Handling Specific Errors

Doug Hennig

Letting a general error handler deal with specific foreseeable errors isn't a good idea, since the handler won't know the environment the errors occurred in nor possible resolutions to the problem. This month's column shows techniques for handling specific foreseeable errors at the appropriate level.

Last month's article presented the design and code for a general error handling scheme. This scheme used a Chain of Responsibility design pattern so error handling can be escalated from the Error method of individual objects all the way to a global error handler providing common services. However, we didn't address what to do about specific errors, such as field or table rule violations or trigger failures. Letting these types of foreseeable errors be handled by the global error handler makes no sense, since it can't possibly be aware of the environment these errors occurred in or know what resolution to take.

This month, we'll take a look at the typical errors a data entry form must be prepared to handle. We'll also examine a workaround for a fairly disastrous problem with the code generated by VFP's Referential Integrity (RI) Builder that under some conditions allows restricted updates or deletions to partially succeed, leaving orphaned child records.

Before we begin, however, I'd like to correct a statement I made last month. I mentioned that you can RETURN TO the routine containing the READ EVENTS for your application to prevent execution from returning to the method that caused an error. In testing, I found RETURN TO would give an error if I tried to return to an object method (such as RETURN TO oApp.Main), so I assumed VFP won't permit this. However, Darrel Miller pointed out that it will work if you leave off the object name (RETURN TO Main). Thus, you *can* have your READ EVENTS in the method of an object and still use this technique.

Handling Specific Errors

The error handling scheme we've looked at so far is generic: every error will cause the "Cancel, Continue, Retry, Quit" dialog to appear. This isn't appropriate for foreseeable errors; it should only be used for unforeseeable errors. Foreseeable errors should be handled in the Error method of objects that may cause them.

As an example of handling specific errors, we'll look the SFMaintForm class (in SFFORMS.VCX), a subclass of SFForm (our form base class defined in SFCTRLS.VCX) designed specifically for data entry forms. This is a good example of how an object has specific knowledge of the environment and foreseeable errors and how they should be handled.

Last month, we saw that the Error method of SFForm calls the HandleError method, which simply passes the error to the SFErrorMgr's ErrorHandler method. Thus, we'll override the behavior of HandleError in SFMaintForm to handle specific data-based errors. Here's the code for this method (constants used in this code, such as cnAERR_NUMBER, are defined in the include file for this class, LIBRARY.H):

```
local lnError, ;
  lcMethod, ;
  lnLine, ;
  llDisplay, ;
  lcReturn, ;
  loObject
with This
  lnError  = .aErrorInfo[.nLastError, cnAERR_NUMBER]
  lcMethod = .aErrorInfo[.nLastError, cnAERR_METHOD]
  lnLine   = .aErrorInfo[.nLastError, cnAERR_LINE]

  do case

* Handle table or database not found in the
* DataEnvironment.

    case (lnError = cnERR_TABLE_MOVED or ;
      lnError = cnERR_FILE_NOT_FOUND) and ;
```

```
      'DATAENVIRONMENT' $ upper(lcMethod)
      llDisplay = oError.lDisplayErrors
      oError.lDisplayErrors = .F.
      oError.ErrorHandler(lnError, lcMethod, lnLine)
      oError.lDisplayErrors = llDisplay
      messagebox(ccERR_TABLE_MOVED, MB_ICONSTOP, ;
        _screen.Caption)
      lcReturn = ccMSG_CLOSEFORM

* Handle table access denied in the DataEnvironment.

    case lnError = cnERR_ACCESS_DENIED and ;
      'DATAENVIRONMENT' $ upper(lcMethod)
      llDisplay = oError.lDisplayErrors
      oError.lDisplayErrors = .F.
      oError.ErrorHandler(lnError, lcMethod, lnLine)
      oError.lDisplayErrors = llDisplay
      messagebox(ccERR_ACCESS_DENIED, MB_ICONSTOP, ;
        _screen.Caption)
      lcReturn = ccMSG_CLOSEFORM

* Handle "table in use" in the DataEnvironment.

    case lnError = cnERR_TABLE_IN_USE and ;
      'DATAENVIRONMENT' $ upper(lcMethod)
      llDisplay = oError.lDisplayErrors
      oError.lDisplayErrors = .F.
      oError.ErrorHandler(lnError, lcMethod, lnLine)
      oError.lDisplayErrors = llDisplay
      messagebox(ccERR_TABLE_IN_USE, MB_ICONSTOP, ;
        _screen.Caption)
      lcReturn = ccMSG_CLOSEFORM

* Handle "DataEnvironment already unloaded" by not
* displaying anything.

    case lnError = cnERR_DE_UNLOADED
      lcReturn = ccMSG_CONTINUE

* Handle a trigger failed.

    case lnError = cnERR_TRIGGER_FAILED
      lcReturn = .ErrTriggerFailed()

* Handle a field rule failed.

    case lnError = cnERR_FIELD_RULE_FAILED
      loObject = .ErrFieldRuleFailed()
      if not isnull(loObject)
        .ActivateObjectPage(loObject)
        loObject.SetFocus()
      endif not isnull(loObject)
      lcReturn = ccMSG_CONTINUE

* Handle a table rule failed.

    case lnError = cnERR_TABLE_RULE_FAILED
      lcReturn = .ErrTableRuleFailed()

* Handle a primary/candidate index violation.

    case lnError = cnERR_DUPLKEY
      lcReturn = .ErrDuplicatekey()

* Handle the case where someone else has the record
* locked.

    case lnError = cnERR_RECINUSE
      lcReturn = .ErrRecordInUse()

* Handle the case where the record was modified by
```

```
* another user during a delete.

    case lnError = cnERR_RECMODIFIED and ;
      lcMethod = 'DeleteRecord'
      messagebox(ccERR_REC_MODIFIED, MB_ICONSTOP, ;
        _screen.Caption)
      .Refresh()
      lcReturn = ccMSG_CONTINUE

* Handle the case where the record was modified by
* another user during an edit.

    case lnError = cnERR_RECMODIFIED
      lcReturn = .ErrRecChangedByAnother()

* Otherwise use the default error handler.

    otherwise
      lcReturn = dodefault()
  endcase
endwith
return lcReturn
```

As you might expect, a routine handling specific errors will likely consist of a set of CASE statements handling all the foreseen errors and an OTHERWISE statement passing any unhandled errors to SFErrorMgr. In the case of SFMaintForm, we'll handle the following errors:

- DataEnvironment errors or trigger or field rule failed: we'll look at these errors in more detail in a moment.
- Table rule failed, primary/candidate index violation, or someone else has the record locked: other than designing a system that minimizes these types of problems (such as using system-assigned keys and using optimistic rather than pessimistic locking), there isn't much we can do about them; it's up to the user to correct the problem. So we'll just call custom methods that display an appropriate error message.
- The record was modified by another user when this user tried to delete it: we'll display a message to the user and refresh the form to show the other user's changes.
- The record was modified by another user when this user tried to edit it: we'll use conflict resolution code to resolve the changes made by each user (we won't look at this code here; feel free to examine the ErrRecChangedByAnother method yourself).

Of course, this isn't the entire range of errors that could be trapped here, but is a good representative sample.

DataEnvironment

Although the DataEnvironment (DE) has its own Error method, classes don't have a DE, so we'd have to manually put code into DataEnvironment.Error for every form we create. Instead, we'll handle DE errors in SFMaintForm's HandleError method. These include "table or database not found", "table access denied", or "table in use" errors (I'm sure you can think of others to trap as well). Generally, there isn't much we can do other than display an error message and close the form. However, we want some error handling services, so we call oError.ErrorHandler after setting the lDisplayErrors property to .F. This causes errors to be logged but not displayed. We'll display our own message before returning "closeform" to the Error method, which causes it to release the form.

One thing to watch out for is a "DataEnvironment already unloaded" error. This may occur as the form is being closed if the DE failed to load because of one of the previous errors, so we'll do nothing when we get one of those errors.

Triggers

If a trigger fails and the object causing the trigger to be called has code in its Error method, that method is fired rather than the ON ERROR routine set up by the trigger (the RIError procedure). This is a big

problem: RIError sets a public variable called pnError to a non-zero value, which is then used by other routines generated by VFP's RI Builder to indicate that the trigger has failed. Imagine the following situation: the user takes some action in a form, such as deleting a record, that causes a trigger to fail. Trigger failure causes the error handler to be called, but since the trigger was fired from an object with code in its Error method, that method gets called rather than the RIError routine in the stored procedures of the database. When the Error method is done, execution returns to the trigger code. However, since pnError hasn't changed from its original zero value, the trigger code doesn't know an error occurred, so it carries on. As a result, the trigger might just partially fail.

Here's a concrete example. CUSTOMER has a cascade delete rule into ORDERS while ORDERS has a restrict delete rule into ORDITEMS. The current CUSTOMER record has two related ORDERS records, the first of which has no related ORDITEMS records and the second of which has one related ORDITEMS record (the sample data at the Subscriber Downloads site has this exact setup). When you delete the CUSTOMER record, what do you think happens? You get an error that the trigger failed but you'll find that the CUSTOMER record and the first ORDERS record were deleted anyway. Only the second ORDERS record exists, and of course, it's now an orphan.

(In case you think this is an arcane example, it actually happened to me with a different database, which is how I discovered this problem in the first place.)

The solution is to have the error handler set pnError to a non-zero value; if you examine the code in the ErrTriggerFailed method of SFMaintForm, you'll see it does just that. However, that doesn't completely solve the problem: a bug in the RIDelete and RIUpdate procedures (which delete or update a child record) generated by the VFP RI Builder must also be fixed (see below for the code in RIDelete). These routines both set llRetVal (the return value) to .F. if pnError is non-zero, which tells other routines that the trigger failed. However, because the RIOpen routine (which is called to open child tables so they can be checked for records related to the parent record) opens tables in non-buffered mode, the next level of trigger (for example, checking into grandchild tables) isn't fired until the UNLOCK statement, which occurs *after* llRetVal is set. Thus, an error in trying to delete or update a grandchild record (which causes the trigger error message to appear) will not cause llRetVal to be set to .F., so this level of trigger doesn't fail even though it should. The solution is to move the llRetVal assignment statement after the UNLOCK command as shown below. Since RIDelete and RIUpdate are generic routines, you could copy the code for these routines in the stored procedures of the database, paste it at the end of the stored procedures (below the RI footer comment line) and make the changes there. This way, you don't have to make the same changes every time you regenerate the RI code.

```
procedure RIDELETE
local llRetVal
llRetVal=.t.
 IF (ISRLOCKED() and !deleted()) OR !RLOCK()
    llRetVal=.F.
  ELSE
    IF !deleted()
      DELETE
      IF CURSORGETPROP('BUFFERING') > 1
        =TABLEUPDATE()
      ENDIF
*** CUT THE FOLLOWING LINE ...
*      llRetVal=pnerror=0
    ENDIF not already deleted
  ENDIF
  UNLOCK RECORD (RECNO())
*** ... AND PASTE IT HERE
  llRetVal=pnerror=0
RETURN llRetVal
```

Field Rule Violation

A bug in the SYS(2018) and AERROR() functions in VFP 5 (including 5.0a) prevents the actual field name from being available when a field rule is violated. Instead, you get one of two strings: either the RuleText property for the field if it was filled in or the generic message "Field <field> validation rule is violated" if not.

So what? Well, what if you want to display a message like "Please enter a valid value for <field caption>" if there is no RuleText for the field? The problem is if you don't know which field rule was violated, how do you know which field to get the caption for? What if you want to set focus to the control bound to that field after displaying the error message, making it easier for the user to edit the value? What if you want to change the background color for that control, making it obvious which fields are in error?

To work around this bug (or "undocumented behavior" as 'Softies like to call it <g>), we need to do one of two things: if the string is "Field <field> validation rule is violated", we'll dig the field name out of the string. If that isn't the string, we have to find out which field in the database has the given string for its RuleText property. The SFMaintForm.ErrFieldRuleFailed method handles this behavior; here's the code:

```
local lcAlias, ;
  lcTable, ;
  lcMessage, ;
  lcField, ;
  lnSpace1, ;
  lnSpace2, ;
  lnI, ;
  lcText, ;
  loObject
with This
  lcAlias   = alias(.aErrorInfo[.nLastError, ;
    cnAERR_WORKAREA])
  lcTable   = cursorgetprop('SourceName', lcAlias)
  lcMessage = .aErrorInfo[.nLastError, cnAERR_MESSAGE]
  lcField   = ''
  if ccVFP_VALIDATION_MSG $ lcMessage
    lnSpace1  = at(' ', lcMessage) + 1
    lnSpace2  = at(' ', lcMessage, 2)
    lcField   = substr(lcMessage, lnSpace1, ;
      lnSpace2 - lnSpace1)
    lcMessage = ccERR_FIELD_RULE_FAILED
  else
    for lnI = 1 to fcount(lcAlias)
      lcText = dbgetprop(lcTable + '.' + ;
        field(lnI, lcAlias), 'Field', 'RuleText')
      if not empty(lcText) and ;
        evaluate(lcText) = lcMessage
        lcField = field(lnI)
        exit
      endif not empty(lcText) ...
    next lnI
  endif ccVFP_VALIDATION_MSG $ lcMessage
  lcField = lower(lcAlias + '.' + lcField)

* Display the error message and find the object whose
* ControlSource is the field.

  messagebox(lcMessage, MB_ICONSTOP, _screen.Caption)
  loObject = .FindControlSourceObject(lcField)
endwith
return loObject
```

After ErrFieldRuleFailed figures out which field has the problem, it calls the FindControlSourceObject method to find which object in the form is bound to that field. If such a object can be found, it returns a reference to the object so HandleError can set focus to it.

Examples

To see examples of how both specific and unforeseeable errors are handled in this error handling mechanism, run MYAPP.APP (available from the Subscriber Downloads site). Choose Error Form 1 from the File menu and click on the "This will cause an error" button. The Click method of this button has two errors in it, so if you choose Continue in the error dialog that appears for the first error, the second error will occur and the error dialog will come up again. If you choose Cancel instead, the second error won't occur but the application stays running and the form is still open. Choosing Quit exits the application cleanly, restoring the menu bar, closing all forms, and resetting the environment to the way it was before the

application was run. Retry, of course, causes the same error message to appear, since the problem hasn't been corrected.

To see how specific errors are handled, choose the Customer form from the File menu. Choose New from the File menu, enter "ALFKI" for the Customer ID, then choose Save from the File menu. You'll get an error that the Customer ID already exists; since that field is the primary key for the table and a record already exists with ALFKI in that field, a primary key violation error occurs and is handled as shown. To see the result of a field rule failure, enter "Test" for the Company; this field has a rule preventing that value from being stored. When you try to leave the Company field, you'll see the error message that results from the field rule failure. However, notice that focus moves to the next field without changing the value from "Test". Move to the City field and enter "Regina", then choose Save from the File menu. First you'll get the field rule error message again, and focus will be set to the Company field. Clear the field or enter something else, then choose Save again. This time, you'll get an error message caused by a table rule failure; the table has a rule that prevents "Regina" from being stored in the City field (after all, who'd want to live there? <g>). Choose Revert from the File menu to remove the newly added record.

To see how DE problems are handled, rename CUSTOMER.DBF to CUST.DBF in the Windows Explorer, then run MYAPP and open the Customer form. Notice that after you respond to the error message that appears, the form closes. Rename CUST.DBF back to CUSTOMER.DBF.

To see the trigger failure problem, exit the application, open the TESTDATA database, and use the CUST_ORDERS view and browse it. This view displays information from CUSTOMER, ORDERS, and ORDITEMS records. Notice the ALFKI customer has several orders but the first one (ORDER_ID = 10062) has no order items (LINE_NO is .NULL.). Run MYAPP, open the Customer form, choose Delete from the File menu, and select Yes when asked to confirm the deletion. Notice the trigger failure message that comes up; this is because of the restrict delete rule between ORDERS and ORDITEMS. However, notice that you get this message several times, and after responding to it for the last time, the ALFKI customer is gone. Exit the application, use CUSTOMERS, and notice that indeed ALFKI is gone. Now USE ORDERS ORDER CUST_ID and notice that lots of orders for customer ALFKI still exist; of course, they're all orphans.

To fix the problem, do the following:

- CLOSE ALL
- Unzip DATA.ZIP (we've messed up the data so we need to put it back the way it was).
- OPEN DATABASE TESTDATA
- MODIFY PROCEDURES
- Move the assignment to llRetVal in the RIDelete procedure to the line after the UNLOCK statement as described earlier. Save and close the code window.
- MODIFY PROJECT MYAPP
- Open the SFMaintForm class in the SFForms class library, open the code window for the ErrTriggerFailed method, go to the bottom of the code, and uncomment the assignment to pnError. Save and close the code window and the class.
- Rebuild MYAPP and run it.
- Open the Customer form, choose Delete from the File menu, and respond Yes when you're asked to confirm the deletion.

This time, you'll get a single trigger failure message and the ALFKI customer still exists. That bug is squashed!

Other things to check out:

- To see how SFErrorMgr logs errors, use ERRORLOG and browse it. Especially notice the MEMVARS memo; it contains the value every variable in each routine in the calling stack had at the time the error occurred.
- Change the value of the Developer entry in APPLIC.INI to Yes. This value is used by STARTUP.PRG to set the lDeveloper property of oError. Open the Windows Explorer and press F5 (this is necessary so VFP sees that the INI file has changed), then run MYAPP. Choose Error Form 1 from the File menu and click on the first button. Notice that a Debug option appears in the error dialog. This is really

useful for debugging an error while the application is still running. Of course, the Trace window displays the code in the ErrorHandler method of SFErrorMgr, since that's where the SET STEP ON command was executed. To return to the code that actually caused the error, you have to click on the Step Out button in the Debugger toolbar once for each routine in the error handling chain.

- The "This should cause an error but doesn't" button in Error Form 1 does just what it says: clicking on it does nothing. If you examine the Click method for this button, this might seem surprising, since there are two errors in the code. However, look at the Error method and notice that unlike other objects, it calls the Error method of its parent object immediately. This is a problem: the button is sitting in a page in a PageFrame, and since the page is a VFP base class page, it has no Error method code, which means that nothing happens when an error occurs. Like ON ERROR *, this is a good way to make your applications error message free. Not error free, just error message free <g>. This is why all base classes in SFCTRLS.VCX look up the containership hierarchy to find the first parent object with Error method code.

Miscellaneous Error Information

Here are a bunch of other things you should know about error handling in VFP.

- In VFP 5, the Error method of a bound control is called when the validation rule for the field the control is bound to is violated. This allows you to control what message is displayed and how. In VFP 3, these errors are untrappable; VFP displays the RuleText property for the field (or an ugly generic message if the RuleText isn't filled in) in a MESSAGEBOX() dialog that you have no control over. Yet another reason to upgrade to VFP 5 if possible.
- Even though variables defined as LOCAL are invisible except in the routine that defined them, the LIST MEMORY command can see all variables defined in all routines in the calling stack. This is a good thing, because it allows you to take a snapshot of all variables at the time the error occurred and log them to a file. The LogError method of SFErrorMgr shows how to do this.
- The LIST STATUS command isn't quite as helpful: it only sees things affected by the current data session (such as open tables and SET settings), so if the error handler is in the default session, it won't see things in the private data session of a form that caused the error. If this is important to you, you could switch to the data session of the form before using this command or you could instantiate an instance of SFErrorMgr into the oError property of the form so it's in the same data session as the form.
- For performance reasons, you might not want to call up the containership hierarchy one step at a time when an error occurs, so you could jump directly from an object to the form's Error method or even SFErrorMgr.ErrorHandler if desired. In my opinion, performance isn't a factor when an error occurs, so I keep the class design cleaner using the mechanism described in this session.
- The ERROR command is interesting: it causes an error to be triggered. It's handy if you want to treat certain types of "soft" errors as if they were VFP errors. For example, if FILE() returns .F. indicating a file is missing, you could use the ERROR command to force the error handler to fire so you get the usual error services (logging, display, etc.). I tend to do this only rarely; after all, why bother checking for a soft error if you're going to treat it like a hard error? Also, depending on where you use the ERROR command, you may not have accurate method and line number information, since they'll reflect where the ERROR command was used rather than where you actually detected the problem.
- If an error occurs in programmatic code called from an object method, the object's Error method is fired rather than ON ERROR. This means two different mechanisms may be used to handle an error in programmatic code, depending on how the program was called. Thus, error handling in programmatic code isn't as simple as it is with objects. This is another reason to move away from PRG libraries and move to object libraries.
- ON ERROR doesn't trap errors in reports and in the SKIP FOR clauses of a menu; these errors are untrappable, so make sure you've tested your reports and your menus under all SKIP FOR conditions. To confirm this, edit APPLIC.INI and set the NoSuchVariable entry to No. Open the Windows Explorer and press F5 (this is necessary so VFP sees that the INI file has changed), then run MYAPP. Click on the menu, and notice the VFP error that appears.

Summary

In the last two articles, we looked at an error handling scheme that uses the best of both worlds: local handling for most errors and global services for the rest. This scheme has been successfully used in several custom applications Stonefield has created for our clients, although we continue to refine it. I hope you find it useful in your applications. Please let me know of any enhancements you add to it or things you think need improvement.

Doug Hennig is a partner with Stonefield Systems Group Inc. in Regina, Saskatchewan, Canada. He is the author of Stonefield's add-on tools for FoxPro developers, including Stonefield Database Toolkit for Visual FoxPro and Stonefield Data Dictionary for FoxPro 2.x. He is also the author of "The Visual FoxPro Data Dictionary" in Pinnacle Publishing's "The Pros Talk Visual FoxPro" series. Doug has spoken at user groups and regional conferences all over North America, and spoke at the 1997 Microsoft FoxPro Developers Conference. He is a Microsoft Most Valuable Professional (MVP). CompuServe 75156,2326 or dhennig@stonefield.com.