When the argument to INPUTN contains a value that cannot be converted to a numeric value, the INPUTN function returns a missing value. If
you tried to convert something like 'OOPS' to numeric, but failed, the
key, then, is to check for a missing value (not _blank_) after applying
the INPUTN function.

NOT AS EASY AS IT USED TO BE (EXAMPLES):
Suppose you are coding an application with three panels. You are on
panel 2. You just realized you made a mistake on panel 2. Remember when
you typed in Panel 2, the BACKUP panel was displayed; you could just fix your mistake and carry on. Not so in
Version 6.06. If you want to back up to the previous screen to correct
something, this feature must be explicitly designed into your system.

It turns out that this is not too difficult to implement. Once again,
the key is handling the right SCL functions to use and how to combine
them. The notes on Panel 1 explain how it works.

ENTRY: EXAMPLE3.PROGRAM ==> BACKUP (Panel 1)

**** DISPLAY ****
PF1=HELP PF3=CONTINUE PF12=QUIT

I Enter a word: &WORD.

**** ATTRIBUTES ****
Window name: Branching/BACKUP Example
Start row: 1 Row: 24
Number of rows: 1 Col: Length 6
Field names: WORD Frame: 1 Choice group: 1
Type: CHRN
Action: NO
Prompt char: ?
Erroroff value:
Erroron value:
Replace:
Replace:
Options: OPS CURSOR REQUIRED AUTOSKP

**** SOURCE ****

INT:

; Purpose:
; Allow user to branch from PANEL1 to PANEL2 to PANEL3.
; Also, if user is on PANELS, for example, and wants to
; backup and change something on PANELS, allow user to do so.

; The trick is to make the BACKUP work area:
; 1. Define your PFZ key to be equal to BACKUP (___KEYS).
; 2. On those screens you want to be able to backup from,
; use the CONTROL ALWAYS command.
; 3. Use the function WORD to query if the user pressed
; the key assigned the function of BACKUP so that you
can do special processing in this case.

; 4. Use symget and symput to make sure all screen values
; are available when needed.

; Because you do not want to BACKUP to the preceding screen
; from here, none of this stuff is implemented on PANEL1.

wordl = symget('wordl');
RETURN;
MAIN:
RETURN;
TERM:
call symput('wordl',wordl);
setstatus();
when('O') return;
when('T') call display('panel2');
return;
RETURN;

1008
ENTRY: EXAMPLES.PROGRAM ==> BACKUP (Panel 2)

ENTRY: EXAMPLES.PROGRAM ==> BACKUP (Panel 3)

ENTRY: EXAMPLES.PROGRAM ==> X-tables (Quota example)
Notice the labelled sections GETROW and PUTROW. GETROW is used to do something with those rows or the table the user initializes the extended table before it is displayed to the user. PUTROW is used to do something with the extended table portion that is displayed on the screen. It is used to display the contents of the extended table to the user. Once the user has completed the form, the PUTROW section will not accept an argument. It must be either a source or a display.

**SOMETIMES IT WORKS FINE, OTHER TIMES IT DOESN'T WORK AT ALL (EXAMPLE 4)**

Experience tells us to expect our solutions to be a lot more complex than this. This first example illustrates a different use of the extended table. It also gets into that last category, the system options. This must be set to extended table. The GETROW section fetches the observations from this dataset and puts them on the screen with their values. The panel is then displayed. Notice the use of arrays. In the GETROW section, the arrays are initialized to blanks. This ensures that the user will see only the values that the user has entered. The PUTROW section displays the form. Once the user has completed the form, the PUTROW section is executed. The user fills in any rows those data values are saved in the corresponding array positions.

Although there are no other key components, they are equally essential to the extended table. The input panel has 2 categories, system options. This must be set to extended table. This is the only way to access those options that are needed. The GETROW section displays the form. It is used to do something with the rows or the table the user initializes. Notice the use of arrays. In the GETROW section, the arrays are initialized to blanks. This ensures that the user will see only the values that the user has entered. The PUTROW section displays the form. Once the user has completed the form, the PUTROW section is executed. The user fills in any rows those data values are saved in the corresponding array positions.

The extended table example below is used to display observations of the dataset and has two columns, the name and the observation. There is little similarity between how this is done using a DATA step and how you do it using SQL. This will be apparent when you examine the INIT section. The call to GETROW is also done in INIT.

The GETROW section fetches the observations from this dataset and displays them with their values. The panel is then displayed. The user then indicates those observations to delete by placing an X in the appropriate field. When the user fills in the PUTROW section executes once for each modified row. What PUTROW actually does is up to the programmer. In the following example, it puts a flag on the dataset indicating that the observation is deleted.

The last thing this example does is actually delete the flagged observations and drop the flag variable from the dataset. Take a look at the example.

**ENTRY:** EXAMPLE4:PROGRW = = X-tables and Friends

**DISPLAY**

- **Command:**
  - PF1=HELP
  - PF3=CONTINUE
  - PF7=UP
  - PF8=DOWN
  - PF2=QUIT
  - X-off those variables you want to eliminate from the dataset
  - VARNAME &
The last point I want to make is about dropping variables from a dataset. I have been unable to find a way to do this using SCL. If you are really adverse to dropping into a SUBMIT block to do this, you can take the approach shown in the closing lines of the **TERM** section. Define a new dataset with all the variables you want to keep and add all the observations from the source dataset to it. Not slick, admittedly, but it does work.

TO CONCLUDE:

In conclusion, for the most part I am impressed by the power of Version 6.05 AF and SCL. It is somewhat difficult to learn because in so many ways it is non-intuitive. The best way to learn it is to work with it, make mistakes; read all the fine print in any reference manual you can get your hands on. Once you get your balance, however, you'll find that the few shortcomings of ths product are far out-advised by its many strong points.

CONTACT INFORMATION
Karen Crandall
Eastman Kodak Company
Kodak Park
Rochester, New York 14650-2204
Telephone: (716) 722-6512

The last point I want to make is about dropping variables from a dataset. I have been unable to find a way to do this using SCL. If you are really adverse to dropping into a SUBMIT block to do this, you can take the approach shown in the closing lines of the **TERM** section. Define a new dataset with all the variables you want to keep and add all the observations from the source dataset to it. Not slick, admittedly, but it does work.

TO CONCLUDE:

In conclusion, for the most part I am impressed by the power of Version 6.05 AF and SCL. It is somewhat difficult to learn because in so many ways it is non-intuitive. The best way to learn it is to work with it, make mistakes; read all the fine print in any reference manual you can get your hands on. Once you get your balance, however, you'll find that the few shortcomings of ths product are far out-advised by its many strong points.

CONTACT INFORMATION
Karen Crandall
Eastman Kodak Company
Kodak Park
Rochester, New York 14650-2204
Telephone: (716) 722-6512