Creating a User-friendly PROC FSCALC Application
Patti Brideson, SAS Institute Inc., Cary, N.C.

Abstract
Setting up a user-friendly PROC FSCALC application requires knowledge of PROC FSCALC and the special tools available for developing such an application. First, the developers must think about how the software works. They are going to be pushing, or sending, commands to the command line using the program screen, and they must always be aware of which screen the command is being pushed to, as well as what effect the command has on that particular screen. Second, they must be aware of the special variables that are available within the FSCALC procedure that allow them to control the environment for the users. Finally, they need to be able to second-guess their users in order to program around any problems users may run into while using the application.

Tools to use
There are many tools available for your use if you are developing a user-friendly application for others. These tools are often used together to make the application easy to use. A description of the commonly used tools is given below.

The _COMMAND_ variable is probably the most important tool you will use. It is a special PROC FSCALC variable that is assigned a character string. This character string consists of FSCALC commands which are separated by semicolons. These commands will be pushed, or sent, to the command line of the FSCALC spreadsheet for execution by the spreadsheet. This method is used to control the commands used by the application.

For example, the FSCALC program statement:

```plaintext
_COMMAND_ = 'FETCH';
```

issues the FETCH command when pushed to the spreadsheet command line and places the user in the FETCH screen as shown below:

```
Again end users would be left in a screen they might not be able to cope with. The command is completed by issuing another END command to end out of the Variable Selection Screen. The full command given is shown below.

```plaintext
_COMMAND_ = 'FETCH; END; END';
```

This series of commands would complete the FETCH of a data set. You should not extend the character string that is being assigned to the _COMMAND_ variable over two lines in your programming statements or it will not be handled properly by PROC FSCALC when it is pushed to the command line.

Another useful tool is the _MESSAGE_ variable. This variable, like the _COMMAND_ variable, is set by using an assignment statement to set the value of the message. When coding the _MESSAGE_ variable, the character string should be restricted to one program statement. Any more text will probably be truncated. This variable is used to produce messages on the message line of the FSCALC spreadsheet. For example, the following assignment statement:

```plaintext
_MESSAGE_ = 'ERROR: This is an error';
```

produces the following spreadsheet.
The @ function, when used in conjunction with special FSCALC variables, is also very helpful. This function is used to return the absolute position of the row or column specified in the spreadsheet and is useful in positioning the cursor to a certain location for the user to enter a value that can then be checked. In the spreadsheet given below, the cursor is placed at ROW34 and MENU. MENU is a column in the spreadsheet and ROW34 is a row in the spreadsheet. No subset command has been given before issuing these statements. The assignment is made through the use of the _CURSORROW_ and _CURSORCOL_ special FSCALC variables. For example, the statements:

```plaintext
_CURSORROW_ = ROW34;
_CURSORCOL_ = MENU;
```

are used to position the cursor.

The PAUSE statement is still another useful tool in FSCALC programming. It is often used with the special FSCALC variables previously discussed: _COMMAND_, _MESSAGE_, _CURSORROW_, and _CURSORCOL_. The PAUSE statement stops execution of the program at the location where the statement is encountered. At this time, the special FSCALC variables listed above are actually executed. Command set up in the _COMMAND_ variable is executed. The message in the _MESSAGE_ variable is displayed, and the cursor is positioned. The spreadsheet is now ready for the user to interact.

The application remains in PAUSE mode until an EXECUTE command is encountered. The EXECUTE, or EXEC, command starts processing of the program at the location the last PAUSE statement was executed. This enables you to define a spreadsheet that prompts the user for a value and then enables you to evaluate what has been entered.

In connection with the PAUSE statement, the function keys are often set to the EXEC command, to enable the programmer to control the user's actions via the program itself. The easiest way to set up the function keys is to create a catalog entry called FSCALC01.KEYS as shown below. This entry in the catalog containing the spreadsheets, is checked before the SASUSER library is checked. PROC FSCALC executes the key values found in this location before any other keys in other locations.

The @ function, when used in conjunction with special FSCALC variables, is also very helpful. This function is used to return the absolute position of the row or column specified in the spreadsheet and is useful in positioning the cursor to a certain location for the user to enter a value that can then be checked. In the spreadsheet given below, the cursor is placed at ROW34 and MENU. MENU is a column in the spreadsheet and ROW34 is a row in the spreadsheet. No subset command has been given before issuing these statements. The assignment is made through the use of the _CURSORROW_ and _CURSORCOL_ special FSCALC variables. For example, the statements:

```plaintext
_CURSORROW_ = ROW34;
_CURSORCOL_ = MENU;
```

are used to position the cursor.

The PAUSE statement is still another useful tool in FSCALC programming. It is often used with the special FSCALC variables previously discussed: _COMMAND_, _MESSAGE_, _CURSORROW_, and _CURSORCOL_. The PAUSE statement stops execution of the program at the location where the statement is encountered. At this time, the special FSCALC variables listed above are actually executed. Command set up in the _COMMAND_ variable is executed. The message in the _MESSAGE_ variable is displayed, and the cursor is positioned. The spreadsheet is now ready for the user to interact.

The application remains in PAUSE mode until an EXECUTE command is encountered. The EXECUTE, or EXEC, command starts processing of the program at the location the last PAUSE statement was executed. This enables you to define a spreadsheet that prompts the user for a value and then enables you to evaluate what has been entered.

In connection with the PAUSE statement, the function keys are often set to the EXEC command, to enable the programmer to control the user's actions via the program itself. The easiest way to set up the function keys is to create a catalog entry called FSCALC01.KEYS as shown below. This entry in the catalog containing the spreadsheets, is checked before the SASUSER library is checked. PROC FSCALC executes the key values found in this location before any other keys in other locations.

The application remains in PAUSE mode until an EXECUTE com-

Other statements that are useful tools in this type of application are the CALL SYMPUT statements. These statements are used to define special macro variables that are used on the screens brought up by the invocation of certain commands. The FETCH command discussed earlier is a clear case of where using CALL SYMPUT would be helpful. Other screens where macro variables can be controlled are the NEW spreadsheet specification screen, the OLD spreadsheet specification screen, the CLEAR screen, the INSERT screen, the CONSOLIDATE screen, and the REPEAT screen.

For example, you can use CALL SYMPUT statements to set up the macro variables used in the FETCH screen by using the following statements:

```plaintext
CALL SYMPUT('FSCFET1', 'IN');
CALL SYMPUT('FSCFET2', 'INVNTRY');
CALL SYMPUT('FSCFET6', '...ROW_');
CALL SYMPUT('FSCFET9', '2');
CALL SYMPUT('FSCFET1D', '1');
```

This would set up the FETCH screen to appear as follows:
Still another tool to use is the _LASTKEY_ variable available in the FSCALC programming language. This variable is useful for controlling the actions that take place based on the key that is pressed. Optimally, the function keys will all be set to the EXEC command. This is important because when a key is pressed, the program can resume processing where it had previously paused, and the _LASTKEY_ variable allows you to verify which key was pressed and what action should take place. For example, assume that a spreadsheet is in PAUSE mode and has asked the user to press certain keys for certain actions. If the user presses the function key #2 or #14, the spreadsheet should be scrolled forward to display more information. The statements for this type of check follow.

```plaintext
IF _LASTKEY_ = 2 OR _LASTKEY_ = 14 THEN DO: 
COMMAND = 'FOR 16'; 
PAUSE; 
END;
```

Another tool that is very useful is the REFRESH OFF/ON command. This command can be pushed through the _COMMAND_ variable to keep PROC FSCALC from displaying all the screens that are executed. In the previous example, the FETCH command was used to bring in a data set. Given the command

```plaintext
COMMAND = 'FETCH; END; END';
```

the FETCH screen would be displayed, then the Variable Selection screen, then control would return to the spreadsheet. You do not want to confuse the naive FSCALC user with these screens. To prevent the intermediate screens, the FETCH and Variable Selection screens, from displaying on the terminal, the statement can be recoded as follows:

```plaintext
COMMAND = 'REFRESH OFF;FETCH;END;END';
```

A final tool to use in user-friendly PROC FSCALC programming is the comment statement that is available. Trying to come back to your application a year after it has been written will convince you to use comment statements. Comment statements can be as long or as short as you desire. There are two ways to create comment statements, but one is by far the best in terms of visibility. A comment statement can be started with an asterisk (*) and ended with a semicolon (;), or it can be started with a slash asterisk (/) and ended with an asterisk slash (*). The latter is much easier to see in the program code. Using a change of color will also help you find comment statements quickly.

Getting started
Once you have discovered all of the tools that are at your disposal, you must decide how to get started. The first thing to do is to decide what the spreadsheet is going to accomplish when it is finished. There could be several choices available to the users depending on actions they take during the FSCALC application. Once you know what the possible endings are for the application, you are ready to consider the program. You may want to consider setting up a skeleton program to get started. The program may appear as follows:

```plaintext
IF _LASTKEY_ = 1 THEN /* go to help */;
IF _LASTKEY_ = 2 THEN /* go to new calc */;
and so on.
```

In the above program, comment statements are placed where programming code will eventually go. These comment statements tell the action that will take place if a particular function key is pressed.

Once the skeleton is completed, you are ready to fill out the program. This is accomplished by taking one section at a time and inserting the program code needed to accomplish that task. The section should be completely tested before starting the next section so that as much testing is done as is possible before the application is finished. The following statements flesh out a section of code given in the skeleton above:

```plaintext
IF _LASTKEY_ = 1 THEN DO:
   MESSAGE = 'You are now entering HELP. Press END to exit.';
   COMMAND = 'RIGHT NAV';
   PAUSE;
   END;
```

Considerations
While you are creating the spreadsheet for user interaction, you consider several things. One consideration is whether you turn NAMES on or off. For some applications it is preferable to turn the spreadsheet by using the NAMES command. The following spreadsheet displays the column names but not the row names.
Function keys can be controlled through the use of FSCALC01.KEYS, the catalog entry discussed earlier in this paper. They can also be changed by using the _KEY_ array available in FSCALC programming. This array can be used to set the function key values. The following programming statements will set all of the function keys to EXEC.

```plaintext
DO I = 1 TO 24;
    _KEY_[I] = 'EXEC';
END;
```

Messages should be controlled through the use of the _MESSAGE_ variable. The text of the message is given within the variable as documented earlier in this paper.

Responses should also be evaluated. There are two common ways to receive a response. One way is by having the user press a function key. Another method is to have a value entered in a cell and a function key pressed to start evaluation. In the first method, the action to be taken is started by the simple process of pressing the function key.

```plaintext
IF _LASTKEY_ = 1 THEN DO;
    _MESSAGE_ = 'You are now entering HELP. Press END to exit.';
    _COMMAND_ = 'RIGHT MAN';
    PAGE;
END;
```

In this example, a check is done to see if function key #1 is pressed. If it has been pressed, a message is displayed and the command to scroll right is given.

It is also possible, however, to check the value of a cell and base actions on that value. One example is to have a section of the application that asks for a response. For example, the application could query end users whether they want a report or not. If yes, the user enters "Y" in a cell and presses any function key. If not, the user enters "N" in the same cell and presses any function key. Evaluation is made on the cell rather than the function key.

```plaintext
IF ROW2?q2 = 'Y'
    THEN DO;
        _MESSAGE_ = 'The spreadsheet is being printed.';
        _COMMAND_ = 'REPORT NAME; SEND; FREE; END';
        PAGE;
        END;
ELSE . . .
```

Further uses

Some other abilities which are not obvious are available in PROC FSCALC. One is the ability to code the help screen within the spreadsheet itself. Usually it is possible to code a help screen in the rightmost or leftmost part of the spreadsheet and use a function key to scroll right or scroll left to reach the help screen.

Another new ability is to use macro variables within title statements. This allows you to define titles from within the program and is useful in defining titles that are blank when not in use or that can be defined when a particular part of the application is used. An instance where this might be helpful would be if you are querying end users whether they want to save the spreadsheet. It would be necessary to know whether the spreadsheet already exists before continuing. If the user issues the BROWSE command and then returns to the spreadsheet, however, the message line will be blanked out. The example below demonstrates what will occur.

```plaintext
The message line above states that any function key should be used to continue, but the real message states that the BROWSE command can be used and the END command can be used to return to the spreadsheet. Another title reinforces the idea that the name must be entered and any function key pressed. Each of these titles has been generated by the following code:
```

```plaintext
CALL SYMPUT( 'msg2', 'You can use the BROWSE command to see if the spreadsheet exists. Type the command END to return here and complete this process.');
```

The message line above states that any function key should be used to continue, but the real message states that the BROWSE command can be used and the END command can be used to return to the spreadsheet. Another title reinforces the idea that the name must be entered and any function key pressed. Each of these titles has been generated by the following code:

```plaintext
CALL SYMPUT( 'msg2', 'You can use the BROWSE command to see if the spreadsheet exists. Type the command END to return here and complete this process.');
```
The spreadsheet before the CALL SYMPUT statements are executed appears as follows:

When the user returns from browsing the catalog directory screen, the message line is blank but the title lines with their associated macro variables still contain their messages.

One further useful ability is that of testing arrays through the use of IF-THEN analysis. This analysis differs if you are testing an entire array for a value or if you are testing each element of the array. To test an entire array to see if all elements meet a certain condition, you need only set up the traditional IF-THEN statement. An example would be:

```
IF COL1 = 'YES'
THEN COL2 = 5;
ELSE COL2 = 6;
END;
```

If every element in array COL1 is equal to the character string 'YES', then every element in COL2 will be set to 5; otherwise every element in COL2 will be set to 6. If you want to check each element separately, however, you would need to code the following:

```
DO I = 1 TO DIM(COL1);
  IF COL1[I] = 'YES'
  THEN COL2[I] = 5;
  ELSE COL2[I] = 6;
END;
```

These statements will enable PROC FSCALC to evaluate each element in COL1 separately and assign values to COL2 based on the equivalent element in COL1.

**Efficient programming**

You want to keep in mind how to do efficient FSCALC programming for the application. When you do array processing instead of explicit cell referencing, you are programming more efficiently. For example, the statement

```
col1 = col2 + col3 + col4;
```

is more efficient than the statements

```
col1.row1 = col2.row1 + col3.row1 + col4.row1;
col1.row2 = col2.row2 + col3.row2 + col4.row2;
col1.row3 = col2.row3 + col3.row3 + col4.row3;
```

It is also more efficient to use temporary arrays or variables to contain values if the values will be used repeated in FSCALC statements than to use references to the spreadsheet cells. In the example given below, COL1 is an array displayed on the spreadsheet. NEWCOL is a new FSCALC temporary variable which is used to store the values in the array COL1. The statements

```
NEWCOL = COL1;
COL16 = NEWCOL + COL2 + COL3;
COL17 = NEWCOL + COL5 + COL7;
COL18 = NEWCOL + COL6 + COL9;
```

are more efficient than the following statements:

```
COL16 = COL1 + COL2 + COL3;
COL17 = COL1 + COL5 + COL7;
COL18 = COL1 + COL6 + COL9;
```

Whenever possible, use the SUBSET statement instead of explicit cell referencing (ROW1.COL1 type of referencing) to restrict the rows or columns that are being used in calculations. The statements

```
SUBSET COL COL1 ROW1 ROW2;
COL1 = COL2 + COL3 + COL4;
```

are more efficient than the following statements:

```
COL1.row1 = COL2.row1 + COL3.row1 + COL4.row1;
COL1.row2 = COL2.row2 + COL3.row2 + COL4.row2;
```

**After the application is developed**

Once the application is developed, you are not quite through. Now you must consider what your end users are going to do to the application that might confuse it or leave them somewhere they can not cope with. You must plan for this in your application. You should go through each option of your FSCALC application and run it. Check to see what would happen if the user presses the wrong function key. Do you issue a proper message, or does your code fall through? You must walk through each section and make sure the users are not going to get lost or do something that will result in their being confused. Check to make sure that there is always a message that tells the users what they should do.

After you have completed the entire application, you are ready to think about how it should be entered. You may want to use the Display Manager's AUTOEXEC processing to enter or in some other way invoke the SAS System. The PROC FSCALC statement should be set up using a four-level name to directly enter the spreadsheet and the NODIR option should be specified to prevent the end user from getting access to the directory screen.

Finally, keep in mind that there are some common sense issues you should deal with before starting. First, PROC FSCALC is not intended to be the end-all procedure. It can not do as efficiently some of the things other procedures are intended to do and is not intended to be a substitute for these other procedures. You will often want to do some preprocessing of the data before entering the FSCALC procedure. For example, it is more efficient to preprocess ten thousand observations in a DATA step or using PROC MEANS or PROC FREQ than to do the same processing in PROC FSCALC.
Summary
Creating a spreadsheet for end users is not complicated once you know the tools available for you to use. You should be aware that you need to set up the FSCALC environment for the user and often need to control that environment. In addition, you must consider how the user can confuse your application and how to do the most efficient FSCALC programming. Finally, don't try to use PROC FSCALC as an end-all catch-all procedure. There are often other procedures that can be used to preprocess the data being brought into an FSCALC application.