

Using Macros to Maintain Program Flexibility and Integrity When Accessing Very Large Datafiles

Edward Jensen, Corbett, OR

ABSTRACT

When writing programs to analyze large datafiles great care must be given to program flow and the accuracy of the procedure otherwise, considerable time is lost in reruns. Macros provide an excellent way to control the program flow and make last minute modifications feasible by allowing changes to be accurately reflected in all relevant areas. This paper will present the use of macros from creating the macro environment with appropriate system options and the Autocall Facility through making the macros flexible with use of both local and global macro variables as well as macro program statements. Various techniques for managing variable lists, standardizing procedures, saving selected macros in autocall libraries, self documenting one time changes, and debugging techniques will be demonstrated. Sample standard macros that have been found useful will be shown. The following techniques are useful for all SAS programs, however, their value is more evident with programs accessing large datafiles when long processing times are required. Emphasis is in the MVS batch environment.

INTRODUCTION

Why use Macros

Because the SAS system is such a great analysis tool, it is used in areas where analysis is performed to support mission critical decisions. This creates a demand for programming techniques that can quickly respond to the requirements and maintain accuracy and reliability.

With SAS macros, code that has been tested and proven reliable can be modularized for use by several programs, modifications documented, and program structure designed for high level control.

TECHNIQUES

General Program Control

Perhaps the most useful application of macros is to break the program into manageable steps that can be

run individually. This works well on programs with many data steps and procedures or programs that are processing large files where each step in the program takes a considerable time to run. Each program step can then be run independently and can easily be controlled.

Setting up a program to be run this way is quite simple. Just write the program as usual, then separate blocks of code by inserting `%MACRO name;` and `%MEND;` - where *name* is the name for that section of code. This defines the code as a series of macros that are individually called later in the program

The example program in Figure 1 shows how a modular program is set up. The program code is grouped into macros in the first part of the listing. Then each macro invocation under the "Execute Macros" banner executes a section of code by macro name. Note that some macro calls are not referring to macros coded in the program, but are calling macros stored earlier in the macro library. These are described later in "Sample Standard Macros".

Controlling the Macros to Execute

Referring to the end of Figure 1, you can see how the program is controlled by commenting or not commenting the macro calls. The program displayed uses `/* comment */` syntax. Notice that a "blank" `/*-----*/` line is available so it can be copied over a macro using the TSO copy-overlay operation.

An alternate is to place an `"**"` following the `"%"`, i.e., `/***macroname;`. However, the `/* comment */` syntax in TSO Edit Color Settings will display the string in cyan - clearly setting it apart from regular code.

Using Macro Parameters

As the program develops, you will find some code can be reused if minor changes are made. Rather than duplicating code and hard coding changes, make the macro modifiable by adding macro variables. Macro variables replace variable code or can be used to control what code is to be generated..

Coder's Corner

To use a macro variable in code, first give a name for a macro variable in the %MACRO statement. Then call it in code. For example:

```
%macro printsom(plclass,dakine);
data printsom;
  set mwork3.pnl&PLCLASS.;
  %if %UPCASE(&dakine.)=FIRST %then %do;
    title5'First 50 of Means File';
  %end;
  %else
  %if %UPCASE(&dakine.)=ALFLG %then %do;
    where TYPE =2;
    title5'Summary by Status Flag';
  %end;
...
%mend;
```

Here, two macro variables: PLCLASS and DAKINE are defined. PLCLASS is used to replace code, DAKINE is evaluated to determine what code is generated. This macro call:

```
%PRINTSOM(NWCLA, FIRST);
```

will literally place NWCLA for &PLCLASS

```
set mwork3.pnlNWCLA;
```

and replace &dakine. with FIRST

```
%if %UPCASE(FIRST)=FIRST %then %do;
```

and evaluate FIRST as

```
%if FIRST=FIRST %then %do;
```

to generate:

```
title5'First 50 of Means File';
```

The %UPCASE macro function sets the incoming value to uppercase to ensure the match. The condition first=FIRST is not true.

This type of macro parameter is a positional parameter. In this example, the first position value NWCLA is assigned to macro variable &PLCLASS, the second - FIRST - to &DAKINE. based on the location in the parameter list. If no value is supplied for a parameter as in

```
%PRINTSOM(, FIRST);
```

or

```
%PRINTSOM(NWCLA);
```

A null value is supplied. This can be very useful as seen later.

Another macro parameter - the keyword parameter - is useful for assigning default values. It is indicated by an "=" and the default value immediately following.

```
%MACRO READLIST(CHRPDNUM,INFILE,ANPOS=1);
  DATA CUAL5R41.&INFILE.;
  INFILE &INFILE. MISSEVER;
```

```
%IF &CHRPDNUM.=CHR %THEN %DO;
  INPUT @&ANPOS. ACCTNO $10.;
%END;
```

In this example, with the call

```
%READLIST(CHR, PREMIER);
```

&ANPOS. will be set to resulting in

```
INPUT @1. ACCTNO $10.;
```

If another value is desired, just code ANPOS=n following the positional parameters.

```
%READLIST(CHR, PREMIER, ANPOS=2);
```

If several keyword parameters are used, they can be coded in any order but must follow the positional parameters.

Managing Variable Lists

Another powerful use of macros is to manage variable lists. Variable lists are used in many statements and procedures: KEEP statements, PRINT VAR statements, PUT statements, INPUT statements to name a few. Matching variable lists is critical in the MEANS procedure where the OUTPUT SUM= and MEAN= statements are mapped by the VAR statement. This can be controlled by creating a modifiable variable list.

Such a macro - MFVARI - is shown in Figure 2.1. Here all variable names are no longer than seven characters with a trailing macro parameter &yr. This variable list is used where two sets of variables are needed, one for year 7 another for year 8. For example:

```
keep acctno aau flag %mfvar(7) %mfvar(8);
```

This macro also controlled some naming exceptions based on the year variable as you can see with the %IF... code.

A more versatile variable list macro is shown in Figure 2.2. This list included an parameter to include a format specification following the variable name. Because of its flexibility, it was used both in a PUT and INPUT statement,

```
put aau 5.
  %putyrvar(7,14.2)
  %putyrvar(8,14.2) ...
```

A MEANS VAR statement (notice the format value is null),

```
VAR ffreq %putyrvar(7) %putyrvar(8);
```

and a PRINT FORMAT statement.

```
format _freq ffreq comma14.0
  %putyrvar(7,comma20.2)
```

```
%putyrvar(8,comma20.2);
```

In these examples, the variable names were no longer than seven characters to allow a variable suffix character to make multiple lists unique. This may require some variable renaming. The variable list in **Figure 2.3** provides an option "REN" that will activate the renaming code so the list can be used in a RENAME statement:

```
merge
dec97.pnldec97(in=indec97
  rename=(%mfvar(7,REN)))
jan98.pnljan98(in=injan98
  rename=(%mfvar(8,REN)));
```

As mentioned earlier, the MEANS procedure requires variable lists to be in matching sequence. An example of such a variable list macro is in **Figure 2.4**. It was used to develop a set of variables for SUM=, MEAN=, and NMISS= OUTPUT options:

```
PROC means DATA=MYLIB.RNKACCTN noprint ;
VAR %MFVARS();
...
sum=%MFVARS(S)
mean=%MFVARS(M)
nmiss=%MFVARS(X)
```

Here, the VAR list variable names had no suffix. Also, an optional parameter was available to control including a subset of variables but it was not used here.

Standardizing Procedures

As shown above, some macros will be useful in several areas of the program or application by just adding a parameter to control the changes. At this point, give thought to keeping your code consistent and regular. This can become more clear as you adapt one set of code to another use. For example, once a variable list is developed that can be used in several places, code will be designed to adapt to it's use and, in return, ideas for making the list more flexible will be discovered.

When you do find a macro that can be modified for additional applications, be sure the default or earlier calls will generate the initial code. As addressed above, a null parameter value can be used to trigger the default condition. This will allow earlier uses of the macro to generate the expected code.

Self Documenting one time changes

In a modifiable macro, the variations for the code's use are readily visible. There is no need to sight compare two similar pieces of code or to rely on possibly outdated comments or documentation.

This idea can be taken one step further. As changes

in requirements are made, each change can be indicated by an optional parameter or new parameter value. Thus, all variations are visible and are available if needed.

Debugging Techniques

When developing a program, often extra code is written to display progress or intermediate results. This can clutter up a program making it difficult to follow the main purpose.

A common method of debugging, is to place PUT statements through out the code to display intermediate results. These can be placed in a macro and called in the program without being so intrusive. An example is shown below:

```
/* SET TO 1 TO RUN DEBUG ITEMS */
%LET DEBUG = 0 ;
%macro TESTADDR(SEQ,OBS
  %IF (&DEBUG) %THEN %DO ;
    if _n_ <=&OBS. then
      put ">>>>> &SEQ. " _n_ = addr2=
    %END;
  %mend;
```

It is called by:

```
%TESTADDR(A2,10);
```

The %LET statement controls all %TESTADDR calls as to whether the PUT statement executed. In this example, macro variable &SEQ. makes the literal associated with the PUT unique, &OBS controls the number of observations PUT is written.

This macro is defined first in the program to be available in all following macros.

An alternate method is to include code in a macro that is controlled by a parameter for that macro.

```
%MACRO PNLPRINT (PLCLASS,DEBUG);
...
put
  %IF &DEBUG=DEBUG %THEN %DO;
    @leftcell ' >>>>>> DEBUG VERSION
    <<<<<< '
  /
  %END;
  @leftcell '
...
@lefttdd '| ' _FREQ_ COMMA17. ' '|
actphhdm comma10.3 ' '|
+13 ' '|
%IF &DEBUG=DEBUG %THEN %DO;
  ' _FREQ_ | ACTPHHDM |
  '|
%END;
...;
```

In this example, additional lines are put and additional data are placed at the end of lines when the DEBUG value of macro variable &DEBUG is given.

Coder's Corner

By using this method, debugging code can be left in the program and called as needed by simply supplying the parameter at run time.

CREATING THE MACRO ENVIRONMENT

System Options

Refer again to **Figure 1**, and the heading:

```
/* -----SYSTEM OPTIONS...
```

The purpose of each option is noted in the figure. It is important that the following options:

```
OPTIONS MAUTOSOURCE SASAUTOS=MACROLIB;
```

be executed first since they set up the ability to use prewritten macros.

Autocall Facility

Macros that are useful over several programs, such as the samples shown below, can be saved in a library and simply called by their name.

The library is allocated to the program using the MAUTOSOURCE and SASAUTOS options as shown above. The SASAUTOS option points to a library where the macros are stored. In the MVS environment, the library is a PDS where the member name is also the macro name. The contents for macro %OPMACRO are saved in member OPMACRO and appear exactly as shown below.

```
/* Sets macro generator */
/* Optional SYMBOL for more gen info
%MACRO OPMACRO(OPMACSYM);
OPTIONS MACROGEN DQUOTE;
%if %upcase(&OPMACSYM.)=SYMBOL %THEN %DO;
OPTIONS MLOGIC SYMBOLGEN;
%END;
%mend;
```

This allows the use of standard code over all programs. Some applications have used macros for storing calculations so that as a calculation is changed, it will be reflected in all programs. A history of changes can be kept in the macro for future reference. As questions arise about a calculation, only one location needs to be reviewed as to the history and current formula.

SAMPLE STANDARD MACROS

Some general use macros that have been found useful are given below. Note the /*...*/ comments in each for more details.

SAS Options

For the macro environment

%OPMACRO

```
/* Sets macro generator */
/* Optional SYMBOL for more gen info
%MACRO OPMACRO(OPMACSYM);
OPTIONS MACROGEN DQUOTE;
%if %upcase(&OPMACSYM.)=SYMBOL %THEN %DO;
OPTIONS MLOGIC SYMBOLGEN;
%END;
%mend;
```

Example: %OPMACRO; Also needs JCL as in the following or equivalent:
//MACROLIB DD DISP=SHR,DSN=MYLIB.SASAUTOS

For hard run errors and nocenter

%OPERRNCT

```
/* Sets hardabend when program abends */
/* Sets report centering */
%MACRO OPERRNCT;
OPTIONS ERRORABEND NOCENTER S=72;
%mend;
```

Example: %OPERRNCT;

Set up dataset compression

%OPCMPRES

```
/* Sets dataset compress options */
%MACRO OPCMPRES;
OPTIONS BUFSIZE=55296 BUFNO=30
        COMPRESS=YES REUSE=YES;
%mend;
```

Example: %OPCMPRES;

Utility

%KILLLIB

```
/* kills an entire library */
/* BE CAREFUL! */
%macro killlib(KILLLIBN);
proc datasets library=&KILLLIBN. kill;
%mend;
```

Example: %macro killlib(MASTRLIB);

%KILLMEM

```
/* DELETES a file from a library*/
/* Option LIST to list library members */
%macro killmem(KILLLIBN,killmemb,list);
proc datasets library=&KILLLIBN.
%if %upcase(&list.)=LIST %then %do;
LIST
%end;
%else %do;
nolist
%end;
;
delete &killmemb.;
%mend;
```

Example:
%macro killmem(MASTRLIB,MASTRMEM,list);

%FLGMRG

```
/* Creates a variable to retain the
   contribution status of an input
   file.*/
%macro flgmerge(flgmrg);
/* Sets up variable to retain */
/* contribution status of merged
   file. */
/* Max length of FLGMRG parm value is 5 */
attrib flg&flgmrg. length=2 format=1.
   label="&flgmrg. File";
flg&flgmrg.=in&flgmrg.;
%mend;
```

Example: %flgmerge(MASTR);

%CONTSALL

```
/* List ALL contents of a SAS library */
/* FORMATS parameter lists the catalog */
/* of a format library and the format */
/* detail. Will abend if library
   does */ /* not contain
   FORMATS.CATALOG */
```

```
%MACRO CONTSALL(CONTSLIB,FORMATS);
PROC contents data=&CONTSLIB.._all_;
```

%if %upcase(&FORMATS.)=FORMATS %then %do;

```
proc catalog cat=&CONTSLIB..formats;
contents;
```

proc format lib=&CONTSLIB..formats fmtlib;

```
%end;
%mend;
```

Example: %CONTSALL(MASTRLIB,FORMATS);

Reporting

%WFBNATTL

```
/* Sets up standard report titles and
   footers*/.
/* TTLMEMBR is program member name */
/* TTLPROJT is project ID number */
/* TTL2 is TITLE2 string - optional */
%MACRO WFBNATTL(TTLMEMBR,TTLPROJT,TTL2);
TITLE "WELLS FARGO BANK, N.A.";
%if %length(&TTL2.)>0 %then
%do;
TITLE2"&ttl2.";
%end;
FOOTNOTE "&TTLMEMBR. - C O N F I D E N T I
   A L - &TTLPROJT.";
%MEND;
```

Example:

%WFBNATTL(MPR12PN8,MAA17,P/L by Product);

%RPTPRTOP

```
/* Sets up report page format options.*/
%MACRO rptprtop;
OPTIONS LS=160 PS=66 CENTER;
%MEND;
```

Example: %RPTPRTOP;

CONCLUSION

Macros take SAS programming to higher levels of control by grouping code into flexible modules. They facilitate standardizing routines, reports, and calculations and they make use of tested and reliable code. By using parameters to control modifications, macros become self documenting.

For programs where the first steps are not dependent on each other, multiple jobs can be submitted to run concurrently.

Because larger programs are easier to manage, the client can continue to proceed with modifying datasets and additional reporting by simply building on the current program. Even if the program may never run completely from beginning to end, the resulting code is a complete program for the analysis. The list of macro calls at the end of the program, serves as an overview of the program flow.

REFERENCES

SAS Guide to Macro Processing, Second Edition, SAS Institute Inc., Cary, NC, USA, 1990.

TRADEMARKS

SAS is a registered trademark or trademark of SAS Institute Inc. in the USA and other countries.

Other brand and product names are registered trademarks or trademarks of their respective companies.

AUTHOR

Ed Jensen has been programming with SAS Software since 1981 when he was at Consolidated Freightways as project lead on the Workers' Compensation System. Since then, he has applied SAS Software at Freightliner Corporation, Federal Reserve Bank of San Francisco, and currently, Wells Fargo Bank, San Francisco. He has a BA in Chemistry
Address: PO Box 283, Corbett, OR, 97019
E-Mail: 74617.1435@compuserve.com
Phone: (503) 880-5463

Figure 1. Overview of Modular Program Structure

```

/*-----JCL-----*/
//STEP1      EXEC SAS
...
//MACROLIB DD DISP=SHR,DSN=MYLIB.SASAUTOS
//SASWRK1 DD DISP=OLD,DSN=MYLIB.WORKLIB1
//SASWRK2 DD DISP=OLD,DSN=MYLIB.WORKLIB2
//SASWRK3 DD DISP=OLD,DSN=MYLIB.WORKLIB3
//SYSIN DD *
/* -----SYSTEM OPTIONS TO ASSIST IN MACRO USE --- */
OPTIONS MAUTOSOURCE SASAUTOS=MACROLIB NOOVP;
/* MAUTOSOURCE - Enable macro autocall feature. */
/* SASAUTOS=MACROLIB - point to library containing prewritten macros */
/* NOOVP - No Overprint to remove duplicate error lines displayed in log */

OPTIONS MACROGEN DQUOTE;
/* MACROGEN - show end code generated by macro */
/* DQUOTE - needed in pre-v6.06 to allow use of macro parameters in quotes */

OPTIONS MLOGIC SYMBOLGEN;
/* MLOGIC - Displays resolution of macro logic such as %IF... */
/* SYMBOLGEN - Displays resolution of macro parameters
/* ----- PROGRAM CODE BLOCKED BY MACROS -----*/
%MACRO mergchan;
    DATA ...;
        MERGE...;
%MEND;

%MACRO PROFIT7;
    DATA ...;
        SET ...;
        WHERE...
%MEND;

%MACRO SORT7;
    PROC SORT ...
%MEND;
.
.
.
/* ***** */
/* EXECUTE THE MACROS */
/* ***** */
/*
    %wfbnatt1(MPR12PN6,MPR12,,MEANS File of Dec97 / Jan98);
/*options obs=1000;
%RPTPERTOP;
/*%mergchan: */ /* merges to get region information */
/*%killlib(SASWRK1);
/*%profit7;
/*%SORT7;
/*%killlib(SASWRK2);
/*%profit8;
/*%mergnew;
/*%mergcls;
/*%killlib(SASWRK3);
/*%mergall;
/*%setall;

...
%PRINTSOM(NWCLA,CNTAU);
%PRINTSOM(NWCLA,MISSA);
%WRITFLAT(NWCLA);
%chekflat;
/*%contsall(SASWRK1);
/*%contsall(SASWRK2);
/*%contsall(SASWRK3);

```

Figure 2.1 Managing Variable Lists

Variable list - suffix for list distinction, select variables depending on suffix

```
%macro mfvari(ys);
  avgbalm&ys.
  tnof&ys.
  svcchrg&ys.
  odrifee&ys.
  ddadvfee&ys.
  poschrg&ys.
  sn_fee&ys.
  othrfee&ys.
  pltrev&ys.
  ...
  admino&ys.
  %if &ys.=7 %then %do;
  admino&ys.I
  %end;
  brncho&ys.
  %if &ys.=7 %then %do;
  brncho&ys.I
  %end;
  direct&ys.
  ...
%mend;
```

Figure 2.2 Making Variable Lists Available for Multiple Uses

Varying Output Format

```
%macro putyrvar(pyr,fmt);
  avgbalm&pyr.    &fmt.
  tnof&pyr.       &fmt.
  svcchrg&pyr.    &fmt.
  odrifee&pyr.    &fmt.
  ddadvfee&pyr.   &fmt.
  poschrg&pyr.    &fmt.
  sn_fee&pyr.     &fmt.
  othrfee&pyr.    &fmt.
  pltrev&pyr.     &fmt.
  tel_c&pyr.      &fmt.
  atm_c&pyr.      &fmt.
  pho_c&pyr.      &fmt.
  chk_c&pyr.      &fmt.
  pos_c&pyr.      &fmt.
  ...
%mend;
```

Figure 2.3 Conditional Generation of Variable Lists

With Renames Option

```
%macro mfvar(ys,ren);
  %if &ren.=REN %then %do;avgbalm  = %end; avgbalm&ys.
  %if &ren.=REN %then %do;tnof     = %end; tnof&ys.
  %if &ren.=REN %then %do;svcchrg  = %end; svcchrg&ys.
  %if &ren.=REN %then %do;odtr_fee = %end; odtrfee&ys.
  %if &ren.=REN %then %do;odri_fee = %end; odrifee&ys.
  %if &ren.=REN %then %do;ddadvfee = %end; ddadvfee&ys.
  %if &ren.=REN %then %do;poschrg  = %end; poschrg&ys.
  %if &ren.=REN %then %do;sn_fee   = %end; sn_fee&ys.
  ...
%mend;
```

Figure 2.4 Maintaining Parallel Variable Lists

Parallel Lists for PROC MEANS

```
%MACRO MFVARS(MFSFX,XNIIADJ);
    avgbalm&MFSFX.  tnof&MFSFX.  svcchrg&MFSFX.  odriffee&MFSFX.
    sn_fee&MFSFX.  OTHRFEE&MFSFX.  DDADVFE&MFSFX.  POSCHRG&MFSFX.
    mocntr&MFSFX.
    %IF &XNIIADJ.=XNIIADJ %THEN %DO; %END;
    %ELSE %DO;
        brncho&MFSFX.
        direct&MFSFX.
    %END;
%MEND;
```

Figure 3 Controlling Test Environments

Using Macro Parameters to control One time/ test runs-Also several examples of using the variable list

```
%macro writflat(plclass,draft);
    ...
    %if &draft.=DRAFT %then %do;
    avgbalm7=(mocntr7/_freq_)*avgbalm7;
    %end;
    put aau      5.
    +1 auname $45.
    dist      10.
    div       10.
    _freq_    8.
    %putyrvar(7,14.2)
    %putyrvar(8,14.2)
    ;
    %mend;

PROC MEANS DATA=ALL;
CLASS FILECLAS;
VAR  ffreq  %putyrvar(7) %putyrvar(8);
OUTPUT OUT=MEANSF SUM=;

PROC PRINT DATA=MEANSF;
format _freq_ ffreq comma14.0
%putyrvar(7,comma20.2) %putyrvar(8,comma20.2);
```