Introduction

For many of us, our early experiences with extended tables can be quite intimidating and frustrating. Since extended tables are not like any other SAS construct, we cannot draw on our previous SAS® experience to help us figure them out. Existing documentation does a good job at describing the components of extended tables; however, it does not provide many examples of how to put everything together. It's not too surprising, then, that we are left with the impression that extended tables are difficult to use.

The purpose of this paper is to carefully examine the issue around timing of events when working with extended tables. The sequence of when the various program sections get executed is not intuitive. All too often, instead of focusing your energy on solving the problem at hand, you get bogged down trying to figure out why part of your code is (seemingly) being ignored.

The intended audience for this paper is people who have a working knowledge of SAS/AF®, Screen Control Language (SCL), and the basic components of extended tables. To get the most out of this paper you should already know how to use extended table constructs such as GATTR, ^^, SETROW, GETROW, and PUTROW.

Five examples are used to provide some insight into how to approach the design and programming of extended tables. The first two illustrate alternative approaches for designing an extended table. Each has its strengths and weaknesses. The remaining three examples illustrate how to implement some common requirements that you would think would be easy to implement—until you actually try to do it. The reason they are not easy to implement is because of the timing issue. Timing will be discussed in the context of these examples.

The Basics

The "array approach" is a good design choice for applications where the size of the table can be fixed—seven days in a week or 12 months in a year. The strength of the "array approach" is that it is easy to understand and code.

***** DISPLAY *****

<table>
<thead>
<tr>
<th>PURPOSE: Array approach to extended tables</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;msg ____________________________________</td>
</tr>
<tr>
<td>ITEMS:</td>
</tr>
<tr>
<td>^^</td>
</tr>
<tr>
<td>Item &amp;N_ ====&gt; &amp;ITEM_</td>
</tr>
</tbody>
</table>

***** ATTRIBUTES *****

<table>
<thead>
<tr>
<th>Window name: Array Approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start row: col:</td>
</tr>
<tr>
<td>Number of rows: cols:</td>
</tr>
<tr>
<td>General help: DISPLAY</td>
</tr>
<tr>
<td>Function keys: DISPLAY</td>
</tr>
<tr>
<td>Lookup data set:</td>
</tr>
<tr>
<td>Command menu:</td>
</tr>
<tr>
<td>Prompt char: ?</td>
</tr>
<tr>
<td>System Options: EXTENDED TABLE</td>
</tr>
<tr>
<td>Parent:</td>
</tr>
<tr>
<td>Window Type: STANDARD</td>
</tr>
<tr>
<td>Scroll Bars:</td>
</tr>
</tbody>
</table>

Field name: N Frame: 2 Row: 1 Col: 15
Length: 3
Alias: ITEMNBR Choice group: Pad: _
Type: FIXED Protect: YES
Format: Just: RIGHT
Informat: 
Error color: RED attr: REVERSE Help: 
List: < 1 25
Initial: 
Replace: Options CAPS AUTOSKIP
The weakness of the "array approach" is that the array dimension must be hardcoded into the program. If your application can work within this constraint, the "array approach" is a good choice.

Most applications, however, are driven by the user's input data. Generally, the size of the array you need is based on the complexity of the user's inputs. Since the array dimension has to be hardcoded, the programmer is forced to either allocate a much bigger array than the application needs (to be on the safe side) or risk abending when the array is not big enough to handle the request. In situations like this, the "dataset approach" works well.

The "dataset approach" is more flexible. Construct a dataset which is essentially a mirror image of what you want the extended table to look like. The number of observations in this dataset will determine how many rows your extended table has. It doesn't matter if you have 5 or 50 observations. Setting up a dataset is certainly more complex than setting up an array; but, since SCL provides so many more functions for managing datasets, the task of managing the table is easier using the "dataset approach".

```plaintext
***** SOURCE *****
array items[25] $ 5 _temporary_;
INIT:
    control always;
    * Set-up:
        * 1. Define an array of 25 positions
        * 2. Define a corresponding x-table
    call setrow(25);
    RETURN;

GETROW:
    * Display blank lines on the screen
    * for user to fill in
    itemnbr = _currow_
    item = items(_currow_)
    RETURN

PUTROW:
    * Save the users responses in the array
    items(_currow_)=item
    RETURN

MAIN:
    RETURN

TERM:
    * Wrap-up
    select(_status_)
    when('C') put 'Request Cancelled'
    when('E') put 'Programming stmts here'
    otherwise put _status_ =
    end
    RETURN
```

```plaintext
***** DISPLAY *****
```

Purpose: Dataset approach to extended tables

```plaintext
&msg ________________________ 

ITEMS:

^
Item &N_ ===> &ITEM_____
```
***** ATTRIBUTES *****

Window name: DATASET Approach
Start row: col:
Number of rows: cols: Banner: COMMAND
Function keys: DISPLAY
Lookup data set:
Command menu:
Prompt char: ?
System Options: EXTENDED TABLE
Parent:
Window Type: STANDARD
Scroll Bars:

Field name: N Frame: 2 Row: 1 Col: 15
  Length: 3
  Alias: ITEMNBR Choice group: Pad:
  Type: FIXED Protect: YES
  Format: Just: RIGHT
  Informat:
  Error color:
    List: RED attr: REVERSE Help:
    Initial: < 1 25
  Replace:
  Options: CAPS AUTOSKIP

Field name: ITEM Frame: 2 Row: 1 Col: 24
  Length: 14
  Alias: ITEM Choice group: Pad:
  Type: CHAR Protect: NO
  Format: Just: LEFT
  Informat:
  Error color:
    List: RED attr: REVERSE Help:
    Initial:
  Replace:
  Options: CURSOR AUTOSKIP

***** SOURCE *****

INIT:
  CONTROL ALWAYS

  length rc 2

  * 1. Define a dataset to correspond with the screen;
  * 2. Add some empty observations
  * 3. Define a corresponding x-table

items = open('items','n')
rc = newvar(items,'item','c',14,'
rc = close(items)

* Add some observations
item = _blank_
items = open('items','u')
nobs = 25
call set(items)
do i = 1 to nobs
  rc = append(items)
end

* Prepare to display the x-table
call setrow(nobs)
RETURN

GETROW:
* Display blank lines on screen for user to fill in
itemnbr = _currow_
rc = fetchobs(items,_currow_)
RETURN

PUTROW:
* Save users responses on the dataset
rc = fetchobs(items,_currow_,'nosey')
rc = update(items)
RETURN

MAIN:
RETURN

TERM:
* Wrap-up
rc = close(items)

select(_status_)
  when('C') put 'Request Canceled'
  when('E') put 'Programming starts here'
  otherwise put _status_ =
end;
RETURN;
For Your Toolbox

Suppose you want to present a list of choices and request the user to select at least one. It seems like this should be easy. There is a built-in routine to limit the number of choices the user may select, however, there is not one to force the user to select at least one. It turns out that this is not a particularly trivial problem. The key to solving it: use a choice group, the 'num-sel' parameter on the SETROW routine, and a tracking variable (keepit) on the dataset to keep track of which choices the user selected.

***** DISPLAY *****

**PURPOSE:** Fill in at least one blank

\&msg

**CATEGORIES:**

\&ITEM

***** ATTRIBUTES *****

<table>
<thead>
<tr>
<th>Field name:</th>
<th>MSG</th>
<th>Frame: 1</th>
<th>Row: 5</th>
<th>Col: 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Length:</td>
<td>72</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Alias:</td>
<td>MSG</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Choice group:</td>
<td>Pad:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Type:</td>
<td>CHAR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Protect:</td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Format:</td>
<td>Just: LEFT</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Informat:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Error color:</td>
<td>RED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>attr:</td>
<td>REVERSE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Help:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>List:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Initial:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Replace:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Options:</td>
<td>AUTOSKIP</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Field name:</th>
<th>ITEM</th>
<th>Frame: 2</th>
<th>Row: 1</th>
<th>Col: 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Length:</td>
<td>14</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Alias:</td>
<td>ITEM</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Choice group:</td>
<td>CHOICES</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pad:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Type:</td>
<td>CHAR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Protect:</td>
<td>INITIAL</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Format:</td>
<td>Just: LEFT</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Informat:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Error color:</td>
<td>YELLOW</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>attr:</td>
<td>REVERSE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Help:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>List:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Initial:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Replace:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Options:</td>
<td>CAPS CURSOR AUTOSKIP</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

***** SOURCE *****

**INIT:**

`control always`;

`length rc 2 keepit $ 1`;

- 1. Define a dataset to correspond with the screen;
- 2. Add some data to this dataset;
- 3. Define a corresponding x-table;

* Define the dataset;

`items = open('items', 'n')`;

`rc = newvar(items, 'item', 'c', 14, 'r')`;

`rc = newvar(items, 'keepit', 'c', 1, 'r')`;

`rc = close(items)`;

* Add some observations;

`category = 'MEAT DAIRY FRUITS VEGETABLES JUNK'`;

`nitems = 5`;

`keepit = ''`;

71
items = open('items', 'U')
call set(items)
do i = 1 to nitems
   item = scan(category, i)
   rc = append(items)
end

* Define and prepare to display x-table on screen
call setrow(nitems, nitems)
RETURN

GETROW:
   * Display list of choices for user to select from
   rc = fetchobs(items, _currow)
RETURN

PUTROW:
   * Indicate on the dataset whether or not
     item was selected
   rc = fetchobs(items, _currow, 'X')
if issel(_currow)
   then do
      msg = _blank_
      keepit = 'X'
   end
else keepit = ''
rc = update(items)
RETURN

MAIN:
RETURN

TERM:
select(_status_)
when('C')
   put 'Request Cancelled'
   if items > 0
      then rc = close(items)
when('E')
   * if putrow did not execute
   * then error-ckg was not done
   if locate(items, 2, 'X') = 0
      then do
         * Need Atleast1
         msg = 'Select at least one category'
         _status_ = 'R'
         return
      end
   else do
      put 'Programming statements here'
      rc = close(items)
   end
otherwise
   put _status_ =
end
RETURN

Another common application is to present an empty list and request the user to fill in all the blanks. This is another of those problems which looks easy to solve, but quickly becomes a real challenge. The last example was a simple matter of selecting/unselecting a row. We were able to use some built-in routines, but the real secret was using the tracking variable to detect when the user didn't select any items. This same tracking variable strategy can be used in solving the 'all rows required' problem. Instead of checking that 'keepit' has been set to 'X' at least once, check that it has been set to 'X' for every observation.

PURPOSE: Fill in all the blanks
&msg ________________________

CATEGORIES:

ITEM

***** ATTRIBUTES *****

Window name: All Rows Required
Start row: col:
Number of rows: cols: Banner: COMMAND

General help:
Function keys: DISPLAY
Lookup data set:
Command menu:
Prompt char: ?
System Options: EXTENDED TABLE
Parent:
Window Type: STANDARD
Scroll Bars:

72
**** SOURCE ****

INIT:

length rc 2 fill_in highlite $ 1

* Define a 'mirror image' dataset
items = open('items', 'n')
rc = newvar(items, 'item', 'c', 8, ' ')   
rc = newvar(items, 'fill_in', 'c', 1, '')
rc = newvar(items, 'highlite', 'c', 1, '')
rc = close(items)

* Setup work variables
nrows = 5
nmissing = nrows

* Add five obs (five rows on the table)
items = open('items', 'u')
call set(items)

item = _blank_
fill_in = _blank_
highlite = 'N'
do i = 1 to nrows
   rc = append(items)
end

MAIN:

* Prepare to display the table
   call setrow(nrows)
RETURN

GETROW:

* Display the x-table on the screen
   rc = fetchobs(items, currow)
   if highlite = 'Y' and fill_in = _blank_
      then erroron item
RETURN

PUTROW:

rc = fetchobs(items, currow, 'noset')

* FILL_IN: put checkmark if row is filled in
select(item)
   when(_blank_) fill_in = _blank_
      erroron item
   otherwise fill_in = 'X'
      erroroff item
end

rc = update(items)
RETURN

TERM:

select(_status_)
   when('C'); put 'Request Cancelled'
      if items > 0
         then rc = close(items)
   when('E'); * make sure all of em are filled in
      if nmissing > 0
         then do; * not OK
            msg = 'All rows are required'
            do i = 1 to nrows
               rc = fetchobs(items, i)
         end
      else do;
         msg = 'All rows required'
         goto IN.rest}

*** SOURCE ***

INIT:

length rc 2 fill_in highlite $ 1

* Define a 'mirror image' dataset
items = open('items', 'n')
rc = newvar(items, 'item', 'c', 8, ' ')   
rc = newvar(items, 'fill_in', 'c', 1, '')
rc = newvar(items, 'highlite', 'c', 1, '')
rc = close(items)

* Setup work variables
nrows = 5
nmissing = nrows

* Add five obs (five rows on the table)
items = open('items', 'u')
call set(items)

item = _blank_
fill_in = _blank_
highlite = 'N'
do i = 1 to nrows
   rc = append(items)
end

MAIN:

* Prepare to display the table
   call setrow(nrows)
RETURN

GETROW:

* Display the x-table on the screen
   rc = fetchobs(items, currow)
   if highlite = 'Y' and fill_in = _blank_
      then erroron item
RETURN

PUTROW:

rc = fetchobs(items, currow, 'noset')

* FILL_IN: put checkmark if row is filled in
select(item)
   when(_blank_) fill_in = _blank_
      erroron item
   otherwise fill_in = 'X'
      erroroff item
end

rc = update(items)
RETURN

TERM:

select(_status_)
   when('C'); put 'Request Cancelled'
      if items > 0
         then rc = close(items)
   when('E'); * make sure all of em are filled in
      if nmissing > 0
         then do; * not OK
            msg = 'All rows are required'
            do i = 1 to nrows
               rc = fetchobs(items, i)
         end
      else do;
         msg = 'All rows required'
         goto IN.rest}

73
if fill_in = _blank_
then do
  highlight = 'Y'
  rc = update(items)
end
end; *do loop
_status_ = 'R'
return
end; *not OK
else do; * All OK
  put 'Programming stmts here'
  rc = close(items)
end
otherwise; put _status_ =
end
RETURN

Timing Issues

The timing of when the different sections execute is what makes extended tables so difficult to use.

- **INIT** executes once before the screen is even displayed. The extended table is defined in INIT.
- **GETROW** always executes immediately following INIT. GETROW executes once for each row of the table; this is how the initial values get assigned. At this point the table is displayed.
- **PUTROW** executes if, and only if, the user modifies a row in the extended table. PUTROW executes once for each row the user modifies. If the user does not modify any rows, the PUTROW section does not execute at all.
- **MAIN** executes once following the execution(s) of PUTROW. If PUTROW does not execute at least once, MAIN does not execute at all (see note about CONTROL ALWAYS below for the one exception to this rule).
- **TERM** executes once as the last step before leaving the PROGRAM screen.

GETROW and PUTROW are the only sections which allow access to individual rows on the screen. Use ERRORON and ERROROFF to set and clear row errors in these two sections only. In an application like "All Rows Required", it is impossible to know if all rows have been filled in by simply looking at a single row. You need to see the whole table. This kind of error checking should be done in the MAIN section. The problem is that you can't access just a single row in the MAIN section. However, you can interrogate the table as a whole and set a "row flag" for each row the user has left blank. Do not attempt to manipulate ERRORON or ERROROFF in the MAIN section. Instead save the error-condition information in a flag and send it back to the GETROW section to handle the ERROR when appropriate.

You may be wondering how we get from MAIN (detect errors and set flag) to GETROW. If the user presses ENTER, this is exactly what happens. But, of course, we cannot count on that. The user might just as well press the END key. In this case, control passes from MAIN directly to TERM. That means there needs to be some logic in TERM to detect if the flag has been set, and, if so to pass control back to GETROW. As it turns out, this part is easy. The automatic variable _status_ = 'R' (resume), passes control directly to GETROW (to re-display the screen).

The pivotal part in all this is that GETROW interrogates the flag and sets ERRORON if the flag indicates a problem. Go back for a moment and walk through this example again; pay attention to how the individual sections interact with one another.

It is a good idea to use CONTROL ALWAYS. This should be the first line in the INIT section. CONTROL ALWAYS forces the MAIN section to execute every time, regardless of whether or not PUTROW was executed. It is comforting to know that the error checking you have programmed will be executed every time -- regardless of what PUTROW is doing.

The TERM section has been programmed for the case when CONTROL ALWAYS has not been used. TERM, then, must be able to detect an error condition in the event that MAIN does not execute. Without CONTROL ALWAYS, if the user does not modify any row, then PUTROW and MAIN never execute. Without the error-detection logic in TERM, the user can get away regardless of how many errors are present. Notice a flag is used along with _status_ = 'R' to detect errors and pass control to GETROW. Remember, ERRORON and ERROROFF should be used only in GETROW and PUTROW, never anywhere else.

One last little tidbit: if you use ERRORON to set an error, the field remains in error until you use ERROROFF to explicitly clear it. The user may correct the mistake, but if you set the error with ERRORON, it won't go away until you clear it with ERROROFF.
"Fill in as many as you want, but no duplicates please."
This example is more complicated than the others, but it employs the same concepts: "dataset approach", tracking variable, timing, ERRORON-ERROROFF. It also illustrates the use of two very useful SCL functions: VARSTAT and LOCATEC.

***** DISPLAY *****

PURPOSE: No duplicates allowed

\&msg ______________________

CATEGORIES: ____

\&ITEM_____

***** ATTRIBUTES *****

Window name: No Duplicates
Start row: col:
Number of rows: cols: Banner: COMMAND

General help: 
Function keys: DISPLAY
Lookup data set: 
Command menu: 
Prompt char: ?
System Options: EXTENDED TABLE
Parent: 
Window Type: STANDARD
Scroll Bars: 

***** SOURCE *****

INIT:

CONTROL ALWAYS

length rc 3 duplicat $ 1
* 1. Define a dataset to correspond with the screen
* 2. Add some empty observations
* 3. Define a corresponding x-table

* Define a dataset
items = open('items', 'n')
rc = newvar(items, 'item', 'c', 14, ' ')
rc = newvar(items, 'duplicat', 'c', 1, ' ')
rc = close(items)

* Add some empty observations
nrows = 8
nmissing = nrows

items = open('items', 'u')
call set(items)
duplicat = 'N'
item = _blank_
do i = 1 to nrows
  rc = append(items)
end

* Define and prepare to display the table
call setrow(nrows)
call setrow(nrows)

GETROW:

* Display a blank list for user to fill in
* Turn error on if appropriate
rc = fetchobs(items, currow)

if duplicat = 'Y'
  then erroron item
  else erroroff item

RETURN
PUTROW:
*Save the user's answer on the dataset
rc = fetchobs(items, _currow, 'noset')
rc = update(items)
RETURN

MAIN:
* Always executed (CONTROL ALWAYS above);
goback = 'N'

* How many unique values did user specify?
rc = varstat(items, 'item', 'nunique', nunique)
rc = varstat(items, 'item', 'nmiss', nmissing)
if mmissing > 0
then nunique = nunique - 1
* missing is a unique value

fill_in = nrows - nmissing
select(nunique = fill_in)
   when(1); msg = _blank_
   when(0)
      * LOCATEC finds 1st occurrence
      * Look from bottom-up to find other
      * obs w/same value. If found, set
      * duplicat='Y'
      msg = 'Error: No duplicates allowed'
do bottomup = nrows to 1 by -1
   rc = fetchobs(items, bottomup)
   select(item = _blank_)
      when(1)
      when(0)
         topdown = locate(items, 1, item)
         if bottomup ^= topdown
         then do
            rc = fetchobs(items, bottomup)
            duplicat = 'Y'
            rc = update(items)
            goback = 'Y'
         end
end; * select(item)
end; * bottomup-loop
end; * select(nunique)
RETURN

TERM:
select(_status_)
   when('C'); put 'Request Cancelled'
      if items > 0
      then rc = close(items)
       when('E'); if goback = 'Y'
      then do
         msg = 'No duplicates allowed'
         _status_ = 'R'
         return
       end
       else if nmissing = nrows
      then do
         msg = 'At least one required'
         _status_ = 'R'
         return
       end
       put 'Programming statements here'
      if items > 0
      then rc = close(items)
      otherwise put_status_ =
       end
RETURN

The GETROW section executes for each line of the extended table. A blank table is displayed on the screen. Since the user has not filled anything in yet, the ERRORON condition has not had the chance to get triggered. The user fills in the table and presses ENTER or END. PUTROW executes for each modified row and updates the corresponding dataset with the user's responses. When PUTROW is done, MAIN executes. The VARSTAT function can tell us how many different values were filled in and how many rows were left blank. Knowing this and how many rows we had to begin with, we can determine if any value was specified more than once. If yes, then we want to find the first row where the value was specified (via LOCATEC). That value is OK. Then we want to look from dataset bottom to top for other occurrences of that value. When detected, flag the tracking variable, 'duplicat'. Set the 'goback' flag to 'Y'. Then TERM executes. If 'goback' = 'Y', control is passed to GETROW. GETROW will detect the error and set ERRORON, otherwise it will clear the error with ERROROFF.

Summary

Extended tables can be intimidating. The syntax itself is not particularly tricky, but it can be a real challenge to get the upper hand in controlling your process. The key is knowing how SAS will execute your program. Once you get a handle on the timing of events, you'll begin to enjoy programming again.
The examples shown in this paper are intended to provide a template to guide you in programming more complicated applications. Having a simple example which illustrates the basics can help make a problem easier to solve. I hope these examples will help you as you work with extended tables.

Contact Information:

Karen Crandall
Eastman Kodak Company
Kodak Park
Rochester, NY 14652-4608
Telephone: (716) 722-6512

SAS is a registered trademark or trademark of SAS Institute Inc. in the USA and other countries.
® Indicates USA registration.