Using %WINDOW to Create Interactive and Completely Re-Usable Code with SAS/BASE

Wade Bannister, Arizona State University, Tempe, AZ

ABSTRACT
Many times we have code for a process that we want to use on numerous projects, with only a small set of definitions or user options, which need to be updated in order to apply it to a new project. Changing libnames, dataset names, input and output files, and even process options can all be handled using the %WINDOW environment. Requiring only SAS/BASE and an understanding of macros, I demonstrate a method of creating an interactive set of windows to allow an end-user to simply enter values and make choices. I also demonstrate code to validate this user input to ensure that it will run as expected. This will allow you to create code that is completely re-usable, never needs to be reviewed after its initial creation, and can be used by employees with limited (if any) SAS experience.

INTRODUCTION
While products such as SAS/AF make it relatively easy to develop a user interface, it is still possible to create rudimentary-looking user interfaces using only SAS/BASE. Creating such a user interface will increase code flexibility, streamline peer review processes, and ultimately reduce development time.

CREATING SIMPLE WINDOWS
In order to get started creating a user interface, one first needs to have a grasp of the %WINDOW and %DISPLAY capabilities within SAS/BASE. The %WINDOW statement creates the definition for the window to be displayed, while the %DISPLAY is called from within a macro and displays the specified window.

Sample code to define a simple window which only displays text to the user is as follows:

```
%window xnames color = orange
/* dimensions of window */
icolumn = 15  irow = 10
columns = 90  rows = 30

/* Window content */

#3     @35 "Title for Window"
           attr = rev video
#5     @10 "Here is some more text"
           attr = underline
       ;
```

Here we have created a window definition to be referenced as “xnames” 30 rows by 90 columns in size. The “#” gives the row number within the window at which to display the text, while the “@” gives the column number to display the text. The text to be displayed in the window is placed inside the quotes, and an optional attribute statement follows, allowing such options as reverse video (`rev_video`) or underlining (`underline`).

Adding user input capabilities to this window is not difficult, and only requires the addition of the following:

```
#7     @10 "* Directory for work files:"
       @55 NewWork 45
       attr = underline required = yes

#9     @10 "* Location & filename of text"
#10     @10 "  file:" @55 TxtFile 45
       attr = underline required = yes

#12    @10 "* Location & filename of
       provider type"
#13    @10 "  definitions file:"
       @55 ProvFile 45
       attr = underline required = yes
       ;
```

This will create a window with three lines for user input, and each line will be stored in a macro variable. In the above example, the “@” following the quoted string specifies the area in the window which the user may type, and this value is stored in the macro variable name which follows. Thus, in line seven of the window, we see that at column 55 the user will type in a value which will be stored as the macro variable “NewWork”. The number that follows the variable name defines the number of spaces allowed for user input, and when followed by the “`attr = underline`” option, this defines the length of the “blank” in which the user types. The “`required = yes`” forces the user to input at least some value before exiting the window.

Displaying this window is simply performed by:

```
%macro macroname;

%* other code;

%display xnames;

%mend macroname;
```

The screen that is displayed by this window definition is shown in Figure 1.

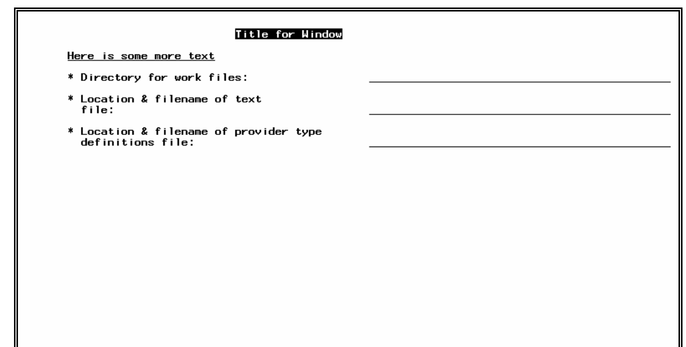

Figure 1: Basic user input screen

PRE-LOADING PREVIOUS USER INPUT
If you are building a system that has numerous user input fields or will be re-run several times with few changes required by the user, it would be advantageous to store the input in a permanent dataset and then reload it the next time the program is run and

pre-fill the user input fields. This will require the user to edit only those fields that require changes and they can merely click through the rest.

One such way of accomplishing this is through the following:

```
%* load any previous user input, if
available;

data _null_;
dsname = "symwork.user_parameters";
if (exist(dsname)) then do;
set symwork.user_parameters;
call symput(ParameterName, ParameterValue);
end;
run;

%display WindowName;

%* build dataset holding all user
parameters for peer review purposes;

data user_parameters;
NewWork = "&NewWork";
TxtFile = "&TxtFile";
ProvFile = "&ProvFile";
run;

proc transpose data=user_parameters
   out=user_parameters2
   (rename = (_name_ = ParameterName
   col1 =  ParameterValue));
var NewWork TxtFile ProvFile;
run;
```

Also note that printing this dataset containing all the parameter values provides an excellent paper trail for peer review purposes, and there will always be a record of exactly what each user inputs.

VALIDATING USER INPUTS

Since exactness is often required in the user input, adding a programming component to validate and change, if necessary, the user inputs can save many hours of frustration for new users. This can be accomplished relatively easily using the dataset created in the previous section which holds all of the user inputs.

For example:

```
data newwork.user_parameters;
set user_parameters2;
if _n_ = 1 then do;

%* insert leading and trailing quotes;
%* for variables that need them;
if substr(ParameterValue,
length(ParameterValue), 1) ~= "'" then
  ParameterValue = trim(ParameterValue)||
  "'";
if substr(ParameterValue, 1, 1) ~= "'" then
  ParameterValue = "'" || ParameterValue;
end;
if _n_ > 2 & _n_ <= 15 then do;
%* remove leading or trailing quotes for;
%* variables that should not have them;
if substr(ParameterValue, 1, 1) = "'"
 then do;
  substr(ParameterValue, 1, 1) = "";
  ParameterValue = left(ParameterValue);
 end;
if substr(ParameterValue,
length(ParameterValue), 1) = "'"
  then do;
    substr(ParameterValue,
    length(ParameterValue), 1) = "";
    ParameterValue = trim(ParameterValue);
  end;
end;

run;

data _null_;
set newwork.user_parameters;
call symput(ParameterName, ParameterValue);
run;
```

The last "`data_null`" step actually reloads the newly fixed user inputs back into the macro variables so that the rest of the program uses the edited values rather than the potentially erroneous values originally input by the user.

CREATING A SET OF DEPENDENT WINDOWS

To most effectively utilize the %WINDOW capabilities, it is often desirable to have a system of windows, all driven by a single main menu. For example, suppose that there are a set of filenames for raw text files to be imported, a set of destinations to write ODS reports to, and a set of program options such as the number of categories to subset upon. Not all of these values need to be re-input every time the program is run, and often none of these values will need to be changed. Rather than force the user to tab through numerous pointless fields, a main menu can be created from which the user can choose sub-menus to edit.

Such a "parent" window can be created as follows:

```
%window frontend color = magenta
       icolumn = 15 irow = 10
       columns = 40  rows = 30
#3 @10 "Select which (if any) options need
        to be set"
#8 @10 option1 1 attr=underline
       @12 "Libraries and Data Filenames"
#10 @10 option2 1 attr=underline
       @12 "Report Locations and Filenames"
#12 @10 option3 1 attr=underline
       @12 "Other Report Options"
#15 @10 "*** After entering choice(s),
       press [ENTER]";
```

The user can select any of these options by typing any letter (for example, an "X" in the blank before the option description. An example of the window produced by this definition is shown in Figure 2.

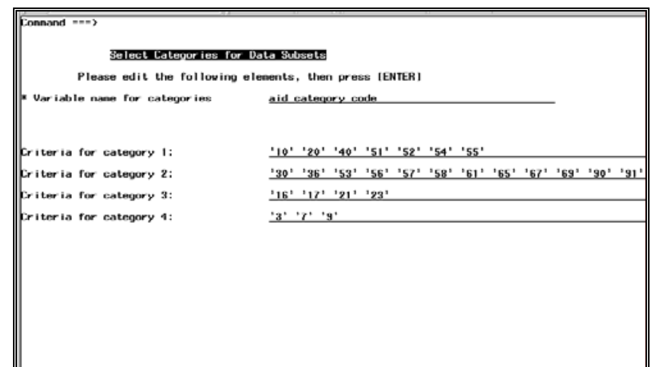

Figure 2: Dynamically generated window

Then, this window is called with the %DISPLAY first in the macro, and subsequent windows are displayed depending upon the options selected by the user. For example:

```
%macro setup;
%* other code;
%display frontend;

%if &option1 ~= &null %then %display
       xnames;
%if &option2 ~= &null %then %display
       reports;
%if &option3 ~= &null %then %display
       other;

%* remaining macro code;
%mend setup;
```

DYNAMIC WINDOW CREATION

It is also possible to create windows dynamically based on options entered in previous windows. For example, one user input window may retrieve information regarding how many data subsets to create, and the next window would then gather user input regarding the output filename for each subset. Since the number of filenames is not known until the user inputs this value in the previous window, the %WINDOW definition must be created dynamically, through macro code. However, since the %WINDOW definitions are compiled prior to them actually being run, the first %WINDOW definition must be both defined and run before the second dynamic %WINDOW definition can be executed. For example:

```
%macro cat input;
%do i = 1 %to &numcats;
#%eval(10 + 2*&i) "Criteria for
  category &i:" @40 cat&i 65
  attr = underline required = yes
%end;
%mend cat_input;

%window cats color = yellow
       icolumn = 15 irow = 10
       columns = 90 rows = 30
#3  @15 "Select Categories for Data
       Subsets" attr = rev video
#5  @10 "Please edit the following
       elements, then press [ENTER]"
#7  @1 "* Variable name for categories"
    @40 catvar 45 attr = underline
        required = yes
%cat_input;
;
```

In the above example, the &numcats variable was input in a previous window, and is used in this window to dynamically generate that many lines for user input. Each user input is then stored in the macro variables &cat1, &cat2, etc.

CONCLUSION

While the appearance of windows created by the %WINDOW and %DISPLAY capabilities are often rudimentary in appearance, they can be quite powerful when applied with a little ingenuity. While surely the SAS/AF package offers substantial improvements over this approach, the %WINDOW method can be an effective one in environments where only SAS/BASE is available.

CONTACT INFORMATION

Wade Bannister
Arizona State University
School of Health Administration and Policy
PO Box 874506
Tempe, AZ 85287-4506

Phone: (480) 965-1623
Fax: (480) 965-6654
Email: wade.bannister@asu.edu