DESIGNING MACRO-BASED SYSTEMS

Jeff Phillips, ORI Inc.

Abstract

This tutorial is a discussion of the SAS Macro Facility as a tool to enhance SAS' performance as a programming language. The focus is on using Macro to construct modular SAS-based software systems. Modular structure, conditional execution of SAS code, and the use of Macro variables as software system parameters are the key elements of the tutorial.

Introduction: SAS and SAS Macro in System Design

The SAS system has experienced phenomenal growth in power and usage since its introduction. Originally designed to provide scientists and engineers with a data analysis tool that was powerful yet easier to use than traditional programming languages, SAS has become a unique hybrid: The SAS DATA step gives users the data manipulation capability of a high level programming language, while over 130 SAS Procedures provide a wide array of easy to use data handling, analysis, and reporting capabilities. The focus of this tutorial is on how the Macro facility can enhance SAS as a programming language.

Where SAS is used as a programming language today, it is often employed to replace traditional languages such as PL/I or COBOL. However, in many, if not most organizations, SAS is relegated to the area of ad hoc information retrieval and quick data analysis. Though no one could argue that SAS should replace traditional languages in all cases, there is ample evidence that it performs well as an alternative language. As an example, consider the 50,000 line production system, written almost entirely in SAS, used to process the U.S. Bureau of Labor Statistics' Consumer Expenditure Survey data. This system's design and implementation is described by Judith Mopsik of ORI, in her paper "SAS Can Replace Traditional Programming Languages", (SUGI, 1984).

This tutorial is designed to show how the Macro facility can enhance SAS' performance as a programming language. To explore this area, SAS must first be placed in its proper context. Figure 1 illustrates the flow of the archetypical SAS job, in which each DATA or PROC step is compiled and executed separately and in sequence. Figure 2 shows the structure of a traditional modular system, in which a main routine controls system execution and calls subroutines as needed. Figure 3 lists four major requirements of software systems, and allows one to compare SAS without Macro to traditional programming languages as to the ability each has to satisfy the four requirements.

The first feature or requirement is modular structure itself. This structure, in which a main routine controls the execution of the system and calls in subroutines as needed, is quite different from standard SAS structure. It is this modular aspect of systems which is the most difficult for SAS to emulate; unfortunately, the other features of software systems are to some extent derived from modular structure. For example, conditional execution of blocks of code is a natural function of the modular structure. It is not a natural function of SAS, in which each step is separate and sequential. Defining system execution parameters (e.g. dates/times, file names, etc.) is likewise natural in modular structure, where such values are passed or inserted in the main routine. In SAS without Macros, however, there is no easy way for a system to place parameter values where they are needed. The user must somehow make the parameter values available to the appropriate DATA or PROC steps.

An important result of modular structure and parameters control is system maintainability. SAS lends itself to developing easily maintainable code, but when system-level requirements like parameter control and conditional execution of code become important, the basic SAS structure is awkward and creates problems for design and maintenance.

The addition of the Macro Facility to SAS solves the central problem in using SAS as an alternative to traditional languages (Figure 4). Macro allows a SAS-based system to have a modular structure in which a main Macro ("main routine") can control execution of the SAS steps, calling in other Macros ("subroutines") as needed. Just as in many traditional systems, such subroutine calling can be nested—Macros called by the main Macro can themselves...
call other Macros, and so on. Macros, then, allow SAS code to be packaged functionally. Since a Macro can contain anything from part of a DATA or PROC step to multiple steps, and can be invoked whenever needed by means of Macro logic, the Macro facility transform the rigid, step-by-step compile/execute sequence that limits SAS capability as a language into a flexible structure that can answer every requirement of even the largest, most complex system. Macro variables become controlling parameters; they can be generated by the system or set by the user. Conditional code compilation and execution become natural features of a SAS Macro based system. Packaging is no longer enforced by DATA and PROC steps— it becomes a tool rather than a limitation. Maintenance of a SAS Macro based system becomes easy, because the great flexibility allowed by Macro makes changes and enhancements easier to implement.

System Components and Their Macro Solutions

1. Main Routines - The heart of any software system is its main routines—that source code which performs the main function of the system. The main routine of a system may not be the physically largest part of the system, but it is its logical core. A main routine in a SAS Macro-based system is a set of SAS and SAS Macro code, placed inside a Macro, that performs a specific function. The Macro may contain DATA and PROC steps, Macro calls, and Macro code used to control execution. Figure 5 shows an example "main routine". The code is packaged as a Macro, using keyword style parameters to govern aspects of its execution. Inside the Macro are "skeletal" SAS code lines consisting of DATA and/or PROC steps whose generation is controlled by the Macro processing; Macro code such as %IF statements to control what portions of the SAS code are generated; and calls to Macros in order to perform specific functions.

This main routine Macro has many advantages, some of which will be explained later, but chief among them are usability, flexibility, and economy. The Macro can act as a "black box" so that the end-user need only supply the Macro required input values through parameters, and the Macro will produce the exact output desired. Only those assigned the responsibility of maintenance need ever even see the actual code. The Macro is flexible because all the code inside it is both skeletal and conditional. The skeletal nature of the code means that such variable entities as data set names need only be specified on the Macro call, not manually inserted wherever they are needed. Main routine Macros are economical because of their conditional capability. Macro logic can avoid the unnecessary invocation of long and complex steps. In other words only the processing required is performed.

2. Subroutines - As Figure 5 showed, a main routine Macro may call other Macros. This nesting feature allows standard sets of code to be stored as Macro and called whenever necessary. Storing code in this manner was possible under old-style (SAS76) Macros, but by using SAS82 Macros parameters can be passed from the main module. Figure 6 shows the Macro "%COMPUTE" that was called by the main Macro in Figure 5. %COMPUTE is a SAS assignment statement or set of statements, and can be inserted wherever needed in the system. One major advantage of such Macros is that they guarantee consistency in the code. Another advantage is modularity. If another calculation had to be added to the system, one change to the %COMPUTE Macro would accomplish this for the whole system.

A special class of subroutines in a system is the utility. Very few if any systems, SAS or otherwise, operate without sorts, prints, copies, and other utility operations. At one level, SAS Procedures, such as PROC PRINT, allow easy subroutine calls, and Macro code can tailor a call to a PROC in terms of data set names, repetitive or conditional execution, etc. On another level, however, certain utility functions which cannot be handled by a simple procedure call but which are generalized enough to be placed in many different applications take on the quality of "tools". David Septoff and Katie Hubbell of ORI, Incorporated discuss the concept of a "SAS Macro Tools Library" in detail (SUGI, 1985). The main feature of subroutines designated as "tools" is that they perform functions common to multiple systems. This type of subroutine Macro, which can be as simple as a print or as complex as a FORMAT table generator or a data set comparison routine can be
stored in a generally accessible library and accessed by any system requiring this function. Figure 7 shows the utility Macro "%NOBS", which is used in the "main routine" (Figure 5) to make sure that the transaction data set is not empty.

3. Parameter Control - Parameters that control SAS Macro system can be of three basic types, as Figure 8 shows. The "global" parameter, such as "%DEBUG" in Figure 8, acts to change a system's run mode in some way. For example, "%DEBUG" in Figure 8 sets some system options. A second form of parameter control is the "set" parameter, which is set by the module itself, and is illustrated by "%ERROR" in Figure 8. If a negative price is encountered in the first DATA step of the module, %ERROR is set to YES, stopping processing and causing a note to the SAS log to be printed. Later, the value of %ERROR is used to conditionally execute PROC PRINT. The third form of parameter is the "structural" kind, such as "%DATA" and "%DATE" in Figure 8. Structural parameters are used to fill in skeletal SAS code, and are usually set by the user.

Macro Techniques for System Building

Having discussed the major SAS Macro structures needed for effective system design, this tutorial will now turn to a more specific discussion of techniques for employing these structures.

1. %INCLUDE: A key element of Macro-based systems is the use of the %INCLUDE statement to bring in appropriate modules, which may be stored in several different locations, to the system. Figure 9 illustrates the use of %INCLUDE, followed by the actual call to the main routine Macro, in a job stream. Note that the NOSOURCE2 option is used on the %INCLUDE statements to suppress the listing of the include Macros.

2. Conditional Execution: Figure 10 illustrates diagrammatically the conditional execution of SAS code. The conditionally executed code illustrated here could be one or more DATA or PROC steps, Macro calls, or Macro operations such as %LET assignment statements. The advantages to this method of conditional execution are obvious: tremendous flexibility and the avoidance of techniques involving external files.

3. Iterative Execution: Figure 11 illustrates how Macro logic can be used to execute code in an iterative fashion. In Figure 11, the calculation is executed once for each value of $NUM where $NUM can be set by a %LET statement, a MACRO parameter, or a SYMPUT call in a previous DATA step. The index variable $ not only specifies repeated execution, but is used to add the proper suffix onto the variable names. A major advantage to this method over other techniques such as ARRAY processing is that the loop happens only once—at compile time, to generate the 20 assignment statements. Any regular SAS loop would happen at each execution of the DATA step.

4. Debugging — The use of %PUT: In addition to facilitating the use of conditionally set system options and conditionally executed error routines based on Macro variable error flags, Macros allow (and in a sense require) another debugging tool: %PUT. The %PUT statement, which writes notes to the SAS log, can of course be used for any message in much the same way as a regular PUT statement. The nature of Macro-based systems, however, leads to the idea of placing %PUT in two specific positions in all Macros. Figure 12 shows %PUT at the top of a Macro, just after the %MACRO statement. The note to the log generated by these %PUTs appears when the Macro begins its expansion/resolution, and indicates that the call to the Macro was successful. (The three asterisks are used to differentiate this note from SAS-generated notes). The second set of %PUT statements, after the %END statement ending the Macro definition, generates a note when the Macro is %INCLUDED, and indicates that the Macro is available for processing when called. These notes are useful for two reasons: first, the nested nature of Macro calls in a Macro-based system makes Macro errors difficult to track without some indication as to how far Macro processing has gone; second, if the system is in production mode, and NOSOURCE is set, the notes allow the reader of the SAS log to track the flow of the system.

Conclusion

While SAS in and of itself is an excellent tool for large scale system implementation, the Macro Facility makes SAS more powerful by an order of magnitude. Macros allow SAS systems to
be truly modular in structure, and can save CPU time and money because they permit conditional compilation and execution of code and step-to-step parameter passing without recourse to external files that must be written to and then included back into the job stream. Even more importantly, Macro-based systems are easily maintainable and give an added dimension of flexibility to the system designer. At the same time, the finely controllable nature of Macro-based systems is such that end users in batch or interactive environments can have great control over this system without ever dealing directly with actual code.

This tutorial has attempted to illustrate some of the techniques and conventions developed to design efficient, flexible, and powerful SAS Macro systems.

ACKNOWLEDGEMENTS AND REFERENCES

The author wishes to thank Jesse Gary, David Septoff, Katie Hubbell, Don Henderson, and Vicky Gardner of ORI, Incorporated for their help in developing this paper.

REFERENCES

Mopsik, Judith, "SAS Can Replace Traditional Programming Languages", SUGI, 1984


SAS is a registered trademark of SAS Institute Inc., Cary, NC, USA.

The authors can be contacted at:

ORI, Inc.
122 C Street, NW
Suite 250
Washington, DC 20001
(202) 737-2666
FIGURE 2: "TRADITIONAL" LANGUAGE SYSTEM STRUCTURE
FIGURE 3: FUNCTIONAL FEATURES OF SOFTWARE SYSTEMS

1. Modular Structure - A main routine governs execution and calls procedures (subroutines) when needed. Each routine performs some specific function.

2. Conditional Execution - Procedures are executed only when required.

3. Parameter Control - Parameters governing an individual run of a system are passed to the system.

FIGURE 4: SAS MACRO COMPONENTS OF SYSTEMS

<table>
<thead>
<tr>
<th>SYSTEM REQUIREMENT</th>
<th>MACRO SOLUTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modular Structure</td>
<td>Macros which package code into &quot;main&quot; and &quot;subroutines&quot;</td>
</tr>
<tr>
<td>Conditional Execution</td>
<td>Macro &quot;%IF&quot; Logic</td>
</tr>
<tr>
<td>Parameter Control</td>
<td>Macro Variables</td>
</tr>
</tbody>
</table>
FIGURE 5: A MAIN ROUTINE MACRO

\%MACRO UPDT (MAST=, NEWMAST=, TRANS=, KEYS=, STMTS=, SORT=NO);
\%LOCAL NOBS;
\%NOBS(DATA=&TRANS) /* A UTILITY MACRO */
\%IF &NOBS NE 0 %THEN
  %DO; /* HAVE TRANSACTIONS-EXECUTE UPDATE */
    %IF &SORT=YES %THEN
      DO; /* SORT TRNS DATA SET */
        PROC SORT DATA=&TRANS;
        BY &KEYS;
        RUN;
      END; /* SORT TRNS DATA SET */
    %IF &NEWMAST= %THEN %LET NEWMAST=&MASTER;
    DATA &NEWMAST;
    UPDATE &MASTER &TRANS;
    BY &KEYS;
    %IF &STMTS NE %THEN %&STMTS;
    RUN;
  %END; /* HAVE TRANSACTIONS-EXECUTE UPDATE */
  %ELSE
    %PUT NOTE:***NO TRANSACTIONS-UPDATE SKIPPED;
  %MEND UPDT;

FIGURE 6: A SUBROUTINE MACRO

\%MACRO COMPUTE;
\%* THIS MACRO IS PASSED INTO THE MAIN ROUTINE MACRO "%UPDT" VIA THE "STMTS" KEYWORD PARAMETER.;
INCOME = NUMBER * PRICE;
%MEND COMPUTE;
FIGURE 7: A UTILITY SUBROUTINE MACRO

%MACRO NOBS(DATA=);  
DATA _NULL_;  
I=1;  
SET &DATA POINT=I NOBS=NOBS;  
IF _N_=1 AND NOBS=0 THEN  
DO; /* DATA SET IS EMPTY */  
   PUT "THE DATA SET &DATA IS EMPTY.;";  
   PUT "PROCESSING CONTINUES.";  
   CALL SYMPUT("NOBS", "0");  
   STOP;  
END; /* DATA SET IS EMPTY */
ELSE  
   DO; /* DATA SET CONTAINS OBSERVATIONS */  
      PUT "THE DATA SET &DATA CONTAINS";  
      PUT NOBS :5. "OBS";  
      CALL SYMPUT("NOBS",LEFT(PUT(NOBS,BEST,»));  
      STOP;  
END; /* DATA SET CONTAINS OBSERVATIONS */
%MEND NOBS;

FIGURE 8: EXAMPLES OF PARAMETER CONTROL

%MACRO MYSYS(DEBUG=, DATA=, DATE=);  
%IF &DEBUG=NO %THEN OPTIONS SOURCE MPRINT  
   %STR( );
%ELSE OPTIONS NOSOURCE NOMPRINT %STR( );
DATA PROFITS(DROP=ERR_MSG)  
ERROR(KEEP=ERR_MSG ITEM=NO PRICE);  
SET &DATA;  
IF PRICE LT 0 THEN  
   ERR_MSG="NEGATIVE PRICE FOUND";  
   CALL SYMPUT("ERROR","YES");  
   OUTPUT ERROR;
END;  
ELSE  
   DO;  
      SALES=PRICE * QUANTITY;  
      PROFIT=SALES - EXPENSES;  
      OUTPUT PROFITS;
   END;  
RUN;  
%IF &ERROR=NO %THEN  
%DO;  
   .PROC PRINT DATA=ERROR;  
   TITLE "ERRORS FOUND IN SALES DATA FOR &DATE";
   RUN;
%END;  
%ELSE  
%DO;  
   .PROC PRINT DATA=PROFITS;  
   TITLE "PROFITS FOR &DATE";
   RUN;
%END;  
%MEND MYSYS;

1137
FIGURE 9:
ILLUSTRATION OF MACRO-BASED SYSTEM INTERFACE TO OPERATING SYSTEM USING INCLUDE

SYSTEM SOURCE CODE
MACRO LIBRARY

TERMINAL SESSION

USER
(HAPPY BECAUSE
HE LOVES MACROS)

UTILITY MACRO TOOLS LIBRARY

SAMPLE
(CS)
JOB
STREAM

DATA TO BE
PROCESSED

//MAJ0E JOB ...
// EXEC SAS ...
// SYSMACRO DD ...
// MACTOOLS DD ...
// DATA DD ...

%INCLUDE SYSMACRO (MODULE1,MODULE2 ...) /NOSOURCE2;
%INCLUDE MACTOOLS (TESTPRN,NOBS ...) /NOSOURCE2

MODULE1 (PARM1=value,PARM2=value ...)
%IF &TYPE = D %THEN
%DO;
           SAS CODE (B)
%END;
%ELSE
%IF &TYPE = I %THEN
%DO;
           SAS CODE (A)
%END;
FIGURE 11: ITERATIVE EXECUTION OF SAS CODE

```sas
/* PERFORM WEIGHTED COST CALCULATION */

%do i=1 %to &num;
    wtcost&i = cost * weight&i;
%end;
```

FIGURE 12: THE USE OF "%PUT"

```sas
%macro dummy;

%put note: "macro dummy is now beginning macro processing;

(text of macro>

%mend dummy;

%put note: "macro dummy is now loaded;

(text of macro>

1140