SCREEN CONTROL LANGUAGE PROGRAMMING TECHNIQUES
FOR CONCURRENTLY ACCESSIBLE DATA
Mike Carney, SAS Institute Inc., Cary, NC

ABSTRACT

When writing an SCL program that reads or updates SAS® data sets concurrently with other SCL programs or users, there are certain techniques you may employ when designing a program to maximize concurrency and ensure data integrity. This paper explains the application of concurrent SCL programming, develops a methodology for producing such a program, and illustrates these techniques with examples.

INTRODUCTION

Screen Control Language (SCL) is a powerful tool for creating custom data entry and report generation applications, but it becomes even more versatile when used in conjunction with SAS/SHARE® software. With the shared data access through a SAS/SHARE server, SCL has the ability to read and update SAS data sets concurrently by other users or SCL programs.

This paper is concerned with data-concurrent SCL applications, meaning those SCL programs which perform concurrent, programmatic update access to one or more SAS data sets. For example, SCL applications that open a SAS data set for update concurrent with other update opens. Subsequent update opens may be done by other instances of the first SCL application, a completely different SAS or SCL application, or even a user running the FSEDIT procedure against the data set. When the other updater modifies the same records and variables that the SCL application may need to alter, special considerations are necessary in programming the application and are the focus of this paper.

A key factor which underlies much of the discussion of this paper is the way in which SAS locks records in data sets that are open for update. Locking a record means preventing any other concurrent [update] open from altering or deleting that record; only the locker may alter the record. This prevents cross-update of records by concurrent opens. In Release 6.06 of the SAS System, only one record may be locked to an update mode open at a time. Thus a data-concurrent application must not rely on the values in the record remaining the same after the application has read another record of otherwise unlocked the former record. This paper will show how to turn this seeming limitation to an advantage in developing data-concurrent SCL applications.

The reader is assumed to be conversant in SCL and necessarily familiar with SAS/SHARE® and SAS/AF® software. Familiarity with the concept of opening a SAS data set, processing selected records, and closing data sets from SCL is important. No knowledge of SAS/SHARE software is necessary for reading this paper.

Several example applications demonstrate the usefulness of concurrent SCL programming and are used in illustrating the methodology which this paper develops. The complete SCL programs are given in the Appendix. The first of these is a simple inventory/order system. You have a list of your store’s inventory which includes product code, description, and number inventoried. You want to automate a system which facilitates the development of orders and, correctly maintains the inventory list while clerks simultaneously write orders for products.

Another more advanced example is drawn from the formal demonstration of SAS/SHARE software at SUGI15. You need a system that allows a number of registrars to simultaneously enter SUGI registration information on behalf of attendees, notably for roundtable attendance. The registrars may add new roundtable sessions for an attendee or change existing selections. You also need a method for roundtable administrators to update the roundtable schedule. You must ensure that registrars do not overbook a roundtable topic and that administrators do not reduce its capacity below the current number of registrants. Of course you want both groups to be able to update the information simultaneously.

The basic functional model of a concurrent data SCL application discussed in this paper may be stated as follows. Given an initial configuration of the data sets which satisfy all intended relations of the data, the SCL application must preserve these relations under any modification to the data by preventing any modification which would invalidate any relation.

DESIGNING THE DATA SET LAYOUT

Initial Organization of Major Data Sets

In architcturing your application you should first organize the data items or variables into the major data sets. Simply enough, begin the design with considerations external to the [concurrent] application. This will likely not determine the final layout of data for your application, but will provide a good baseline from which to deviate.

Guideline 1: Use the most data-natural organization possible. This means putting data items together that best hang together and avoiding replication of specific data items in separate places, including the separation of data items which are equivalent or imply each other. For example you would not want to have a variable z in data set ana which is defined as the sum of variables x and y in data set two.

For the inventory/order example this means starting with the obvious organization of two data sets. The first will be the inventory data set which has one record per product and the following variables:

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>code desc</td>
<td>Char 4</td>
<td>Product code</td>
</tr>
<tr>
<td>invent</td>
<td>Num 8</td>
<td>Number of units inventoried</td>
</tr>
</tbody>
</table>

The other would be the order data set which would have one record for each product ordered and would be replicated for each customer, day, and so on. It would have at least the following variables:

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>product</td>
<td>Char 4</td>
<td>Product code</td>
</tr>
<tr>
<td>quantity</td>
<td>Num 8</td>
<td>Number of units ordered</td>
</tr>
</tbody>
</table>
Guideline 2: Select an organization expedient for processing the data sets external to your application. This includes any off-line processing (data set(s) may only be modified by that application) or on-line processing which does not modify concurrent variables nor care if they are modified at any point in its execution.

In the roundtable administration example, say that you wanted to periodically generate a report of the attendees for any roundtable section in a straightforward manner. You might try to structure the roundtable data in a single large record, one for each roundtable section, with the variables:

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>top_num</td>
<td>Num 8</td>
<td>Number of the topic</td>
</tr>
<tr>
<td>room</td>
<td>Char 4</td>
<td>day of the roundtable</td>
</tr>
<tr>
<td>attend01</td>
<td>Num 8</td>
<td>Room in which the roundtable will be held</td>
</tr>
<tr>
<td>and so on ...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>attend10</td>
<td>Num 8</td>
<td>First Attendee's Id number</td>
</tr>
</tbody>
</table>

Modification of Data Set Organization for Reducing Conflict

In the second round of design you should consider the likelihood of conflict among updaters to your application's data sets because the SAS System does not allow concurrent update access to a given record. Clearly you want to minimize the likelihood of conflict, thus possibly necessitating trade-offs with the goal of having all closely related data together in the same record. The questions to be weighed include:

- Who (what group) will need to alter which variables?
- How many concurrent updaters is each group likely to include?
- Do all members of an updating group have equal authority to the updatable variables?

In the roundtable administration example, there are two distinct groups of updaters; the registrars and the roundtable administrators. The registrars will need to update the roundtable registration for a given attendee at a time. The administrators will need access to the data associated with a given roundtable section. To minimize the likelihood of conflict among all updaters, reorganize this data into two data sets. Place the roundtable information in one data set with one record for each roundtable section. Include the variables:

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>top_num</td>
<td>Num 8</td>
<td>Number of the topic</td>
</tr>
<tr>
<td>room</td>
<td>Char 3</td>
<td>day of the roundtable</td>
</tr>
</tbody>
</table>

Place the registrant part of the information in another data set (probably containing other registrant information) using two variables:

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>table1</td>
<td>Num 8</td>
<td>First roundtable topic [number] to attend</td>
</tr>
<tr>
<td>table2</td>
<td>Num 8</td>
<td>Second roundtable topic [number] to attend</td>
</tr>
</tbody>
</table>

Because an attendee can attend a maximum of two roundtables, this is a general structure.

Modification of Data Set Organization for Relation Maintenance

In the design of your application system's data structure you have not yet considered the set of relations which you want to enforce between the data of your system. It is important to identify these relations precisely at this point and facilitate their maintenance by the organization of the system's data. The Inventory/order example requires the simple one-sided relation, 'inventoried number of units' >= 0. In consideration of orders. The roundtable administration system implies a two-sided relation, 'roundtable topic capacity' >= 'roundtable topic number of registrants'. Note that by implication, the validity of other data items must be maintained, for example, 'beginning number of units' = 'ordered number of units' + 'inventoried number of units' in the first example application.

Within the defined data sets identify the variables/records which establish the relation. For example, in the inventory/order system this is the 'inventoried number of units'. Maintaining the selected relationships becomes easier if the data can be isolated in a single record or at least in single records in different data sets. Remember that you can lock a single record in a SAS data set that is open for update so its contents may not be altered by another user or application. In this way you may utilize the single-record locking which the SAS System employs to preserve the integrity of your data relationships. Techniques for obtaining and handling locks are presented in the next section.

As you can see, the data structure of the Inventory/order system as proposed above isolates the data which establishes (each specific instance of) the relationship on single records. There is one record per product on the inventory data set and it contains the 'inventoried number of units' variable.

In the roundtable administration example, there are no single variable/records which establish a single instance of the desired relationship. For topic x, it is the "number of sections of topic x times 10" (assuming a capacity of ten per section) and the "total number of attendees registered for topic x." The solution to protecting this relationship lies in creating an additional data set called a control data set. Each record will pertain to one roundtable topic and will contain three variables:

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>title</td>
<td>Char 80</td>
<td>Roundtable topic title</td>
</tr>
<tr>
<td>capacity</td>
<td>Num 8</td>
<td>Capacity of the roundtable [topic]</td>
</tr>
<tr>
<td>current</td>
<td>Num 8</td>
<td>Total number of registrants for the topic</td>
</tr>
</tbody>
</table>

You may wonder why the roundtable administration data set could not be modified to serve as the control data set. Functionally it could by replicating the section-specific set of variables, day and room, for the maximum number of sections allowed. Then each record would represent a roundtable topic and the aforementioned capacity and current-registration variables could be added. This is probably not a good idea because the large record will be locked as long as a roundtable administrator purports and enters modifications to it. You would need to lock that record at least briefly each time a registrar enrolls or disenrolls an attendee. The scenario assumes that a SAS/FSP procedure would be employed by the administrator.

The concept of a control data set becomes even more attractive when more grandiose or esoteric relations are
desired in a system. What if you wanted to maintain a cap on the number of roundtables scheduled on a given conference day? Merely create a control data set with one record for each conference day containing the total number of roundtables scheduled for that day. The record for a given day would be locked (and thus the relationship could be enforced) whenever an administrator added or deleted a roundtable section.

FRAMEWORKS FOR CONCURRENT SCL APPLICATIONS

A Screen Control Language program is written for use with a SAS/FSP procedure or for use in a SAS/AF program. Each of these underlying procedures implies a functional template into which the application must fit. These templates provide special opportunities and present special requirements on data-concurrent SCL applications. The most common use for concurrent SCL is in validating modifications and additions to SAS data sets under the FSEDIT procedure.

The FSEDIT and FSBROWSE procedures effectively give the SCL programmer five exits in which to structure the SCL program. The FSEINIT section is useful for opening auxiliary data sets and locating their variables, and for performing any once-per-execution initialization. The converse exit, FSETERM, is useful for closing auxiliary data sets.

The INIT section is useful to concurrent applications because the current values of a pre-existing observation may be preserved in SCL variables herein. The values are useful for several reasons:

1. The application can determine if a modified window variable actually contains a value different from its initial, validated value.
2. The application can restore initial values in case the modifications to the FSEDIT DATA=data set are not allowed because they would violate a data relation.
3. The application can maintain the specific relationship(s) implied by the init (now updated) values.

The MAIN section and possibly also the TERM section will enforce the system relations and one or the other must update auxiliary data sets. Select fields may be restored from initial values in either section if all modifications are not allowed. It is important that no record of a control data set remain locked between these sections. That would lock other users or applications out of making modifications to variables participating in the locked relation for a truly arbitrary amount of time. Also remember that when the TERM section is executing, the user is moving on the current edit observation and the associated SCL application cannot prevent this by issuing an ERRORON for any window variables.

Several FSEDIT [user] commands are likely to require action in your SCL application. The FSEDIT DELETE command can be sensed in the MAIN section using the WORD function. Note that is also necessary to set CONTROL ENTER (or higher) so the MAIN section will be executed when the DELETE command is entered. It is not possible to handle the DELETE command completely satisfactorily because it is not possible to restore the values of a deleted observation when the SCL application discovers that system relations cannot be enforced given the delete. In this case the application can only log an error message containing the deleted observation's former values or add the deleted observation to a special auxiliary data set for later off-line processing.

If your SCL application needs to distinguish new observations from existing ones, as the example applications do, you need to likewise check for the DUPLICATE FSEDIT command. The problem is that a new duplicated edit observation looks just like an old observation in the INIT section in that its variables have non-missing values. When the application senses a DUPLICATE command executing its MAIN section it must interpret the very next execution of its INIT section to pertain to a new observation. This is true unless a window variable becomes flagged as in error on that sensing execution of MAIN.

In SAS/AF programs the INIT and TERM sections are analogous to the FSEINIT and FSETERM sections in FSEDIT and FSBROWSE procedures for the purpose of this discussion. Because there is no inherent editing of a data set when a SAS/AF program is run, there is no issue of restoration of previous values or maintenance of old specific relationships. The MAIN section subsumes all the duties of FSEDIT's MAIN and TERM sections (that are applicable).

The formulas of the FSVIEW procedure do not provide a practical framework for a concurrent SCL application. Whereas it is possible to cleverly effect a one-time-only, FSEINIT-like initialization in an FSVIEW formula, there is no way to simulate a FSETERM-like termination section. It is of course completely impractical to open and close auxiliary data sets within each formula, especially when the data sets must be accessed through a SAS/SHARE server. Further, it is impossible to preserve an observation's initial values for comparison or restoration.

TECHNIQUES FOR MAINTENANCE OF DATA INTEGRITY/RELATIONS

It is the job of the SCL application to maintain the integrity of the data in the application and enforce the desired data relations. Its general strategy is embodied in the following steps:

1. Lock the necessary, instance of the data relation by locking one or more data set records.
2. Make / allow data modifications which preserve the relationship.
3. Unlock the relation instance.

Locking and Unlocking Relationships

As discussed earlier, you have isolated the data which establishes each [instance of the] relation(s) of the system into the variables of a single record, or possibly several records, per data set. Once that application locks the relation instance by fetching (and locking) one or more records, it may proceed to validate the requested modifications.

Remember that in Release 6.06 of the SAS system only the lock on the current record accessed in an open data set is retained. Once that lock is released by reading another observation on that open or by one of the other methods discussed below, your application can no longer be sure of the values that observation contains. Another updater may have already modified the values so that any data
modifications made based on the old values may destroy the data integrity of the system. Always consider that an arbitrarily long period of time has elapsed between a previous and subsequent read and lock of a record.

A record (or conceptually a relation) lock may be enforced at initial fetch time or at the time the control data set is updated. Enforcing the lock at fetch ensures the validity of the obtained data and represents the most straightforward application of the general strategy. However, this may not be desirable for all purposes, especially when the application would like to merely check the current validity of a change or addition, but make no immediate update. Of course if a lock is not enforced and update made [to the control data set] at the time a modification is validated, the modification must be reevaluated under lock at update time. The inventory/order application example illustrates validation at update time and the roundtable administrator applications demonstrate validation at initial fetch.

Regardless of when your application attempts to enforce the record lock, the attempt may fail due to a conflicting lock. It would be unseemly to report a lock error to the user upon a single failed attempt to secure it. One of the techniques for waiting on a lock is called spin locking. The application fetches the fetch of the observation number till it obtains the lock. The roundtable administrator applications demonstrate this method.

Spin locking is appealing in its simplicity, but would prove impractical if another application or user might lock the target observation for a long period while your application is running. An example of this is a user running the FSEDIT procedure against the inventory data set of our example. Your SCL application would not give control back to the clerk if the FSEDIT user entered or a product that the lock attempted to order. The clerk would think something was amis in the application after waiting a few moments to receive back control. Modifying spin locking to limit the number of spins will alleviate such a problem. After some number of failed attempts to obtain the lock, return to the user reporting the failure. The user can try again or proceed with another task. Timing tests may be desirable to determine a good retry limit. The inventory/order application exhibits limited spin locking.

An FSEDIT based application needs to be prepared to handle the lock unavailable error condition if it can occur. The situation is similar to the application refusing a modification/addition of data due to violation of a data relation. The application may restore the values of window variables to their initial values and report the error to the user. Alternatively, the application may only report the error condition to the user but leave window variables as modified. These would be restored to initial values (in the TERM section) only if the lock/validation/update is never realized. Obviously, either approach implies that the initial values of window variables be preserved at INIT time.

In a concurrent SCL application, it is as important to properly unlock a record as it is to lock one. The application should unlock the record as soon as possible to minimize conflict with other applications/users. There are several ways to unlock a locked record from SCL. The most efficient method is to issue the SCL UNLOCK function. It leaves the data set at its current position and does not update the data set Data Vector (DDV). Another way to unlock a record (and not lock another new record) is to position to a dummy observation, such as the first or last observation in the data set. This is practical if the dummy observation is trivial to access, especially via observation number. With this method, the current record or position of the data set is always the same when no record is locked. Both unlocking methods are used by the example applications.

Locating/Fetching Control Records

SCL provides a set of functions useful for locating and fetching the required control records in a data-concurrent SCL application. A caution is in order concerning the use of these functions for concurrent applications. The return code, obtained directly from the called function or from the SYSRC function, must be checked to ensure that a lock was obtained or that an update was successful. The return value, %SYSRC(SNWNGUPD), is generated when a fetch or update function fails to lock or update the record due to it being locked by another application, except as noted below.

FETCHB(S) is a good function to use when the observation number can serve as the relation key. It is used in the roundtable administration example applications. Remember that this function accepts a relative observation number by default that may or may not equate to the physical observation number. If deleted records in the control data set are possible, you probably want to specify the ABSN option (for absolute record numbering) to the function. Warning: For Release 6.06 of the SAS System, this function returns a zero when the function cannot do binary searches on compressed sets.

LOCATEC/N function is good for finding unique keyed observations in small data sets (say of less than 100 observations) when the data can remain sorted by the key and a binary search specified. Outside of this scenario, the overhead of searching in the SCL function probably indicates use of the WHERE function. The reason for this is that in a shared-data environment, each checked record must be requested from the SAS/SHARE server and transmitted to the user's execution. The SYSRC function must be queried for warnings when LOCATEC/N finds a record because the function only returns a zero-positive indicator for did not find record or found record. Remember that LOCATEC/N cannot do binary searches on compressed data sets, views, or data sets with deleted observations.

The more general and usually more efficient manner to locate a record employs the WHERE function followed by a FETCH function call. The efficiency is particularly good on large data sets that are indexed, if the where subset is empty [record not found], FETCH returns a -1 indicating End Of File. Note that if the where clause is cleared by issuing a null WHERE function call, the data set is positioned to the first observation which will be locked (if available). In this case either the first observation must be a dummy observation or UNLOCK must also be called.

The DATAUSTC/N function is indeed useful for helping a user pick a valid selection for a variable. This function actually fetches the entire selected record into the DDV. Unfortunately it is not useful to use as a search key for one of the other functions. In this capacity and especially on larger data sets, it would be preferable to use this function to access a read-only index of keys. This would allow DATAUSTC/N to access the data locally instead of through a SAS/SHARE server and would improve the performance of the application system.
SUMMARY

The methodology which this paper discusses for developing data-concurrent SCL applications falls into three interrelated areas. The first of these deals with the data set design which satisfies the requirements of the concurrent SCL application to maintain a desired data relationship. The second relates to the data to minimize the likelihood of conflict among updates. Finally, tweak the design to allow your concurrent SCL application to maintain all desired interdata relations, by identifying those relations and by specially isolating the data which establishes them.

The SAS System framework which will contain and interact with your SCL application provides its own set of special considerations. FSEDIT and FSBROWSE provide five SCL application exits, each of which must be properly coded to maintain the integrity of your system. The DELETE and DUPLICATE FSEDIT commands that a user enters probably need to be detected and dealt with in your application. AF programs have only three SCL application sections, and do not provide the product interface complications which FSEDIT does because there is no editing of data outside the control of the application. FSVIEW is not a useful framework for a data-concurrent SCL application because it does not provide the necessary SCL exits.

Once the data set layout is determined and the gross framework of the SCL application is known, the application(s) must be written to maintain data integrity and enforce the desired data relationships. This is accomplished generally by locking the relation instance, making or allowing data modifications which preserve the relationship, and unlocking the relation instance. This paper has discussed different techniques for obtaining a lock and for releasing it. An assortment of useful SCL functions that locate and lock data set records are discussed in the final subsection.

APPENDIX: EXAMPLES

/*
 * Inventory/order system SCL application for use
 * with PROC FSEDIT when editing an order data set
 * Issues an error message to the SAS log upon failure to update inventory when a delete is issued.
 * A update failure when adding an order will leave
 * a null order in the order data set.
 */
length rc 8; /* System return code storage */
length invent 11 ,
length length Invent 11 ,
length wordi 40; /* Fetch area for FSE command */

FSEDIT: /*
 * Open the product control data set and save the
 * needed variable nos. The open actually locks the
 * first record but it is released immediately in
 * /EXIT. Specify "control enter" to always run NAYS
 * to catch illegal input values as a new obs., to
 * catch deletions, and to forecast when the next
 * (order) record is actually a new duplicative one.
 * "Control term" issues that deleted and dupl'd
 * (new) records actually update the inventory data.
 * /init the duplicate flag, dupflg, here since it
 * cannot be reinitialized by RUN.*/

codeid = open('mylib.inventor', 'U');
wordi = varnum(codeid, 'wordi');
length wordi 40;
 invent = varnum(codeid, 'invent');
control enter term;
dupflg = 0;
return;

MAIN: /*
 * Entered upon each user <enter> or function key,
 * /this section verifies the product code and
 * quantity requested against the inventory data
 * set. It also helps the program remember when
 * an order has been deleted or duplicated.*/
wordi = wordi(1, 'U');
if (substr(wordi, 1, 3) = 'DEL' or deleted ) then do;
delorder = 1;
return;
end;
else if (substr(wordi, 1, 3) = 'DUP' ) then dupflg = 1;
if (product = ' ' ) then link needcode;
else {if (sav_ex = quantity or 'olddord') then do;
link getrec;
rc = unlock(codeid);
rc = field('product', 'product');
return;
end;
end;
 oleorder = 1;
link getrec;
rc = unlock(codeid);
rc = field('product', 'product');
return;

getrec: /*
 * Generally, this section fetches the desired
 * record of the inventory data set. If a valid
 * product code is specified, the binary searching
 * LOGICED call will fetch (and lock if possible)
 * the inventory record. DATALIBF is employed to
 * provide a selection list to the user and perform
 * an non-locking fetch of the selected product.
 * The product code will always be available at
 * observation termination time and thus LOGICED
 * will be used to lock and read the record.*/
cursor product; getrec = 0;
if (substr(product, 1, 1) = '?' ) then do;
/* Handle prompt from user */
product = ' ';
product = datalname(codeid, 'code invent desc');
/* Please select product code, 'Y'*/
if (product = ' ' ) then do; /* No selection*/
if (sysarc > 0 ) then do; /* Error*/
sav_ex = sysarc();
return;
end;
/* Check the amount available */

/* Generally, this section fetches the desired record of the inventory data set. If a valid product code is specified, the binary searching LOCATEC call will fetch (and lock if possible) the inventory record. RARELISTC is employed to provide a selection list to the user and perform an non-locking fetch of the selected product. The product code will always be available at the beginning of the data set and will be used to lock and remap the record. */

if ( sav_prod = product & oldorder ) then
/* Just a quantity change, add back old value */
invent = invent + savquan;
elseif ( quantity <= 0 ) then do;
   _msg = 'Please enter quantity to be ordered';
   erroron quantity;
   return;
else if ( quantity > invent ) then do;
   _msg = 'ERROR: Available stock is ' || put(varn(invent),best,);
   erroron quantity;
   return;
else do;
   _msg = 'Press any key to enter another product';
   _loop_cnt = 1;
   return;

needcode:
/* Ask user to enter product code. Set ERRORON to prevent exiting the observation (in NAER). */

/* Everything */

erroroff _all_;
return;

RETURN:
/* Termination: Close the lookup data set if it was indeed successfully opened. */
if ( codeid > 0 ) then rc = close(codeid);
return;

if ( erron ) then do;
   product = ' '; quantity = 1;
   return;
if ( deleted ) then put 'ERROR: Inventory from deleted order consisting of quantity units of product number ' product ' could not be restored to the inventory data set. '; _msg = _msg || ' Update failed.';
else do;
   product = ''; quantity = 1;
   return;

if ( field('ERROR','product quantity') ) then
/* An error exists, it is a no-go. */
goto restore;

if (sav_quan="quantity or deleted or not oldorder) then do;
   if ( deleted ) then do;
/* Initial values are effective on delete */
product = sav_prod; quantity = sav_quan;
end;
/* Try to lock inventory record to update */
loop_count = 0;
link getrec;
if ( not getrec ) goto restore;
if ( locrec = keyrec(codeid) ) then do;
   if ( loop_count < 500 ) then goto loop_count;
   _msg = 'ERROR: Product was locked. '
   goto restore;
end;
if ( deleted ) then
/* Need to add back the order quantity */
invent = invent + quantity;
else do;
/* Need to recheck and debit the inventory */
link chckquan;
if ( not qatch ) then goto restore;
invent = invent - quantity;
end;
putvarn(codeid,viewn,inventory);
calc putvarn(codeid,viewn,inventory);
if ( syrac() > 0 ) then do;
   _msg = syzmsg(); goto restore;
end;
rc = unlock(codeid);
/* In case user didn’t leave observation */
/* clarify that this order is saved. */
sav_prod = product; sav_quan = quantity;
oldorder = 1;
end;
return;

restore:
/* Restore the order’s previous values upon exiting */
/* after an unsuccessful validation of the order and update to the inventory data set. A new observation must be explicitly assigned nil values in case it was duplicated and not just “added”. */
if ( oldorder ) then do;
   product = sav_prod; quantity = sav_quan;
   return;
else do;
   product = ''; quantity = 1;
   return;
*/

run;
Because registrars and roundtable administrators are editing two distinctly different data sets, two different SCL programs are required to operate the PROC FSZEDIT invocations. The registrar’s program ensures that new registrations do not exceed the capacity of any roundtable topic and that dropped registrations are correctly added back to their topic capacities. The roundtable administrator’s SCL program will increment the capacity of a topic when an administrator adds a new section of a topic and decrement that capacity when a section is dropped; however, it will not allow a drop in capacity below the current number of registrants for the topic.

```plaintext
/* The Roundtable Administrator’s Program - for use */ /* with PROC FSZEDIT editing the roundtable schedule data set. Observations may not be deleted while */ /* editing this data set; removing a roundtable */ /* section is done by blanking the topic number. */ /* The topic number may not otherwise be changed */ /* in a pre-existing section. */

length word $ 40; /* Fetch area for FSH command */ length operation $ 3; /* Operation to perform on */ /* section = “add” or “del”/sets*/

PROCEDURE:

/* Open the control data set and save the interesting */ /* string variable now. No need to move off the first */ /* record because it is a dummy record. */ /* Specify “control” enter to always run MAIN to */ /* catch illegal initial values on a new obs, to */ /* catch deletions, and to forecast when the next */ /* (order) record is actually a new duplicated one. */ /* Initialize the duplicate flag, duplicate here */ /* since it cannot be done unconditionally in INIT. */

INIT:

/* Save initial roundtable section values. */ /* Set the “~ num” in “missing” to indicate a new */ /* section when it was duplicated, its a new order. */ /* For a pre-existing order, remind the user how a */ /* delete is affected. */ /* When the topic number is available for the */ /* observation fetch the topic title from the */ /* control data set, not forgetting to unlock the */ /* record by moving to the dummy observation. */

/*+---------------------------------------------*/
say_title: say room: room: say_tnum: top_num:
if ( duplicate ) then do;
   say_tnum: = ;
   duplicate: = ;
end;
if ( say_tnum ) then
   _msg_ : “Blank top_num to delete this section”;
if ( top_num > 0 ) then do;
   rc = fetchobs( t_inv, top_num );
   title: = getvar( t_inv, t_tie )
   rc = fetchobs( t_inv, 1 );
end;
else do;
   _msg_: “Please enter the Topic Number.”
   curor top_num:
end;

return;

MAIN:

/* Entered upon each user “enter” key on function key, */ /* this section verifies the addition or deletion */ /* of a roundtable section and updates the control */ /* data set. It also senses whether an order has */ /* been duplicated. */

word = word(0,"V")
if ( substr(word,1,3) = “DEL” ) then do;
   _msg_ = “ERROR: Delete is not allowed.”
   return;
end;
else if ( substr(word,1,3) = “DUP” ) then duplicate = 1;
/* No change means nothing to do */
if ( say_tnum = top_num ) then do;
   if ( top_num < 1 ) then do;
      _msg_: “Please enter the Topic Number.”
      curor top_num;
   return;
end;
/* If reached this point, something has changed */
/* or this is a new, duplicated observation. */
if ( say_tnum ) then do;
   if ( top_num )
      _msg_: “ERROR: Topic may not be changed.”
   | “Previous value restored.”
   top_num = say_tnum;
   errornum top_num;
   return;
end;
/*(say_tnum = & top_num = ) => delete section*/
/*else operation = “del” */
end;
/* say_tnum = . and top_num = . - add section */
/*else operation = “add” */
end;
/* Split till the lock is not denied */
/* Check that the fetch worked, this validates */
/* the topic number. */
do until( syserr( "tsyserrammed") )
   rc = fetchobs( t_inv, top_num );
end;
if ( syserr( > 0 ) ) then do;
   /* Error */
   errornum top_num;
   _msg_: syserr( ) | “Fetch failed.”;
   return;
end;
/* Got the record! Get the current enrollment */
/* and capacity variables and verify the */
/* operation. If a capacity reduction attempt */
/* fails, set an appropriate message and restore*/
/* initial values. */
curr = getvar( t_inv, t_curr );
capacity = getvar( t_inv, t_cap );
if ( operation = “add” ) then capacity = capacity + 10;
else do;
   capacity = capacity - 10;
if ( current < 0 ) then do;
   _msg_: “You cannot reduce capacity below”
   | “the current number of registrants.”
   day = save_day: room = save_room: top_num = save_tnum;
   errornum top_num;
   return;
end;
/* The modification must be ok if got to this */
/* point, so update the roundtable topic capacity. */
/* If successful, the new day, room, and topic */
/* number becomes the “old” values. */
```
call putvar( t_inv, t_cap, capacity );
re = update( t_inv );
if ( syserr() > 0 ) then do; /* Error */
   erroron top_num;
   msg =sysmsg(); || 'Update failed.'; return;
end;
re = fetchnbs( t_inv, 1 );
say_day, sav_room, room; say_table = top_num;
return;

TERM:
/*-----------------------------------------------*/
// Nothing to do at observation termination.
-----------------------------------------------*/
return;

ERROR:
/*-----------------------------------------------*/
// Terminate Close the lookup data set if it
// was indeed successfully opened.
-----------------------------------------------*/
if ( t_inv > 0 ) then re = close( t_inv ); return;

/*-----------------------------------------------*/
// Roundtable Administration System SQL Program II:
// The Registrar's Program - for use with PROC FSEDIT
// editing the attendee registration data set.
// issues an error message to the SAS log upon
// failure to update control data set to reduce
// current enrollment in a roundtable topic.
-----------------------------------------------*/
length re $ ; /* System return code storage */
length wordl $ 48; /* Fetch area for FSR command */

PAR1NT:
// Open the control data set and save the interest-
// ing variable nos. - No need to move off the first
// record because it is a dummy record.
// Specify 'control enter' to always run MAIN to
// catch illegal initial values on a new obs, to
// catch deletions, and to forget what the next
// [order] record is actually a new duplicated one.
// Initialize the duplicate flag, duplicat, here
// since it cannot be done unconditionally in INIT.
-----------------------------------------------*/
t_inv = open( 'mylib.tblInvWen', 'U' );
t_cap = varnum( t_inv, 'CURRENT' );
t_room = varnum( t_inv, 'CAPACITY' );
control enter;
duplicat = 0;
return;

INIT:
/*-----------------------------------------------*/
// Save initia1 roundtable section values turn off
// the deleted flag.
// Set the 'say_tbl1's to missing to indicate a new
// section when it was duplicated.
-----------------------------------------------*/
/*-----------------------------------------------*/
if ( duplicat = 0;)
say_tbl1 = .; say_tbl2 = .; duplicat = 0;
end;
else do;
say_tbl1 = table1; say_tbl2 = table2;
end;
return;

MAIN:
/*-----------------------------------------------*/
// Entered upon each user <enter> or function key.
// this section uses "table" to sense and handle
// any change in the attendee's roundtable
// registration. It also senses whether an order
// has been duplicated or deleted.
-----------------------------------------------*/
if ( deleted ) then return;
wordl = word(1, 'U');
if ( substr(wordl,1,3) = 'DEL' ) then do;
deleted = 1; table = .; table = .;
end;
else if ( substr(wordl,1,3) = 'DUP' ) then duplicat = 1;
table = table1; say_tbl1 = say_tbl1;
link doable;
if ( errorflg ) then do;
table = say_tbl1;
erroron table;
return;
end;
say_tbl1 = table1;
table = table2; say_tbl1 = say_tbl2;
link doable;
if ( errorflg ) then do;
table = say_tbl2;
erroron table;
return;
end;
say_tbl2 = table2;
return;

detable:
/*-----------------------------------------------*/
// If the saved table topic differs from the current
// one then there is something to do here.
// Call 'detable' to verify the change in the
// attendee's roundtable registration and make the
// change. 'change' is the increment to the current
// roundtable topic registration.
// The new roundtable topic must have its current
// enrollment incremented by one and the old topic's
// enrollment must be decremented by one.
// If the addition of the new topic fails, don't
// remove the old topic so the caller can restore.
-----------------------------------------------*/
errorflg = 0;
if ( table = say_tbl1 ) then do;
change = 1; link doable;
if ( errorflg ) then return;
table = say_tbl1;
change = -1; link doable;
if ( errorflg ) then do;
   put 'ERROR: Unable to reduce the current '
      'enrollment of Roundtable table ';
errorflg = 0;
end;
end;
return;

detable:
/*-----------------------------------------------*/
// For a nonmissing topic, fetch the control record,
// change the current enrollment and verify it, and
// then update the control record.
-----------------------------------------------*/
if ( table = . ) then do;
link fetches;
if ( errorflg ) then return;
current = current + change;
if ( current > capacity ) then do;
   msg = 'Error! There is no more room in '; ||
      'Roundtable table '; return;
errorflg = 1; return;
end;
link updatrec;
end;
return;

fetchrec:
/Spin till the lock is not denied.
/check that the fetch worked, this validates
/the topic number.
/When the record is fetched get the current
/enrollment and capacity variables.
/-------------------------------------
do until( syield() = syield(_sysmode) );
  rc = fetchobs( t_inv, table );
end;
if ( syield() > 0 ) then do; /* Error */
  _msg = sysmsg(); "Fetch failed.';
end;
case do;
current = getvar( t_inv, t_cur );
capacity = getvar( t_inv, t_cap );
end;
return;

updatrec:
/Attempt to update the control data set and
/unlock the control record.
/-------------------------------------
call putvarn( t_inv, t_cur, current );
rc = update( t_inv );
if ( syield() > 0 ) then do; /* Error */
  errorflag = 1;
  _msg = sysmsg(); "Update failed.';
end;
rc = fetchobs( t_inv, 1 );
return;

TERM:
/Nothing to do at observation termination.
/-------------------------------------
return;

PATTERN:
/---------------------------------------------------------------------
/Termination: Close the lookup data set if it
/was indeed successfully opened.
/---------------------------------------------------------------------
if ( t_inv > 0 ) then rc = close( t_inv );
return;

REFERENCES
SAS, SAS/AF, SAS/FSP, and SAS/SHARE are registered trademarks of SAS Institute Inc. Cary, NC, USA.