DATASET ========*/

DATA ESCPU (KEEP=RMS_ALL) REGCPU (KEEP=RMS_ALL)
TSOCPU (KEEP=RMS_ALL) TSOCONN (KEEP=RMS_ALL) SETUP (KEEP=RMS_ALL)
CARDSIN (KEEP=RMS_ALL) REMOTIN (KEEP=RMS_ALL)
DISKIO (KEEP=RMS_ALL) TAPEIO (KEEP=RMS_ALL);
SET DAySMF.JOBS;
RMS FRMT IF JOB TSO='J'
THEN DO; LINK ESCPU; LINK REGCPU; LINK SETUP; LINK CARDSIN; LINK REMOTIN;
END;
ELSE DO; LINK TSOCPU; LINK TSOCONN; END; LINK DISKIO; LINK TAPEIO; RETURN;
```

```/* WRITE OUT INDIVIDUAL */
/* DATASETS. */

DATA ESCPU (KEEP=RMS_ALL) REGCPU (KEEP=RMS_ALL)
TSOCPU (KEEP=RMS_ALL) TSOCONN (KEEP=RMS_ALL) SETUP (KEEP=RMS_ALL)
CARDSIN (KEEP=RMS_ALL) REMOTIN (KEEP=RMS_ALL)
DISKIO (KEEP=RMS_ALL) TAPEIO (KEEP=RMS_ALL);
SET DAySMF.JOBS;
RMS FRMT IF JOB TSO='J'
THEN DO; LINK ESCPU; LINK REGCPU;
LINK SETUP;
LINK CARDSIN;
LINK REMOTIN;
END;
ELSE DO;
LINK TSOCPU;
LINK TSOCONN;
END;
LINK DISKIO;
LINK TAPEIO;
RETURN;
```
Commentary: Don't ever use text flow to save space or for other machine efficiency reasons! Consider human efficiency instead. The second example illustrates several of the guidelines. Notice the generous use of white space, the indexing and explanatory comments, the use of indentation, and no more than one statement to a line. Depending on the sophistication of the intended reader, some of the explanatory comments may be unnecessary, as the variable names and dataset names were chosen to be self-explanatory. Notice also that the author is identified at the beginning. Lastly, note the modularity of the code as shown in the LINKS and IF-THEN-ELSE statements.

Bad:

```
DATA TEMP;
INFILE RM2000 END=E:
FILE RM3000;
IF N =1 & E=1 THEN X=3: RETAIN X;
INPUT FILLER $ 1-5 Y 6-10:
Z=SQRT(X*X+Y*Y);
PUT Z;
```

Better:

```
DATA _NULL_; /*----COMPUTE LENGTH OF HYPOTENUSE WITH ONE SIDE = 3 METERS----*/
INFILE VARSIDE END=EOF;
FILE HYPOSIDE;
IF N =1 AND NOT EOF THEN FIXDSIDE=3: /*SET FIXDSIDE AT START*/
RETAIN FIXDSIDE;
INPUT @6 VAR SIDE 5.;
HYPOTNSE= SQRT(FIXDSIDE * FIXDSIDE) + (VAR_SIDE * VAR_SIDE);
PUT HYPOTNSE;
```

Commentary: The second example illustrates several documentation principles as well as some efficiency considerations. Notice that more meaningful names were used in the second example. Also notice the use of explicit parentheses in the calculation. The columnar style input is also done and the unneeded filler is not read (this will increase SAS efficiency). Additionally, since the dataset TEMP is not used, it is replaced by _NULL_ which should also increase SAS's efficiency.

Conclusion:

This completes the discussion of SAS documentation guidelines. You'll probably find some other rules which will make your code easier to read. Be sure to use them and share them with others; more readable code benefits everyone, so be alert to the need to make your programs readable.