

Mighty Macros Manage Many Monstrous Tasks!

Roger Konyndyk, Steelcase Inc.

Abstract

Faced with a need to generate a large number of sales reports with similar formats from the same dataset, we turned to the SAS® macro language. By writing a system of macros, we were able to generate new reports very quickly, while assuring consistency in data, labeling, and computation. We ended up with an extremely flexible code generation system that not only handled production needs, but also took care of 95% of our *ad hoc* requests. Finally, FSEDIT screens allow non-programmers to enter the *ad hoc* report requests into SAS datasets. Batch programs translate the observations in those datasets into SAS macro calls to create the reports.

Background

Following a reorganization of our sales force, all of our production sales reports were obsolete. Furthermore, users might request information arranged or selected by any of a large number of variables: dealer, product, customer type, account representative, country, discount level, internal business unit, and so on. Whatever reports we provided, there were always other ways to look at the data, and we could be sure that sooner or later, a user would need to look at the data in a way we hadn't programmed for. Couple the great need for information with limited programming resources, and you have potential for seriously frustrated users. To keep life interesting, continuing changes to the organization meant continuing changes to the reports. Since the basic data set for the reports has a couple million records, debugging can be slow and painful.

A Flash of Insight...

In looking at the reports which I was to replace, I noticed that there were only three or four basic types of reports. There were account listings, which showed sales by special quote number, reports ranking dealers by sales, and many PROC TABULATE reports showing sales and discounts for the preceding month and the fiscal year-to-date, with comparisons to the same periods in the previous fiscal year. What was different in the reports was the data on which they were based, sort orders, by-variables, and the variables in the row table space in the TABULATE reports. After looking at the reports long enough, I realized that each of the report types had many common elements, and I began to think about ways to write a unified system using macros.

We have a departmental production macro library, and we have all our SAS sessions and batch jobs set up to use it. Thus we are well set up to write and use our own macros. Having such a macro library is crucial to an endeavor such as this one. Any well-run group of SAS programmers or users should have such a macro library.

Disadvantages of Macros

- Macros require more knowledge and expertise than ordinary code.
- Writing and debugging is more difficult than working with ordinary code. There are two levels of debugging: the macro, and the SAS code generated by the macro.

- Macros can get quite complex when they must offer many options and cover many different situations.

Advantages of Macros

- Maintenance is only needed in the macro library, instead of being spread through many programs.
- By having everyone use the same macros, you guarantee that everyone uses the same variable names, labels, and computational methods.
- Because of the consistency that macros provide, it is easier for one person to maintain a program written by someone else.
- Macros can be programmed to be extremely flexible.
- Using the same macros instead of writing custom programs is a great time-saver.
- Debugging is done when the macro is written. A programmer can regard a macro as already-debugged code.

Our System of Macros

Having decided that this task cried out for the SAS macro language, we must decide what types of macros we should have. As the project went on, we ended up with some groups of macros necessary for anyone attempting a project like this:

- *Setup macros* that do things like format a report I.D. number, set page dimensions, create useful global macro variables, and make formats and datasets to use later.
- *Data selection* macros that extract the right data from our sales history file, while adding some useful additional variables.
- *Report generation* macros that take the data selected from our sales history file and create the desired reports.

Our macros are geared to our specific application, and so the code is not likely to be of general interest. However, as an example, here are the macros that we use and the tasks they perform. We begin with the setup macros:

%rptname takes the SYSPARM parameter passed to the job and uses it to format a report I.D., which is put in the upper left corner of the first title line of each report.

%mkmacvar creates several global macro variables: the current fiscal year, the last month for which we have sales data, the name of that month.....

%setpage sets the *pagesize* and *linesize* options to insure compatibility with our company's printers. This macro consists only of an *options* statement that sets the *pagesize* and *linesize* system options. We use this macro, because one day changes to our company's computer network reduced the number of lines per page that our printers could handle. When I realized that I had to change options settings in over 40 production programs, I decided to write this little macro, so I would not have to go through that again.

%rptprep reads various SAS datasets and sets up several formats used in creating variables later. The formats do such things as relate a special quote number to the name of the sales representative handling the account, or relate the state and county to current company market area boundaries. By doing this in real time, we save disk space and guarantee that we will always be using the latest version of our data.

%libname functions like a system catalog. This macro will issue a libname statement

for any of our department's production SAS data libraries. For example, *%libname(usageog cangeog)* will allocate our U.S. and Canadian geography data libraries. By using this macro to allocate SAS data libraries, we can move libraries to different disks and make changes to only one macro, instead of every program that uses the SAS data library. Further, we insure that everyone uses the same libnames for the same libraries.

Two macros handle the data selection:

%fytdsumm extracts data for the current and previous fiscal year-to-date from our sales history summary file. It creates several new variables with information about our company's sales territories and field sales people. Output from this macro is often stored in a permanent dataset so that it can be used by several jobs. By doing this in a macro we insure that all these new variables have consistent definitions, names, label, and formats.

%summextr extracts data from the dataset created by the *fytdsumm* macro for use in reports. For example, we may want data only for the United States, or one region in the U.S., or one market area, or for one internal business unit, or only sales to government agencies. This macro handles the data selection and creates some title lines

which will appear on the reports. This way the reports will be labeled to reflect the data that they contain. Also, if a report has data for both the U.S. and Canada, this macro decides to convert Canadian dollars to U.S. currency, and puts that fact into a title line. The title lines are all set up as global macro variables, so they are available to the later macros that format the reports.

Finally, the macros that generate reports:

%fytdtab is our most popular report macro. It takes the data from the *summextr* macro and puts in into a PROC TABULATE, allowing for one to three category variables on the left side, a possible page variable. If by-variables are desired, it sorts the data by those variables and inserts a by-line.

%fytdlrs ranks our dealers by sales.

%fytdacct lists special quotes, the associated customer, the dollar amount, and the discount. It allows for sorting by account name, quote number, or dollar amount. Accounts are always grouped by type of customer.

By using these macros, a typical production program becomes merely a series of macro calls:

```
%rptname      We begin with the "setup" macros described earlier.
%mklibnm
%libname(rptdata)
%setpage
options missing=' ' mprint;
** Set up macro variables          **;
%mkmacvar
** The RPTPREP macro prepares data and formats for the other *;
** macros.                        *;
%rptprep

%let prefix=SHIP;   Ask for shipments, not orders acknowledged.

%let bu = 1;
```

Now the data selection

```
** Reports for Domestic U.S. Sales Only **;  
%summextr(busunit=&BU,      Extract data for the U.S. only, for just one business unit  
  indata=rptdata.&prefix.fytd,  
  sellunit=US      )  
%fytdgraf(fiscyear=BOTH, sumvar=&nnvar,      Report generation macros  
  grafvars=rptfy,  grpvar=prodline,  
  title2=Sales by Product Line and Collection Code)  
%fytdtab(col1=prodline, col2=collcode,  
  pagevar=busunit,  
  title2=Sales by Product Line and Collection Code)  
%fytdtab(col1=prodline, col2=discpct, Tabulate report with discounts by product  
  title2=Discount Analysis for Sales by Product Line)  
run;
```

General observations about writing macros

- Any time you see a task, even a simple one, that must be performed many times, write a macro to do it.
- Make your macros flexible! Input and output dataset names should be macro parameters that user can determine. Provide lots of macro parameters that the user can control.
- Where possible, provide sensible default values for macro parameters.
- Document your macros. Comments at the beginning of each macro should describe what the macro does, how to use it, and the meaning of each of the macro parameters. That way the user can see how to use the macro just by looking at the macro code. We also have documentation stored on-line, in using a popular word processor.
- Include comments within your macro code, too, so a person doing maintenance on your code (such as yourself!) will know what the code is doing. This can be more important for macro code than ordinary SAS code.
- Note that you can use the %sysexec function to issue operating system commands and even check their return codes. You can even check what operating system you're in, and issue the appropriate command based on that. We

use this capability to do such things as uploading data to our mainframe and submitting batch jobs on it.

- Use global macro variables to pass information between macros.
- Macros that build datasets should also create descriptions of the data. For example, our macro variables that extract data also create global macro variables describing the data. These are automatically included in the title lines of all reports. This is crucial to avoid confusion as to what data the report is based on.

Handling ad hoc requests

Once the macros have been written, you can handle *ad hoc* requests by stringing together appropriate macro calls. However, this still requires some knowledge of programming. Even writing a macro call may worry some inexperienced SAS users. It's possible to make things much easier. Suppose you had a SAS dataset where each variable corresponded to a parameter in a macro call, with one observation for each report request. Some of the variables would determine which data to extract, another would decide which report macro to call, and other variables would give the values of the parameters in that report macro call. Once you have this, you could write an FSEDIT screen to facilitate entering the report

requests. The only challenge then remaining is to translate the observations in the dataset into appropriate macro calls to generate the desired reports.

First, you need a SAS dataset in which to store data and report requests. The concept is quite simple: each observation in the dataset corresponds to a report request, including information about what data to use. Each variable corresponds to a parameter in a macro call. For example, we have the following variables, among others:

- PREFIX** - Can take on the values ACKN or SHIP - indicates whether to base the report on orders acknowledged or on shipments.
- BUSUNIT** - Take data for just one of our internal business units, or all of them?
- ASDAREA** - Take data for just one area that an Area Sales Director is responsible for?
- IFSTT** - An free-form IF statement to allow more flexibility in choosing data
- WHERECL** - A free-form WHERE clause to apply to the data
- RPTTYPE** - Tells what report type to use; i.e., which report macro to call.

There have to be enough variables to cover all the available report macro parameters, too. To keep things reasonable, the report macros use the same parameter names, where feasible.

- BYVARS** - Variables to include in a BY-line.
- COL1, COL2, COL3** - In the FYTDTAB macro, that formats a PROC TABULATE report, these give the variable names that are used as CLASS variables in the left-hand columns. Only the first of these must be given a value.

There are several more variables, but this should suffice.

Once we have the dataset, we need to translate it into appropriate macro calls. This is not too difficult. A SAS program uses the dataset as input to a `_null_` data step which use `put` statements to write the macro calls to an external file. Here is a simplified version of such a data step:

```
data _null_;
  set ...
  file 'temp-pgm.dat';
  put '%let prefix=' prefix ';';
  put 'summextr(asdarea='
      asdarea $3. ', busunit='
      busunit $1.;
  if wherecl ne ' ' then put ',
      wherecl=%str(' wherecl ')';
  if ifstt ne ' ' then put ',
      ifstt=%nrstr(' ifstt '),';
  put ')';
  put '% rpttype '(byvars=' byvars
      ', .... )';
```

Obviously, you need to be careful about getting your quote signs in the right place!

Once this is done, a `%INCLUDE` statement brings in the sequential file for immediate execution. Finally, if the program executed without error, we wipe out the report requests in the request dataset:

```
data dept.fytdrpt;
  set dept.fytdrpt (obs=0);
run;
```

By setting options `errorabend` before this data step, we leave the report requests in place if there was a problem. We have set this program to run at 6 P.M. each weekday.

By now you are probably aware that the report request dataset must be in the proper form for this to work. Thus there must be an FSEDIT screen (accessible through SAS/ASSIST) to prevent the user from making mistakes.

```

+-----+
FSEdit DEPT.FYTD RPT-----+
      Fiscal Year-to-Date Reporting Based on the Sales Summary File
PREFIX:  .... (ACKN=Orders, SHIP=Shipments)  SELLUNIT:  ...
CUSTTYPE: ... Customer Type  ASDAREA: .....  BUSUNIT:  . (Blank=ALL)
WHERECL: .....
IFSTT:  .....
REPORT:  ..... Report to produce  BYVARS: .....
      Title Lines:
      _____
      _____
      _____
Items for the FYTD Sales Summary Report (TABULATE):
PAGEVAR: ..... Page-Break  Left-hand Cols: .....
Account List: FCASTS: . Show Forecasts?  OTHRSORT: .....
Dealer Rankings: RANKBY: ..... Rank Dealers by NDN or Net-Net?
Hbar Charts: FISCYEAR: .... (CURRENT, PREVIOUS, or BOTH FY's?)
      .... Type of statistic to graph for the variable .....
      .... Ordering of bars (blank = default)
      ..... Variable(s) to graph
      (One bar per group for each value)
      ..... Group the bars by this variable (optional)
+-----+

```

Admittedly, the screen is rather cluttered, and I would probably use AF/Frames and break it up into more than one screen if I were to write it today. However, note the following features: Help screens pop up valid values for many variables to be filled in. Depending on which report type is chosen, fields not used for the report are blanked out and protected, while a value can be demanded for other fields. The free-form IF statement and WHERE clause can't be checked completely, but at least I can check for an odd number of quotes (a fairly common problem). I won't claim that this system is truly user-friendly, as it requires a knowledge of the sales history file that contains the data. Still it's a vast improvement over programming from scratch. With only a small number of people actually using the screen, the system works well.

There's one other "detail." We also have forecasts by fiscal year for our sales by product and by quote number. This macro

system also allows comparison of actuals to forecast, where appropriate. Another macro calculates the forecasts, whether by product or by customer. The report macros call the forecast macro as needed, and incorporate the forecasts into the reports.

We're not done! These reports are all based on comparing the current and previous fiscal year-to-date. We must also handle report requests for arbitrary time periods. The system just described was used as the basis for a second system to handle these needs. In fact, the vast majority of our ad hoc requests are for arbitrary time periods.

Finally

Anyone with an understanding of our sales history summary file and a little knowledge of SAS can generate ad hoc reports. We have freed up our SAS programmers and macro experts to work on other things. By moving *ad hoc* reporting to people with a lower skill level, we have greatly improved

our productivity as a department. A person needing information calls our department, and the request is taken care of by filling in the FSEDIT screen. The next morning, the reports are waiting to be printed off.

Because our ad hoc reports use the same macros as our production reports, we are assured that the numbers on ad hoc reports will always match those on production

reports. We don't have to worry about whether a programmer included the right data.

To sum up, this paper is an advertisement for the SAS macro language. It's well worth mastering, and it can be a great productivity boost for your entire department. Other people might look at you not only as a SAS guru, but also as a hero!

SAS is a registered trademark of SAS Institute Inc. in the USA and other countries.
® denotes USA registration.

Contact information:

Roger Konyndyk
CD-5W-08
Steelcase Inc.
901 - 44th Street SE
Grand Rapids, MI 49508

Telephone: (616) 698-4518
FAX: (616) 698-4502