A Bag of FSEDIT Tricks

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

Abstract

For data entry applications, the FSEDIT procedure provides a good user interface. With reasonable care and knowledge, an FSEDIT screen will yield well-organized, reasonably clean data for analysis. However, what happens when that reasonable care and knowledge are not a given, especially when that application is to be remotely supported, and/or extensive training is not an option? How can you help improve the data quality while retaining that user-friendly, non-technical interface? Screen Control Language (SCL) makes it possible, but there are several non-obvious techniques that also help accomplish that end. In addition, there are other, non-SCL methods that will aid the FSEDIT screen programmer in eliminating data problems at entry.

It is always easier to build an application for yourself than to build one for other people to use, because you know how everything is supposed to work. This paper details five solutions to problems that have been presented to us by our users. They are: how to annotate a specific item in a data set; improving data quality at entry; maintaining an intuitive interface when there is a category called "other (specify)"; how to protect existing data from inattentive entry people; and finally, how to handle phone numbers for painless entry and storage.

These are some of the tricks we have developed and use in the distributed entry applications we have created. The methodologies shown can be extrapolated to many other situations.

Problem #1: Annotating a data set

When dealing with data in human subjects research, there are times when it is helpful to know what the person administering the test saw, particularly if something out of the ordinary occurred. After three years of data collection, these things can understandably become vague and confused. Therefore, it would be useful to be able to annotate any given data item. This isn't a problem in and of itself: with the SAS System®, it is easy to make a dataset specifically for textual (or coded) notes. The problem arises with the question of how to build this into an application without forcing the user to explicitly open a separate dataset, or worse, making the user enter a form, then enter comments as a separate form. If the data entry system is operating in real-time (e.g., data is entered with the subject present), both of these options are not only clumsy, but inconvenient and impractical.

One possible solution is to store comments in the form dataset itself. This has some disadvantages. During analysis, the comments are not necessarily attached to any particular field, and some preliminary work to coordinate the comment with its relevant data must be done prior to analysis. Text comments inflate the size of a dataset. Since most observations won't have full comments, much of the space allocated for comments will be unused, and therefore, wasted. In addition, developing a database in this fashion requires a pre-data collection estimate of a reasonable amount of space for comments to reduce the amount of wasted resources. This means that ultimately, there will be some finite limit to the number/amount of comments. Finally, the problem of not entering the comment while the tagged data item is being entered still remains.

Our solution is to use a separate notes dataset that can be easily accessed from any field on any form, with the relevant identifying information pre-loaded. Multiple notes may be entered for any given field, so there is no theoretical limit on the length of a comment.

The Hidden Note

This trick takes advantage of the ability to create custom commands with SCL. Using macro variables and a developer-defined function key, the notes dataset is only a key press away. It contains the ID number of the subject, the name of the field being annotated, the date that the note was entered, and, for multiple-form applications, the name of the form where the field is located. These fields are pre-filled and protected when the notes screen appears, leaving only the comment and a user identification to be filled in by the user.

You can create different sets of function keys for different FSEDIT screens. To do this, open the keys window and save it with a name other than "FSEDIT" in the same SAS catalog that contains the FSEDIT screen. For example: given a screen named "survey1.screens.demog.screen", save the keys with the following command:

SAVE *survey1.screens.demog.keys*

This will create a keys entry in the SAS catalog survey1.screens. If you have more than one screen in a catalog, then it is a good idea to name each key entry the same as its corresponding screen entry. This way, there is no confusion over which set of key definitions go with which screens. Link the keys entry to the screen by changing the "Keys name" parameter in the "Modification of General Parameters" selection in the FSEDIT screen modification menu. In the above example, the keys name would be *DEMOG*.

To define function keys, assign commands to the list of function keys displayed in the keys window. It is not necessary to fill in something for each key shown, and you may change the SAS defaults if you wish. It is, however, a very good idea to keep the same common keys for all FSEDIT screens in an application. The only possible problem with creating custom commands is that the potential for disabling a SAS command exists. If your custom command has the same name as an existing SAS command, your command will execute instead of the SAS command. While this could work to your advantage in the case of "delete", it would be disastrous if you created a command called "end" that did not terminate the FSEDIT session.

After you have your command's name ("NOTEX" in the example) put the code associated with the command in the MAIN section of the SCL code. It is also imperative to add the ALLCMDS option to the CONTROL statement in order for custom commands to work.

Annotation Function Key Code:

```
cmd = UPCASE(LASTCMD());
var = CURFLD();
IF cmd EQ "NOTEX" THEN DO;
  CALL SYMPUT("id",id);
  CALL SYMPUT("passvar",var);      ①
  CALL SYMPUT("passform",form),
  CALL SYMPUTN("notedate",date);
  nota = 'notedsn (WHERE=(id EQ "' || id |'      ②
        '" and form EQ "' || trim(form) ||
        '" and item EQ "&PASSVAR"))';
  CALL FSEDIT(nota,scrname,'ADD');          ③
  CALL NEXTWORD();
END;
```

In the above code, *notedsn* is the name of the notes dataset (e.g., "survey1.notes"), and *scrname* represents the name of the screen for the notes dataset (e.g., "survey1.screens.notes.screen").

① indicates the macro variables where the field name, form name, and date are stored for passing to the notes dataset. The field name is obtained by using the CURFLD() function, and the form name is hard-coded in the INIT segment of the screen's SCL code. ② is where the name of the notes dataset is assembled. Using the WHERE dataset option insures that the notes dataset is opened only for that subject, for that field, concerning that particular form. In short, the note will be restricted to the specific field where the key press occurred. ③ is the invocation of the FSEDIT screen for the notes dataset. Note that it is opened with the "ADD" attribute; this prevents an inattentive user from overwriting any existing notes.

The second part of this trick has to do with how the notes screen processes the information that is passed from the main form. This is what a typical notes dataset screen looks like:

ANNOTATE DATA ITEM SCREEN

Subject ID: _____ **FORM NAME:** _____

DATA ITEM: _____ **Date of comment:** _____

COMMENT: _____

SCL For The Notes Dataset FSEDIT Screen

```
FSEINIT:
CONTROL label enter allcmds;
LENGTH cmd $ 80;
/* Make sure that user must press 'Enter' key to
move on */
CALL SETCR('stay','return','modify');
CALL EXECCMD('zoom on');   /* Fill whole screen
*/
RETURN;

INIT:
PROTECT id form item date;
IF id EQ ' ' THEN
  id = SYMGET('id');
IF form EQ ' ' THEN
  form = SYMGET('passform');  ①
IF item EQ ' ' THEN
  item = SYMGET('passvar') ;
IF date EQ . THEN
  date = SYMGETN('notedate');
CURSOR comment;
RETURN;

MAIN:
cmd = UPCASE(LASTCMD());
IF cmd EQ 'CANCEL' THEN DO;
  _STATUS_ = 'H';
  RETURN;
END;
RETURN;

DATE:
CURSOR user;
RETURN;

USER:
CURSOR comment;
RETURN;

COMMENT:
_MSG_ = 'Press F8 to END and SAVE ' ||
        'and return to the form';
ALARM;
CURSOR comment;
RETURN;

TERM:
RETURN;
```

In this code, ① indicates how the macro values that were stored in the function key code are received by the notes screen. They are all conditionally executed, so that any

existing values are not overwritten. This also allows the values to be pre-filled again if the user needs to add more comments for that field. The next point of interest is another little trick: at ②, if the user issues a "CANCEL" command, we terminate the screen immediately by using "_STATUS_ = 'H'". This allows the user to stop the current note without adding an observation to the dataset.

The remaining code is there for one reason: users can always find a way to get somewhere they don't belong. Defining the actions to occur on each field regardless of whether they should be there, insures that users don't end up stuck and frustrated. A brief message and beep after they have entered the comment cues them to the default action that should be taken.

Problem #2: Checking for Male/Female or Yes/No

Inexperienced data entry people tend to prefer mnemonic categories for items such as sex, or yes/no answers. Sometimes it is also important to store these as character variables, since as a single character it occupies 67% less space than the smallest numeric variable. If disk space is at a premium and you're dealing with large amounts of data, it becomes imperative to take advantage of all such compression.

The problem is, how do you check that the user has entered a valid value for these types of questions? For that, you must go...

Beyond Maximum and Minimum...

PROC FSEDIT provides range checking for the fields in a screen. You may define the maximum value and the minimum value for any field on any screen. Unfortunately, sometimes your range is not contiguous; there are both valid and invalid values between the two endpoints. While you could use an IF statement in labeled SCL code to check the validity of any given field, this is a large waste of programming time, because there is an easier way: custom INFORMATS.

With informats, you can define all possible responses for any question. It has the additional benefit of being transparent. It does not have to be invoked with the FSEDIT command. Once the informat is built using PROC FORMAT, associate it with the variable (or variables) when you define the dataset attributes. Now you don't have to worry about it while designing the custom FSEDIT screen. Instead of having to code valid responses for each screen field with SCL, one informat can handle an infinite number of fields with the same response categories. It is also not restricted to a specific screen. As long as the informat exists in a permanent format library ("LIBRARY=LIBRARY" in the PROC FORMAT statement), and you have defined the libname LIBRARY somewhere in your application before using the informat, it automatically works. Here's an example of how to do it for male/female forced choice coding:

```
PROC FORMAT LIBRARY=LIBRARY;
INVALUE $GENDER (UPCASE)
  'M','MALE'='M'
  'F','FEMALE'='F'
  OTHER=_ERROR_
;
```

Anything other than "M", "F", "MALE", or "FEMALE" will be rejected during entry. The (UPCASE) option renders the comparison case-insensitive without forcing the field on the screen to be upper case.

What if your yes/no variables are numeric in the dataset, but your entry people are used to using Y/N? Here's another example:

```
INVALUE YN (UPCASE)
  0,'N','NO','0'=0
  1,'Y','YES','1'=1
  ' '=.
  OTHER=_ERROR_
;
```

This permits your users to enter it however they are most comfortable, but the value that winds up in the dataset will still be 0, 1, or missing. **Note:** If the blank (' ') is omitted from the informat, it will become a forced choice question, since blank would then fall under the "OTHER" category, and be interpreted as an error.

The final example, below, shows how to force the entry person to enter something while giving the subject the option of replying. This has the additional advantage of clearly indicating that the question wasn't inadvertently skipped:

```
INVALUE YN (UPCASE)
  0,'N','NO','0'=0
  1,'Y','YES','1'=1
  'R'=.R     /* Special missing value .r */
  OTHER=_ERROR_
;
```

The letter "R" (for "Refused") will be translated into the special missing value .R if the subject doesn't want to answer the question. Leaving the field completely blank is not an option (note that blank is not specified as a valid variable), but the value stored in the dataset is analyzed as a missing value.

Problem #3: "Other (specify)"

How do you make sure that the question is asked when someone selects this category? How do you maintain an answer field so that it makes sense to people just looking at the screen? Also, when it comes time to analyze this, do you consider looking at "other" as its own variable? What do you do to keep everybody happy?

The Incredible Reappearing Question!

Here is a categorical question and the possible response categories:

Welcome to Quark's

1. Please indicate your species:
1: Human
2: Bajorn
3: Cardassian
4: Klingon
5: Ferengi
6: Tellaxian
7: Other (specify) _____

For this trick, you will need to create and define a screen variable. Screen variables belong to FSEDIT screens, but do not go into the dataset. This is done from the "Screen Modification and Field Identification" selection of the FSEDIT Modify menu. Before defining the fields on the customized screen, PROC FSEDIT asks if you created any computational or repeated fields. Answer 'Y', and a screen will appear with spaces for variable names, their type (character, numeric, or repeated), and any informat or format to be applied.

In this example, we add a variable named SPECIFY to the list. Its only function is to display the question, so it is not needed in the dataset. To render the screen field "invisible," set the pad character to a blank (by hitting spacebar) in the "Assign Special Attributes to Fields" selection of the FSEDIT Modify menu. The created variable is also set to blank in the INIT segment of the SCL program. Here is the code that performs the reappearing act:

```
SPECIES:
IF species EQ 7 THEN DO;
  specify = "Please specify species";
  CURSOR species2;          ①
  UNPROTECT species2;
END;
ELSE DO;
  PROTECT species2;
  specify = ' ';
  species2 = PUT(species,rce.);  ②
  CURSOR educ;
END;
RETURN;

SPECIES2:
CURSOR educ;
RETURN;
```

When the "Other" category is chosen (①), SPECIFY is given a value, and seemingly by magic, the question appears on the screen. The field for the response (SPECIES2, which is in the dataset) is unprotected and may now be entered. If one of the pre-defined categories is selected (②), SPECIFY is reset to blank while SPECIES2 is filled in with the formatted value of the response category. SPECIES2 is then protected so that the user cannot change that value. Now there are two variables that represent the species question in your dataset: a categorical numeric one and a standardized text version of that variable.

Problem 4: Well I Didn't Know, I Just Started Typing!

One recurring nightmare that data managers have is doing an audit and finding out that the first N observations have changed. Worse, that data is old, so it should not have changed. A call to the field gives you the answer. The new entry person has been entering a pile of forms using the forward key. Upon further questioning, you discover that no one told them about needing to add a blank observation before entering a new form.

There are two possible ways to handle this. The first solution is the easy one. Just start the PROC FSEDIT with the "ADD" option. Or, if this is part of a larger application, use "CALL FSEDIT(*dataset name,screen name*,'ADD');. That will automatically start the screen with a blank observation ready for entry. A slightly more sophisticated method is:

The Smart Lock

Why is there a "smart lock?" The real-life example is a form whose data comes in bits and pieces, but the form needs to be entered as soon as <u>any</u> data is available. Existing observations are constantly in flux, and the possibility always exists for writing over valid data. If the default mode of operation were to add a new observation, the entry people would have to: cancel the new blank observation, rewind the dataset, and search for the observation they need to update each time they open the screen. To insure a reasonable amount of care in the entry and modification of this data, we lock each existing observation on the fly.

```
INIT:
IF id ne ' ' THEN DO;
  lokit = "Record is LOCKED. Use F7 to
UNLOCK.";          ①
  CURSOR alert ;
  PROTECT _all_;
END;
ELSE DO;
  UNPROTECT _all_;  ②
  lokit = ' ';
  PROTECT lokit;
  CURSOR id;
END;
```

The ID field determines whether or not the lock is "armed" for any given observation. If the ID number already exists (①), then the record exists, and is automatically "locked" by using the "PROTECT _ALL_" statement. Also, a screen field called LOKIT is now set to display a message to that effect. If it is a new observation (②), then the fields are unprotected, and the message field is blanked.

However, every lock needs a key, so a custom function key is defined as "UNLOCK". In the MAIN section of the SCL program:

```
IF cmd EQ 'UNLOCK' THEN DO;
   lokit = ' ';
   alter = TODAY();
   alterby = SYMGET('userid');
   UNPROTECT _all_;
   PROTECT id lokit;
END;
```

In the above code, the variables on the form are unprotected for editing, except for the ID, and the screen variable used to display the lock message. Audit control is provided by the non-screen dataset variables ALTER and ALTERBY, which track who made the last change to the observation and when. While this does add a step to the editing process, it is not overly cumbersome and will reduce the amount of accidentally overwritten data.

Problem 5: Those Messy Phone Numbers!

Phone numbers are some of the most difficult data to enter. There are so many common ways of representing a phone number that it is easy to wind up with a dataset containing all of them. Given the different delimiters that can be used, the easiest method for painless entry is to define phone numbers as character variables.

Unfortunately, this makes it almost impossible to check them for validity or to present them in a uniform fashion for reports. While training and standardizing on a specific input method (e.g., enter as 314-555-1212 ONLY) can help, phone numbers are frequently written according to how the person is used to writing them, and deviations from the standard format will occur.

The obvious solution to this problem is to define phone numbers as numeric variables. However, trying to read a phone number without delimiters (e.g., 3145551212) is confusing. If it's hard to read, then it's hard to enter. Long strings of digits are prone to winding up with extra or missing digits. So what do you do?

Try this SAS user-defined format. It will delimit those long strings of numbers:

```
PROC FORMAT LIBRARY=LIBRARY;
PICTURE PHONE
. = ' '
2010000-9999999 ='000-0000'
2012010000 -9999999999 =
'000) 000-0000' (PREFIX='(')
20120100000 -99999999999 =
'000) 000-0000 Ext 0' (PREFIX='(')
202010000000  -999999999999 =
'000) 000-0000 Ext 00' (PREFIX='(')
2012010000000 -9999999999999 =
'000) 000-0000 Ext 000'   (PREFIX='(')
20120100000000-99999999999999 =
'000) 000-0000 Ext 0000'  (PREFIX='(')
201201000000000-high=
'000) 000-0000 Ext 00000' (PREFIX='(')
OTHER='ERROR'
;
```

By using this picture-style format, numeric values are automatically converted to a standardized representation of phone number. This has the added benefit of formatting any number entered with fewer than 7 digits, or a number beginning with "1" as "ERROR". Therefore, it will be easier to find in any report.

So now you have a format for your FSEDIT screen and your reports. Any number entered should be neatly represented. However, that is only half of the problem with phone numbers. There is still the problem of data entry. What happens when the number is entered as something other than all numbers? Doesn't that cause an error? Don't the users get frustrated? And once it gets formatted, what happens if they need to change it? Doesn't it still have these formatting characters that will cause an error?

The answer to all the above questions is YES. To deal with this, instead of solving these questions individually, we eliminate them by a nice bit of sleight-of-SCL called:

The Phone Trick

The ability to create FSEDIT screen fields that are not in the dataset comes in handy here. The phone number is kept in the dataset, but is not displayed on the screen. What the FSEDIT users see and interact with is a something that is only defined as a part of the FSEDIT screen. It is defined as a character variable that represents the nicely formatted value of the phone number. The formatting of the phone number is done "on the fly" using the PUT() function. This is half of the trick.

Still, what about all those different ways that people write phone numbers? How do you turn that messy collection of numbers and delimiters into a number ready for formatting? First, remember that the phone field on the FSEDIT screen has been defined as a character variable. Therefore, phone numbers may be entered in any way, and the screen will take them without complaining. However, this doesn't achieve the desired formatting, nor does it store them in your dataset as numbers. The SAS function COMPRESS() removes the formatting characters, and the function INPUT() turns that string of numbers into a numeric value.

On the other hand, what if the phone number is entered as a string of digits without delimiters? Again, since it's a character field, there is no problem with the input itself. However, you will have to explicitly apply the phone format to the field to get your standardized display. In the SCL below, PHONEC is the name of the character variable defined in the screen, while PHONE is the numeric variable in the dataset.

Make sure that you use the LABEL option on the CONTROL statement in your SCL, or the phone trick will not work, since it is dependent upon execution of labeled code.

Phone Trick SCL:

```
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 temporary to numeric, store in
dataset variable */
  phone = INPUT(tmp2,20.) ;
END;
ELSE
/* If no delimiters, convert entered field to
numeric, store in dataset variable */
  phone = INPUT(phonec,20.);
/* Put formatted dataset variable into screen
character variable */
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 dataset while the value on the screen will have a nice, easy-to-read format.

If you were to stop here though, PHONEC would be blank for existing observations where the phone number has already been entered. Since PHONEC is only a part of the of the FSEDIT screen but is not in the dataset, it will be blank when an observation is loaded. To avoid this perplexing problem, include the following in the INIT section of the SCL code:

```
phonec = LEFT(PUT(phone,phone.));
```

This defines the character variable on the screen each time an observation is loaded, so that the field is not blank unless PHONE is missing.

Summary

These are only some of the tools in our PROC FSEDIT programming kit. We also use other screen design techniques which aid in yielding better FSEDIT applications. For example, standardized page design and consistent use of colors and function keys across screens helps to create more ergonomic, and therefore, better data entry applications. The five tricks described above improve the quality and functionality of our data.

The general techniques demonstrated here may be extrapolated to solve other problems in FSEDIT or FRAME applications. As an example, customized key commands can be employed to print form letters or to display a reference dataset.

In developing an FSEDIT application for others to use, always consider the following: if by spending a little more time in the development phase, you can reduce the opportunity for error at entry time or reduce the users' level of frustration, you will get a better system. A better system means that your users will spend more effort entering data and less in fighting the program.

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