

Handling production macros in the FRAME system: A Tutorial in the use of the SAS/AF and FRAME systems

Paul A. Thompson, Ph.D.

Division of Biostatistics, Washington University, St. Louis, MO

ABSTRACT

A production macro is defined as one which is used frequently over time, to perform a well-defined task which is not modified often. Such macros must be maintained to retain compatibility. It is important to control the use of the macro from improper use by non-developers. SAS/AF[®] methods, using the very helpful FRAME classes and widgets, can be used to control the access and usage of the macro. Here, explicit macro calls are replaced by storage of all macro values in one or more datasets, with a generalized call of the macro in the SCL code. The EXTENDED TABLE widget is used to access the dataset for adding values, processing the values through the macro, deleting values and retaining information about which values have been processed. The PMENU system is used to control various aspects of the system. An example is used to demonstrate the various aspects of this system.

INTRODUCTION

In using the SAS system for information processing (SAS), macros are commonly used to analyse data in a production mode, where analyses are run repetitively, frequently and the macro is seldom changed. The use of macros has many advantages:

1. It allows the user to concentrate upon the variable information, while keeping the constant and unchanging things, in fact, constant and unchanging.
2. Changes made in one run of the macro system will be incorporated every subsequent time the macro is use, if the macro is kept in a common place (such as a macro catalog or macro subdirectory).
3. Macro structuring frees up the designer from the specifics of one particular set of information and forces concentration upon ensuring that the macro will run under a variety of conditions.

There are down-sides to this, of course.

1. Macros can force the incorporation of code for exceptions, which are often quite tangential.
2. Any change in the list of calling parameters for the macro must be made in all existing macro calls. If

there are any appreciable number of such calls, this can be a tedious and boring process.

3. The macro is often intelligible only to the writer. If anyone else is to use the macro, an extensive effort must be undertaken to eliminate idiosyncratic and obscure approaches.

Using SAS/AF and FRAME methods to handle a production macro provides a protected interface for: 1. the macro parameter entry process; 2. the storage of macro parameters; 3. the actual macro calls; and 4. the use of the macro system by other persons. When a macro is a heavily used, production macro, this can be a good approach. If the macro is not used often, the amount of time needed to set up the macro in FRAME terms usually precludes this approach. However, a macro which has been set up in this manner become easy to use and difficult to destroy. These tradeoffs should be carefully considered before committing to the process.

The Key Idea

The key idea here is to replace macro calls with observations in one or more SAS datasets. A portion of the FRAME SCL program will translate the values of the dataset variables into macro variables. At that point, the macro can be called easily, using SUBMIT CONTINUE blocks in the FRAME SCL program. Changes in observations in the SAS dataset enables the actual conditions to run the macro to change. Changes in the single SAS macro can again be done easily. There is never a problem with an inconsistent argument-parameter relationship. In addition, the interaction of the user with the macro is "protected" by the FRAME entry, so that naive users can work with the macro without damaging the system.

Plan for the paper

This paper discusses the steps of converting a SAS MACRO into a FRAME GUI. These steps are:

1. Decomposing and restructuring the macro call.
2. Forming a main dataset and a number of secondary datasets to store the information.

3. Storing current information in these datasets.
4. Setting up the macros for easy use.
5. Setting up SCL code to manage the secondary dataset information.
6. Setting up SCL code to manage the main dataset information.
7. Transferring information from dataset variables to macro variables.
8. Setting up a menuing system for the macro.
9. Running the macro.
10. Modifying the macro.

A example case

A case study for this process will be used. I have recently been consulting with a number of companies in conjunction with meetings, in which I provide "on-site" instant data analysis using optical scanner methods. In this approach, we handle a large number of evaluation instruments, each of which have different numbers of items, on a generic scanning form. When this form is used, we handle all item and scale differences internally, using SAS formats to indicate item differences and other macro calls to indicate differences in the number of items per instrument, etc. Each of the different locations requires different titles to ensure that proper names are used. There was not one macro used, but actually three, and one macro had a variety of sections, controlled by other macro parameters.

STEP 1: MACRO ANALYSIS

Here are some examples of the macro calls which were used to run this macro:

```
%rtotal(_tape=0203, _gsr=0, _src=f1_h1,
_nitem=17, _items=1, _itemf=17, _maxv=6,
_fwin=0, _namef=fha17, _minv=0, _strt=1,
_ronly=1, _scale=hx, _sname=HAM-D, _markout=1,
_locat=flor1, _fmti=%str(hamdfmt.), _cent=1,
_longtab=1, _useadj=0, _namet=fha17);
```

In this macro, there are a number of classes of information, which we can consider thusly:

1. The same `_tape` (tape number) is unique to each tape;
2. The `_nitem` (number of items), `_items` (first item), `_itemf` (last item), `_fmti` (format name) and `_maxv` (maximum value) macros parameters are associated with the `_sname` parameter.
3. The `_scale` (specific scale term) parameter is associated with a given scale, but is incorporated into various names at each use. If a scale is used twice at a site, this value must be unique.
4. The `_locat` (subdirectory name) parameter indicates the subdirectory that the information is stored on.
5. A gold standard is used sometimes (`_gsr=1`) and not used in others. This depends on the tape and the scale.
6. The `_src` (source file name) indicates the name of the data file containing the information to be read in. The formats for the data are controlled partially by the `_nitem` parameter, partially by a stored macro read from a consistent location (a dataset called `TAILOR.SAS`, found in each subdirectory and read at the start of the macro run).

The same tape may be used at several sites.

Other information about each situation was handled idiosyncratically. Some of the information (a preset `TITLE` statement, for instance) was stored in a file which was input by the macro as one of the first operations performed.

STEP 2: FORMING DATASETS

In converting this to `FRAME`-base GUI, it became apparent that there are 4 separable and distinct sets of information in the macro call: 1. Instruments: 6 parameters; 2. Tapes: 2 parameters; 3. Sponsors: 5 parameters; 4. Locations: 9 parameters (which included information about the sponsor) If the values used in the macro call were converted into the values of a single dataset, this would require that many variables would have duplicated observations; in terms of database design, such a dataset would not be *first-order normal*. A dataset with such a characteristic has unique identifiers and no repeated information across observations. If duplicated variable values exist and needed to be changed, they would need to be changed in all observations. This would be difficult to do and is inefficient.

Thus, the first process to be accomplished lies in setting up a good database structure, which is first-order normal. The structure is shown here:

1. One dataset, `RAPRESP.RECORDS`, contains the key information about each invocation of the macro. It contains key variables which are used to link in information in other databases. Each secondary dataset was related to the primary dataset by a

one-many relationship (one record in each secondary dataset could be related to one or many records in the primary dataset).

2. There are several secondary databases, which are all subsidiary to the main database:

- (a) The tape dataset, `RAPRESP.TAPES`, contains tape names and numbers.
- (b) The scales dataset, `RAPRESP.SCALES`, contains scale information, such as format names, number of items, and lowest and highest values.
- (c) The contacts dataset, `RAPRESP.SPONSOR`, contains contact information about the firms with which we work on these projects.
- (d) The sites dataset, `RAPRESP.SITES`, contains information about the locations in which the presentations are given.

At this point, the essential macro has been changed. The macro is called with the following macro call:

```
%let _sb=1;%let _se=4;  
%_rtotal(_tape=0203,_scale=hx,_locat=flor1);
```

The remaining portions of the macro have been converted into observations in various datasets. Each dataset is set up with relevant variables to contain the macro variable information.

STEP 3: STORING INFORMATION

At this point, the various macros calls are surveyed. If they are to be retained in the system, the information in the macro calls must be transferred into these datasets. This is done using standard SAS tools, is tedious, and needs no further discussion.

STEP 3: SETTING UP THE MACROS

For the most easy maintenance of the macros, they should be set up to be stored on a compiled macro catalog. This is done using the `MSTORED` option:

```
OPTIONS MSTORED SASMSTORE=MAIN;
```

In addition, macros must be set up to be stored on the macro catalog:

```
%macro _prln(_lnx)/STORE DES="Print a line";  
HLA=&_lnx.;PUT HLA;  
%mend _prln;
```

Using this mechanism, macros do not need to be read in and processed, but are always available in a compiled fashion.

STEP 5: THE SECONDARY DATABASES

Within the SCL code for the GUI module being constructed, some method must be chosen to manage the information in each secondary dataset. This can be easily done using either `FSEDIT` or `FSVIEW` (either of which has direct implementation in SCL, and may be called as a function). In some cases, more labor-intensive approaches can be set up using `EXTENDED TABLES` implementations. These are only needed when special things need to be done, and often take much more programming to accomplish. Here is an example of the `FSVIEW` statement used to edit one of the secondary datasets:

```
call fsview("NAMEXI","edit");
```

This is a very simple approach. Within the `FSVIEW` process itself, new observations may be added, old ones deleted or modified, and the dataset saved easily.

STEP 6: THE MAIN DATABASE

In this particular case, a `FRAME` tool which was flexible and extensible was needed. This suggests the use of the `EXTENDED TABLE` approach. This tool is very flexible, allowing the user to determine whether 1 or 500 different calls are examined at a time.

Flexibility comes at a price, of course; extended tables are somewhat difficult to work with. I have evolved a general approach to the use of extended tables, which is generally flexible and pretty easy to use. In addition, since it uses arrays to handle all references to variables, widgets and lists, it is also easy to move from one location to another.

My approach proceeds as follows:

1. The `EXTENDED TABLE` widget is defined. I always define the widget using the following choices:
 - (a) In the `Options` section, I check `Dynamic Table`, `Selection list` and `Vertical scroll bar`. I sometimes check `Horizontal scroll bar`, but not often.
 - (b) In the `Table size` section, I set 0 for `Maximum rows`, 0 for `Maximum spacing` and 0 for `Row spacing`.
 - (c) In the `Putrow options` section, I usually select `Run GETROW` for all displayed

rows. Sometimes, this can really slow down processing, if many rows are to be displayed. In those cases, I select Run GETROW for all selected rows. However, this is done after observing the results of the process.

- (d) In the Selection list options section, I use a large number of cases for the Maximum selections choice. In this case, I use 101. This is definitely overkill for the present case, but it does not hurt to have a little overkill at this point.
2. The widgets for the different variables are defined. For the purposes of this example, I will define 2 numeric value widgets and 2 character value widgets. For this example, these are called S_NA and S_NB (for the numeric widgets) and S_CA and S_CB (for the character widgets).
3. The values from the dataset are called VNA, VNB (numeric variables), VCA and VCB. These variables must exist in the dataset (stop me if I am being too obvious).
4. One final element allows different variables to be read from the file than are posted. In many cases, the designer wishes to read all variables, but only post a portion. For this purpose, a secondary set of arrays is defined which contain variable numbers. Variables on this list are posted and handled in the extended table. Others are read from the file.
5. The values from the datasets are transferred into lists for each variable. I use lists for all temporary handling of variables for three reasons:
 - (a) The dataset can be opened during the transfer, then closed for the use of the dataset. This ensures that I am always clear about the status of the dataset. In multi-user systems, it is also important to control the open status of the dataset carefully as well.
 - (b) The number of cases which are being used at the time is always very clear.
 - (c) When saving values, the values in the lists can be easily transferred back from the lists to the dataset.

In this case, the lists are called L_NVA, L_NVb, L_CVA and L_CVB.

6. Widgets, lists and variables from the datasets are all handled using arrays. This gives an enormous amount of flexibility in the use of the EXTENDED TABLE widget. In all cases, separate arrays must be used for character and numeric variables (and their associated widgets and lists). Here are the various arrays for the current situation:

```

/* ***** */
/* Widget name arrays - A_DN, A_DC */
/* Widget arrays - A_WN, A_WC */
/* Variable name array - A_VN, A_VC */
/* List name array - A_LN, A_LC */
/* Position number array - A_PN, A_PC */
/* ***** */
ARRAY A_DN(2)$ _TEMPORARY_ ("S_NA" "S_NB");
ARRAY A_WN(2) S_NA S_NB;
ARRAY A_VN(2) VNA VNB;
ARRAY A_LN(2) L_NVA L_NVb;
ARRAY A_PN(2) _TEMPORARY_ (1,2);
ARRAY A_DC(2)$ _TEMPORARY_ ("S_CA" "S_CB");
ARRAY A_WC(2) S_CA S_CB;
ARRAY A_VC(2) VCA VCB;
ARRAY A_LC(2) L_CVA L_CVB;
ARRAY A_PC(2) _TEMPORARY_ (1,2);

```

7. Data are transferred from the variables in the RAPRESP.RECORDS dataset to the lists using the following code. After this section is processed, an additional blank item is added to each list, to enable more information to be added to the values in the table. The code is as follows:

```

/* ***** */
/* Begin by initializing the list */
/* ***** */
do i=1 to dim(a_lc);a_lc[i]=makelist(0);
end;
do i=1 to dim(a_ln);a_ln[i]=makelist(0);
end;
/* ***** */
/* Read from the dataset ds_rec until */
/* the end of the dataset */
/* ***** */
r_vz=0;do until(r_vz);r_vz=fetch(ds_rec);
if (r_vz) then LEAVE;
/* ***** */
/* Read from the dataset ds_rec until */
/* the end of the dataset */
/* First, read the char values */
/* ***** */
s_cnt+1;
do s_i=1 to dim(a_vc);
r_vc=setitemc(a_lc[s_i],getvarc(ds_rec,
varnum(ds_rec,a_vc{s_i})),s_cnt,"Y");
end;
/* ***** */
/* Next, read the num values */
/* ***** */
do s_i=1 to dim(a_vn);
r_vc=setitemn(a_ln[s_i],getvarn(ds_rec,
varnum(ds_rec,a_vn{s_i})),s_cnt,"Y");
end;
end;

```

```

/* ***** */
/* Add an extra element to each list */
/* is blank, to allow the user to add */
/* values dynamically */
/* ***** */
link ADDONE;

```

8. The GETROW section transfers information from the lists to the widgets. The number of items in the first list is used to control the number of items presented in the extended table:

```

GETROW:
/* ***** */
/* Dynamic tables must be informed when */
/* there is no more data to post */
/* ***** */
if (_currow > listlen(a_cls{1})) then do;
call notify("ET_SCALE","_ENDTABLE_");
end;
else do;
/* ***** */
/* s_tx: 1 if cursor was on num var */
/*       2 if cursor was on char var */
/* This is set during MAIN section */
/* ***** */
if (s_curp = _currow) then do;
if (s_tx=1) then
call notify(a_wc{s_p},"_CURSOR_");
if (s_tx=2) then
call notify(a_wn{s_p},"_CURSOR_");
end;
/* ***** */
/* Post numeric values */
/* If the value is missing, set _BLANK_ */
/* There is probably a better method */
/* ***** */
do s_i=1 to dim(a_wn);
a_wn{a_pn{s_i}}=
getitem(a_ln{a_pn{s_i}},_currow);
if (getitem(a_ln{a_pn{s_i}},
_currow)=.) then
a_wn{a_pn{s_i}}=_BLANK_;
end;
/* ***** */
/* Post numeric values */
/* If the value is missing, set _BLANK_ */
/* There is probably a better method */
/* ***** */
do s_i=1 to dim(a_wc);
a_wc{a_pc{s_i}}=
getitemc(a_lc{a_pc{s_i}},_currow);
if (getitemc(a_lc{a_pc{s_i}},
_currow) = " ") then
a_wc{a_pc{s_i}}=_BLANK_;
end;
end;

```

RETURN;

9. Data are recovered from the widgets and transferred back to the lists in the PUTROW section:

```

PUTROW:
/* ***** */
/* Remember the current row position */
/* ***** */
s_curp=_currow_;
/* ***** */
/* Recover numeric values */
/* ***** */
do s_i=1 to dim(a_wn);
r_vc=setitem(a_ln{a_pn{s_i}},
a_wn{a_pn{s_i}},_currow_,"Y");
end;
/* ***** */
/* Recover char values */
/* ***** */
do s_i=1 to dim(a_wc);
r_vc=setitemc(a_lc{a_pc{s_i}},
a_wc{a_pc{s_i}},_currow_,"Y");
end;
/* ***** */
/* If there is a value in the blank, */
/* add a new blank to each list */
/* ***** */
if (s_curp = listlen(a_lc{1}) &
getitemc(a_lc{1},s_curp) ^= "") then
link ADDONE;
RETURN;

```

10. Finally, data are replaced in the dataset. Generally, I assume that if the controlling variables are found in the dataset, other values are to be replaced. If the controlling variables cannot be found, a new observation is to be added to the datasets. The statements are as follows:

```

do s_cnt=1 to listlen(a_lc{1});
/* ***** */
/* Ensure that this is a real obs */
/* ***** */
if (getitemc(a_lc{1},s_cnt)
not in (""," ")) then do;
/* ***** */
/* Check for the values of the obs */
/* ***** */
r_vc=where(ds_rec,"direc="||
quote(getitemc(a_lc{4},s_cnt)));
if (r_vc) then _msg_=sysmsg();
r_vc=fetch(ds_rec);
/* ***** */
/* Obs does not exist - add new one */
/* ***** */
if (r_vc ^= 0) then r_vc=append(ds_rec);
/* ***** */
/* Transfer char values to dataset */
/* ***** */

```

```

/* ***** */
do s_i=1 to dim(a_vc);
call putvarc(ds_rec,varnum(ds_rec,
  a_vc{s_i}),getitemc(a_lc{s_i},s_cnt));
end;
/* ***** */
/* Transfer num values to dataset */
/* ***** */
do s_i=1 to dim(a_vn);
/* ***** */
/* Set up the default value with */
/* missing value for this var */
/* ***** */
s_bval=((s_i in (1))>0)*1+
      ((s_i in (2))>0)*0;
if (getitemc(a_lc{s_i},s_cnt) = .) then
  r_vc=setitemc(a_lc{s_i},s_bval,s_cnt);
call putvarn(ds_rec,
  varnum(ds_rec,a_vn{s_i}),1);
end;
/* ***** */
/* Update dataset to save values */
/* ***** */
r_vc=update(ds_rec);
end;
end;
r_vc=where(ds_rec);
r_vc=sort(ds_rec,"direc","/nodupkey");
if (r_vc) then _msg_=sysmsg();

```

The only flaw to this general approach is if the controlling variable values are changed during the course of the processing of the active lines, the system will not consider these lines to be replacements of old lines, but rather new lines. This leads occasionally to unnecessary duplication of entries. This is not a serious problem, as duplicate lines can easily be deleted.

STEP 7: SETTING UP MACRO VARIABLES

There are two approaches to the problem of obtaining information from secondary datasets. In both cases, the macro values are inserted into macro variables using CALL SYMPUT. Both approaches have similar needs for generalizability in approach. In both cases, the macro variables will have global scope.

1. The datasets can be opened in the calling SCL program. Macro variables set up in the calling SCL program have global scope with respect to a macro run within a SUBMIT CONTINUE block in that SCL program.
2. The dataset can be opened in a SUBMIT CONTINUE block which runs the macro itself, either as a part of

the macro, or prior to the macro being used. In this case, the macro variables should have global scope as well, although this is not as clear. In this case, the dataset is opened using a DATA _NULL_ construction.

The SCL approach

Here is some code which functions to work with the contents of a given secondary database to set up macro variables in the SCL program:

```

/* ***** */
/* SCL Code to convert dataset value to macro */
/* ***** */
ds_scl=open("rapresp.scales");
r_vc=locatec(ds_scl,varnum(ds_scl,"SCALE"),
  s_scalnm);
s_nitem=getvarn(ds_scl,varnum(ds_scl,"NITEM"));
call symputn("_nitem",s_nitem);
if (ds_scl) then ds_scl=close(ds_scl);

```

This is generally straightforward, and can be shortened slightly (embedding the GETVARN construction in the CALL SYMPUT statement. However, it can become cumbersome and difficult, if there are a large number of variables. Here is a somewhat more complex case, which illustrates a general approach to the transfer of values to macro variables:

```

/* ***** */
/* SCL Code to convert dataset value to macro */
/* ***** */
\begin{enumerate}
\item
\end{enumerate}
array a_nvn (4) $ _TEMPORARY_
  ("NITEM" "FITEM" "LITEM" "MVALUE");
array a_nmn (4) $ _TEMPORARY_
  ("_nitem" "_items" "_itemf" "_marv");
array a_nsn
  (s_nitem s_items s_itemf s_marv);
...
INIT:
...
ds_scl=open("rapresp.scales");
r_vc=locatec(ds_scl,varnum(ds_scl,"SCALE"),
  s_scalnm);
do s_i=1 to dim(a_nvn);
a_nsn{s_i}=getvarn(ds_scl,
  varnum(ds_scl,a_nvn{s_i}));
call symputn(a_nmn{s_i},a_nsn{s_i});
end;
if (ds_scl) then ds_scl=close(ds_scl);

```

There are several points to this code:

1. The array specifications are easily extended to include new variables. The user must be sure to keep numeric and character variables separated, however.
2. The use of a single CALL SYMPUT can reduce coding errors.
3. The DIM specification ensures that all of the variables are converted into macro variables, regardless of the number of variables.

This can be placed at several points in the SCL code. Naturally, the value of the variable `s_scalnm` must be known at the point that the code is executed.

The SUBMIT CONTINUE block

The second approach lies in performing the conversion from dataset values to macro variable within the SUBMIT CONTINUE block, inside a DATA _NULL_ construction. The data from the main dataset `RAPRESP.RECORDS` and each of the subsidiary datasets are obtained by code such as the following, included in the body of the macro:

```
DATA _NULL_; /* Select cases first scale &_scn*/
SET RAPRESP.SCALES(WHERE=(SCALE=&_scn));
/* ***** */
/* Handle character variables */
/* ***** */
CALL SYMPUT("_fmt1",FMTNM);
CALL SYMPUT("_xds1",DS1);
CALL SYMPUT("_xds2",DS2);
/* ***** */
/* Handle numeric variables */
/* ***** */
CON=1;
DIFF=MAX-MIN+1;
CALL SYMPUT("_cont",COMPRESS(PUT(CON,1)));
CALL SYMPUT("_minv",COMPRESS(PUT(MIN,4)));
RUN;
```

This code is difficult to extend, and is not systematic and reusable. Reusable code can be generated by the use of arrays; in the array approach, one statement (the array itself) must be rewritten to add new variables to the process. Rather than using a single CALL SYMPUT statement, the use of arrays enables the designer to generalize the process. This is done as:

```
DATA _NULL_; /* Select cases first */
SET RAPRESP.SCALES(WHERE=(SCALE=&_scn));
/* ***** */
/* Handle character variables */
/* Arrays define vars, SYMPUT converts */
/* ***** */
```

```
ARRAY MC(3) $ _TEMPORARY_
      ("_fmt1","_xds1","_xds2");
ARRAY VC(3) FMTNM DS1 DS2;
DO I=1 TO DIM(MC);CALL SYMPUT(MC{I},VC{I});END;
/* ***** */
/* Handle numeric variables */
/* Arrays define vars, SYMPUT converts */
/* ***** */
ARRAY MN(12) $ _TEMPORARY_ ("_cont","_minv",
      "_marv","_items","_itemf","_nitem","_marit",
      "_xnbr","_xis1","_xis2","_rif1","_rif2");
ARRAY VN CON MIN MAX ITEMS ITEMF NITEM
DIFF NBR ITS1 ITS2 ITF1 ITF2;
DO I=1 TO DIM(MN);
      CALL SYMPUT(MN{I},COMPRESS(VN{I},8));END;
RUN;
```

Adding new variables into either of the systems (character or numeric) requires the addition of the variable name to the name array (VNS or VCS), and adding the name of the macro and incrementing the count of the macro array (MNS or MCS) into that macro array. The use of the DIM specification ensures that the translation is done automatically. Care must be taken to ensure that the number of variables in the different arrays are kept synchronized. A two-way array cannot be used for this process, because variable types cannot be mixed in SAS arrays.

Thus, this phase of the conversion process is completed by using a set of statements like those shown above, for the main datasets `RAPRESP.RECORDS` and for each of the subsidiary datasets (`RAPRESP.TAPES`, `RAPRESP.SPONSOR`, `RAPRESP.SCALES`, `RAPRESP.SITES`). What is left unstated is some error checking information; since important macro variables are set in each of the selection DATA _NULL_; data steps, the designer must include macro variable controls for failure to select appropriate information.

STEP 8: SETTING UP A MENU SYSTEM

The PMENU approach to handling access to the macro can make a very professional looking GUI look even better, and enables the user to have good control over the window. Essentially, the PMENU approach enables the user to control the user's access to both standard SAS utilities, as well as present the user with a clear interface for the choices to be made during the macro use.

The PMENU used for this analysis is shown here:

```
/* ***** */
PROC PMENU CAT=MAIN.DWMNEW;
MENU RAPRESP;
ITEM "Exit" SELECTION=E_X;
ITEM "Locations..." MENU=LOC_M;
```

```

ITEM "Analysis..." MENU=ANALY_M;
ITEM "Edit..." MENU=EDIT_M;
ITEM "Other info..." MENU=INFO_M;
MENU EDIT_M;
ITEM "Source" SELECTION=E_F
ITEM "Names" SELECTION=E_N;
MENU INFO_M;
ITEM "Scales" SELECTION=I_S;
ITEM "Locations" SELECTION=I_L;
ITEM "Tapes" SELECTION=I_T;
MENU LOC_M;
ITEM "Select" SELECTION=L_S;
ITEM "Use selected" SELECTION=L_U;
MENU ANALY_M;
ITEM "Read and check" SELECTION=A_A;
ITEM "Initial run" SELECTION=A_B;
ITEM "Full prep" SELECTION=A_C;
SELECTION A_A "A_A";
....
....
SELECTION S_V "S_V";
RUN;

```

In using the PMENU approach, there are a number of choices. I try to simplify things by setting up a SELECTION statement which has the same command output as the SELECTION specification in the ITEM statement. This is simply my style, however.

This PMENU specification must be invoked during the INIT: portion of the SCL program (to invoke the menu from the beginning of the display of the window). Use of the NOAWSMENUMERGE option early in the INIT: block:

```

/* ***** */
OPTIONS NOAWSMENUMERGE MSTORED SASMSTORE=MAIN;

```

will ensure that your menu is not merged with the default SAS macro.

This menu is actually invoked using the following codelet:

```
r_vc=pmenu("main.dwmnew.rapresp");
```

The menu operates by placing a term on a stack accessed with the WORD function as:

```

/* ***** */
/* Get the command selected by the menu */
/* Clear remaining (?) commands */
/* ***** */
s_pmenu=word(1);
call nextcmd();
select (s_pmenu);
when ("A_A") link READCHK;

```

```

when ("A_B") link INITRUN;
...
otherwise;
end;

```

STEP 9: RUNNING THE MACRO

In running the macro, the elements described above are all put together and used in an integrated way. Here is the basic method:

1. The user uses the menu system to get the various invocations of the macro which are to be run into the EXTENDED TABLE interface. As few as 1 invocation may be included into the interface, with no actual limit to the number which may be selected.
2. The user selects the particular cases to be analyzed. This is done by clicking on the row containing the case to be run.
3. The user uses the menu system to select the type of analysis to be performed.
4. The menu item is sent to the MAIN block for processing, using the menu-handling code described above.
5. The macro is run out of the PMENU structure. There are several approaches to running the macro, which differ from one another by the use of various macro control variables to run specific sections of the macro using a %if (&var.) %then %do; approach. Here is the overall mechanism to run the macro:

```

do s_i=1 to listlen(a_cls{1});
call notify("ET_SCALE","_ISSEL_",s_i,s_selx);
if (s_selx) then do;
link SETMACS;
link OVERRUN;
end;
end;

```

This operates by checking each row of the displayed table and determining if the row is selected. If the row is selected, it is run. For those rows, the system sets up macro variables using the mechanism described above for the conversion process. Following macro variable conversion, the system uses the OVERRUN block to run the macro:

```

OVERRUN:
r_vc=libname("C"||s_sclctr,
s_rrt||"\\"||s_rxt||"\\"||s_dirct);
SUBMIT CONTINUE;
options fmtsearch=(rresplib library);
%macro _overrun;

```

```

%put *.*****.
%put *.RTOTAL macro start *;
%put *.*****.
%if (&_zswa | &_zswb) %then %do;
%rtotal (_tape=&_tape., _secb=&_zsb.,
 _sece=&_zse., _cencoun=&_cencoun,
 _ronly=&_ronly., _centers=&_centers.,
 _longtab=&_longtab., _markout=&_markout,
 _locat=&_direc., _extadd=&_extadd.,
 _extname=&_extname., _extpos=&_extpos.,
 _scale=&_scletr., _src=&_src., _run=&_orun.,
 _sname=&_sname.);
%end;
%mend _overrun;
%_overrun;
ENDSUBMIT;

```

The macro is run after embedding it inside another macro. This enables the use of macro %do loops and other such control structures.

6. The print stream to the OUTPUT window and the LOG window are diverted to files using the PRINTTO mechanism. Several things can be done with these streams: they can be printed using the External File Viewer widget, they can be routed to a printer, or they can be run through a input process to look for problem codes. In this case, the External File Viewer is used to view the log, and there is no output to be examined.

STEP 10: MODIFYING THE MACRO

To ensure that the system can be easily used, a method for modifying and compiling macros should be included. I store macros as a SOURCE module on the same catalog as the FRAME and SCL modules; whenever I export the catalog, I export the macro SOURCE module as well. By running the macros, I store them on a macro catalog on a continually-activated system library.

SAS has several methods for working with lines of code. One could %include such files into the PROGRAM EDITOR, but this is not a well-protected interface. Another approach lies in the use of the PREVIEW buffer, which includes an editor much like the PROGRAM EDITOR. Here is an example:

```

/* ***** */
/* Identify the file (z_macros.source) */
/* Include file into preview buffer */
/* Set up three lines into a LEGEND window */
/* Start up the LEGED window */
/* Add lines #s into the edit buffer */
/* Edit the file - control goes there */
/* After editing, copy lines back to file */

```

```

/* Clear preview buffer */
/* Set up a SUBMIT CONTINUE block */
/* Include the macro file in, compile it and */
/* store it on the macro catalog */
/* ***** */
fr_ni=" ";r_vc=filename(fr_ni,
 "main.dwmnew.z_macros.source","catalog");
r_vc=preview("include",fr_ni);
do l=1 to 1;call putlegend(l,a_expl{1});end;
call legend("Explanation");
s_cmd="num on";call exccmd(s_cmd);
r_vc=preview("edit","Macros Analysis");
r_vc=preview("file",fr_ni);
r_vc=preview("clear");
submit continue;
options source source2;
proc printto print="__res.out"
 log="__res.log" new;run;
filename macin catalog
 'main.dwmnew.z_macros.source';
%include macin;
proc printto;run;
endsubmit;

```

Using this method, the user is in very good control of the editing process of the macro. You never save the macro back into the second file that is opened!

There is one important thing that must be remembered in using this approach for editing: either the preview buffer or some aspect of the transfer mechanism appears to have a fixed and immutable line length of 80 characters. This can have a spectacular effect on a macro with lines longer than 80 character, of course. In my case, terminal */ characters, which end comments, were often placed in columns 82-83. In addition, %do statements were often found out there. Fixing such a disaster can be quite difficult.

CONCLUSIONS

Converting a macro into the protected mechanism of the GUI can be a time-consuming process. It probably took about 24 hours for the initial conversion process. In total, possibly 30 hours (and this may be an underestimate) were required for this conversion. For macros which must be used by others than the author, this time expense is necessary. In addition, macros set up in this way are more durable and less easily "broken" than other ways of maintaining the system. In addition, this approach centralizes the processing. All in all, such an approach can be helpful, but is time-consuming to initially set up.

ABOUT THE AUTHOR

Paul A. Thompson, Ph.D., is Research Associate Professor of Biostatistics, Washington University, St. Louis, MO. He has been a SAS user for 23 years, since graduate school at the University of North Carolina, Chapel Hill, NC. His interests in the SAS include data management for clinical trials, linear models and mixed models methods, multivariate methods, IML methods and using the SAS in conjunction with other systems for intelligent high-volume paper and form production, using SAS, TeX and L^AT_EX. He may be contacted at:

Paul A. Thompson, Ph.D.
Division of Biostatistics
Washington University School of Medicine
Campus Box 8067, 660 Euclid Av.
St. Louis, MO 63110-1093
Phone: (314) 747-3793
FAX: (314) 362-2693
e-mail: pat@po.cwru.edu

ACKNOWLEDGMENTS

SAS, SAS/AF, SAS/FRAME, SAS/FSEDIT and SAS/FSVIEW are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

REFERENCES

SAS Institute Inc. (1996). *SAS/AF® Software: FRAME Class Dictionary, Version 6, First Edition*. Cary NC: SAS Institute Inc.

SAS Institute Inc. (1996). *SAS/AF® Software: Usage and Reference, Version 6, First Edition*. Cary NC: SAS Institute Inc.