Abstract
This tutorial is intended to present a conceptual model that has proven useful in planning and designing interactive production applications using the SAS System. It recommends a framework for the design, development, and maintenance of such applications at a strategic level, one that addresses the usual concerns of the systems analyst and manager:

- minimizing the time required to design and implement the application;
- minimizing the time required to train the user to operate the application, and making users' skills transportable from one application to another;
- insuring that the resulting applications are easy to maintain by promoting the use of modern, structured programming principles.

The first section of the tutorial is an overview of the characteristics or capabilities of interactive applications that are easily attainable using SAS software in most environments. This discussion is aimed at providing the reader with a basis for deciding if a particular application is suited for implementation in an interactive mode, and for establishing design goals that completely utilize these characteristics.

The second section describes a set of techniques that applications can use to interact with users which are more structured and thoroughly defined than most analysts consider when designing this type of application. Rather than viewing all program interaction with the user as simply printing to the console screen or reading what is typed, we will attempt to classify the reasons for interacting with the user into distinct categories, and develop a set of methods and rules for designing the interactive processes in each category.

The third section looks at some techniques for implementing the various user-interface techniques as software tools, utilizing the capabilities of various configurations of the SAS System products. Emphasis will be placed on the design of generalized macro-based modules that can be used by all applications.

The reader should be experienced in the use of the SAS System as a programming language for data-processing applications, including the DATA step and the Macro Facility. Familiarity with the %PUT/%INPUT statements in an interactive environment, the use of the SYMPUT and SYMGET functions, SAS/FRM, SAS/AIP, and the WINDOW/DISPLAY statements in PC/SAS would be helpful.

Characteristics of Interactive Applications
What is it about interactive applications that make them desirable for certain projects? What can be added to the functionality of an application when it is implemented in an interactive mode, that wouldn't be possible in a batch mode? These questions are very common when the prospect of interactive system development is first introduced at a site, and are often not adequately addressed. By classifying and defining the capabilities that are attainable when programming in an interactive environment, and assessing the need for each of these capabilities for the project at hand, the analyst can not only make sure that these capabilities are fully utilized by the application, but can also determine if the added expense of operating the application interactively is justified.

Conversational Processing
An application (or one of its components) is said to have conversational characteristics if it has the ability to stop at some point in its execution and converse with the user, pointing out the occurrence of some significant event, possibly asking for additional information it needs before proceeding, or allowing the user to request some expansion or refinement of the scope of the process.

In traditional batch-mode applications, control-card or command-statement parameters are the usual mechanisms by which the user specifies run-time options. The principal difference between batch processing and conversational processing is that while batch jobs, the parameters must be coded completely and correctly prior to execution. With conversational processes, the user must know only how to start the job. Once it is executing, the process itself determines what information is needed from the user, and can prompt for it in a natural manner. When something goes wrong, an interactive application can inform the user of the exception exactly when it occurs, possibly presenting the user with options for resolving the problem.

For example, consider the simple application of printing a data set from a pre-specified SAS library. The SAS System provides a non-interactive application suited to this task: PROC PRINT. In order to use this application, one needs to know ahead of time the name of the data set to be printed and the specific variables that are to make up the body of the report.

An interactive application, on the other hand, could be designed to eliminate the need for the user to predetermined all this information. By using PROC CONTENTS, the application could analyze the pre-specified library and present the user with a list of valid data set names to choose from. The same PROC CONTENTS information could be used by the application to derive a list of valid variable names for the selected data set and to prompt the user to select those that belong in the body of the report, and those that are to be used as row-labels (ID statement variables).

The application could be designed to suggest appropriate answers for its own questions, for instance, if the example printing application found only one data set on the library, it could suggest that data set as the likely one to be printed, allowing the user to confirm this suggestion or cancel the report request entirely.

Also, it is often necessary to have the application alert the user when it encounters undesirable circumstances, ones that make completion of all or part of the application impossible or senseless. If our example printing application found no data sets on the SAS library, it should provide the user with an informative description of the problem, rather than just stopping short or presenting an empty data set selection screen. If the user selected a data set that the application found had no observations in it, the application should inform the user that the selection was invalid for that reason, and could then ask if the user wished to select an alternative data set.

At first, this may seem like a great deal of effort to go through for such a simple process, but in many circumstances it is a justifiable expense. If these types of reports are required on a frequent basis, novice users would be able to execute the interactive application themselves with minimal training and very little need to interrupt a programmer for help. Such users forced to use the batch application would be much more likely to introduce errors that would require the entire application to be
re-executed, in many cases making the batch application more costly in the long run. We will come back to this example and discuss it in more detail in the next section.

Conditional & Iterative Processing
Like any program, an interactive application can take advantage of two logical design characteristics known as conditional and iterative execution. A process within an application is said to be conditionally executed when its use by the application depends on the completion of a logical test, and is commonly implemented using the IF THEN-ELSIF structured statements provided by the SAS DATA step and macro language. An iteratively executed process is a specialization of this characteristic wherein the process is repeatedly performed, parametrically referencing a series of logically related objects such as an array of variables, data sets, or values, and is implemented using the DO WHILE/UNTIL-END structured statements. The number of iterations again depends on the successful completion of some logical test.

The use of these two characteristics is fundamental to the development of structured applications regardless of the mode of execution. In the context of interactive applications, however, they offer some unique capabilities to the designer, because the conversational characteristics of these applications allow the user to contribute to the conditional tests used to control each process, using information gathered and reported by the application at execution time.

Following on with the simple printing example began above, suppose it is desirable for the users to be able to request that the data set be sorted before it is printed. After the application has presented the user with the data set selection screen, it could display a screen asking the user if the selected data set is to be sorted, and by which variables. The user's response to this query would then be used as the criteria for conditionally executing a PROC SORT step.

Presume now that the user is likely to want to print more than one report at a session; it would be desirable to design the system so that they select all the data sets at one time, so that they do not have to exit and re-invoke the application several times, possibly selecting the same data set twice as an inadvertent mistake. This could easily be accomplished by allowing the user to select all desired data sets from the initial screen, and then defining the rest of the application, including the sort query, as an iteratively executed process, to be repeated for each selected data set.

Modal Processing
Many projects often require the use of a group of functionally independent processes that support the distinct tasks required for the project. For example, a typical management information system may have separate processes for data collection, analysis, graphics, and reporting. In a batch environment, each process would be designed and coded as a separate application. In an interactive environment, however, each process could be coded as separate modules of a single application that would allow the user to easily invoke a specific one as needed. This central application would present the user with a screen describing all of the processing alternatives that the user could select from, and would automatically invoke the right one, handling any necessary data set allocation or other system control needed for the process once the user selected it.

Applications that are designed in this manner—to allow users to perform separate sub-processes at their request—are referred to as "modal" applications, since the user relates to each subprocess as a separate mode of operation (data-entry mode, graphics mode, reporting mode, etc.). The most common technique for implementing a modal application is through the use of menus screens, presenting the user with options to choose from, each representing a separate mode. While the concept of modal application design is rapidly becoming out-dated in favor of more flexible ones on systems such as microcomputers that can afford to dedicate more system resources to support user-interface activities, it remains the best technique on mainframes.

User Interface Tools
At a very simplified level, the characteristics that an interactive application must exhibit are indeed very simple: it must be able to transmit information to the user by displaying messages on the console screen; and it must be able to receive information from the user by pausing to read what is typed in response. In order for these applications to be easy to operate, not to mention easy and efficient to design and program, some structure and planning must be given to the way in which these two capabilities are implemented. Like any programming project, it is to the advantage of the eventual user to be able to anticipate the wants, needs, and work-habits of the users will be more easily transferred to new and existing applications.

This section describes such a set of rules for planning the interactive characteristics of applications. Much of this material has been borrowed from recent developments in software design for microcomputers, and has been modified to match the capabilities of the SAS System in a mainframe environment.

The advantages derived from designing a more structured definition of the way in which a program may interact with users are threefold:

• The system analyst can develop a set of re-usable software tools that can be stored and reused by each application;
• The user can develop skills that are transportable across applications, because the different types of displays and the methods for responding to each display type are more easily "standardized";
• Experience gained in designing various types of applications, in coding and implementing the designs, and in understanding the wants, needs, and work-habits of the users will be more easily transferred to new and existing applications.

There are many different methods for implementing each of the concepts discussed here, depending on the environment in which you are working and the SAS software products available at your site. Examples using the SAS/GRAPH macro facility, SAS/FSP procedures, and Version 6 SAS WINDOW and DISPLAY statements will be described in the following section.

Alerts
There are many instances when an interactive application must convey a message to the user regarding an unusual event or circumstance that occurred during the course of executing a process. For instance, if the user invokes a process that requires reading a specific SAS data set, and that data set is not available or is empty, the process cannot execute. Rather than allowing the application to end with a SAS error message, or worse, to ignore the user's request without informing them of the nature of the problem, a well-designed application will inform the user of the nature of the difficulty, possibly offering a suggestion for resolving it. As another example, consider an application that allows users to create and save SAS data sets on a permanent library, using names that they specify at some point during the session. When the user completes the session that creates the data to be stored in the data set, the application should be designed to check and see if a data set
with the user-supplied name already exists on the library. If so, the user should be warned that if the process continues, this previous data set will be destroyed.

These types of messages are referred to as alerts, and can constitute up to half of the conversational activity of a well-designed interactive application. Alerts are used in many different situations, but all have the same basic function: they display a complete descriptive message to the user that informs them of an important event or situation. The way they differ is in the way the user responds to the message. As you will see in the discussion below, this difference determines the appropriate use for each type of alert.

Dead-End Alerts

As its name implies, a dead-end alert is one that informs the user that the active process has reached a dead-end, a point from which the normal execution of the process can not continue due to some unusual circumstance. In the example below where the application could not find a data set necessary for the process at hand, the message telling the user that the requested process could not be executed because of this missing data set would be a dead-end alert. This example is diagrammed below:

**Dead-End Alert Processing**

1. Start
2. Check for required data set
   - Yes: Generate reports
   - No: Does data set exist?
      - Yes: Alert user!
      - No: SAS Library
3. End

A dead-end alert is used to indicate rare and unrecoverable circumstances. Its purpose is simply to inform the user that their task cannot be carried out as requested, and therefore requires no intelligible response from the user. You do not, however, want the process to simply print the message on the console and then to go on its way, returning to the menu screen or exiting the application entirely, because you will not be able to insure that the user saw the message. It is very possible that the user, expecting a normally slow or lengthy process could not be executed

Confirm-or-Cancel Alerts

This second type of alert is not meant merely to inform the user of an unexpected situation, but rather to warn them that a potentially undesirable or destructive situation could occur if the current course of action has not been carefully considered. It is intended to give the user an opportunity to reconsider this course, and to back out if the user wishes. In the example described earlier, wherein the user chooses a name for a new data set that is already used by a previous data set on the same library, a confirm-or-cancel alert would be appropriate to warn the user that if the process continues as instructed, the data contained in the previous data set will be lost. If the user desires that this data set be replaced by the new one, they would respond in a positive manner to confirm the current process. If they decided that the old data set should be retained, they would cancel the current process and re-invoke it using a unique name for the new data set.

It is this ability to respond positively or negatively, to confirm or cancel the potentially dangerous process, that makes this type of alert different. As with the dead-end alert, the alert message should include explicit instruction to the user on how to respond either way. An alert screen that could be used for the described example is shown below:

**Warning:** The SAS data set REPORT1 already exists on the MISLIB library. If you continue, the data stored in this data set will be lost! If you cancel, you will be prompted for a new data set name to be used to store the new data.

Enter a C to continue, or an X to cancel

Since this alert requires a response from the user, the application must be designed to act on the user's response, conditionally executing the appropriate routine based on whether the user confirms or cancels the alerting process. In the case of our example, if the user enters an 'X' in response to the alert, the process should re-execute the routine that determines the user's desired name for the new data set. If the user confirms the process by entering a 'C', it can ignore the previously-existing data set and replace it with the new data. A diagram showing how this alert process might be implemented is shown at the top of the next page.

Positive Alerts

The two types of alerts described above are alike in that they display their messages and then stop the execution of the current process and wait for the user to respond. When one of these types of alerts are used, the situation always requires a user response, even if that response is only a confirmation of the fact that the user has read the message and hasn't been inconvenienced. There are times when you will want to accommodate the possibility that, during the execution of a lengthy procedure, a user may be left staring at the screen for a very long time, not always sure that the process they have requested is being performed or not. Naive users especially have a tendency to get very nervous when the screen goes blank and doesn't give them the report or the data-entry screen or the graphic that they requested right away. In order to avoid this "Blank-Screen Anxiety", it is desirable to display messages on the screen telling the user that the process currently underway involves a bit of preparation, and may take a while to complete.

On the other hand, you do not want the display of such messages to interrupt the process and wait for the user to confirm that they have read the message; users that become accustomed to the wait will undoubtedly plan to do other things during times like these. It is not important for you to insure that the user has a chance to read these messages—they are simply there to
Creating a temporary data set in order to a central SAS data set, as well as one or more look-up routines that only accepts a positive or negative response does not several IIUnU!es before the PROC FSPRINT steps, :MERGEs, and associated PROC SORTs that such an information at the screen using PROC FSPRINT. As an example, consider a reporting application that must access the applicatlonL provides us with the capability to allow the application to display a query on the screen, pause execution of the process, wait for the user to type a response to the query, and then store this response for later use by the application.

Dialogs
We have already discussed several examples of interactive processes that needed to ask the user for information necessary to complete its task, such as the name of a dataset, a list of variables, or a data value. In these situations, a simple alert routine that only accepts a positive or negative response does not provide sufficient capabilities. A more advanced model referred to as a dialog provides us with the capability to allow the application to display a query on the screen, pause execution of the process, wait for the user to type a response to the query, and then store this response for later use by the application.

Recall the example discussed in the first section of the interactive application that printed a SAS dataset chosen by the user from a specific SAS library. If asked to design such an application, many would simply ask the user to type the name of the desired SAS dataset, using a display such as this:

```
*** Please enter the name of the SAS data set you wish to print, ***
*** and press ENTER.
```

While this approach is very easy it implement and satisfies the basic definition of a dialog, it does little to justify the added expense of implementing the application in interactive mode. Most users that would be happy with this would be just as happy using PROC PRINT in a prewritten batch job. The motivations behind an interactive application such as this should include:

1. shielding the user from JCL and the need to use traditional programming tools, such as program editors;
2. preventing the user from executing erroneous or useless processes;
3. enabling untrained, less costly personnel to operate the application.

The example above accommodates the first motivation rather nicely, (assuming that the application is installed properly as a TSO CLIST, CMS EXEC, ISPF option, or MS-DOS BAT routine). The second goal can be attained by making sure that the application checks to see if the user-specified name is a valid SAS name, that a data set does exist on the SAS library, and that it is not empty. The %NOBS macro described in the SAS Macro Course would be ideal for this purpose. The third item definitely requires some additional thought, however. In order to enable as many people as possible to operate this application, the dialog must provide enough information to adequately respond to the query.

The terms “enough” and “adequately” are relative ones, subject to the expected capabilities of the users, the overall complexity of the application, and the value of the effort required to develop the application. It may be judged more parsimonious to simply include a list of the data sets that should be stored on the SAS library, along with a description of their contents, in the application’s documentation. But in cases where the data set list can be expected to change, or when users are likely to be operating the application without ready access to this documentation, we need to place more responsibility for informing the users on the dialog itself.

By designing the application to first compile a list of the data sets stored on the library, the dialog can be designed to display the list, giving users a complete set of valid responses. PROC CONTENTS can be used to compile a table of all the SAS data set names, the number of observations on each, and, assuming that the applications that created the data sets support the data set LABELS option, can also contain a brief description of the contents of each data set. The list can then be edited to remove those data sets that are empty, and then a dialog such as the one shown below can be created from the remaining data set entries on the table:

```
Data set list for the Sales Reporting Library:

<table>
<thead>
<tr>
<th>Number</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>JANSALES</td>
<td>sales data for the month of January</td>
</tr>
<tr>
<td>2</td>
<td>FEBSALES</td>
<td>sales data for the month of February</td>
</tr>
<tr>
<td>3</td>
<td>MARSALES</td>
<td>sales data for the month of March</td>
</tr>
<tr>
<td>4</td>
<td>APRSALES</td>
<td>sales data for the month of April</td>
</tr>
</tbody>
</table>

Please select the SAS data set you wish to print by typing its corresponding number and pressing ENTER.
To cancel, press ENTER without typing anything.
```
By expecting the user to simply type a number, the application further reduces the chance of error that would result from typing errors that might result from typing the full data set name.

Note that the dialog gives the user the option to cancel the execution of the print step by not typing a data set number. This is a very important consideration for any dialog—giving the user an opportunity to change their mind, or to back out of a process if they realize that they need more information before proceeding. In this case, if the user the chance to cancel the application if they discover that the data they intended to print is not available on the library.

If your site has the full-screen procedures offered by SAS/FSP or SAS/AF, this dialog could be implemented as a full-screen display such as the one shown here:

![Full-screen dialog example]

The availability of full-screen procedures opens up a great many possibilities for developing much more sophisticated dialogs than our examples have demonstrated so far. If your application requires more than one piece of information from the user (and most will) you can combine the separate queries into a single dialog, allowing the user the luxury of considering all the queries at once, entering and editing the responses as a group before “submitting” them to the application. In many cases, being able to read all the questions before composing the responses makes it easier to understand them and respond properly. It is also possible to make iterative processes more convenient for the user with a full-screen dialog. In the example above, if the users often need to print more than one data set in a given session, the application could be designed to allow the user to place an “X” next to it in the “Select” column, and pressing the “END” (F11) key. To cancel, press “END” without selecting a data set.

The “Separate-but–Equal” approach is depicted in the next figure. A second approach is used whenever the nature of the application is such that one of the modes is more frequently used or is otherwise more “important” than the others. For example, imagine an application designed to manage a mailing list maintained in a SAS data set for use by a newsletter publishing operation. One mode of this application would be a PROC FSEDIT step and associated processes for adding, deleting, and updating the addresses, and would be used on a daily basis as changes were received. Other modes would use PROC FSLETTER and PROC FORMS to produce renewal notices, invoices, and mailing labels on a much less frequent schedule, probably only once a month.

It would most likely be more convenient for the user if the “primary” mode of the the application, the FSEDIT session for editing the address file, were automatically invoked at the start of the application. When the user exited FSEDIT, the application would then display the menu screen, allowing the user to either select one of the “secondary” modes, or exit the application entirely. If a secondary mode is selected, it would return the user to the primary mode when it is completed. Most applications that use an on-screen data edit/entry process as the principal source of data for other reporting or analysis processes, are usually more convenient for the users if implemented in this manner. As a matter of fact, PROC FSEDIT and FSCALC, both modal applications themselves, use this “primary-over-secondary” menu model to implement their built-in reporting and screen-layout modes.
Once the number of modes necessary to support a given application becomes too large, the simple menu models shown above are too limited. A menu screen with too many options can be cumbersome and intimidating to many users. This problem can be addressed by grouping together logically-related collections of sub-processes, and "nesting" these options into menus that are subordinate to the primary menu. For instance, if an example application had several report-generation alternatives to offer the user, we wouldn't need to list them all on the primary menu screen; a single item on the primary menu designated something like "Generate Reports" would, when selected, invoke an secondary menu screen that listed all the specific reporting options. The user could then select the specific report needed from the secondary menu.

Help Screens

The structures we have discussed so far, alerts, dialogs, and menus, have two purposes to make the application easier for the user to operate, and to enable the application to communicate with the user so that it may perform its designated tasks. The last structure that we will discuss, the help screen, is committed to only one of these functions. Its sole purpose is to make the job of the user as easy as possible, by integrating documentation for using the application within the application itself, ready to be consulted whenever the user indicates a need to do so.

Like the other models, there are many ways to implement a help screen within interactive applications. The simplest form of a help screen is to simply set up a help mode within the menu structure of the application. When the user needs help, they return to the primary menu and select the help mode option, which will prompt the application to print the documentation to the screen for the user to read. If the SAS/FSUP product is available, PROC FSUST provides an excellent facility for this purpose, allowing the user to scroll forward and back through the help file, or search using the FIND command, until the desired topic is located, and then to cancel immediately and return to the menu once the necessary information has been read.

The problem with this sort of approach arises when applied to relatively complex applications. The user may not realize that help is needed until a particular process has been selected, and the user may have already responded to several dialogs. If the user has to cancel at this point to return to the menu and enter help-mode, the entire process will have to be repeated. As the complexity of the processes and their associated interactions with the user increases, this becomes more of a burden to the user than an aid. In these cases, in order to implement a help facility that the users are likely to take advantage of, you must resort to a context-sensitive model.

In a context-sensitive help facility, rather than having a single help mode that displays the instructions for the entire application, you would introduce a method for the user to invoke the help screen from crucial points in each process. Exactly where the help screen is accessible by the user, and how they indicate the need to invoke it, depends entirely on the design and complexity of the application itself. You would still want to put a help mode option on the menu screen (on each menu screen if you are using nested menus), that contained a brief description of each of the modes accessible from that menu. You would also want to define a method for accessing a help screen for each dialog that the application used, that provided more detailed information necessary to adequately understand and respond to the dialog query. One method for doing this is to allow the user to respond to the dialog with a question-mark ("?"). The application would have to be designed to test for this response first, and if found, invoke the specific help-screen process for that dialog. Once the user dismissed the help screen, the application would release the dialog so that the user could respond again with the proper value for the application to use.

Implementation Techniques

Now that we have completed a discussion of the concepts and guidelines to use when designing interactive applications, we can examine the various approaches to implementing these concepts for various combinations of SAS software products. Keep in mind that the code examples shown are meant to be just that—examples that may or may not be the best approach for your site. The purpose of this tutorial is to teach how to design these types of applications. The examples are intended to demonstrate the interactive capabilities of the various SAS products, and to suggest approaches that have been used successfully in the past.

Macro-Driven Screens

As mentioned earlier, the base SAS Macro Facility provides the elementary tools necessary for interacting with the user by means of the %PUT and %INPUT statements. The %PUT statement is used to write text to the console screen, while %INPUT is used to stop program execution and wait for the user to type a response. Whatever the user types in response to a %INPUT can be stored in macro variables for use as symbolic-substitution code or as a test value for conditional or iterative execution control.

We can use these basic interactive statements along with the rest of the macro programming language to build higher-level interactive structures that conform to the requirements of our interactive models. The %CCAlert macro shown at the top of the next page is one example of an implementation of the "Confirm-or-Cancel" alert model.

The function of this macro can be outlined as follows:

- There are six parameters, "msgl" through "msg6", that the programmer can use to define up to six lines of text for the message of the alert.
- A global variable with the same name as the macro is defined. This variable is used to convey the value of the user's response to the macro that calls the alert. If the user responds positively, i.e. "confirms" the alert, &CCAlert will resolve to a "1" or true value. If they "cancel" the alert, &CCAlert will resolve to a "0", false value. The variable can therefore be used as a test to conditionally execute the process that the alert is warning the user about.
Macros:

- %CCAlert:
  - Macro to produce a "confirm-or-cancel" alert.
  - Parameters to allow programmer to specify:
    - Param to allow programmer to specify.
    - msgfmt=macro:
    - up to 6 lines of text for the alert message.
  - Global:
    - %local:

- %PUT
  - %put str1 ( );

- %do loop
  - %do i = 1 to 6;

- %if
  - %if %quote(&sysbuffr) eq C
  - %then %let CCAlert = 1; /* return a true */
  - %else %if %quote('upcase(.osysbuffr)) eq X
  - %then %let CCAlert = 0; /* return a false */

- %ELSE:
  - %ELSE:
    - %ELSE:
      - %ELSE:
        - %ELSE:
          - %ELSE:

- %PUT statements are used to start printing a box ofasterisks that will enclose the alert message. As a convention, this is recommended since it dresses up the display, and stresses the importance of the message to the user.

- A %DO loop is used to print those message-parameters that are used by the programmer when the macro is called. In order to maintain the "boxed" look of the alert, the programmer must start and end each line of text with an asterisk, and pad the message with blanks in order to balance the text within the box properly. A possible improvement for this macro would be to add mechanisms for automatically centering the supplied text between the asterisks, so that the programmer need only supply the text itself.

- More %PUT statements print the bottom of the asterisk "box", including instructions to the user on how to respond to the dialog. The %INPUT statement stops execution of the entire application until the user responds.

- After the screen is cleared to remove the alert message, the value of the user's response, stored in the automatic macro variable "sysbuffr" by the %INPUT statement, is tested for validity and the value of &CCAlert is set accordingly. If the response is invalid (not a "C" or "X"), an error message is printed informing the user of that fact, and the macro re-executes itself recursively until the user gets it right.

Implementation of this macro could look something like this:

```sas
%macro %ccalert (data=; libname=; name=); /* Macro to produce a "confirm-or-cancel" alert */
%local nobs;
%do i = 1 to 6;
  %if %quote(&sysbuffr) eq C
  %then %let CCAlert = 1; /* return a true */
  %else %if %quote('upcase(.osysbuffr)) eq X
  %then %let CCAlert = 0; /* return a false */
%end;
%else %do i = 1 to 6;
  %if %quote(&sysbuffr) eq C
  %then %let CCAlert = 1; /* return a true */
  %else %if %quote('upcase(.osysbuffr)) eq X
  %then %let CCAlert = 0; /* return a false */
%end;
%end;
%end;
%end;
%end;
%end;
%end;
%end;
%end;
```

Assuming that the macro variable "&userData" contains the name to be used to store the user data set. The %nobs macro is used to test for the existence of a data set with the name we are about to use to store our user's data in. If the macro variable %nobs (created by the %nobs macro) has a value of 0, either the data set does not exist or it is empty, and thus our data set name should be safe to use. If it has any other value, however, we are in danger of replacing data, an action which we must give the user the opportunity to confirm or cancel. Execution of the %CCAlert macro with the appropriate message informs the user of the problem, and returns the user's decision as the value of the &CCAlert macro variable. If this value is true, the user has confirmed the action of replacing the previous data, and we use PROC COPY to do so. Otherwise, the user has cancelled, and we execute a dialog to query the user for a new name. Since the new name may also be in use, the entire test/PROC COPY/dialog routine is repeated until the user either selected an unused name or confirms the use of an existing one.

It should be very easy to take this example and expand on it to develop similar macros for simple dialog and menu functions. For example, with the %Dialog macro used above, the macro structure would be essentially the same as %CCAlert except that the externally referenced macro variable would return the actual value of the user's response, rather than a true-or-false flag value. To implement a menu macro, you would need to replace the message parameters with pairs of parameters, to allow the programmer to define the value that the user should type and the process description for each of the menu options. The macro could be designed to automatically invoke a dialog derived from the user option that would execute the corresponding modal process, re-invoking the menu macro when the modal process completed.

Once these macros have been implemented, it is often desirable to develop higher-level macros for alerts and dialogs with specific purposes that many applications have in common. For example, the dialog that presents the user with a list of existing data sets from a specific library may have uses in many applications at some sites. Developing a macro that accepts a list of names as a parameter value, executes PROC CONTENTS for this list, does the symbol-table construction, displays the dialog, and returns the selected data set name to the module that invokes the macro, would enable all these applications to utilize this capability very easily. Like any type of programming, the best way to take advantage of the benefits offered by interactive programming in a cost-effective manner is to look for and identify these types of potential "building-block" routines, and implement them as easily and as possible.

**Macro-driven SAS/FSP**

If your site has the full-screen procedures, it is often desirable to use the macro structure to allow the user to specify the desired features of the %PUT/%INPUT statements for interacting with the user. By using PROC FSEDIT as the basis for your dialog screens, for example, you could create a single screen that queried the user for several items of information. PROC FSEDIT's capability to define field-attributes such as length, type (numeric or character), and input/display format (which will automatically convert date strings to numeric values, for instance) all contribute to the convenience of both the programmer and the user. You can even pre-assign default or suggested values as answers to the queries when appropriate. Variable fields can be assigned a "protected" status, enabling you to pass variable values into the display for titles and instructional messages. Finally, color and highlighting can be used to draw the user's attention up the display when appropriate.

Since the appearance of the display, including these field attributes, is controlled by the layout of the FSEDIT screen catalog entry used by the procedure, it remains separate from the

57
code. It is therefore very easy to modify this layout to fine-tune the titles and instructions using FSEDIT's full-screen editing facility, without modifying any of the macro code. Also, many specialized dialogs, such as the example that uses PROC CONTENTS output to allow the user to select an existing data set name, could have uses in more than one application. It is likely, however, that each application would require a different screen layout (titles and instruction to the user), but the code and actual fields used on this screen would be exactly the same. A single macro written to perform the complete functionality of this dialog could be used by each application, referencing a different screen catalog entry as a parameter value. In this way, the macro can display a screen that is customized for the application, containing the titles and instructions it needs, without the need to customize any of the code.

However, the macro-driven SAS/FSP approach has its drawbacks: since the FSEDIT procedure deals with SAS data set variables, and most applications will need the user responses stored as macro variables in order to be useful in symbolic-substitution or conditional/iterative execution control, you need to take a somewhat round-about approach to reading the user responses:

- First, a temporary data set with a single observation must be created in a DATA step, with variables for each user response you want to collect. These variables should be of the proper type and length for the values expected.
- Invoke PROC FSEDIT referencing the data set created, and the screen catalog containing the desired display layout (you will have to do this once outside the application in order to create this screen layout using FSEDIT screen-modification mode).
- Follow the FSEDIT session with a DATA _NULL_ step that uses the SYMPUT function to transfer the value of each data set variable to a corresponding macro variable.

An example of a general-purpose dialog macro developed along these lines is shown below:

```sas
%macro FSMDialog; /* Macro to produce dialog screens using PROC FSEDIT */
  screen=; /* Name of FSEDIT screen catalog containing display layout */
  fields=; /* List of field names, separated by single spaces */
  format=; /* List of corresponding formats, separated by spaces */
  global attmap=; /* Define field list on global symbol table */
  local attmap=; /* Define attributes for each field */
  value=; /* Define values for each field */
  %local i n;
  %global &fields;
  %do i = 1 %to &n;
    %let n = &n + 1;
    output; /* Create a single observation & stop */
    stop;
  %end;
  pproc fsedit data=work:dialog screen=screen; /* Display the dialog */
  clear;
end;
```

A similar macro for controlling menu functions can be developed this way, with the following exceptions:

- Only one variable need be defined in the temporary data set used for the FSEDIT session, for the field that the user will enter some value designating the desired option.
- After the FSEDIT session, the DATA _NULL_ step will SYMPUT the value of this field to a macro variable, as before, but the macro should not stop here. Once the value of the user's option choice is known, it can be used to invoke the macro that controls the processing for that option. When this macro has completed, and execution control returns to the menu macro, it can re-invokes itself, allowing the user to return to the menu and select a new option.

Modal applications would be implemented with this macro by first writing macros that invoke the processes for each mode. These macros would be named according to a simple convention that would enable the macro to re-invokes itself, allowing the user to return to the menu and select a new option. The example implementation shown below should clarify this:

```sas
%macro main; /* Main menu */
  %fsmenu(prefix=main, /* prefix for "main" process macro name */
    screen=sasfile.mainmenu, /* main" menu screen layout */
    exitopt=x /* value of exit option */
  )
end main;
```

```sas
%macro main1; /* code for main option 1 process */
end main1;
```

```sas
%macro main2; /* code for main option 2 process */
end main2;
```

```sas
%macro main3; /* code for main option 3 process */
end main3;
```

```sas
%macro report1; /* code for report option 1 process */
end report1;
```

```sas
%macro report2; /* code for report option 2 process */
end report2;
```

```sas
%macro report3; /* code for report option 3 process */
end report3;
```

This code shows how to implement an example discussed earlier: a modal application with three options offered at the primary menu, where one of these options invokes a "nested" menu with two report options offered. The first macro, %main, is the module invoking the primary menu. Its name is used as the prefix for the names of the macros corresponding to the three mode options it offers. If the user enters a "1" at this main menu, the %main menu macro will invoke the macro named %main1; a "2" will invoke the %main2 macro; and so on. After each of these modal macros has completed, %main will re-invokes the %main macro, which will start the whole process over again. Since we have designated mode 3 as a nested menu, we see that %main3 in turn invokes a macro named %report, which is simply another %fsmenu module. For this menu, we use the name the macro "report" as the prefix, so that it will invoke the %report1 and %report2 macros as its modes, and return to the nested report menu on completion.
Both %fsmenu calls use the "exitopt" parameter to define the value of "x" as the option to be entered by the user to exit the menu. The "exit" parameter is used to define the macro to be invoked when this exit option is selected by the user. It is used on the %fsmenu call in the %report macro to designate that the %main macro is to be invoked; in this way, the user will return to the primary menu when work from the report menu is completed. Since no "exit" parameter is used on the %fsmenu call in the %main macro, %fsmenu will execute the default exit procedure, an ENDSAS statement that terminates the SAS session.

An example of one way to code this %fsmenu macro is shown below:

```sas
%macro fsmenu(); /* Macro to implement menu control using PROC FSEDIT */
  screen=; /* Name of FSEDIT screen catalog with display layout */
  prefixed-option, prefix used for naming modal macro definitions */
  exitopt=, /* Option value to be entered to exit the menu routine */
  menu=; /* Name of macro to be executed upon exit of the menu */
  options noclone; /* Turn off macro-resolution validation checking */
  data work.menu; /* Create menu data set */
  options nomerror; /* Turn off macro-resolution validation checking */
  call symput('option='); /* Define field for user option-choice value */
  stop;
end;

proc fseedit data=work.menu screen=screens; /* Display menu screen */
  class;
  run;

data _null_; /* Obtain the user's response as a macro variable */
  set work.menu; /* The name of the appropriate model process macro should be derived */
  options nomeclat; /* by concatenating the value of prefix with the user's option code */
  call symput('option=',upcase(compress('prefix='|option) ));
  stop;
end;

/* Compare resolved & unresolved macro text to see if macro exists */
  if %quote(=option=) ne '%quote(-option=) then
    options macro; /* Define macro */
    invoke option macro =;
    %prefix= /* Return to current menu */
    options macro=; /* Option specifies a defined macro */
  end;

/ * Macro for user's option does not exist */
  options macro;
  prefix=&option; /* Invoke option macro */
  %prefix= /* Return to current menu */
  options macro=; /* Option specifies a defined macro */

/* Macro for user's option exists */
  options macro;
  prefix=; /* Invoke option macro */
  %prefix= /* Return to current menu */
  options macro=; /* Option specifies a defined macro */

/* User has entered the "exit" option */
  options macro;
  prefix=; /* Invoke option macro */
  %prefix= /* Return to current menu */
  options macro=; /* Option specifies a defined macro */

/* Execute exit macro */
  options macro;
  prefix=; /* Invoke option macro */
  %prefix= /* Return to current menu */
  options macro=; /* Option specifies a defined macro */

/* User has entered the "exit" option */
  options macro;
  prefix=; /* Invoke option macro */
  %prefix= /* Return to current menu */
  options macro=; /* Option specifies a defined macro */

/* Invalid option alert */
  options macro;
  prefix=; /* Invoke option macro */
  %prefix= /* Return to current menu */
  options macro=; /* Option specifies a defined macro */

/* Macro for user's option does not exist */
  options macro;
  prefix=; /* Invoke option macro */
  %prefix= /* Return to current menu */
  options macro=; /* Option specifies a defined macro */

/* Macro for user's option exists */
  options macro;
  prefix=; /* Invoke option macro */
  %prefix= /* Return to current menu */
  options macro=; /* Option specifies a defined macro */
```

**SAS/AF Software**

The SAS Applications Facility (SAS/AF) product is a set of procedures designed specifically for implementing the very type of interactive application we are discussing here. It combines the symbolic-substitution and conditional-execution control structures of the Macro Facility with the full-screen interactive capabilities offered by the SAS/FSP procedures. It allows a programmer to easily implement very sophisticated applications using menus, dialogs, alerts, and context-sensitive help without the need for complex macro programming. The sole purpose of the SAS/AF product is to provide all the implementation support necessary for the interactive functionality of the application. Once the application-specific functions are coded and tested, they can be integrated with the user-interface and control functions with very little additional coding.

The design strategies and user-interface models presented above apply very well to the use of this product, but an example that sufficiently demonstrates its capabilities is beyond the scope of this tutorial. A very brief discussion of the basic design and philosophy of the software follows; for additional details and examples, the reader should consult the appropriate documentation.

SAS/AF software presents a bit of a departure from the traditional method of coding SAS applications, in that the SAS code is stored as a set of screen entries in a SAS library member, the same way PROC FSEDIT screens and PROC FSPCALC spreadsheets are stored. Each entry is defined as one of four types, based on its specific role in the development of the application:

- **Program Screens**, used to define dialogs, contain both the full-screen display layout and the SAS code affected by the user's response.
- **Menu Screens**, as the name implies, are used to define menus which link program screen sequences together into modal applications.
- **Help Screens** are used to define displays for providing context-sensitive help.
- **CRT Screens** enable the development of Computer-Based Training applications, which can function as a tutorial mode for other applications or stand alone for topics not specific to any application.

Applications are implemented by first designing the program screen sequences containing the SAS code that make up the steps of each process along with any dialog and alert displays needed. The program-screen modules are then linked together, using conditional and iterative processing if necessary, into the processes that make up the separate modes. These modal processes are then linked together using menu screens, which allow very easy definition of complex nested menu structures. Help screens are then used to define context-sensitive help displays that can be linked to (and invoked from) any menu screen, program screen, or individual response field.

This unique approach to defining and storing program modules and execution logic as screen catalog entries poses some problems, however. Programmers experienced in traditional SAS macro-based application development often have difficulties working in this environment, especially if macro-based modules must be used to share code between different applications. The SAS/AF product does, however, provide adequate mechanisms for integrating the screen-based modules with the macro facility, and the program screens support the use of the %INCLUDE statement. Macro-based code can therefore be stored and maintained in the traditional manner and included into the program screens for use by the SAS/AF-based application.

---

Version 6 (PC) SAS

SAS Institute announced a new release of the base SAS product for microcomputers last year, and with it they introduced a powerful new method for implementing user-interface techniques through the DATA step language. By using two new DATA step statements, WINDOW and DISPLAY, programmers can define customized windows of varying size and location on the screen, each with its own command line, text messages, titles and user-response fields. The complete facility of the DATA step language can be used to validate and process user responses, making it especially well-suited to sophisticated data-capture and editing applications.
The downside of this capability is that Version 6 does not yet have a complete implementation of the macro facility. The SYMPUT function, desirable for translating data set variable values captured with the WINDOW/DISPLAY into macro variables, and the %DO-END and %IF-THEN-ELSE structured statements needed to use these values for execution control, are not available in the current release.

Menus and dialogs for execution-control queries can still be implemented by using the DATA step language itself as a program-text generator, the same way that the macro language would be used. A DATA_NULL_step can be used to display the window, and code appropriate to the user's response can be written to an external file using FILE and PUT statements. This code can then be submitted for execution using the %INCLUDE statement. The simple example below demonstrates this.

```sas
/* saved as 'MKTGIS/TEMP.SAS' */
data _null_; /* Data step to manage menu display logic */
  window menu /* Define the menu screen display */
  %include 'MKTGIS/MENU.SAS'; /* Include information system primary menu */
  %include 'MKTGIS/TEMP.SAS'; /* Include temporary file */
end; /* End primary menu */

/* Enter your option here: 'option 01. a-rev_video*/
file 'MKTGIS/TEMP.SAS'; /* Temp. file to hold generated code */
okay = 1;
do until (okay); /* Exit when valid option entered */
  display menu; /* Display menu */
  okay = 1;
  select (option); /* Select option */
    when ('1') do; /* Option 1 selected */
      okay = 1;
      put 'proc print data = mktg.sales uniform;';
    end;
    when ('2') do; /* Option 2 selected */
      okay = 1;
      put 'proc fsed!t data = mktg.prospects screen = mktg.proscr;';
    end;
    when ('3') do; /* Option 3 selected */
      okay = 1;
      put 'proc fsedit data = mktg.execs screen = mktg.execscr;';
    end;
    otherwise do; /* Invalid option entered */
      okay = 0; /* Report loop using message area for alert */
      put 'The option entered was invalid. Please try again.;'
    end;
  end;
end; /* Do until option selected */

/* Do until select */
stop; /* Stop execution */

/* Include 'MKTGIS/MAIN.SAS'; execute the generated code */
/* Include 'MKTGIS/MAIN.SAS'; re-execute the menu driver code */
```

If the code to be generated with the PUT statements is lengthy, it can be pre-written and stored in separate files, and the PUT statement used to generate an %INCLUDE to these files. In this way, these DATA_NULL_step modules can be nested within each other and used to create nested menus, dialogs, alerts, and any other specialized user-interface displays necessary for your application.

Conclusion

By categorizing the user-interface processes used by interactive applications into a few simple groups, and implementing software tools that perform these processes and can be shared by all applications, the development team can save a great deal of time, effort, and frustration over the long term. The model presented here has proven to be simple enough for development teams that are inexperienced with this type of project, yet comprehensive enough to provide a good basis for getting started. It is not, however, intended to be the final model, comprehensive enough to meet all the needs of an organization.

As the development team gains experience, they should constantly examine the functionality of the software tools, looking for ways to further enhance this functionality by adding capabilities to the basic tools, and adding new tools for specialized functions.

The author can be contacted at the following address:

Software Applications and Training Division
ORI, Inc.
601 Indiana Avenue, NW, Suite 1000
Washington, DC 20004
Telephone: 202/737-2666

Acknowledgements

Much of the content of this paper is a result of work done at the Blue Cross and Blue Shield Association Federal Employee Program Director's Office, and as a result, credit is due to Dorothy Thomas, Jackie Nelson, Patrick Quigley, Frank Velez, Morton Rumberg, and Newton Plaisance of the Office of Data Systems Management for their valuable contributions.

SAS, SAS/FSP, and SAS/AF are registered trademarks of SAS Institute, Inc.