Paper 10-25

Building A Better Data Entry Application Using PROC FSEDIT

Derek Morgan, Washington University Medical School, St. Louis, MO
Michael Province, Washington University Medical School, St. Louis, MO

Abstract

It was once said that, "If you build a better mousetrap, the world will beat a path to your door." However, for data entry system programmers, "a better mousetrap" is designed to keep people away from our doors. We don't want visits from users with questions about using the system, nor from analysts who want to know why there are 500-pound children sprinkled liberally through the database. Finally, we definitely don't want administrators and supervisors to come knocking, demanding to know why the users and analysts are calling them.

This advanced tutorial will cover items beyond the basics of the FSEDIT procedure with the goal of building a better data entry system. Familiarity with SCL is helpful. The topics to be covered include:

- Screen Ergonomics (use of color and placement of items)
- The difference between the FSEINIT section and the INIT section
- Cursor control—when NOT to use the defaults
- Screen variables vs. data set variables
- Obtaining data from an existing data set
- Creating custom commands
- Conditional entry ("skip logic")
- Field validation at entry
- Using selection lists to fill a field
- Customized PMENUS and customized function key assignments
- Control over deleting records

Introduction

PROC FSEDIT at its most basic is a utilitarian data entry tool. However, with some SAS® System programming tricks using Base SAS software and the SAS Component Language (SCL), a PROC FSEDIT screen can become a robust self-contained data entry application instead of just a data entry tool. The ideal data entry system is intuitive, self-documenting and yields clean data. This tutorial is designed to bring a basic PROC FSEDIT screen closer to that point by illustrating some of the methods we have developed, and providing code that can be imported into any FSEDIT screen.

Analysis Issues

A good FSEDIT application begins like any other application: with a concept. The more clearly envisioned the concept, the better the application. For a data entry screen, however, the concept frequently begins and ends with, "turn this form into an FSEDIT screen." This is not as simple, nor as well developed, a concept as it appears. The complete data entry concept has to take into account the entry method, the skill level (or range) of the users, the data that should result from this entry, and form layout, both on paper and on the screen.

Entry methods can include real-time paperless data entry, paper form transcription, and double data entry. The skill level of the users may range from a computerphobic clinician to 10,000 keystrokes/hour experienced data entry operators. The data that is obtained by this entry can be thought of as one large data set or several smaller data sets. Finally, form layout has a definite impact: it is not always easy to create a mirror image of the form on a computer screen. All these issues affect the design of an FSEDIT screen.

Single entry forms can be transcribed from a paper form, or for those who prefer working without a net, entered during real-time in-person or telephone interviews. If the questions are on paper, then what is on the screen does not necessarily have to correspond exactly with the words on the paper. This also makes it easier to synchronize the screen pages with the paper pages. Paperless data entry turns the FSEDIT screen into both data capture method and interview instrument. The text on the paper template must be echoed exactly on the screen. It would be a mistake to read from the paper and enter the response on the screen, especially in a face-to-face interview situation. It would be too easy to get out-of-sync or to delay in conducting the interview while the question/answer combination is verified.

If double entry is to be used, there are methods for point-of-entry verification inside the FSEDIT screen itself, as well as methods for automating the traditional enter/compare/report/correct cycle. Whichever of these methods is chosen will have an obvious impact on the programming of your screen.

User skill level is also a consideration in the design of a robust screen. Extra care must be taken for inexperienced data entry people, or the potential for incorrect data is much greater. If too much care is exercised when an experienced data entry operator is the primary user, then the frustration will increase, and you will either lose the operator, or suffer a reduction in data quality due to operator error.

After all the above factors have been considered, and any related questions that may arise have been answered (e.g., how often will these forms and the related data change over the life of this project?), the concept is much more clearly defined. Now you have enough information to design a robust data entry system.

Design Theory

Reducing user fatigue through screen ergonomics is perhaps the easiest way to improve the data. First, there's the issue of the color scheme. A user should not have to work to read the screen. A good color scheme should be easy to read and will create de facto visual cues. Select a color scheme and stick with it, especially in multiple-form, multiple-screen applications. The color scheme I prefer is:

Background: Black
Form Text: Cyan
Response: Yellow
Valid Response Categories: Reverse Green
Operator Information: White
Important Information: Reverse White
Calculated or Protected Fields: Reverse Cyan

The black background is used because almost all colors will show up against it. The others are purely arbitrary, but have been endorsed by our users. We use these colors in every FSEDIT screen in every application.

Next, there is the matter of screen legibility and how closely it corresponds to the form being entered. In general, keeping screen and form pages in parallel depends on three things: screen size, available fonts, and the design of the paper form itself. If the paper form is crowded, forget about using one screen page for one paper page. This is unimportant as long as the FSEDIT application handles the screen paging without user intervention. The CURSOR statement in SCL allows you to move the cursor to any field, so that you have control over cursor movement.

Remember to use blank space liberally, especially in paperless data entry, where the user has to read the screen constantly. Too much text in not enough space is a bad thing. Eyestrain and the resulting fatigue can cause needless entry errors.

Making It Happen: Using The Power of SCL

To this point, data entry system development has been discussed in philosophical and general terms. The SAS Component Language (SCL) is what allows FSEDIT to become a complete application. Using SCL, you can cause things to happen on a field-by-field basis; whenever the application is opened or terminated; and/or when a record is displayed on the screen. This allows for tight programmer control of FSEDIT screen applications. The remainder of this paper will demonstrate techniques used to build a robust FSEDIT screen. This SCL code has been used on a PC platform under release 6.11 of the SAS System.

SCL Labels in FSEDIT Screens

An SCL program can have several labels. There are five predefined labels that serve specific purposes within a program: FSEINIT, INIT, MAIN, TERM, and FSETERM.

Screen code starts with either an FSEINIT or an INIT section. The most important distinction between the pair is that FSEINIT code gets executed one time per screen invocation, while INIT gets executed each time a record appears on the screen. If there is something to be done only once at the start of the session, use FSEINIT. SCL list creation, opening auxiliary data sets, defining functional control of the screen with the CONTROL or CALL SETCR are all prime examples of the tasks suited for the FSEINIT section. Any non-record dependent actions or things that will not change with each record can be done here.

The INIT section is for actions that need to occur once each record, when the record is placed onto the screen. Opening a specific record in an auxiliary data set, creating a data-sensitive SCL list, or initializing fields for the record being displayed are examples of tasks that will only work in the INIT section. As a general rule of thumb, anything that is record-oriented (data dependent) must be done in the INIT section.

MAIN

This section can be summed up by saying that this is where code that is not specific to a field, or does not belong in one of the opening or ending labeled sections, should go. The specifics of when this code executes depend on the CONTROL and CALL SETCR parameters in effect. However, this is where any interrupt processing or custom commands are coded. Additionally, any data-based calculations can be done in the MAIN section, especially if the calculation depends on more than one field on the screen. If any of the fields are changed, the calculation will need to be redone. So, if two fields are used, the calculation code (or a LINK statement to the calculation code) will need to be in the labeled code for both fields. If the code is in MAIN, it will be done whenever the MAIN section executes. A LINK statement in labeled code is more execution-efficient, but leaving it in MAIN is probably the easiest to maintain.

Closing It Down: TERM and FSETERM

At the close of the record or screen, the special labels are TERM and FSETERM. They can be thought of as the inverses of INIT and FSEINIT, respectively.

The TERM section executes when a displayed record is closed (usually with an END, FORWARD, or BACKWARD command.) However, this only occurs if the record has been modified, or the TERM option has been specified in the CONTROL statement. The FSETERM statement executes when the FSEDIT session ends. What do you do in these sections? Close or update any auxiliary data sets that you've opened. Generally, if you open a data set in the INIT section, close it in the TERM section; if it was opened in FSEINIT, close it in FSETERM.

The TERM section can also be used to perform any final calculations, or to set modification variables in a record. For example, a timestamp for a record may be useful. You would do this in the TERM section, without CONTROL TERM in effect. That way, the timestamp would only be changed if the record was modified. You could also copy the record to a transaction data set in this section. Again, it would only be copied if a modification was made. Closing down the record and application can be as important as the work done to open them up, so these sections should not be thought of lightly.

Example: TERM

```
TERM:
entryid = SYMGET('userid');    ①
touchdat = DATETIME();         ②
CALL PUTVARN(stat,VARNUM(stat,'chk'),1);  ③
rc = UPDATE(stat);             ④
rc = CLOSE(stat);              ⑤
RETURN;
```

This example shows some of the uses of the TERM section. In ①, a data set variable is set using an existing macro variable, while ② indicates the creation/updating of a timestamp on the record. Items ③, ④, and ⑤ change a variable in an auxiliary data set by writing it, updating the record, and closing the data set.

These are the five predefined SCL labels in an FSEDIT program. You can create as many additional labels in your application as memory will allow, and later I'll show what you can do with your own labels.

Total Control: Timing and Cursor Movement

To determine the way that the code in a screen application will execute, use the CONTROL statement. The options for the CONTROL statement allow you to:

- Enable the use of custom commands
- Cause the MAIN section to execute only when a window variable is modified or an error is detected
- Execute specified code when a user break is signaled
- Execute a labeled section of code when a field is modified
- Force execution of code in the TERM section regardless of whether the record was modified.

With CONTROL ALLCMDS enabled, the MAIN section executes before any command is processed (including SAS system commands). You can then create your own custom command (or hijack a SAS command) by processing the word (or words) to execute SCL statements.

CONTROL LABEL causes a labeled section of code to execute when a field with the same label is modified. This can be used to modify flow of entry based on the value of a field, provide value verification, or perform calculations. As you will see in a moment, the exact definition of "modified" can be changed to suit your purposes.

There are other CONTROL options available as well. CONTROL BREAK can be used to prevent users from hitting <CTRL-C> and shutting the application down prematurely. CONTROL ENTER will cause the main section to execute whenever the <ENTER> key is pressed. CONTROL ERROR causes the MAIN section to execute when an error condition is detected. Finally, CONTROL TERM will cause the TERM section of code to execute regardless of whether the record has been modified or not. Without this option, the TERM section of code only executes if the currently displayed record has been modified.

CONTROL ALWAYS includes the ENTER and ERROR options. Specifying ALLCMDS in FSEDIT is the same as using ALWAYS.

The next level of control in FSEDIT screens is defining how the SCL program handles the <ENTER> key press, using the CALL SETCR statement. This is more important than it may seem, because you can use this to change the movement of the cursor in response to <ENTER>. You can make the cursor move to the next horizontal or vertical field. You can also move the cursor to the command line. Or you can make the cursor stay where it is.

CALL SETCR can also cause <ENTER> to be interpreted as a modification of a field, whether or not the field has been changed, and is another way to effect CONTROL ENTER. To create a tightly controlled application, the options in CALL SETCR should be "STAY", "RETURN", and "MODIFY". "STAY" prevents the cursor from moving automatically, "RETURN" makes the MAIN section execute with each <ENTER>, and "MODIFY" insures that labeled code will execute (in conjunction with CONTROL LABEL) whenever <ENTER> is pressed and the cursor is on a field. This means that you have complete control of the cursor. Labeled code will execute even if the field hasn't been modified, so the screen will react the same way at each field, regardless of user input.

Any discussion of cursor movement would not be complete without taking note of the default AUTOSKIP field attribute. Under the FSEDIT Screen Menu, item 4 ("Assign Special Attributes to Fields") contains various parameters for every field on the screen. One of them is called NOAUTOSKIP. If you toggle this by putting an "N" on the field, the cursor will not move when the maximum number of characters has been entered in the field. A tightly controlled application needs to have all fields set to NOAUTOSKIP, so that only the SCL is responsible for cursor movement.

The Form — Building an Application

As mentioned earlier, every FSEDIT screen starts with an idea. Our "idea" is a two-page medical survey form that requires:

1) Confidential contact information stored in another data set
2) Verification of numerical values
3) Yes/No/Don't Know questions
4) Skip patterns
5) A time stamp on each record
6) Text entry that needs to be coded with a code book

Later, contact with the person's physician may be required to verify any prescription medications. This form may be read over the phone to a subject and the answers entered directly into the database.

Let's start our analysis with the last point. The screen must have the entire text of the form on it since it may be used in a paperless situation. Also, given that we don't know the exact size of the screen being used (although we can say that it will be 1024×768 resolution), trying to format the screen exactly the same as the form is not recommended.

Our data entry people are trained interviewers, not necessarily experienced data entry operators. In an interview situation, you don't necessarily want the operator to have to remember the coding system for male/female and Yes/No/Don't Know. We now have a relatively well-defined application, and can begin our screen.

Fields and Variables - What's The Difference?

FSEDIT users enter data into fields on the screen. These fields may or may not be associated with variables in the FSEDIT data set. Variables that are in the data set do not have to be fields on the screen, and vice versa. You can define any variable from the data set being edited as "unwanted" during the field identification phase of the screen modification menu item, and it won't be on the screen.

To create variables that are on the screen, but not in the primary data set, answer "Y" to the question about creating any "computational or repeated fields" after you have created the text for your entry screen. This will bring up a screen for you to fill in the field names, types, and any formats or informats to be associated with these

fields. Any fields on your screen that aren't in the data set are defined here.

So before you start coding, look at the variables on the form, in the data set and the fields that are to be on the screen. In most cases, it is good to have some sort of record identifier on each screen page. For our sample application, this would be the ID and the subject's name. You also need to be aware of any information that is not in the data set being edited, but needs to be on the screen. This is where the difference between screen fields and data set variables becomes important.

A repeated field is one that is going to be on the screen more than once. It does not have to be associated with a data set variable, but it can be (e.g., to put the ID number on each screen page, you would define a repeated field called "ID" for each page). A "computational" field is one that is not in the data set, but is to be displayed on the entry screen. It does not necessarily have to be the result of a computation.

So, to put the subject ID and name on each screen page, we would use the "R" field type for ID and name, indicating that these are repeated fields. For the contact information, we would define a screen field for each variable from the contact data set (use the contact data set variable names for all the screen fields except the phone number; phone numbers are handled in a special fashion). Once we have associated fields on the entry screen with their variables, we are ready to begin coding.

First, let's think about any auxiliary data sets we'll need. Item 6 on the previous page indicates that we will be using a codebook for the medications. Instead of having the user look it up in a book and enter the code, we will build a "mini-application" using a medication dictionary data set. Since this data set is universal across the application, we will open it at the beginning and close it at the end of the application.

The contact information for each person is also in a separate data set. This data should be available for correction, but not stored in the survey database. Since this will change for each record, we will have to open it at the start of each record, and close it at the end of each record. Therefore, this task clearly belongs in the INIT section. We have now defined the program tasks that need to occur in the FSEINIT and INIT sections, in addition to specifying control of the entry screen.

Example 2: FSEINIT for the Sample Application

```
FSEINIT:
CONTROL LABEL ALLCMDS
CALL SETCR('STAY','RETURN','MODIFY');
CALL EXECCMD('ZOOM ON');
LENGTH cmd tmp $ 100;
med_dictionary = OPEN("meds.medcode","I");
RETURN;
```

Notice that the variable name for the medication dictionary is longer than eight characters. You can do this in SCL under version 6! We've just opened the data set for use; nothing else needs to be done with it yet. The CONTROL options are LABEL, since you want to have certain things occur at each field and ALLCMDS since you will be creating a custom command to print out the information for the physician contact. This will also make sure that MAIN is always executed.

The CALL SETCR options indicate the tightest control possible. The cursor will not move when <ENTER> is pressed unless directed to by the SCL program. CALL EXECCMD is used to maximize the entry screen. The length statements are optional. However, all SCL character variables are 200 characters in length unless explicitly defined otherwise. In version 7 and beyond, character variables may be up to 32K in length, but the default is still 200 characters. Before we move on to the code for the INIT section, let's look at using auxiliary data sets in FSEDIT.

Using Auxiliary Data Sets in FSEDIT

The primary data set in FSEDIT is the one specified in the CALL FSEDIT or PROC FSEDIT statement. Fields are automatically aligned with the variables during the field definition phase of screen development. What happens when you have to get information from a different data set for use in the screen as with our contact data set?

The OPEN and FETCH SCL statements are what you use. OPEN allows you to use another data set, and FETCH will get the information from the current record available in that data set. There are several ways of selecting the correct record so that you can FETCH it.

1) Use an SCL data set search command: LOCATEC will allow you to search for a character value, LOCATEN for numeric values.
2) FETCHOBS will get a specified record by its record number. This takes the place of the FETCH statement.
3) Use NOTE to mark a record in an auxiliary data set, and then find it with POINT later.
4) Use the SCL WHERE command.
5) Use the WHERE= data set option when you OPEN the data set. I prefer this method if the where clause and data set name will fit in a character variable. Store the data set name to be opened into a variable, and use dsid=OPEN(*cvar,attribute*); where *cvar* is the name of the character variable with the data set name and where clause.

Once you have FETCHed the data, you have to get the information into your screen. What FETCH actually does is move data from the data set into the data set data vector, which is different from the fields on your screen. To move this data into the fields on your screen, you will use CALL PUTVARN, CALL PUTVARC, or CALL SET. CALL PUTVARN(C) moves the numeric (N) or character (C) data from one variable into the SCL data vector (SDV). You can then assign it to a variable in your SCL program or a field on your screen. CALL SET will move the data from all the variables in the auxiliary data set to the SDV. If fields on your screen have the same name as variables from the data set, they will automatically be loaded.

CALL SET is very convenient and very powerful, because it can take the place of many CALL PUT/GETVARN(C) statements. However, this convenience comes with a price: the possibility of inadvertently altering variables in either your screen data set or the auxiliary data set. If there are variables with the same name in both data sets, a fetch command will overwrite the fields on your screen with the values from the auxiliary data set. Similarly, if the auxiliary data set is opened in "update" mode, any

changes made to screen fields with the same names as variables in the auxiliary data set may change the values in the auxiliary data set.

To illustrate, say that there is a variable named "DATE" in both the auxiliary data set and on your screen, but they represent different dates. Using CALL SET would write the date from the auxiliary data set into the DATE field on the screen. This is a problem if we want the current date instead of the date from the auxiliary data set. It would become a larger problem if you were to change the date on your screen, and then update the auxiliary data set. Then the original date on the auxiliary data set would disappear, replaced by the date that was entered on your screen. How do you get around this without coding a CALL GETVARN(C) for every variable you want, so that you don't get the ones you don't want involved? Easy. Use a DROP or KEEP data set option when dynamically defining the data set to be opened. This will allow you to restrict the variables you are using from the auxiliary data set, just as the WHERE clause restricts the record. The example below demonstrates how to code the use of an auxiliary data set.

Example 3: INIT for the Sample Application

```
INIT:
IF id EQ ' ' THEN          ①
   CURSOR id;
ELSE DO;
   PROTECT id;        ②
   dsn = "secure.contact (DROP=date " ||
         "WHERE=(id eq '" || id || "'))";      ③
   hs2inf = OPEN(dsn,'u');      ④
   CALL SET(hs2inf);      ⑤
   fail = FETCH(hs2inf);      ⑥
   IF fail NE 0 THEN DO;
     ERRORON id;
     rc = CLOSE(hs2inf);          ⑦
     RETURN;
   END;
   CURSOR fname;
phonec = LEFT(PUT(phone,phone.));    ⑧
END;
RETURN;
```

Example 3 is the INIT section for the sample screen. Note that it executes based on the presence of an ID number. The key assumption made here is that a blank ID indicates a new record. If it's a new record, put the cursor on the ID field (①). If it is not a new record, then we need to get the information from the contact data set.

First, protect the ID field to prevent any changes (②). This will insure that the interviewer doesn't inadvertently alter the wrong person's data. Now select the record by opening the contact data set with a where clause. This is done by dynamically creating the name of the data set to be opened (③). The DROP= option keeps us from accidentally involving the DATE variable from the contact data set. We open the data set (④) in the "update" mode. This will allow our interviewers to correct the contact information in its data set if necessary.

⑤ is the all-important CALL SET. The FETCH command (⑥) actually loads our record from the contact data set. No searching for the correct record is required since we used a WHERE clause to select one record when we opened the data set. If FETCH returns something other than 0, there's an error condition, and the user needs to be alerted (⑦). This concludes our use of the contact data set in the INIT section. The next step is to place the cursor on the first field to be edited. Finally, with ⑧, we initialize the phone number field with a character representation of the phone number. This is done as a part of the "phone trick," which eliminates many of the problems with phone numbers. The phone number is stored in the data set as a numeric value, but is not displayed on the screen. What the users see and interact with is a character screen field that represents the nicely formatted value of the phone number.

Now it's time to define the MAIN section for our sample screen. What needs to be done here? The custom command to print out the physician and medication information has to be coded, but there isn't anything else necessary for this application.

Example 4: MAIN for the Sample Application

```
MAIN:
cmd = UPCASE(LASTCMD());    ①
IF cmd EQ 'ADD' THEN          ②
   CALL NEXTWORD();
IF cmd EQ 'PRINTOUT' THEN DO;                ③
   CALL EXECCMDI('SAVE');
   tmp = "sample.app (WHERE=(id eq '" || id || "')";
   CALL LETTER('hg.fsletter.contact','print',tmp);
   CALL NEXTWORD();
END;
RETURN;
```

Example 4 shows how custom commands work. We get the command by using the LASTCMD() function. A command is entered by the user through a mouse click on a menu selection, function key press, or by typing on the command line or in the command box (①). If you have multiple word commands that need to be parsed, you should use the WORD() function. The UPCASE() function is used to make the command case-insensitive.

② is an example of "hijacking" a SAS command. In this case, the "ADD" command is effectively disabled by discarding the command using the CALL NEXTWORD() function. This particular technique is useful if new records are added to the data set using the "ADD" option on a CALL FSEDIT statement as part of a larger application, but you don't want the user to be able to add a new record while in the FSEDIT application.

③ is the coding of the custom command called "PRINTOUT". When ① returns the word "PRINTOUT", the current record being edited is saved, then an FSLETTER application opens it for printing. The "PRINTOUT" command can be defined in a PMENU or associated with a function key for the user. In both ② and ③, the CALL NEXTWORD() function makes sure that once the MAIN section has executed, the command is not still there for processing by the SAS System. Without it, the command is in the command buffer, and SAS will process it.

The next code to be written is for the TERM and FSETERM sections. In addition to closing the auxiliary data sets in use, we are supposed to include a timestamp on each record. So, the timestamp will be assigned to the record being edited, and you would update and close the contact information data set in the TERM section, since both tasks are record-specific. The closing of the medication dictionary will take place in FSETERM, since it

is not record-specific. Once that has been done, the predefined sections of the application are finished. Now we can proceed to programming the fields.

Handling Individual Fields in an FSEDIT Application

This is where most of the time in developing a tightly controlled, robust FSEDIT application is spent. The better job you do at this level, the less time you will spend in user support. In a tightly controlled application, almost every field will have a corresponding section of labeled code, if for no other reason than to move the cursor. If you use the "STAY" option on the CALL SETCR statement, the cursor won't go anywhere you don't tell it to. I highly recommend this strategy if your users have enough knowledge to be dangerous.

What can you do in a field-labeled section? You can implement "skip logic", providing flow-of control based on a response to a question. Consider the section below from a survey form. If the answer to question 4 is "yes", then 4a) and 4b) should also be answered. Otherwise, the interviewer should go on to question 5.

DID YOU EVER HAVE:

4) Heart attack? ____ *(Yes/No/Don't Know)*

4a) At what Age? ____

4b) Were you hospitalized? ___ *(Yes/No)*

5) Bypass Surgery? ____ *(Yes/No/Don't Know)*

Example 5: Skip Logic

```
MI:
IF mi EQ 1 THEN DO;
  UNPROTECT miage mihosp;          ①
  CURSOR miage;
END;
ELSE DO;        ②
   PROTECT miage mihosp;
   miage=.p;
   mihosp=.p;          ③
   CURSOR bypass;
END;
RETURN;
```

The field name is MI, so the labeled section is "MI". First, we test the value of the field. If it is yes, then in ①, the fields for questions 4a) and 4b) are unprotected, and the cursor is moved to question 4a). If the answer isn't yes (②), then the fields for 4a) and b) are protected preventing any input. ③ is optional, but may come in handy at analysis time: the variables for 4a) and b) are set to a special missing value indicating that they were skipped due to protocol. The cursor is then moved to the field for question 5, and the user doesn't have to think about manually skipping the questions.

At this point, it might occur to you that if the entered value is available in labeled code, shouldn't you check it there? Why isn't there something in example 5 that checks to see if the answer entered is yes, no, or don't know? Field validation is a whole topic in itself, and doesn't have to be handled in code at all.

Field Validation

The FSEDIT procedure has a built-in tool for field validation. Under option 4 of the Screen Modification Menu, you have the ability to define a maximum and minimum value for any field on the screen. This is quick, relatively painless, and somewhat useful. The reason it is only somewhat useful is that valid answers are not necessarily encompassed in a continuous range. Let's use question 4 from the previous example. It will either be "Y", "N" or "D". If you choose "D" as the minimum and "Y" as the maximum, SAS will accept all the letters in between as valid answers, so you'd still have to code something in the labeled section. This isn't what you want.

As an alternative, why not say that yes is 1, no is 0, and don't know is 2? Then you can use 0 and 2 as the minimum and maximum. This is a possibility, and it depends on how well your entry people will deal with turning a "yes" they hear on the phone into a "1" on their entry screen. Also, analysts prefer non-answers like "Don't Know" to be coded as missing values.

Finally, there is one large drawback to using maximum and minimum: they are fixed, and have to be changed by modifying the screen. So, for certain items, you need to be sure that there's no way a value is going to be out-of-range, or there's going to be a great deal of programmer time spent accommodating these changes.

There is an easier way. Use PROC FORMAT to create a custom informat. Associate it with the variables in the data set when you build the data set using the INFORMAT statement in Base SAS. One custom informat can be used for an infinite number of variables across an infinite number of projects. With custom informats, you can define all valid values by range, or individually. More important, you can define everything that is not a valid value as an error. Then the SAS System will handle the error, notifying the user and keeping the cursor on the field in question without any additional coding on your part.

Example 6: Using PROC FORMAT to Validate Fields

```
PROC FORMAT LIBRARY=LIBRARY;      ①
INVALUE YND (UPCASE)      ②
        0,'N','NO','0'=0
        1,'Y','YES','1'=1
        2,'D',"DON'T KNOW",'2'=.D
        ' '=.
        OTHER=_ERROR_      ③
;
RUN;
```

This is an example of a custom informat for Yes/No/Don't Know questions. Every question on a survey that has the answer categories "yes," "no," and 'Don't know" can be taken care of by this informat. ① uses the LIBRARY= option, which will store this informat in a permanent format library. The UPCASE option (②) performs all comparisons in upper case (i.e., makes it case-insensitive.) Finally, the informat keyword "OTHER" is set to the special value _ERROR_ (③), which is what triggers the error handling in FSEDIT (and SAS/AF®).

If you look at the values that have been specified as valid input, you will see a mixture of character and numeric values. However, the value that is returned is numeric (0,1, or special missing value .D). This means that during

the input of any variable associated with this informat, "Yes" will be translated to 1, while "No" will be translated to 0, and those numeric values will be stored in the data set. This allows you to accommodate those who like yes and no, as well as those who prefer 0 and 1. It also makes the analysts happy because they don't have to recode character yes/no as numeric for analyses.

There is also another advantage to using this method. If someone decides to add another category, this is the only thing that you have to modify, since it is a data set attribute and not a screen attribute. Were you to code your validation in labeled sections, you would have to change it in every label that checked Yes/No/Don't Know. If you used maximum and minimum, you would have to change the maximum in each affected field on every screen.

Special Validation Cases

This is where coding your validation in the labeled section is useful. Telephone numbers, and possible, but not probable values fall into this category. We have a standard way of handling phone numbers in our applications. First, we define our phone numbers as numeric variables without delimiters, which makes range checking them for errors easier. By using the "phone trick"[1], we remove the hassle from the data entry operator's point of view. No matter what delimiters are used when entering a phone number, it will appear on the screen in a standardized fashion. To make the process of input more painless, we use a character screen field that is not in the data set. Example 7 is code and comments for the label associated with this field.

Example 7: The Phone Trick

```
PHONEC:
/* Check for delimiters in field value.  */
IF INDEXC(TRIM(LEFT(phonec)),'/()- Ext') GT 0 THEN
DO;
/* If present, remove them and store entered data in
a temporary character variable */
        tmp2 = LEFT(COMPRESS(phonec,'/()- Ext'));
/* Convert temp to numeric, store in data set
variable */
        phone = INPUT(tmp2,20.) ;
END;
ELSE
/* If no delimiters, convert entered field to numeric,
store in data set variable */
        phone = INPUT(phonec,20.);
/* Put formatted data set variable into character
screen field */
phonec = LEFT(PUT(phone,phone.));
RETURN;
```

The functions INPUT() and PUT() convert between character and numeric variables. Now, any time that the field PHONEC is changed, the resulting number will be stored in the data set while the value on the screen will have an easy-to-read format. You just have to remember to initialize this field for existing records during the INIT section (see example 3).

The other special case is in-place verification of what was entered. We have found this especially useful for anthropometric measurements. It can be thought of as selective double data entry for cases where the entry is possible, but out of an expected range.

Example 8: In-Place Verification

```
IF ht LT lowlim OR ht GT hilim THEN DO;     ①
   IF NOT chk OR (chk AND oops NE ht) then do;
      oops = ht;      ②
      ht = .;
      ALARM;
      _MSG_ = 'Is the value ' ||     ③
      TRIM(LEFT(PUTN(oops,'BEST.'))) ||
      ' correct?  Re-enter same value to verify.';
      chk=1;    ④
      CURSOR ht;     ⑤
      RETURN;
   END;
END;
CURSOR weight;
chk = 0;              ⑥
RETURN;
```

The value of the field HT is checked against upper and lower limits (①), which if exceeded, trigger the verification. Next, we test to see if this value has not been verified, or if it's been typed differently the second time. If either is true, we put the current value in a holding variable and clear the field. ③ puts an error message on the message line. We set the verification flag in ④, while ⑤ makes sure the cursor stays on the HT field. Then we let the user have another shot at it. If the value checks out or the verification is not triggered, the cursor is moved to the next field, and the flag is reset to 0 (⑥). This results in any suspicious values being entered twice.

The various methods of field validation in FSEDIT detailed here work for data that are entered by the user. What happens when you have a list of possible responses for a question and you want to allow the user to pick one of these choices?

Using Selection Lists in FSEDIT Applications

SCL provides functions to do selection lists. A selection list is a pop-up window that will allow the user to select from the items listed within and automatically use that selection as input for a field. The functions are DATALISTC, DATALISTN, LISTC, LISTN, and SHOWLIST. With SHOWLIST, up to 13 character items can be defined in the statement itself, of which one will be selected by the user and stored in an SCL variable.

LISTC and LISTN operate from SCL lists and return character or numeric selection(s), respectively. You can have an infinite number of selections from these windows. DATALISTC and DATALISTN are similar, except that they use SAS System data sets for the contents of the list. Which one should you use? If you have numeric input, you can't use SHOWLIST, because it only returns a character value. If the items for your selection list are already contained in a data set (especially if the data set has many records), then use the appropriate DATALIST function. Otherwise, create an SCL list and use the LIST functions.

Our selection list is from a medication coding dictionary data set, and we want to get the code numbers for drug names. From example 2, our medication dictionary file indicator variable is "med_dictionary".

Consider the following series of questions on a form:

	Medication Name		Code
11a)	_____	11b)	_____
12a)	_____	12b)	_____
13a)	_____	13b)	_____
14a)	_____	14b)	_____

15) Do you take ASPIRIN or a product containing ASPIRIN daily? ____ (Yes/No/Don't Know)

If the variable names are MED1-MED4 for the drug names and MCODE1-MCODE4 for the codes, this is what the labeled code for the MED1 field will look like.

Example 9: MED1 coding

```
MED1:
IF med1 NE ' ' THEN DO;
      medname = med1;       ①
      LINK codeit;
      mcod1 = codex;        ②
      IF codex NE . THEN
         CURSOR med2;
      ELSE
         CURSOR med1;
END:
ELSE
      CURSOR aspirin;       ③
RETURN;
```

Since this has to be done multiple times, we will create a labeled section of code that will be called each time. The label "CODEIT" does not correspond to any field on the screen, or else it would be executed when that field was modified.

If the drug name field is left blank, then it is assumed that no more medications are to be entered, so the application skips to question 15 (③). If something is entered in the field, that value is going to be passed to CODEIT as the variable MEDNAME (①). The linked section "CODEIT" will return the value CODEX, and MCODE1 will be assigned that value (②). If CODEX is missing, the cursor won't move, because there's a drug without a code assigned to it. Example 10 below illustrates the basics of using a selection list.

Example 10: Using A Selection List

```
CALL WREGION(16,1,21,74);    ①
codex = DATALISTN(med_dictionary,"medcode
        dname", "Medication Dictionary");      ②
RETURN;
```

It's that simple. The CALL WREGION (①) sizes and places the selection list window. It usually takes a couple of tries to get this right, because in most cases, you don't want the selection list taking up the whole screen and covering your FSEDIT session. I like to place my selection lists near the field that they are going to be used to fill. ② is the actual selection list. The data set being used, variables in that data set to be displayed, and a label for the window are all defined in the DATALISTN statement. Since there will only be one selection, and this is the default, there are no other parameters used.

This is an example of the importance of thorough system analysis. It is advantageous to have a grasp of your application's origin and purpose. For our application, a simple selection list is inadequate. Why? Because any reasonably comprehensive medication coding dictionary has thousands of entries. It can take a long time for someone to find the correct medication if they have to search the whole dictionary. Therefore, we are going to create a mini-application to help the interviewer by intelligently narrowing down the number of choices from which they have to select. Our selection list is restricted in several different ways through use of WHERE clauses that are defined by the input from the user.

We have created a variable SOUNDX in the medication dictionary data set by using the SOUNDEX() function on the drug name (without spaces) in the data set and taking the first six alphanumeric digits. This allows us a way of determining if the spelling (or phonetic transcription, since this may be read over the phone) is "close" to the correct name of the drug.

Example 11: Selection List Mini-Application

```
CODEIT:
rc = WHERE(med_dictionary,"dname EQ '" ||
      medname || "'");
IF ATTRN(med_dictionary,'any') GT 0 THEN DO;
       rc = FETCH(med_dictionary);
       codex = GETVARN(med_dictionary,1);          ①
       RETURN;
END;
rc = WHERE(med_dictionary);       ②
smname = SOUNDEX(medname);
rc = WHERE(med_dictionary,
     "soundx CONTAINS '" || smname || "'");     ③
CALL WREGION(16,1,21,74);
rc = VARSTAT(med_dictionary,'code','n',possible);     ④
IF possible EQ 0 THEN DO;
    ALARM;
    _MSG_ = "Code not found in dictionary for " ||
                medname;
    rc = WHERE(med_dictionary);      ②
    rc = WHERE(med_dictionary,"dname LIKE '" ||
         SUBSTR(medname,1,1) || "%'");      ⑤
    str = "Codes for Meds beginning with " ||    ⑥
          SUBSTR(medname,1,1);
END;
ELSE
    str = "Codes matching sound-alike Med Name";   ⑥
codex = DATALISTN(med_dictionary, "medcode
          dname", str);      ⑦
IF codex EQ . THEN DO;
  ALARM:
 _MSG_ = "No code selected from list!";
END;
rc = WHERE(med_dictionary);    ②
RETURN;
```

This mini-application is designed to handle user input in one of three ways. If the drug name entered is an exact match, the code for that exact match is automatically returned (①). We use the ATTRN() function to see if any records are returned after the WHERE clause has been applied. GETVARN , not CALL SET is responsible for the transfer of the medication code to the SDV because there's only one variable involved, and it doesn't have the same name as any one field.

If there is no exact match, the WHERE clause is cleared (②), and a match based on SOUNDEX is attempted by applying a new WHERE clause (③). The VARSTAT()

function (④) is used to determine if there are any matches based on the soundex algorithm.

If there are no matches to SOUNDEX, we clear the WHERE clause again, and apply a new one based on the first letter of the drug name (⑤). We define the label for the top of the selection list window in the variable STR (⑥). The DATALISTN statement (⑦) displays the window and gets the selection. Finally, before the routine returns to its calling point, any WHERE clauses are cleared so that it is ready for the next invocation. This now gives the user a fighting chance to find the correct code in a reasonable amount of time.

In the above example, only one selection is allowed from the list. However, you can set the number of selections obtained through a selection list from zero to infinite. No-response lists can be used to display information in a window, while you can use multiple-response lists for "Check all that apply" situations. If there is not enough room on the screen to display response categories, make a selection list pop up when the user enters an invalid response. There are many creative uses for selection lists in FSEDIT screens.

Function Keys and PMENUS

A robust FSEDIT application also has to allow the user easy access to necessary commands. You can define what commands are easy to get to by defining a function key catalog entry and/or a PMENU catalog entry, then associating them with your screen. Which method is employed depends on your users. If they prefer using point-and-click, then the PMENU entry should be carefully designed. If they like using the keyboard, then you should take care in defining the function keys and display them prominently on your screen pages. In our applications, the keys available for use are displayed across the bottom three lines of **every** screen page.

While displaying function keys isn't necessary for PMENU users, you should also define function keys for them, just in case a user decides to hit one. Never assume that the SAS system defaults are harmless to your application or your users' mental health. We use a standard basic key definition in all of our applications.

Example 12: Basic Function Key Definitions

```
F1   HELP
F2   RFIND
F3   LEFT
F4   RIGHT
F5   BACKWARD; =1
F6   FORWARD; =1
F7
F8   END
F9   ADD
F11
F12  PREVCMD
```

For applications that only need a single screen page, F3 and F4 are left blank so that there is no beep when these keys are pressed and there's no other page to go to. F5 and F6 make sure that the first screen page is always shown when the user steps through the data set record-by-record. Keys F7 and F11 are blank, so that they can be used for any custom commands created in the applications. A key is defined to ADD a record, but depending on the application, the command may be thrown away.

To create a key set, open the keys window and save it with a name in the same catalog that contains the FSEDIT screen. For example, given a screen that is named survey1.screens.demog.screen, save the key set with the following command:

save survey1.screen.demog.keys

This will create an entry in the catalog survey1.screen. If you have multiple screens in one catalog, then it is a good idea to name the key set entries the same as the screen entries; this way there is no confusion over which key set goes with which screen. Link the keys entry to the screen by changing the "Keys name" parameter in the "Modification of General Parameters" selection of the FSEDIT screen modification menu.

Unfortunately, PMENU creation is too complex to be covered in this paper. However, the principle is the same; to make sure the user can only access the "harmless" commands easily. Associating a PMENU with a screen is done in the SCL code with the PMENU() function. You would code it in the FSEINIT section because it defines an application parameter, and will probably be in effect during the entire FSEDIT screen session.

Deleting Records

Deleting records in FSEDIT is easy. Just use the "DELETE" command. For some applications, this is too easy. If you don't want the user to have access to the DELETE command, you can make it impossible to delete a record in the "Modification of General Parameters" menu item. What if you want the user to have the ability to delete a record, but want to make them verify it? The example below pops up a selection list for the user to verify the delete command.

Example 13: Verifying A Deletion

```
FSEINIT:
CONTROL ALLCMDS;
RETURN;

INIT:
flg = 0;                ①
RETURN;

MAIN:
cmd = UPCASE(LASTCMD());
IF (NOT flg AND (cmd EQ 'DELETE'))
   AND id NE ' ' THEN DO;          ②
   verify = "NO";
   verify = SHOWLIST("NO","YES",          ③
         "Delete this record?");
   IF verify EQ "YES" THEN DO;
      flg = 1;                          ④
      CALL EXECCMD('delete');
   END;
   ELSE DO;
      CALL NEXTWORD();      ⑤
   END;
END;
RETURN;
```

Only the relevant code is given here. ① clears the deletion flag when a record comes on the screen. ② checks to see if the delete command has been issued and that it has not been verified. It also automatically allows deletion of records without ID numbers. ③ displays the selection list window. If you use this, you might want to

define the position of the window first. If the user selects "YES" (④), the flag is set, and the delete command is reissued. Otherwise (⑤), the delete command is flushed from the buffer using CALL NEXTWORD().

What if you want to put the deleted record in a transaction data set? First, outside of FSEDIT, you need to create a transaction data set with the same variables as your primary data set. Use the code below to "move" the observation.

Example 14: Copying the Current Record to Another Data Set

```
MAIN:
cmd = UPCASE(LASTCMD());
IF (cmd EQ 'DELETE') AND id NE ' ' THEN DO;
   copy = OPEN("trans.survey","u");
   CALL SET(copy);                          ①
   rc =APPEND(copy);
   rc = CLOSE(copy);
END;
RETURN;
```

The code in ① copies the record to the data set *TRANS.SURVEY*. *TRANS.SURVEY* must have the same variable names as the primary data set because CALL SET is used to align the variables in the SDV with those in the record to be written. No CALL NEXTWORD() function is used to remove the DELETE command from the command buffer, so the deletion occurs when the RETURN statement is executed. You can use this method and this code to copy a record to another data set for any reason.

Versions 7 and higher of the SAS System have built-in audit trail capacity for data sets. It will automatically keep track of deleted, added, and modified records. If space considerations do not rule out keeping audit data sets, it is a good idea to take advantage of this feature, especially for interactive data entry and editing applications.

Summary

These are the processes necessary to create a robust FSEDIT application. Emphasis should be placed on the system analysis phase, because this is where many steps can be taken to improve the quality of your data and your application before a single line of code is written. Establishing standards across applications is very useful as well. If you create a standard interface for your FSEDIT applications, training and coding costs will be decreased.

The 14 examples given indicate some of the possibilities (and necessities) of using SCL in FSEDIT applications. There are, of course, many more. They will spring from your understanding of the task, knowledge of your users, the uses of the data being collected, and your imagination.

References

1. Morgan, D. and Province, M. (1997), "A Bag of FSEDIT Tricks," *MWSUG '97 Conference Proceedings*, 359-363

Acknowledgements

This work was partially supported by NHLBI grants HL 54473 and HL 47317.

Further inquiries are welcome to:

Derek Morgan
Division of Biostatistics
Washington University Medical School
Box 8067, 660 S. Euclid Ave.
St. Louis, MO 63110
Phone: (314) 362-3685 FAX: (314) 362-2693
E-mail: derek@wubios.wustl.edu

The sample application and this paper are available via the World Wide Web at:

http://www.biostat.wustl.edu/~derek/sasindex.html