LOW-MAINTENANCE PROGRAMMING TECHNIQUES
by Robert Virgile
Independent Trainer and Consultant

Abstract

Elements of programming style and technique affect the amount of effort required to maintain programs. In many cases, the "necessity" to maintain programs is a direct result of the tools and techniques used when writing the original program.

An ultimate goal would be zero maintenance: a program that can be submitted at any time, with no changes to the code, for any set of input data. In practice, this approach demands a broad range of skill and knowledge using SAS software, creativity, and a willingness to put in extra programming effort up front. The final program should require little or no attention on the part of the programmer.

This paper investigates some of the techniques that reduce maintenance, gives examples of relevant syntax, and applies the techniques to sample programming problems.

Where Does the Time Go?

A few categories stand out as major time gluttons. For example:

**Figuring out what a program does.** How many times have you had to modify a program that was 200 lines long but contained no comment statements, not a single line of documentation?

**Writing new programs for every application.** This reinvention of the wheel can multiply the time required to write and maintain programs.

**Regularly making small, standard changes.** Such as modifying a range of dates used to subset observations. Often this takes the form of hard-coding similar changes in multiple sections of a program.

**Using personal effort to process data.** The computer, not the individual, should process data. In many instances, we don't allow the computer to perform as much of the work as we should.

Let's take a closer look at these categories, and see what we can do to reduce maintenance efforts.

What Does the Program Do?

How would you like to modify a 250-line program that began like this:

```sas
DATA D (KEEP=DISP IRNUM MAT_COST HRS);
SET INPUT16.ACTIVE; DATE=TODAY();
THSMONTH=INTNX('MONTH',DATE,0);
LSTMONTH=INTNX('MONTH',DATE,-1);
IF DAT_CLOS=. THEN DELETE;
IF DAT_CLOS < LSTMONTH OR DAT_CLOS >= THSMONTH THEN DELETE;
DISP=D_DISP_C; MAT_COST=M_COST;
HRS=M_HRS+PA_HRS+ENGRHS;
IF HRS=. THEN HRS=1;
PROC SORT DATA=D; BY IRNUM;
DATA D2; SET D; BY IRNUM; IF FIRST IRNUM;
DATA I (KEEP=DISP IRNUM MAT_COST HRS);
SET INPUT17.IRS; DATE=TODAY();
THSMONTH=INTNX('MONTH',DATE,0);
LSTMONTH=INTNX('MONTH',DATE,-1);
IF D_CLOS=. THEN DELETE;
IF D_CLOS < LSTMONTH OR D_CLOS >= THSMONTH THEN DELETE;
DISP=DSPI;
HRS=M_HRS+E_HRS+O_HRS+O_HRS;
IF HRS=. THEN HRS=1;
```

For many reasons, this program is difficult to decipher. Contributing factors include:

1. No comment statements
2. Poor spacing/indentation
3. Unrevealing names for file references
4. Different variable names representing Date Closed.

Although this list could be extended, one point is already clear. The easier you make it to figure out what your program does, the easier it will be to maintain that program. Ingrain appropriate techniques as a matter of habit. Comment statements in the source code are basic. However, there are many other ways to document what a program does. Consider these steps:

1. Use meaningful names for datasets, variables, and file references. Date of birth, for example, could be stored under a variable name like DOB, D_BIRTH, or BIRTHDAY. Quick and dirty statements like DATA D; or INPUT V ARI; make programs more difficult to comprehend. If the same variable exists in more than one dataset, use the same variable name to represent that quantity across all datasets.

2. Label your variables. Remember, you can use PROC CONTENTS to easily review all variable names and labels. If appropriate, label your datasets as well. For a large system with many SAS datasets, consider reserving a separate dataset for documentation purposes. This dataset could contain one observation for each of the datasets in the system, with text data values that describe the contents of each dataset. Similarly, one member of a PDS might document the purpose of each of the other members.

*SAS is a registered trademark of SAS Institute Inc., Cary, NC 27511-8000.*
3. Stick to coding conventions, such as indentation and spacing, to make programs readable. Some reasonable conventions might include indenting all statements in a DO group, or beginning all statements on a new line. Whatever conventions you select, stick to them. One good set of conventions can be found in an article entitled "Improving Programmer Productivity through Coding Conventions" from the 1984 SUGI proceedings.

4. Specify data set names on all DATA and PROC statements. Make it easy to see which dataset is being processed in each SAS step.

5. Broaden your knowledge of SAS software to ensure that you are using the best tools for the job at hand. The sample code above uses assignment statements for DISP and MAT_COST where RENAME would have accomplished the same result using less CPU time. This adds to the maintenance work, because the programmer now must figure out whether there is a valid reason for using assignment statements.

6. Modularize code within a program. This does not necessarily involve MACRO language or %INCLUDE. Within reasonable limits (such as avoiding "spaghetti code"), two statements within a DATA step should not perform the same function. Consider these statements taken from a customized report:

```sas
IF SEX='M' THEN PUT @10 'MALE' @21 AGE 3. @30 WEIGHT 3. @40 HEIGHT 2. @48 SMOKER $3. @56 DRINKER $3. @64 HELTHNUT $3.;
ELSE IF SEX='F' THEN PUT @10 'FEMALE' @40 HEIGHT 2. @48 SMOKER $3. @56 DRINKER $3. @64 HELTHNUT $3.;
```

Since the second halves of both statements are identical, the program should be modified to read:

```sas
IF SEX='M' THEN PUT @10 'MALE'
   @21 AGE 3. @30 WEIGHT 3.
   @40 HEIGHT 2. @48 SMOKER $3.
   @56 DRINKER $3. @64 HELTHNUT $3.;
ELSE IF SEX='F' THEN PUT @10 'FEMALE'
   @40 HEIGHT 2. @48 SMOKER $3.
   @56 DRINKER $3. @64 HELTHNUT $3.;
```

A New Program Every Time?

Many times, different programs perform similar functions, functions that are similar enough to warrant "sharing" sections of code. A simple way to save time is to cannibalize one program to produce another -- copy the original and modify it as necessary for the new application. While this saves programming time, it still doesn't reduce maintenance time. If future requirements call for modifications to be made to one program, chances are all programs will need to be changed.

Sophisticated techniques save on maintenance time by storing separately the shared sections of code and permitting each program to call upon those stored sections. This approach may include:

1. Storing source code in a separate partitioned data set and %INCLUDEing the code as needed.

2. Storing MACRO code in a separate sequential or partitioned dataset and %INCLUDEing all or some of it as needed.

3. Creating permanent format libraries that incorporate programming functions.

All three techniques permit changes to be made in one location which will affect many programs. The first two methods are well known and are covered extensively in existing literature. The third method, format libraries, is not covered so thoroughly. A straightforward application would call for modification of a commonly used format. For example, the RGN. format below might be referenced in many programs:

```sas
PROC FORMAT LIBRARY=SASLIB;
VALUE RGN
   1='NORTH'
   2='SOUTH'
   3='EAST'
   4='WEST';
```

By storing the format permanently, it can easily be referenced or modified. A simple PROC PRINT could refer to the format, as could many other programs:

```sas
PROC PRINT DATA=CITIES;
VAR CITY STATE REGION;
FORMAT REGION RGN.;
```

If the RGN. format needed to be changed, the change would only have to be made once, and all programs using the format (such as the previous PROC PRINT) would automatically be updated:

```sas
PROC FORMAT LIBRARY=SASLIB;
VALUE RGN
   1='NORTH'
   2='SOUTH'
   3='EAST'
   4='WEST'
   5='NORTHEAST'
   6='NORTHWEST'
   7='SOUTHEAST'
   8='SOUTHWEST';
```

More complex applications could use a format library to embed programming logic common to many programs. In the following example, many programs share a common objective of computing the number of working days between two date variables. As initial input, there exists a known list of all holidays from 1987 through 1996. The holidays are stored in a sequential file, one holiday to a line, in MMDDYY form. Assume 8 holidays per year, 80 in all. A portion of the file might read:

```
01/01/87
02/16/87
04/25/87
07/03/87
```

A New Program Every Time?
Armed with a list of holidays, plus the SAS software's ability to distinguish weekends from weekdays, we possess all the tools needed to compute working days. A sample dataset might contain two date variables, START and FINISH. Computing the difference in calendar days is easy:

\[
\text{CALENDAR} = \text{FINISH} - \text{START};
\]

Even without a format, existing tools permit computation of working days. For example:

```
DATA HOLIDAYS;
  INFILE RAWHOLS;
  INFORMAT HOLIDAY MMDDYY8.;
  INPUT HOLIDAY MMDDYY8.;
```

Structure of the SAS dataset HOLIDAYS:

- One observation
- 80 variables named HOLIDAY
- All variables are unformatted SAS dates

```
DATA TASKS;
  SET TASKS;
  IF _N_=1 THEN SET HOLIDAYS;
  WORKDAYS=-1;
  DO DUMMY = START TO FINISH;
    IF Weekday(DUMMY) < 7 THEN WORKDAYS = WORKDAYS + 1;
  END;
  ARRAY HOLS (_1_) HOLIDAY MMDDYY8;
  DO _I_=1 TO 80;
    WORKDAYS = WORKDAYS -(START <= HOLS < FINISH);
  END;
```

The variable WORKDAYS contains the number of working days between the START and FINISH variables.

Some notes about the syntax:

1. The WEEKDAY function returns a value of 1 for Sunday and 7 for Saturday.
2. The expression \((\text{START} <= \text{HOLS} < \text{FINISH})\) returns a value of either 0 for false or 1 for true, and thus can be used in an arithmetic computation. (Specifying \(< \text{FINISH}\) instead of \(\leq \text{FINISH}\) is arbitrary and is done here to make this solution equivalent to a later solution.)
3. We have assumed that all values for \text{START} and \text{FINISH} fall within the ten-year range. (Other approaches that follow also make this assumption.)
4. Tasks are scheduled to begin and end at the same time of day. Therefore, \text{START} should never equal \text{FINISH}.

This program represents a lot of processing as well as a fair number of lines of code that must be written and maintained in numerous application programs. A low maintenance approach would create a format to transform dates from a scale of calendar days to a scale of working days. The scales:

<table>
<thead>
<tr>
<th>Calendar</th>
<th>Working</th>
</tr>
</thead>
<tbody>
<tr>
<td>C + 0</td>
<td>W + 0</td>
</tr>
<tr>
<td>C + 1</td>
<td>W + 1</td>
</tr>
<tr>
<td>C + 2</td>
<td>W + 2</td>
</tr>
<tr>
<td>C + 3</td>
<td>W + 2</td>
</tr>
<tr>
<td>C + 4</td>
<td>W + 3</td>
</tr>
<tr>
<td>C + 5</td>
<td>W + 3</td>
</tr>
<tr>
<td>C + 6</td>
<td>W + 3</td>
</tr>
<tr>
<td>C + 7</td>
<td>W + 4</td>
</tr>
</tbody>
</table>

On the working day scale, each day is assigned an integer value. From one working day to the next, this integer value increases by 1. All dates in the relevant range (from 1/1/87 to 12/31/96) will be assigned a working day value. Non-working days will be assigned the same value as the next working day. For example, Saturday and Sunday will be assigned the same working day value as the following Monday.

The objective is to compute working days as:

\[
\text{WORKDAYS} = \text{FINISH} \text{ (transformed from the calendar day scale to a working day scale)} - \text{START} \text{ (also transformed from the calendar day scale to a working day scale)}
\]

All the needed computations would be performed once, and the results stored in a permanent format library. Any programs that need to compute working days would convert from calendar days to working days by referring to the format. Computation of calendar days from \text{START} to \text{FINISH} would call for:

\[
\text{CALENDAR} = \text{FINISH} - \text{START};
\]

Similarly, computation of working days would call for:

\[
\text{WORKDAYS} = \text{FINISH} \text{ (transformed from the working day scale to the calendar day scale)} - \text{START} \text{ (also transformed from the working day scale to the calendar day scale)}
\]

The original tools are sufficient for creating the working format -- SAS software plus a sequential file containing all 80 holidays. One workable approach to creating the format is:

```
DATA _NULL_;
  INFILE RAWHOLS END=EOF;
  INPUT HOLIDAY MMDDYY8.;
  FILE VALUE1 NOTITLES;
  IF _N_=1 THEN PUT 'VALUE HOLIDAYS';
  PUT HOLIDAY = HOLIDAY;
  IF EOF THEN PUT;
PROC FORMAT LIBRARY=SASLIB;
%INCLUDE VALUE1;
```

This program represents a lot of processing as well as a fair number of lines of code that must be written and maintained in numerous application programs. A low maintenance approach would create a format to transform dates from a scale of calendar days to a scale of working days. The scales:
DATA _NULL_;  
FILE VALUE2 NOTITLES;  
WORKDAY=9999;  
PUT 'VALUE WORKING';  
DO DATE = '31DEC96'D TO '01JAN97'D BY -1;  
   IF (1 < WEEKDAY(DATE) < 7) AND  
       (PUT(DATE,HOLIDAYS.) NE 'HOLIDAY')  
       THEN WORKDAY = WORKDAY - 1;  
   PUT DATE ' = I I I WORKDAY 4.';I I I  
END;  
PUT ' ';  
PROC FORMAT LIBRARY=SASLIB;  
%INCLUDE VALUE2;  
This four-step program passes the data along until  
producing the final WORKING format. The first  
DATA step creates a VALUE statement which identifies  
holidays. Next, PROC FORMAT executes that VALUE  
statement. The second DATA step examines each day.  
determines whether it is a weekday or not (using the  
WEEKDAY function), determines whether it is a holiday  
or not (using the format just produced in the prior  
step), numbers the day accordingly, and writes the  
results to a second VALUE statement. Finally, PROC  
FORMAT executes the second VALUE statement to create  
the permanent WORKING format.

Some additional notes:
1. This program arbitrarily sets December 31, 1996  
equal to 9998 on the working day scale analogously to  
the way January 1, 1960 is set to 0 on the calendar  
day scale.
2. The statement  
   PUT DATE ' = I I I WORKDAY 4.';I I I  
writes one line of a VALUE statement using four  
arguments. For the first date (December 31, 1996),  
the arguments are:
   DATE 13514  
   WORKDAY 9998  
which translates to  
   13514 = '9998'  
3. Moving backwards through the dates (BY -1 in the  
   DO loop) makes it easy to equate each non-workday  
   with the next working day. This means that a task  
   scheduled to begin on Sunday will not actually begin  
   until Monday, while a task scheduled to end on Sunday  
   will not be finished by Friday, but will take till  
   Monday.
4. The final VALUE statement is thousands of lines  
   long and therefore requires considerable CPU time and  
   region size to execute. Resource constraints may  
   necessitate splitting this program into two, each  
   containing a DATA step and a PROC FORMAT.

Reducing Small, Routine Changes
An initial extra effort can reduce or eliminate the  
need to make small, routine changes to programs.  
MACRO language may come into play here, but even  
without MACRO language, good programming techniques  
can reduce changes. Consider this report on the total  
cost of fruit purchases:

TOTAL COST OF ALL FRUIT PURCHASES
   RATES: 0.55 PER APPLE  
         0.40 PER ORANGE  
         0.18 PER BANANA

TOTAL FRUIT EXPENDITURES:
   APPLES: 42.90  
   ORANGES: 36.10  
   BANANAS: 21.78

This report is based on a SAS dataset containing two  
variables: FRUIT (type of fruit purchased) and  
QUANTITY (number of pieces purchased). There  
may be many observations for each value of FRUIT.  
The report makes use of cost per fruit in two  
sections: (1) printing headers, and (2) computing  
total expenditures. The following high maintenance  
program hard codes cost in both sections of the  
program:

DATA _NULL_;  
SET FRUITS END=EOF;  
IF FRUIT='APPLE' THEN  
   Q_APPLE + QUANTITY;  
ELSE IF FRUIT='ORANGE' THEN  
   Q_ORANGE + QUANTITY;  
ELSE IF FRUIT='BANANA' THEN  
   Q_BANANA + QUANTITY;  
IF EOF THEN DO;  
   FILE PRINT NOTITLES;  
   C_APPLE = Q_APPLE * .55;  
   C_ORANGE = Q_ORANGE * .40;  
   C_BANANA = Q_BANANA * .18;  
   PUT 'TOTAL COST OF ALL FRUIT PURCHASES';  
   / @20 \  
   'RATES: ' @33  
   / @33 \  
   '0.55 PER APPLE' /  
   / @33 \  
   '0.40 PER ORANGE' /  
   / @33 \  
   '0.18 PER BANANA' /  
   'TOTAL FRUIT EXPENDITURES:' /  
   / @20 \  
   'APPLES:' / @30 C_APPLE 6.2  
   / @20 \  
   'ORANGES:' / @30 C_ORANGE 6.2  
   / @20 \  
   'BANANAS:' / @30 C_BANANA 6.2;  
END;

This program would have to be modified any time the  
price of fruit changes. With very few modifications,  
we can localize and reduce the changes that must take  
place, as in the next program.
DATA _NULL_;  
SET FRUITS END=EOF;  
RETAIN R_APPLE .55 R_ORANGE .40 R_BANANA .18;  
IF FRUIT='APPLE' THEN Q_APPLE + QUANTITY;  
ELSE IF FRUIT='ORANGE' THEN Q_ORANGE + QUANTITY;  
ELSE IF FRUIT='BANANA' THEN Q_BANANA + QUANTITY;  
IF EOF THEN DO;  
FILE PRINT NOTITLES;  
C_APPLE Q_APPLE * R_APPLE;  
C_ORANGE = Q_ORANGE * R_ORANGE;  
C_BANANA = Q_BANANA * R_BANANA;  
PUT '/ ' @20  'TOTAL COST OF ALL FRUIT PURCHASES' /// @25 'RATES:' /// @33 R_APPLE Z4.2 ' PER APPLE' /// @33 R_ORANGE Z4.2 ' PER ORANGE' /// @20 'TOTAL FRUIT EXPENDITURES:' /// @20 'APPLES:' @30 C_APPLE 6.2 /// @20 'ORANGES:' @30 C_ORANGE 6.2 /// @20 'BANANAS:' @30 C_BANANA 6.2;  
END;  

Now, any changes in the RETAIN statement will automatically be passed through to both relevant sections of the program.

When working with date variables, it is often possible to eliminate maintenance entirely. A common task is subsetting observations based on date. One form of this task involves printing all data from the previous month. For example, during February 1988 we might code:

DATA SUBSET;  
SET ALL;  
IF '01JAN88'D <= DATE < '01FEB88'D;  

During March 1988, we might modify this code to read:

DATA SUBSET;  
SET ALL;  
IF '01FEB88'D <= DATE < '01MAR88'D;  

The INTNX function can eliminate entirely such hard-coding of dates into the program. The only requirement would be that the program be run during the proper month.

DATA SUBSET;  
SET ALL;  
IF _N_=1 THEN DO;  
REFDATE=TODAY();  
START=INTNX('MONTH',REFDATE,-1);  
FINISH=INTNX('MONTH',REFDATE,0);  
END;  
RETAIN START FINISH REFDATE;  
IF START <= DATE < FINISH;

So long as this program is run during March 1988, it assigns START the value of February 1, 1988 and FINISH the value of March 1, 1988. Program maintenance can be eliminated in exchange for an obligation to run the program before the end of the next month. In addition, if the report had to be backdated, only one statement (REFDATE=) would need to be changed.

Personal Effort vs. Computer Effort

The programmer instructs the computer; the computer processes the data. By failing to instruct properly, the programmer may spend a good deal of time personally processing data. Here are some basic examples.

A SAS dataset with credit card information contains four variables:

CARD: Credit Card Number ($13)  
DATE: Date of Purchase (numeric)  
COST: Cost of Purchase (numeric)  
ITEM: Item Purchased ($20)

Since each observation represents a purchase, there will be many observations with the same value for CARD.

Various programming objectives can be accomplished efficiently (by letting the computer process the data) or inefficiently (by personally processing the data).

Objective #1: Identify all purchases made during 1986 where the amount was more than $500.

Time-Inefficient:

DATA PURCHASE;  
SET PURCHASE;  
IF YEARDATE)=1986 THEN FLAG1='*';  
IF COST> 500 THEN FLAG2='*';  
PROC PRINT DATA=PURCHASE;  
VAR CARD DATE FLAG1 COST FLAG2 ITEM;  
FORMAT DATE DATE9.;

Time-Efficient:

DATA COSTLY;  
SET PURCHASE;  
IF YEARDATE)=1986;  
IF COST> 500;  
PROC PRINT DATA=COSTLY;  
VAR CARD DATE AMOUNT ITEM;  
FORMAT DATE DATE9.;

TITLE '1986 PURCHASES OVER $500';

Objective #2: Print a report for all the data, with a title that reads:

PURCHASES FOR ALL CREDIT CARDS

where the blank gets filled in with the total number of credit cards in the dataset. Sort by credit card number, and print the total cost of purchases for each credit card.

1246
Time-Inefficient:

PROC FREQ DATA=PURCHASE;
TABLES CARD;

(Programmer counts the output lines and replaces the blanks in the title of the second program.)

PROC SORT DATA=PURCHASE;
BY CARD;

PROC PRINT DATA=PURCHASE;
VAR COST ITEM DATE;
BY CARD;
SUM COST;
SUMBY CARD;
TITLE 'PURCHASES FOR ALL CREDIT CARDS';
FORMAT DATE MMDDYY8.;

Time-Efficient:

PROC SORT DATA=PURCHASE;
BY CARD;

DATA _NULL_;  
SET PURCHASE END=EOF;
BY CARD;
IF LAST.CARD THEN COUNT + 1;
IF EOF THEN DO;
FILE TEMP NOTITLES;
PUT 'TITLE "PURCHASES FOR ALL CREDIT CARDS"';
END;

PROC PRINT DATA=PURCHASE;
VAR COST ITEM DATE;
BY CARD;
SUM COST;
SUMBY CARD;
%INCLUDE TEMP;
FORMAT DATE MMDDYY8.;

Now the entire result is achieved by running one program instead of two, with no intermediate work on the part of the programmer.

Putting It All Together

This final example builds a program that uses many of the previous techniques. The objective is to plot the quality of 100 different toys produced by a toy factory. The data are already stored in a SAS dataset containing three variables:

TOY: Toy ($8) takes on a unique value for each type of toy manufactured (sample values: DOLL, TRAIN, BALL). Hundreds or thousands of each toy are manufactured every month. There are 100 unique types of toy.

DATE: Date (numeric) on which the toy was tested. Every toy is tested before it is shipped.

RESULT: Result ($1) of testing the toy. If the toy passes inspection, RESULT will be 'S' (success). If the toy fails inspection, RESULT will be 'F' (failure).

The program(s) should generate a graph for each type of toy produced (i.e., each value of TOY), 100 graphs in total. Graphs should illustrate the defect rate over time. Some sample graphs are illustrated below.

The specifications include:

1. Using PROC PLOT, produce a plot for each toy.

2. Plot failure rate by month. Failure rate is the total number of failures for the month (RESULT='F') divided by the total number of tests for the month.

3. The X-axis: Include the most recent full 12 months only. For example, if the program is run on any day during February 1988, include February 1987 through January 1988.

4. The Y-axis: Select one of two scales, based on the data. Most of the time, the monthly failure rate will remain below 20%. The y-axis should then be scaled 0 to 0.2 by 0.02. Occasionally the failure rate rises above 20%. In that case, the y-axis should be scaled 0 to 1 by 0.1. The y-axis may be different for different toys.

High-maintenance programs might solve this problem in steps. The programmer modifies an initial program to compute and print the failure rates for the proper time period. Next, the programmer inspects the printout to locate any failure rates over 20%. Finally, the programmer modifies and runs a second program that plots the test results. (Final modifications concern which toys should be plotted on which scale for the y-axis.) The programs below illustrate this approach.

Program 1:

DATA ONEYEAR;
SET TEST.RESULTS;
IF '01FEB87'D <= DATE <= '31JAN88'D;
DATE=INTNX('MONTH', DATE, 0);

PROC SORT DATA=ONEYEAR;
BY TOY DATE;

DATA TEST.RATES;
SET ONEYEAR;
BY TOY DATE;
IF FIRST.DATE THEN DO;
TESTS = 0;
FAILURES = 0;
END;

PROC PRINT DATA=TEST.RATES;
BY TOY DATE;

The dataset TEST.RATES contains one observation per toy per month, including a variable that represents failure rate for that toy-month combination. If inspection of the PROC PRINT output revealed that three toys (AIRPLANE, GIRAFFE, and FUN FOAM) exhibited a failure rate over 20%, the second program would be:

```
DATA TINY BIG;
SET TEST.RATES;
IF TOY='AIRPLANE' OR TOY='GIRAFFE' OR TOY='FUN FOAM' THEN OUTPUT BIG;
ELSE OUTPUT TINY;
PROC PLOT DATA=TINY;
BY TOY;
PLOT FAILRATE*DATE='*' / HAXIS = '01FEB87'D TO '01JAN88'D BY MONTH VAXIS=0 TO 0.2 BY 0.02;
FORMAT DATE MONYY5.;
PROC PLOT DATA=BIG;
BY TOY;
PLOT FAILRATE*DATE='*' / HAXIS = '01FEB87'D TO '01JAN88'D BY MONTH VAXIS=0 TO 1 BY .1;
FORMAT DATE MON YY5.;
```

Other formatting statements such as titles, variable labels, and the NOLEGEND option might be included in the actual programs. However, those statements are not related to the issue of program maintenance.

These programs have one thing going for them—they work. However, on a month to month basis, the amount of maintenance becomes excessive. Each month, the programmer must:

1. Select the proper observations by date (modify line 3 of the first program).
2. Inspect the printout from the first program to discover any failure rates over 20%.
3. Based on the results of the inspection, modify the first DATA step of the second program.
4. Change the dates on the x-axis of both PLOT statements in the second program.

All of these tasks can be handled by the computer, if the programmer employs the techniques outlined earlier.

The Failure Rate

Rather than search by eye for failure rates over 20%, the programmer should spend time reprogramming in a way that will allow the computer to do the work. One simple modification would be to print only those failure rates over 20%:

```
IF FAILRATE > .2 THEN PUT TOY= ;
```

This statement could be added to the first program, eliminating entirely the PROC PRINT. Once we begin to let the computer select failure rates over 20%, we can build along those lines and combine the original two programs into one:

```
DATA ONEYEAR;
SET TEST.RATES;
IF '01FEB87'D <= DATE <= '31JAN88'D;
DATE = INTNX('MONTH',DATE,0);
PROC SORT DATA=ONEYEAR;
BY TOY DATE;
DATA PLOTME (DROP=MAXFAIL)
SIZE (KEEP=MAXFAIL TOY);
SET ONEYEAR;
BY TOY DATE;
IF FIRST.TOY THEN MAXFAIL=0;
RETAIN MAXFAIL;
IF FIRST.DATE THEN DO;
TESTS = 0;
FAILURES = 0;
END;
TESTS + 1;
IF RESULT = 'F' THEN FAILURES + 1;
IF LAST.DATE THEN DO;
FAILRATE = FAILURES / TESTS;
OUTPUT PLOTME;
MAXFAIL = MAX(MAXFAIL,FAILRATE);
END;
IF LAST.TOY THEN OUTPUT SIZE;
DATA BIG TINY;
MERGE PLOTME SIZE;
BY TOY;
IF MAXFAIL > .2 THEN OUTPUT BIG;
ELSE OUTPUT TINY;
PROC PLOT DATA=TINY;
BY TOY;
PLOT FAILRATE*DATE='*' / HAXIS = '01FEB87'D TO '01JAN88'D BY MONTH VAXIS=0 TO .2 BY .02;
FORMAT DATE MONYY5.;
PROC PLOT DATA=BIG;
BY TOY;
PLOT FAILRATE*DATE='*' / HAXIS = '01FEB87'D TO '01JAN88'D BY MONTH VAXIS=0 TO 1 BY .1;
FORMAT DATE MONYY5.;
```

Notes about the syntax:

1. The dataset PLOTME contains failure rate information, with one observation per toy per month. These observations will be displayed using PROC PLOT. In prior programs, PLOTME was stored as a permanent dataset. Since only one program is now necessary, PLOTME becomes a temporary dataset.
2. The dataset SIZE contains one observation per toy. The key variable here is MAXFAIL, the maximum failure rate encountered during the past 12 months. This variable determines which y-axis should be used for each toy.
3. By merging PLOTME and SIZE, the program segregates plottable data into two datasets, BIG and TINY. BIG will be plotted on the larger y-axis (0 to 1 by .1) while TINY will be plotted on the smaller y-axis (0 to .2 by .02).
Selecting Observations
The program must still select the proper observations based on date. With the extensive capabilities built into SAS software, it ought to be possible to let the computer perform date-related maintenance tasks. Selection by date is relatively simple:

```
DATA ONEYEAR;
SET TEST.RESULTS;
IF _N_=1 THEN DO;
   RENEQ = TODAY();
   START = INTNX('MONTH', RENEQ, -12);
   FINISH = INTNX('MONTH', RENEQ, -1);
END;
RETAIN START FINISH;
IF START <= DATE <= FINISH;
   TEND = INTNX('MONTH', DATE, 0);
END;
```

Now, as long as the program is run during the month of February 1988, it will automatically select records dated from February 1987 through January 1988. Presumably, it takes little or no effort on the programmer's part to ensure that the program is run during the correct month.

One additional section of the program requires date-related maintenance: specification of the x-axis. Obviously, all the information required to specify the x-axis is available as early as the first DATA step above. The question is how this information can be passed to the later PROC PLOTs. There are three general approaches for passing information from a DATA step to a subsequent procedure:

1. Process the dataset in the procedure.
2. MACRO language.
3. Write to a file in the DATA step, and %INCLUDE that file in the procedure.

In this case, option 3 is most suited to our needs. The general method involves two steps:

1. In the first DATA step, write to two temporary files. Each temporary file will contain the entire PLOT statement to be used in a later PROC PLOT.
2. In each PROC PLOT, replace the entire PLOT statement with a %INCLUDE statement.

By employing appropriate techniques, the following program is virtually maintenance-free:

```
DATA ONEYEAR;
SET TEST.RESULTS;
IF _N_=1 THEN DO;
   RENEQ = TODAY();
   START = INTNX('MONTH', RENEQ, -12);
   FINISH = INTNX('MONTH', RENEQ, -1);
   CUTOFF = INTNX('MONTH', RENEQ, 0) - 1;
   FILE BIGAXIS NOTITLES;
   PUT 'PLOT FAILRATE*DATE=*1, / ';
   'VAXIS = 0 TO 1 BY 0.1 HAXIS =';
   'START TO FINISH BY MONTH;';
   FILE TINYAXIS NOTITLES;
   PUT 'PLOT FAILRATE*DATE=*1, / ';
   'VAXIS = 0 TO .2 BY .02 HAXIS =';
   'START TO FINISH BY MONTH;';
END;
RETAIN START FINISH CUTOFF;
IF START <= DATE <= CUTOFF;
   DATE = INTNX('MONTH', DATE, 0);
PROC SORT DATA=ONEYEAR;
   BY TOY DATE;
DATA PLOTME (DROP=MAXFAIL);
   SIZE (KEEP=MAXFAIL TOY);
   SET ONEYEAR;
   BY TOY DATE;
   IF FIRST.TOY THEN MAXFAIL = 0;
   RETAIN MAXFAIL;
   IF FIRST.DATE THEN DO;
      TESTS = 0;
      FAILURES = 0;
   END;
   TESTS + 1;
   IF RESULT = 'F' THEN FAILURES + 1;
   IF LAST.DATE THEN DO;
      FAILRAT = FAILURES / TESTS;
      OUTPUT PLOTME;
      MAXFAIL = MAX(MAXFAIL, FAILRAT);
   END;
   IF LAST.TOY THEN OUTPUT SIZE;
DATA BIG TINY;
   MERGE PLOTME SIZE;
   BY TOY;
   IF MAXFAIL > .2 THEN OUTPUT BIG;
   ELSE OUTPUT TINY;
PROC PLOT DATA=TINY;
   BY TOY;
   %INCLUDE TINYAXIS;
   FORMAT DATE MONYY5.;
PROC PLOT DATA=BIG;
   BY TOY;
   %INCLUDE BIGAXIS;
   FORMAT DATE MONYY5.;
```

1250
The author welcomes comments, questions, and even criticism. Mail all to:

Robert Virgile
4 Colonial Village Drive
Arlington Heights, MA 02174

Reference Article