ABSTRACT

This paper will describe various ways to trouble-shoot Version 5.03 SAS/AF and SAS/FSP applications. Common problems encountered in the writing of an application and how to avoid these problems in the future will be discussed. Also, how to avoid or eliminate common problems associated with the writing of an SCL based application will be illustrated.

TROUBLE-SHOOTING SAS/AF APPLICATIONS

SOME HELPFUL HINTS

Use the SWAP command when building an AF application to swap between BUILD windows. The NEXT command will display all open windows, including display manager windows. Place all entries for your application in one catalog. Then you do not have to specify the libref and catalog names on the ATTR panels. Also, library management is much easier when only one catalog is involved.

To create a uniform display for your application, the following design principles should be considered:

• consistent indentation and spacing across displays
• consistent use of color and highlighting
• concise and unambiguous text
• make the application fit the smallest device (window size) to be used by your application.

INDIVIDUAL EDPARMS ENTRIES

Create an EDPARMS entry for each type of AF entry and store these in your SASUSER.PROFILE catalog. Once you have established your own individualized EDPARMS entries, all new entries will use these settings instead of the system defaults. This is particularly useful in selecting the color scheme for your AF applications. Also, set +b specifications to the beginning of columns that will be used on many windows to ensure uniformity of displayed information.

MENU WINDOWS

Menu windows can be created with MENU or PROGRAM entries in the BUILD procedure. Creating MENU windows with MENU entries will be discussed, followed by a discussion on using PROGRAM entries as MENU windows. First, some general hints on MENU windows will be addressed.

MENU DISPLAY AND FUNCTION CONSIDERATIONS:

Consider the following when designing your menus:

• limit the number of options on a menu — use submenus with logical grouping of options
• always provide a selection to return to the main menu from all submenus
• create HELP entries for the general use of your menu system
• create HELP entries for specific menus that may require additional instructions
• use full-word choices (see below for full discussion).

MENU entries used to create MENU windows:

MENU windows designed with numeric selections are by far the easiest to develop with MENU entries. With numeric selections, fast branching is simple — for example, 3,2 means to go to the second item on the third submenu. If you decide to use words or letters instead of (or along with) numeric selections you must be aware of the following:

• abbreviations, especially single letter designations, may cause problems
• valid DMS or FSP/AF global commands (ex. STORE and FREE) must be avoided.

The following examples demonstrate the problems that can be encountered when non-numeric selections are provided. Any abbreviations that can be expanded to fit a DMS, FSP, or AF global command will cause an ERROR message. For example, if a user picks option A, the user would see the following:

TESTAF
Select Option = = >
ERROR: Two or More Commands start with a. Please reenter

PICK ONE OF THE FOLLOWING MENU OPTIONS

<table>
<thead>
<tr>
<th>Selection Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>First option</td>
</tr>
<tr>
<td>STORE</td>
<td>Store reports off-line</td>
</tr>
<tr>
<td>X</td>
<td>Leave this application</td>
</tr>
</tbody>
</table>

Use of menu choices that are any of the DMS, FSP, or AF global commands will also cause that command to be executed instead of the branching you desired. Consider what will happen when the user picks the STORE option for the above menu:

TESTAF
Select Option = = >
ERROR: Store cannot be handled by this window.

In this case, the DMS STORE command was attempted and the intended branching was not executed.

Finally, if the user picks option X, the user would be sent out to the host operating system, and would have to issue a command to return to the application.

If you do decide to provide the user KEYWORD options (after careful checking to avoid DMS, FSP, and AF global), use the MENU LINK commands MLOPTS and MUNK to allow the user the ability to select items on submenus just by typing the selection name on higher menus. Make sure that the selections on the submenus have names that do not appear on higher menus. Note: If you add additional options to your MENU window, you must issue another MUNK to establish the correct menu linkage.

You can allow the user the ability to have multiple methods to pick a selection, by using the ATTR window to give both a numeric and a word selection for each menu selection.

PROGRAM entries used to create MENU windows:

You can add additional functionality to your MENU windows by creating them with the PROGRAM entry. In SAS/ASSIST™ software all menus are PROGRAM entries. The use of PROGRAM entries allow more control by the application developer over the user-interface to the MENU window.

The ability to select menu items (or selection fields) by placing the cursor on a word and pressing ENTER minimizes errors resulting from typographical errors and improves the ease of...
use of your application. Because a MENU entry only allows a user-entry on the command line, a PROGRAM entry must be used to produce a menu with selection fields.

The SCL CURWORD function lets you determine the cursor location when ENTER is pressed. This function returns the word in which the cursor resides. The SCL for a simple MENU using selection fields is shown below. The SCL program determines the cursor placement and branches to the desired location in the application based on the cursor placement.

```
INT:  
  control enter;  
return;  
MAIN:  
  curloc=upcase(curword()); /* location of cursor */  
if curloc='VBAR' then call display('vbarr.program');  
if curloc='HBAR' then call display('hbar.program');  
return;
```

The SAS/ASSIST Primary Menu is an example of a sophisticated menu created with a PROGRAM entry.

```
SAS/ASSIST:Primary Menu

TUTORIAL DATA REPORTS GRAPHICS
STATISTICS ENTRY PROGRAMS SETUP
PROCEDURE/EXIT
```

Memory Concerns When Using PROGRAM Entries for MENU windows:

If you use PROGRAM entries instead of MENU entries for MENU windows, be aware that more memory will be needed to run the application. If a series of PROGRAM entries are called in succession with CALL DISPLAY, all the PROGRAM entry windows will be open at the same time. This can result in insufficient memory to run the application. MENU entries use less memory because the MENU window closes when a selection is made and is opened again when the user return to the MENU.

PROGRAM ENTRIES

Action fields are given field names based on their position in the display window — the first action field is assigned FIELD1, and so on. If you add a field to the display window, you may change the assigned field names and cause problems with any SCL code you wrote using the assigned field names. You may also lose special attribute settings for that field. If you assign an alias (on ATTR window) you can use the alias name in place of the field name in all your SCL code. Then if you add a field or change the display, your SCL program and your attribute setting for those action fields will not be affected.

Make sure the format and informat specifications for each field in the ATTR window are compatible. The informat specification must accept the format's output as input. For example, if you use the DOLLAR format for a numeric field, use the COMMA informat for that field. Informat verification takes place before the SCL code executes.

CBT ENTRIES

Problems with branching on final CBT frame:

If you want unconditional branching on the final frame of a CBT, then make the entry you want to branch to the child. If you want branching based on the user's answers to a question, then use the > > branch indicator. Note: if you use the WRONG=, the > branch indicator, or a FFP branch on the last frame, the student will go from the branched window directly to the first child just as if the student had pressed ENTER after finishing the final frame.

CBT checkpoints:

CBT checkpoints are suppressed under TESTAF. You can branch out of the CBT entry, but your return branches will always go to the original entry's first frame. When you assign a HELP entry from a CBT entry, you are returned to BUILD under TESTAF not to the parent entry as it does under AF. Thus, to check branching of CBT entries you should use the AF command.

HELP WINDOWS

HELP windows can be created with HELP or CBT entries in the BUILD procedure. Use CBT entries instead of HELP entries when multiple pages of information are needed. Note: use of CBT entries for HELP windows will probably increase the memory requirements for your application.

Unless you have a definite reason, leave the parent blank on HELP windows and the user will go back to the window he/she asked for help from. When you assign a HELP entry from a CBT window, do not assign the same CBT entry as the parent. You will tie the user up in a knot of multiple task invocations.

TESTING SAS/AF APPLICATIONS

Compile all programs and analyze any compile errors. You may want to do this in a batch job so that all messages will be routed to the SAS LOG. Run through your application using the AF command and check for all of the following:

- does it flow correctly — do you branch correctly
- can the user exit at any time
- can the user get help from any window
- are correct results generated
- do incorrect answers get trapped
- is incorrect user input handled correctly

AF entries can be tested from within the BUILD procedure with the TESTAF command or from any display manager window by issuing the AF command. Although TESTAF is convenient for testing while creating entries, you should also test your application with the AF command.

DOCUMENTING A SAS/AF APPLICATION

PROC BUILD allows you to generate a hardcopy of your application for documentation. The BUILD statement in the BUILD procedure controls the part of your application that is printed. You can control what is printed as follows:

- a listing of the catalog directory
- the display panel of an entry
- the display panel of a PROGRAM entry — with or without pad characters displayed
- the source panel of PROGRAM entries
- the attribute information associated with each entry
- a cross-reference table showing the entries that call, or are called by MENU or CBT entries.
TROUBLE-SHOOTING SAS/FSP APPLICATIONS

SOME HELPFUL HINTS

Place forms for the printers you will use in your application in a SASUSERPROFILE catalog. Make sure to copy this catalog and make it available to the user as their SASUSERPROFILE catalog. Then all printing you setup in your applications will use these default forms. Note: this is the only way you can use customized forms with the FSPRINT and FSUST procedures.

PROC FSEDIT SCREENS

DISPLAY DESIGN:

Fields are identified on the screen based on the position of the underscored lines associated with the field. Once you change the position or length of an underscore, the field must be re-identified. Note: You must have at least one space between any text and the underscored field. The length of the underscore does not have to match the length of the variable in the data set.

LABEL Statement:

To make it easier to create a new screen, use a LABEL statement for the variables and specify the LABEL option on the PROC FSEDIT statement. The labels you have specified will appear on the new screen instead of the variable names.

NON-DISPLAYED VARIABLES:

The three different ways suppress the display of variables within your data set:

- use the VAR options on the PROC statement to subset the variables to be displayed
- set a variable to UNWANTED during the FSEDIT identify phase
- set the non-display attribute in FSEDIT parameter modification mode.

In the first two cases, the status of the variable to FSEDIT is unwanted. Unwanted fields are not displayed on the screen and are not available for use in SCL. However, they remain in the data set unmodified. In the last case, the variable is not displayed but is available for use in SCL.

COMPUTATIONAL and REPEATED FIELDS:

FSEDIT allows you to create fields for screen only variables (computational fields) to be used with Screen Control Language and repeated fields which allow you to display the same field in many different places throughout your application. Repeated fields are useful in multiscenario applications for displaying identifier variables (for example, name or ID number) on each screen.

LONG FIELDS:

If your data set has character fields that are longer than the width of your screen, you can display the entire field on multiple lines by putting an asterisk (*) at the end of each line that needs to be continued. Note: when you set attributes for these fields, you must type the flag letter on each line. For example, type P on all lines of a continued field to set the PROTECT attribute for the entire variable.

FSEDIT FIELD ATTRIBUTES

To further customize your display and validate fields, you can set attributes for any field on your screen. The following are some hints for using these attributes:

- Use the MAX and MIN attributes to set maximum and minimum values. If an incorrect value is entered, the field will be marked in error, and the user will not be able to leave that observation until the field is corrected.

- If a field is set to REQUIRED, the user will not be able to leave a new observation until a valid value is entered in that field.

- Use the FCOLOR & FATTR attributes to call attention to important fields or distinguish between different types of fields.

- Use the PROTECT attribute to protect fields so that the user cannot make modifications.

- NON-DISPLAY will allow a field to be used in the application by SCL but not display it on the screen. The NON-DISPLAYED attribute can also be used for user passwords.

SEARCHING IN PROC FSEDIT

The FSEDIT search commands search from the current observation in the data set forward, so it is often best to begin your searches at the first observation. If you know the observation number you are looking for, simply type that number on the command line.

The FSEDIT Parm window allows the user to set default variable values for both the NAME and STRING commands. If you have a customized screen and have forgotten the name of a data set variable, place your cursor on the field and press HELP to get information on the variable name and type.

HELPFUL HINTS FOR CREATING DATA SETS

Field identification will be easier if you create variables in the order you plan to display them on the screen.

If the data set is one that will not have observations added to it, just fields updated and/or added, then sort the data set the way that is best for searching and other needs and then assign an ID number to each observation with the statement \n N. This will give each observation a unique identification number for searching, merging, and updating. If you are going to be adding observations and want to keep the observations in chronological order, you can do the same thing.

TROUBLE-SHOOTING SCL APPLICATIONS

SCL in SAS/FSP under PROC FSEDIT

To save your FSEDIT modified screen containing your SCL program, you must assign a permanent SAS catalog with the SCREEN= option of the PROC FSEDIT statement. An FSEDIT SCL program is compiled when the END command is issued to close the FSEDIT PROGRAM window. Messages from the compile and/or execution of an SCL program are sent to the SAS LOG window.

SCL in SAS/AF under PROC BUILD

An AF SCL program must be compiled before it can be used. Messages from the compile and/or execution of an SCL program are sent by default to the BUILD MESSAGE window. Note: if you are in TESTAF you must end TESTAF and go back to a BUILD window to view the MESSAGE window. If you would like to print your messages to the SAS LOG window, issue the NOMSG command with your PROC BUILD statement.

HELPFUL HINTS

COMPILE:

When writing SCL code, write your program in sections and compile it frequently. Compiling brings to the surface many syntax and logic errors and allows you to fix errors as they occur and avoid one small error re-occurring throughout your program. This becomes particularly important in SCL when the result of one function may be passed as an argument to many other functions throughout the program. It is recommended that you clear the LOG window (FSEDIT) or the MESSAGE window.
Another good practice to develop is to make back-up copies of your SCL programs. You can either make a second copy of your catalog or make a flat file of your SCL program. Having a flat file copy is an easy way to create similar SCL programs for many different screens (FSEDIT) or program entries (BUILD).

**SCL LABELS**

Reserved labels:

- **FSEINIT**: Executes before the first observation is displayed (for FSEDIT applications only).
- **INT**: Executes each time a new observation is displayed (FSEDIT) or when the AF task begins (AF).
- **MAIN**: Executes each time a user modifies a field and presses ENTER or a function key.
- **TERM**: Executes when the user leaves an observation to go to a new observation or leave the application (FSEDIT) or when the user leaves the application (AF).
- **FSETERM**: Executes when the user leaves the application by entering END (for FSEDIT applications only).

User-defined labels:

Use user-defined labels with **GOTO**, **IF/THEN** and **LINK** statements in your SCL programs to add functionality and to make your programs modular. For example based on some condition in MAIN you could either continue, go to another label for further processing, or jump directly to the **TERM** section. In the example below, if X is greater than 100, the program will jump to **MYLABEL** execute the statement in that label and then return to execute y=x+1.

```
MAIN:
  if x>100 then LINK MYLABEL;
  y=x+1;
  RETURN;
MYLABEL:
  _msg_ = "MYLABEL executed - X is greater than 100";
  RETURN;
```

**OPENING DATA SETS**

Data sets should be opened and closed once for the application. In other words open data sets in the **FSEINIT** label for FSEDIT applications or in the **INT** label for AF applications. This assigns a data set id for each data set which you can then reference anywhere in the SCL program.

An open statement in the **MAIN** label could needlessly open the data set many times depending on how often **MAIN** is executed. Once a data set has been opened, it should be closed before exiting the application. Close the data set in the **FSETERM** label for FSEDIT applications or in the **TERM** label for AF applications.

Use a **SET** statement to reference variables in an opened data set. The **SET** statement links variables in a SAS data set with the variable of the same name and type in an SCL program or on the display. The **FETCH** and **FETCHNOBS** functions will load the values. If you will be changing the values of these variables during your application, you must use the **UPDATE** function to update the changes from the application to the data set.

**RETURN CODES**

It is the responsibility of the SCL programmer to check the return codes to ensure correct working of the application. The majority of SCL statements give the programmer a return code, often abbreviated as **rc** in the documentation. Return codes are used in two ways.

First, a return code will tell you if the function worked correctly (valid syntax). For example if you open a data set, a dsid (data set id) is returned if the dsid is greater than 0, the data set was successfully opened if the dsid is less than or equal to 0, the data set was not opened.

Second, a return code may be passed to another function as an argument. For example to find out how many observations are in an opened data set use the **NOBS** function. The dsid from the open function is passed to the NOBS function and the number of observations is returned.

```
rc = NOBS(dsid)
```

It is often helpful to give more meaningful names to your return codes especially in larger applications where there may be numerous functions with return codes. For example:

- dsid might be id_one for opening data set one.
- rc for nobs might be num_obs instead of rc.
- rc for varname might be var3_one for the third variable of data set one.

The meaning of the return code varies from function to function. Check the documentation for the function you are using to determine the meaning of its return code.

The SCL programmer should check to see that the open function returns a valid dsid (for example, the data set does indeed get opened) before passing that dsid to another function as an argument, as with the **NOBS** function. Using the **MSG** variable, there are a few ways to display the value of a return code. Use the **SYSMSG()** function to return a system message for the last function that was executed.

```
INT:
  dsid = open('work.one','input');
  if dsid < 0 then _msg_ = SYSMSG();
  else _msg_ = "The data set was successfully opened";
  RETURN;
```

In the above example, if the dsid is less than or equal to 0 the **SYSMSG()** function will print the following message on the message line:

```
ERROR: file work.one.data does not exist.
```

This message will signal you to return to your program and correct your error. Without this message you would not have known that the data set was not opened and later in the program when the dsid was passed to another function, the program would have halted because of an invalid argument. At that point you might have thought the error was with the function and not the data set id.

Some return codes return a character or numeric value that you were trying to query. The return code can be displayed on the window by associating it with a user field. You can also have the return code displayed in a message. For example:

```
numobs = NOBS(dsid);
  _msg_ = "The number of observations is ", PUT(numobs,8.);
```

Note: if you wish to put a numeric return code into a message, use the **PUT** function to prevent warnings about numeric to character conversions.
ATTRIBUTES vs SCL

Both the BUILD and FSEDIT procedures allow you to set various attributes for the screen fields. These attributes are validated when the application is displayed. In addition, SCL provides statements to perform some of the same functions. In SCL however, execution depends on where (under which label) the statement appears in your SCL program. You can also make the setting of attributes conditional. For example, assume that, with a particular data set, if the value of variable X is greater than 100, you do not want any changes made to the variables NAME or X. The statement

\[
\text{if } x > 100 \text{ then } \text{PROTECT NAME X;}
\]

would allow you to protect those fields if the variable x is greater than 100 but still allow changes to be made to any observation whose value of x is less than 100.

The PROTECT attribute prevents the user from modifying that field on the window, but does allow modification through SCL. SCL has both a PROTECT and an UNPROTECT statement. Protecting a field in SCL that has already been protected by the attribute has no effect (in other words: it does not act like a toggle). You can use the SCL UNPROTECT statement to unprotect a field that has been protected by the attribute. Again, the advantage of the SCL PROTECT is that it allows conditional setting of the attribute. Also you can issue a PROTECT_ALL statement which will protect all the fields and allow no modifications.

CONTROL FEATURES

The following SCL functions allow the SCL programmer to override the default actions of the SCL language or the PROC itself. These functions should be used very carefully. Misuse could lead to corrupted data.

LABEL CONTROL STATEMENTS:

The following powerful functions should be used with care.

- **CONTROL ENTER** allows the execution of the MAIN label after the user has pressed ENTER or a function key — the user does not have to modify a field first.
- **CONTROL ERROR** allows the MAIN label to execute when fields are in error — this normally is not allowed.
- **CONTROL ALWAYS** combines the features of CONTROL ENTER and CONTROL ERROR.

ERRORON and ERROROFF:

The ERRORON and ERROROFF functions allow you to turn error flags on and off for situations that the attributes may not cover. For example,

- to set the error flag on a field and force the user to change the field before the user may continue or to draw special attention to a field without forcing any changes.

For example if you do not want any data entered for a student named JANE, then the following statement would cause the error flag to be set on the name field if JANE is entered.

```sas
if name = 'JANE' then do;
   ERRORON name;
   msg = "You may not enter the name Jane."
end;
RETURN;
```

The user would have to change the name in order to continue the application. If you just want to draw attention to the fact that the value of the NAME field is JANE then add the CONTROL ERROR statement and the field will be marked in error, you can display a message, but the user will be able to continue with the application.

The ERROR function will return a 0 or a 1 to tell you if a field is marked in error. This is also useful with CONTROL ERROR, so that the user can continue, but you will know that fields are in error.

OTHER CONTROL FUNCTIONS:

The EXECCMD function will collect the command or commands issued and place them on the command line of the next window to be displayed, where they are executed before that next window is displayed. They are not carried out in relation to where they are placed in the SCL program. Valid commands which may be passed to the EXECCMD function include commands for the procedure you are in as well as any valid Display Manager commands. A note about issuing display manager commands — be sure to leave a way to get back to your FSEDIT or AF application — don't get stuck in Display Manager.

DEBUGGING FEATURES

In addition to using messages to access return code values from your SCL functions, there are two SCL statements to assist you in debugging your application.

The ROPT statement allows you to set runtime options. The following options are available with the ROPT statement:

- **LOOPCHECK** is set by default to check for infinite loops and stop after executing 200,000 statements. Specify NOLOOPCHECK to turn off this limit. As with the CONTROL statements, you should be very careful when using this feature.
- **The MAXSTMT** option allows you to set the maximum number of statements that can be executed under an SCL label.
- **The TRACE option** will track the execution of the program in the LOG window. TRACE is a very valuable option for isolating code errors. It is best used as TRACEON and TRACEROFF to block off areas of code where problems are occurring.

A simplified version of the SAS PUT statement is available, to aid in debugging SCL programs. The PUT statement can be used to verify values of screen variables. The PUT statement takes the form

```
PUT fieldname1 = fieldname2 =;
```

and will write the value of the specified variable at that point in the program to the LOG. PUT ALL will write the value of all fields to the LOG. The PUT statement can be used instead of ROPT TRACE to get information at an isolated location in the program.

SAS/AF SPECIFIC SCL

The **_STATUS_** Variable:

As an SCL programmer, you can use the **_STATUS_** variable to query how the user of yours application exited the main section, or you can force the user to stop executing the main label based on some criteria. For example:

```
if x > 100 then _STATUS_ = 'H';
```

would halt the execution of statements in the MAIN label when x was greater than 100. This will close the display without further input from the user. Note: _STATUS_ = 'H' does not execute until the next RETURN; or STOP; statement.

In the **TERM** label you can force the user back to the MAIN label.
ignoring the user’s END or CANCEL command, by setting _STATUS_ = ‘R’ for RESUME. Use caution with this option. A statement like

if _STATUS_ = 'C' then _STATUS_ = 'R';

could leave the user “stuck” in main if they entered the cancel command because they were unable to correct a field in error.

**SUBMIT BLOCKS:**

The statements in a submit block are not executed based on where they occur in the SCL program, but are collected in the PREVIEW window and submitted to the display manager when specified. Statements are often submitted in pieces to the PREVIEW window and do not execute until the last submit block is reached or a submit with an option is given. The statements within a submit block are not checked for proper syntax by the SCL compiler. This allows for the piece by piece submission of statements. The submit block statements are not checked for syntax until they are run, so the SCL programmer should be careful and not depend on the SCL compiler to find errors.

The timing of the submission of the statements is controlled by an option following the SUBMIT statement these are:

- CONTINUE: Submits the code in the PREVIEW window at this time and returns to execute the statements following the ENDSUBMIT.
- IMMEDIATE: Submits the code in the PREVIEW window and returns control to the procedure (returns to the parent entry). Note: statements following the ENDSUBMIT statement will not be executed.
- PRIMARY: Submits the code in the PREVIEW window, returns the user to the entry specified with the C= option in the AF command. Note: statements specified following the ENDSUBMIT statement will not be executed.
- TERMINATE: Submits the code in the PREVIEW window and terminates the application.

When no option is specified on the SUBMIT statement, the SCL code is retained in the PREVIEW window and is executed when the AF task ends.

You can visually verify the code being generated in a SUBMIT block by using the PREVIEW function. For example, the following would display the generated code, then clear the window:

```sas
submit /* continue */; /* for debugging only */
proc print data = &dsname &options; run;
endsubmit;
call preview("display"); /* for debugging only */
call preview("clear"); /* for debugging only */
```

When the code has been verified, add back the SUBMIT CONTINUE statement and remove the calls to the PREVIEW function.

A submit block can submit multiple SAS statements within an IF/THEN statement without requiring a DO/END block. The submit block replaces the normal DO/END block.

**REPLACE STATEMENT:**

The REPLACE statement defines a character string that can be used to replace a field value (&field) in a submit block. Using the REPLACE statement can help you avoid much of the if/then/else logic used to create submit block statements based on user input. In the example which follows, if the user does not supply a data set name then the data= statement is not submitted and the code will not contain syntax errors.

```sas
TERM:
replace &dsname 'data = &dsname';
submit;
proc print &dsname; run;
endsubmit;
RETURN;
```

**VIEWING OUTPUT:**

If you want the user to view the OUTPUT window for the results of a submit block, you must code this into the SCL program. You can use the call EXECCMD function explained earlier in this section, or you can issue the DM statement and give it the commands you want to execute. After your final submit block, you could add the following line to take the user to the OUTPUT window.

```sas
call EXECCMD('output;zoom on;top');
```

Remember that EXECCMD commands are “held” until the control is returned to the procedure, and thus are not executed in relation to where they appear in the program. Use the REFRESH statement to force this command to execute immediately.

The following SCL code would place the user in a zoomed output window:

```sas
TERM:
submit;
DM 'zoom on;output;' output;
proc print &dsname &options; run;
endsubmit;
RETURN;
```

**CONCLUSION**

This paper discusses helpful hints and ways to avoid common problems encountered in the writing of Version 6 SAS/AF and SAS/FSP applications. Specific information is provided to help with the writing of SCL based applications in both the BUILD and FSEDIT processes. It is hoped that the information provided will facilitate the development of sophisticated interactive applications using the Version 8 SAS System.

**ACKNOWLEDGEMENTS**

The authors wish to thank Sue McGrath, Glen Itano, Scott Bass, and Tom Sherron for technical review and Bud Whitmeyer for aid in typesetting the paper.

**REFERENCES**


Base SAS®, SAS/AF®, and SAS/FSP® are registered trademarks and SAS/ASSIST™ is a trademark of SAS Institute Inc., Cary, NC.