Paper 258-2012

Common Sense SAS® - Documenting and Structuring Your Code

Gary Pool, BNSF Railway, Ft. Worth, TX, USA

ABSTRACT

Job security does not result from being the only one who can fix your programs. It results from being the one who is most productive and able to pass projects on to someone else. The most frightening thing that confronts a SAS® programmer, whether amateur or professional, is to be presented with the challenge of debugging or updating someone else's code. Two simple things can make your code more valuable: effective documentation and good code structure.

This paper shows the importance of documentation and offers suggestions on how to make it effective. It demonstrates simple techniques for creating clear, easy to read, easy to debug, and easy to update code.

INTRODUCTION

Documentation is the first step in a project, not the last. The most efficient coders document the bulk of a system before the first line of code is written. A well-documented process provides a springboard for coding since each step is already defined. Inserting code between existing lines of documentation often is all that is required.

This paper will also discuss:

- Block style and brief style of documentation,
- The use of asterisks,
- Macro code and macro variable considerations, and
- Knowing when to quit.

Well-structured code is similar to an effective résumé. Most résumés have too many words and not enough white space. In like manner, most code does not have enough white space. Résumés traditionally follow an accepted template, making it easy for the reader to know what to expect. Good code will do the same.

PSEUDOCODE

By the time a project gets to a coder's desk, it has been through several steps, including conceptual development, requirements analysis, and system design. The documentation steps have already started, and you will continue those steps. As you put the requirements in your own words, you are creating pseudo code, or an English-like version of what the code will actually do. Here is an example of pseudo-code and the accompanying SAS code. Notice how the pseudo-code serves as the documentation.

Determine the number of stores in each country selling boots.
    Sort unique subsidiaries by region if they have boots.

List the store with the highest volume in each region.
    Summarize the sales by region and store

Sort the stores by descending sales
    Display only one observation for each region

From here, it should be clear what code needs to be implemented to achieve the desired results. More importantly, the next coder will easily understand the reason for the steps you've taken in your code.

```
/*** Determine the number of stores in each country selling boots. ***/
Proc means data=sashelp.shoes NWAY MISSING NOPRINT SUM;
      where product = 'Boot';
      class region subsidiary product;
      var   stores;
```

```
      output out=storesum (drop=_freq_ _type_) sum=;
      run;


/*** List the store with the highest volume in each region  ***/
Proc means data=sashelp.shoes nway noprint missing sum;
      class region subsidiary;
      var  sales;
      output out=salessum (drop = _freq_ _type_) sum=;
run;

/*** Sort the stores by descending sales   ***/
Proc sort data=salessum out=topsales;
      by  region descending sales;
run;

/*** Display only one observation for each region   ***/
Proc sort data=topsales out=topsalesbycountry nodupkey;
      by  region;
run;

Proc print uniform data=topsalesbycountry;
run;
```

The most efficient coders document the bulk of a system before the first line of code is written.

BLOCK STYLE AND BRIEF STYLE OF DOCUMENTATION

Block style documentation is used when introducing a system, a process, a section of code, or providing a lengthy explanation.

Every well documented program begins with a block style section of documentation. This large area of comments includes the name of the program, a high level description, the identity of the coder, a summary of the process completed in the code, the date of the project, a change log, reflecting the date and purpose of any changes to the code, and any other items your installation may require or value.

```
/*** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** ***/
/* PROGRAM:    B802550.TEST.EDREDO.PROGRAMS(ALLEVTS)
/* PURPOSE:    COMBINE THE DATA GATHERED FROM SEVERAL SOURCES,
/*             PREPARE FOR ANALYSYS AND REPORTING
/* PROGRAMMER: GARY D. POOL
/* PROCESS:    COMBINE ALL THE EVENTS - BE SURE TO FILTER EVENTS
/*             BASED ON THE START AND END DATES - KEEPING THE CURRENT
/*             YEAR AND THE PREVIOUS YEAR.
/*
/*             CREATE DATASETS FOR REPORTING:
/*                - PROJECT TO DATE
/*                - PROJECT YEAR TO DATE (FROM JANUARY 1ST)
/*
/* DATE:       MARCH 14, 2010
/*
/*    NOTE: THE PROJECTS GATHERED ARE FOR A FULL TWO YEAR PERIOD.
/*          EVENTS ARE BETWEEN THE FIRSTMVDATE AND LASTMVDATE.
/*          REPORTING IS YTD, SO PROJECTS WITH A LASTMVDATE PRIOR
/*          TO THE FIRST OF THE REPORTING YEAR CAN BE ELIMINATED FROM
/*          CONSIDERATION.
/*
/* CHANGE LOG:
/* Date        By   Purpose
/* 10/18/2011  GDP  CHANGE THE INCREMENTAL LOGIC TO TEST FOR REVENUE,
/*                  CONTRIBUTION AND UNITS.
/*** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** ***/
```

Program Introduction Documentation

A smaller block of documentation is used to explain a process or a particular step in you code. Sometimes you need to answer the question “Why did you do what you did?” In this example, a person new to this system would have to do the same research and “digging” as the original programmer to understand the code that follows if this block of documentation is not provided.

```
*** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** ***;
*** STATION AND STATE VARIABLES REPRESENT THE LOCATION THAT     ***;
***    ACCOUNTS FOR THIS RECORD BEING INCLUDED.  THE ORIGIN     ***;
***    OF THESE FIELDS VARY BASED ON THE EVENT.                 ***;
*** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** ***;
***                     STATION    STATE          PROJ633       ***;
***                     -------    -----          -------       ***;
***    AP AND RI         STN_333    STN_ST            N/A        ***;
***    RR EVENTS        ORIG_333   ORIG_ST          SHIPPER      ***;
***      OR (LATER)      STN_333    STN_ST          SHIPPER      ***;
***    INTERMODAL        STN_333    STN_ST         BENE_633      ***;
***    DD EVENTS    OP_CITY_333     OP_ST         OPCST633      ***;
***      OR (LATER)    DEST_333   DEST_333   CUST_633_CNSG      ***;
*** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** ***;
                    Block Program Documentation
```

Granted, the above example contains railroad jargon, however it is language familiar to railroaders, so no further explanation is necessary.

The brief style of documentation, which I often refer to as “snippets”, might also be called in-line commentary. This is documentation that shares the physical line with other code. For example:

```
Proc Printto Print=LP123456   /* Printer by 3rd floor mail room */
```

This helpful note will benefit future users of your code, and takes little time or space. Remember, the information you put in comments does not slow down your program or take any system resources to process. It is there for your benefit and the benefit of those who will follow in your footsteps.

THE USE OF ASTERISKS

There are two methods for inserting documentation in you code. You have seen both methods in the examples in this paper.

The first is the asterisk used to begin a comment. The comment continues until a semi-colon is found. The second method also includes asterisks, surrounded by the slash character (/* */). A common practice is to use a full line of asterisks to start a section of code or bring attention to an especially important comment. A convention we have agreed upon on my team is to use a series of three asterisks and a space to create that line.

```
*** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** ***;
```

or

```
/*** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** ***/
```

The biggest advantage of this method is to allow you the option of 4 asterisks to help you find a spot in your code. That spot may be a place where you have a “hard coded” variable or a location you need to find on regular basis. Simply searching for the string “****” gets you there in a hurry.

MACRO CODE AND MACRO VARIABLE CONSIDERATIONS

Macros are the blessing and curse of SAS code. When used well, they can save a great deal of time in coding. When used poorly, they can create a great deal of confusion. Proper documentation will help mitigate the challenges of macros.

First, a shortcut you may find helpful. Often, there are blocks of code that, for one reason or another, need to be skipped during the development process. Surround the code to be skipped with %MACRO skipme -- %MEND. When you want to run the block, simply include %skipme after the %MEND statement. When you do not want to run the code, put an asterisk in front of the %skipme statement.

You will want to have a standard for block documentation for macros. Here is a suggestion:

```
*** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** ***;
*** PROCESS ALL OF THE DAILY RECORDS                              ***;
*** MACRO NAME: FORDAY
*** MACRO IN:   SORTORD - Sort Order determined by Forecast Group
***             SEGMENT - Summary level
***             MEASURE - Variable being summarized
***             MTHNUM  - Month being considered
***             LABEL   - Descriptive results name
***             WHERE   - Where clause for filtering results
*** MACRO OUT:  CSV file appended to COMMA dataset
*** MACRO PROCESS: Summarize data based on input parameters
***                Append results to CSV file
*** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** ***;
                          Macro Documentation

%MACRO FORDAY(SORTORD, SEGMENT, MEASURE, MTHNUM, LABEL, WHERE);

PROC SUMMARY DATA=HISTDAY&MTHNUM NWAY MISSING;
    &WHERE;
    CLASS &SORTORD &SEGMENT DAYNUM SASDATE;
    VAR   &MEASURE ;
    OUTPUT OUT=HIST1(DROP=_TYPE_ _FREQ_) SUM=;
RUN;

PROC SORT DATA=HIST1 NODUPKEY OUT=HIST2;
    BY &SORTORD &SEGMENT DESCENDING &MEASURE ;
RUN;

DATA HIST2(KEEP=&SORTORD &SEGMENT DAYNUM &MEASURE);
    SET HIST2;

    FILE COMMA MOD;
    FORMAT SASDATE MMDDYY10.;
    BY  &SORTORD &SEGMENT DESCENDING &MEASURE;
    IF  FIRST.&SEGMENT THEN
        DO;
            PUT &LABEL "," SASDATE "," &SEGMENT "," &MEASURE;
            OUTPUT;
        END;
 RUN;
%MEND FORDAY;


%FORDAY (ORDERBU, BU, CLOSENET, 0, 'NET REV BU ALL', WHERE TOTAL = 'BNSF');
```

Any time you use macro variables, you will benefit from noting the definition of the variable. An old Chinese proverb says that “the faintest ink is more powerful than the strongest memory.” This adage has great applicability when discussing the best ways to document macro variables, especially when you use multiple ampersands. I do not profess to be an expert in &&&macro variables. I find them useful, but tend to forget how many amper’s are needed. The best way to remember is to write it down.

KNOWING WHEN TO QUIT

Documentation for documentation's sake is of no value whatsoever. Too much of a good thing is a bad thing. For example,

```
PROC PRINT DATA=SASHELP.SHOES   /*** Print the SASHELP Shoes data  ***/
```

While this example is pretty obvious, it is still quite commonly found. Just as not enough documentation makes code difficult to work with, too much low value documentation adds a "fog factor" that negates the value of even the good documentation in a program.

STRUCTURING SAS CODE

This is not a coding class, just a series of suggestions that have made my life as a SAS professional much easier – both for me and for others.

Structuring SAS code is a highly personal thing among many coders. It has little to do with your coding style. It has more to do with the use of white space.

Let me show you again an example of what is commonly referred to as "spaghetti code". This is code with no meaningful documentation. It also resembles a run-on sentence. It is difficult to follow and even more difficult to debug.

```
Data ethanolraw; set ethanolraw;
if act_date_nf = &rptday then do; units=mtdunits;grossrev=mtdgross;end;else
do;units=0;grossrev=0;end;run;
Proc means data=ethanolraw nway missing noprint sum;
by shipper consignee ultimate orig_333 orig_st carron onjct onjctst offjct
offjctst  carroff  dest_333 dest_st;
output out=ethanolsum sum=;run;
```

Honestly, this isn't spaghetti code in the true sense of the term. It is just unstructured and hard to follow. Here are a couple of examples of how I structure code.

Here is a Do/End followed by an If/Then/Else. Notice that the code lines up to make it easy to see where things start and stop.

```
DO;
     QTYWEEKS = 0;
     TOTUNITS = 0;
     SERVED   = 0;
END;


IF  BUS_GRP_CD = '02' THEN        /*** INDUSTRIAL PRODUCTS  ***/
    RTM = 474;
ELSE
IF  BUS_GRP_CD = '04' THEN        /*** AG COMMODITIES       ***/
    RTM = 474;
ELSE
IF  BUS_GRP_CD = '03' AND IML_EQP = 'C' THEN
    RTM = 312;                    /*** CP CONTAINERS        ***/
ELSE
IF  BUS_GRP_CD = '03' AND IML_EQP = 'T' THEN
    RTM = 212;                    /*** CP TRAILERS          ***/
ELSE
IF  BUS_GRP_CD = '03' THEN
    RTM = 132;                    /*** AUTOMOTIVE           ***/
ELSE
IF  BUS_GRP_CD = '01' THEN
    RTM = 812;                    /*** COAL                 ***/
ELSE …
```

Here, in a simple sort step, the variables in the BY are lined up, making them easy to read and identify. There is no reason to do this other than for readability, but what better reason could there be?

```
PROC SORT DATA=VENDORS NODUPKEY OUT=TRANLOAD.VENDLIST;
    BY STN_ST   STN_333 TRACK   FACID
       BULK     DIMEN   WAREHSE BNSFOWN
       RAILROAD PREMIER MEGAPOL VEND633 VENDNAME;
RUN; QUIT;
```

Finally, here is a DATA statement.

```
DATA ALLEVNTS;
    INFILE EVENTS;
    INPUT @01 WB_ID           $26.
          @27 XLD_STN         $09.
          @36 XLD_ST          $02.
          @38 TRK_NUMB        $04.
          @42 CAR_INIT        $04.
          @46 CAR_NUMB        $10.
          @56 EVT_CD          $02.
          @58 EVST_CD         $02.
          @60 EVT_DT_C        $10.
          @70 OPCST633        $12.
          @82 CAR_KIND        $04.
          ;
```

```
DATA ALLEVNTS; INFILE EVENTS;
INPUT @01 WB_ID $26. @27 XLD_STN $09. @36 XLD_ST $02. @38 TRK_NUMB $04.
@42 CAR_INIT $04. @46 CAR_NUMB $10. @56 EVT_CD $02. @58 EVST_CD $02.
@60 EVT_DT_C $10. @70 OPCST633 $12. @82 CAR_KIND $04.;
```

Yes, both samples are correct and operate properly. If you were to inherit a program, with which one would you prefer to work?

Enterprise Guide can be used as a tool to structure code. It is designed to take that spaghetti code we've seen and turn it into structured code.

Here's our poor code:

```
Data ethanolraw; set ethanolraw;
if act_date_nf = &rptday then do; units=mtdunits;grossrev=mtdgross;end;else
do;units=0;grossrev=0;end;run;
Proc means data=ethanolraw nway missing noprint sum;
by shipper consignee ultimate orig_333 orig_st carron onjct onjctst offjct
offjctst  carroff  dest_333 dest_st;
output out=ethanolsum sum=;run;
```

Here is what SAS© Enterprise Guide does to it:

```
Data ethanolraw;
       set ethanolraw;

       if act_date_nf = &rptday then
              do;
                     units=mtdunits;
                     grossrev=mtdgross;
              end;
       else
              do;
                     units=0;
                     grossrev=0;
              end;
run;
```

```
Proc means data=ethanolraw nway missing noprint sum;
       by shipper consignee ultimate orig_333 orig_st carron
          onjct onjctst offjct  offjctst  carroff  dest_333 dest_st;
       output out=ethanolsum sum=;
run;
```

If SAS thinks it is important enough to provide this functionality, it should be clear that we, as professionals, create code that does not need such a tool!

IN CONCLUSION…

Documentation and structure are often ignored. You have the opportunity to create value by writing code that is easy to read and understand. In doing so, you can easily turn projects over to other people and move on to bigger and better challenges. Unstructured, poorly documented code will become a burden to you and your team. Do you want to be known for creating value or producing a burden? The popular author, John Maxwell says, “If you don’t have time to do it right the first time, when will you find time to do it over again?”

Do yourself a favor…the first time.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Name:	Gary Pool
Enterprise:	BNSF Railway
Address:	2650 Lou Menk Drive
City, State ZIP:	Ft. Worth, TX 76131
Work Phone:	817-352-6308
Fax:	817-352-7374
E-mail:	gary.pool@bnsf.com
Web:	www.bnsf.com