Error Handling in Visual FoxPro

By Doug Hennig

Introduction

Like most things, error handling is more flexible but a lot more complex in Visual FoxPro than it was in FoxPro 2.x. While objects have Error methods to provide local error handling, how do you provide common, global error handling services to your application? How do you recover when an error occurs? This session looks at a proven strategy for implementing error handling in Visual FoxPro applications, starting from individual controls and working up to a global error object.

Error Handling Basics

There are several aspects involved in error handling: setting up the error handler, determining the cause of an error, informing the user what happened (and possibly logging it to a file for later analysis), and trying to resolve the problem (trying to execute the command again, returning to the statement following the one that caused the error, quitting the application, etc.).

Setting up the Error Handler

Setting up a global error handler in VFP hasn't changed from FoxPro 2.x: you still use the **`on error`** command. Here's an example:

```
on error do ERR_PROC with error(), sys(16), lineno()
```

These parameters tell the error handler the error number, the name of the routine, and the line number of the code executing when the error occurred. You can pass any parameters to the error handler you wish.

VFP has supplemented the global error handler approach with the ability to provide individual local handlers by implementing the Error event. Every object in the Visual FoxPro event model has an Error event. Of course, not every object will have an Error method. If this distinction isn't clear to you, remember that an event is an action triggered by something the user or the system does (such as a keystroke, a mouse click, or something that Visual FoxPro thinks is an error), while a method is the code that executes when the event occurs. The code for a method will also execute when a message is passed to an object telling it to execute that method. With many events, such as a mouse click, if the object doesn't have any code for the appropriate method, the event is ignored or default behavior is executed. However, when an error occurs, what happens depends on a number of things.

The Error method of an object will be called if it exists and an error occurs in a method of the object or in a non-object program (such as a PRG) called by the object. What happens if the object doesn't have an Error method? When I first started working with VFP, I assumed the Error method of the object's container (such as a form) would be called. However, that's not the case. Instead, if any object on the call stack has code in its Error

method, that method is called; if not, the **`on error`** routine (if there is one) is called. If there's no **`on error`** routine, Visual FoxPro does its own error handling (the infamous Cancel/Ignore dialog), and we consider that the application has crashed.

Determining the Cause of the Error

Several functions in FoxPro 2.x and VFP help determine the cause of an error, including **`error()`**, **`message()`**, **`lineno()`**, **`sys(16)`**, and **`sys(2018)`**. VFP also provides an **`aerror()`** function, which places information about the most recent error into an array. Although some of the information in the array can be obtained from other functions, for some types of errors (such as OLE, OBDC, and triggers), **`aerror()`** provides information not available elsewhere (such as which trigger caused the error).

Informing the User What Happened

Informing the user that an error occurred is fairly straightforward, and is often combined with allowing the user to determine how the problem should be resolved. The main concerns here are deciding what to tell the user and what choices to present. The message to display will vary with the type of error, and should be worded in a calm tone to prevent the user from panicking and doing a "three-finger salute". It could even provide information about how to resolve the problem. For example, something simple like the printer being off-line can be handled by asking the user to ensure paper is properly loaded in the printer, that it's turned on and connected to the computer, etc. The user can be given the choice of trying to print again or canceling the print job. If a user tries to edit a record that's locked by another user, you might tell the user that someone else is editing the record right now, and give them the choices of trying again or canceling the edit.

Given that there's over 600 possible errors in VFP, you're probably horrified thinking about coming up with meaningful messages for all those errors. Fortunately, if you look at the VFP help topic on error messages, you'll see that most of them fall into one of a few categories:

- "this'll never happen unless something is seriously hosed"
- "this wouldn't have occurred if the programmer wasn't drinking Tequila the night before writing this code"
- "you mean more than one user is going to use this system at a time?"
- errors that need to be handled individually

In the first category, your application really has no safe choice other than to shut down. Errors in the second category should be caught in testing, but if they aren't, the application should report and log the error, then shut down. Errors in the third category should also be caught in testing, but a simple "you can't do this right now" type of message should suffice if not. It's really only the fourth category of errors that you need to specifically address. Some examples of these types of errors are field or table validation rule failure, primary key failure (this will be minimized if you're using system-assigned or "surrogate" keys), trigger failure, and file not found (which you should usually handle by ensuring the file exists rather than letting the error handler catch it).

Before displaying the error message to the user, many developers like to log the error to a error log file. This has proven to be invaluable time and again, because users are frequently vague about the specifics when they report an error to you. The error log file can be a text file, but I prefer a table with fields for the date and time of the error (this can be a DateTime field in VFP), name of the user, error number and message, line number and code where the error occurred, and a memo field containing the current contents of memory variables.

Resolving the Error

Resolving an error can be complicated. The **`retry`** command tries to re-execute the command that caused the error, but with most errors, this doesn't help since the ability of the user to resolve the problem themselves before attempting to retry is limited. **`return`** continues execution from the line following the one that caused the error, but since the command that caused the error in effect did not execute, frequently a second error will occur because something didn't happen, such as a variable not being created (after all, if the command that caused the error could be bypassed without a problem, what's it doing there in the first place? <g>). **`cancel`** isn't a realistic option because it terminates the application, which may not have a chance to properly clean up after itself. Shutting down the application in a controlled manner *is* a valid option, since many errors result from a programmer or system error, and there's not much point in carrying on until the problem is resolved. As a developer, you'd also like an option to cancel the application and return to the Command window. The code doing that should clean up the environment as much as possible.

Another option is to use **`return to master`** or **`return to <program>`** to exit the program that caused the error and go back to the main program or some other specific program. You need to be careful with this option, since it might leave the system in a messy state: forms and windows will still exist, tables or cursors will still be open, etc. It's a good idea to clean such things up as much as possible before using the **`return to`** command. We'll look at this more closely later.

Designing an Error Handling Scheme

A global error handler and an object's Error method represent two opposite ends of a spectrum:

- The global handler is far removed from the source of the error while the Error method is part of the object that caused the error. Thus, the Error method should know a lot more about the environment it's in, the potential errors that could occur, and how to resolve them. For example, the CommonDialogs ActiveX control (which displays file, print, color, and printer dialogs) can cause an error if the user chooses Cancel. It'd be dumb to let the global error handler try to handle that error, since it would only see it as an OLE error of some kind and wouldn't know what to do about it. It makes more sense to put code into the Error method of the CommonDialogs control that knows how to handle a Cancel situation.

- The global error handler is called outside VFP's event handler using FoxPro's older "ON" event scheme (this scheme is still used by menus and on key labels). This

means you can't use object syntax like **`Thisform`**, and issues like private datasessions can complicate error resolution.

- The global error handler can efficiently consolidate error handling services (such as error logging and display) into *one* place. It's inefficient to try to handle most types of unanticipated errors (like a network connection going down) in the Error method of every object in your application.

Let's look at a design of an error handling scheme that incorporates the best of both worlds. We want to handle errors in the most efficient manner possible, yet still provide the ability of individual objects to handle their own specific errors. Here's the strategy we'll use:

- Like most kids, an object knows more about what's really going on than its parents do, so the Error method of an object will handle any errors it can. It will pass those it can't handle up the class hierarchy using **`dodefault()`**. Each subclass in the class hierarchy will do the same. If a subclass or instance of a class don't need to handle any specific errors, no code is placed in the Error method, causing the parent class code to automatically be used. Thus, SFDeepSubClassTextBox.Error calls SFSubClassTextBox.Error which calls SFTextBox.Error.

- The Error method of the topmost parent class for the object will handle any errors it can. It will pass those it can't handle to its container using **`This.Parent.Error`**.

- Because the container classes work the same as the control classes (they pass unhandled errors to their parent classes, and the topmost parent class passes errors to *their* container classes), the net effect is that we move up the class hierarchy then up the containership hierarchy.

- The Error method of the topmost parent class of the outermost container will handle any errors it can. It will pass those it can't handle to the global error handler.

This is a Chain of Responsibility design pattern: each object in the chain either handles the error or passes it on to the next object in the chain. In this multi-layered scheme, error handling gets less specific and more generic as you move from the object to the global error handler, allowing errors to be handled at the appropriate level. Figure 1 illustrates this strategy.

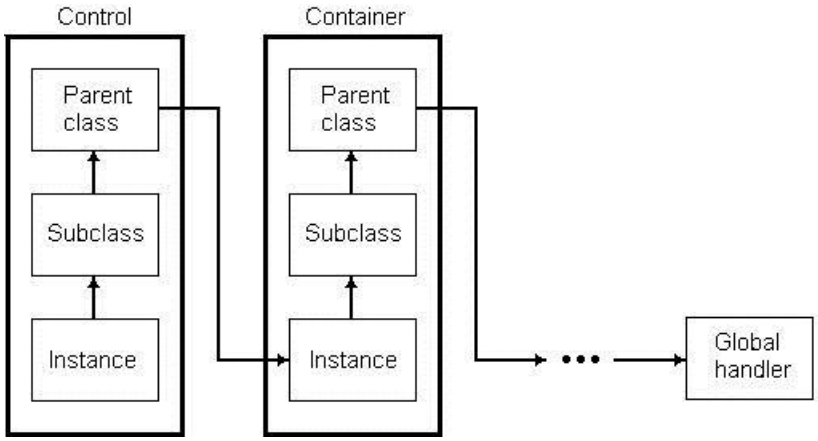

Figure 1. Error handling strategy.

I decided to make the global error handler an object that's instantiated into the global variable oError from the SFErrorMgr class at application startup. One of its methods (ErrorHandler) is called both directly by objects as described above and indirectly since it's also the `on error` handler. The error handling object should have a simple interface (meaning the programmatic, not user, interface), so SFErrorMgr accepts only the same parameters as the Error method of objects (the error number, method, and line number) and returns a string indicating what choice the user (or object) made for resolving the error. The error object is at the end of the chain of responsibility, so it doesn't know much about the environment it was called from (it might be several objects removed, in a different data session, etc.). As a result, it can't really "handle" (that is, resolve) much. Its purpose is to display a message to the user, log the error for post-mortem purposes, and either decide what action to take (under certain foreseeable conditions) or more likely ask the user what action to take. Thus, the error object should really only be used to handle foreseeable errors you haven't yet foreseen (once they occur, you'll change the object, class, or routine that caused the error to handle that case) and unforeseeable errors (true bugs or unforeseeable environmental conditions).

The global error handler may take a global resolution itself (bring up the VFP Debugger or shutting down the application) or may allow the object originating the error to have the final resolution. To allow the latter, each step in the error handling chain returns a resolution code to the previous level. For simplicity, I decided to return a string indicating what resolution is chosen: "retry" to retry the command that caused the error, "continue" to return to the line of code following the one that caused the error, or "closeform" to close the form the control is sitting on. Each object then takes the appropriate action based on the return message. Because the Error method of a container object may have been called from a member object or by an error that occurred in one of its own methods, the container must decide whether to pass the return message on or process it itself. We'll see the code for this later.

This scheme has one problem: controls sitting on VFP base class Page, Column, or other containers with no Error method code essentially have no error trapping because they call an empty method! The solution is to travel up the containership hierarchy until we find a parent that has code in its Error method. If we can't find such a parent, then display a generic error message (this isn't likely, since I base all forms on the SFForm class, which does have Error method code).

On thing to keep in mind is that the complete error handling chain must be the most bug-free part of your application, since the only fallback if an error occurs in any code called while in the error state is the VFP Cancel/Ignore dialog. Fortunately, since you can put most of the error handling code into your framework, once you've got it working, it won't be much of a concern (although flaky environmental conditions can still cause the error handler itself to fail). Don't bother trying to create an Error method in the error handling object: it doesn't get called when an error occurs in the error handler itself.

The Error Method

Let's look at the strategy in more detail. The starting point when an error occurs is the Error method of the object the error occurred in, so let's start there.

The code for the Error method of most classes in my application base classes, which are contained in SFCTRLS.VCX, is listed below (constants such as ccMSG_RETRY are defined in SFERRORS.H, which is included in SFCTRLS.H, the include file for each class). I say “most classes”, because top-level containers like forms and toolbars have to work a little differently. This is one of the few times I wish VFP supported multiple inheritance; as it is, you need to use the VB method of subclassing to put the same code into the Error method of all classes (select the code in the method to subclass, press Ctrl-C, put the cursor in new method, and press Ctrl-V to tell it to use the desired parent class code <g>).

```
lparameters tnError, ;
  tcMethod, ;
  tnLine
local laError[1], ;
  lcMethod, ;
  loParent, ;
  lcReturn, ;
  lcError

* Get information about the error.

aerror(laError)
lcMethod = This.Name + '.' + tcMethod

* If we're sitting on a form and that form has a
* FindErrorHandler method, call it to travel up the
* containership hierarchy until we find a parent that
* has code in its Error method. Also, if it has a
* SetError method, call it now so we don't lose the
* message information (which gets messed up by TYPE()).

if type('Thisform') = 'O'
  loParent = iif(pemstatus(Thisform, ;
    'FindErrorHandler', 5), ;
    Thisform.FindErrorHandler(This), .NULL.)
  if pemstatus(Thisform, 'SetError', 5)
    Thisform.SetError(lcMethod, tnLine, @laError)
  endif pemstatus(Thisform, 'SetError', 5)
else
  loParent = .NULL.
endif type('Thisform') = 'O'
do case

* We have a parent that can handle the error.

  case not isnull(loParent)
    lcReturn = loParent.Error(tnError, lcMethod, tnLine)

* We have an error handling object, so call its
* ErrorHandler() method.

  case type('oError.Name') = 'C'
    oError.SetError(lcMethod, tnLine, @laError)
    lcReturn = oError.ErrorHandler(tnError, lcMethod, ;
      tnLine)

* A global error handler is in effect, so let's pass the
* error on to it. Replace certain parameters passed to
* the error handler (the name of the program, the error
* number, the line number, the message, and SYS(2018))
* with the appropriate values.

  case not empty(on('ERROR'))
    lcError = strtran(strtran(strtran(strtran(strtran( ;
      strtran(upper(on('ERROR')), ;
      'SYS(16)',   '"' + lcMethod + '"'), ;
      'PROGRAM()', '"' + lcMethod + '"'), ;
      'ERROR()',   'tnError'), ;
      'LINENO()',  'tnLine'), ;
```

```
      'MESSAGE()', 'laError[2]'), ;
      'SYS(2018)', 'laError[3]')

* If the error handler is called with DO, macro expand
* it and assume the return value is "CONTINUE". If the
* error handler is called as a function (such as an
* object method), call it and grab the return value if
* there is one.

    if left(lcError, 3) = 'DO '
      &lcError
      lcReturn = ccMSG_CONTINUE
    else
      lcReturn = &lcError
    endif left(lcError, 3) = 'DO '

* Display a generic dialog box with an option to display
* the debugger (this should only occur in a test
* environment).

  otherwise
    lnChoice = messagebox('Error #: ' + ;
      ltrim(str(tnError)) + ccCR + ;
      'Message: ' + laError[2] + ccCR + ;
      'Line: ' + ltrim(str(tnLine)) + ccCR + ;
      'Code: ' + message(1) + ccCR + ;
      'Method: ' + tcMethod + ccCR + ;
      'Object: ' + This.Name + ccCR + ccCR + ;
      'Choose Yes to display the debugger, No to ' + ;
      'continue without the debugger, or Cancel to ' + ;
      'cancel execution', MB_YESNOCANCEL + MB_ICONSTOP, ;
      _VFP.Caption)
    do case
      case lnChoice = IDYES
        lcReturn = ccMSG_DEBUG
      case lnChoice = IDCANCEL
        lcReturn = ccMSG_CANCEL
    endcase
endcase

* Ensure the return message is acceptable. If not,
* assume "CONTINUE".

lcReturn = iif(vartype(lcReturn) <> 'C' or ;
  empty(lcReturn) or not lcReturn $ ccMSG_CONTINUE + ;
  ccMSG_RETRY + ccMSG_CANCEL + ccMSG_DEBUG, ;
  ccMSG_CONTINUE, lcReturn)

* Handle the return value.

do case

* It wasn't our error, so pass it back to the calling
* method.

  case '.' $ tcMethod
    return lcReturn

* Display the debugger.

  case lcReturn = ccMSG_DEBUG
    debug
    suspend

* Retry the command.

  case lcReturn = ccMSG_RETRY
    retry

* Cancel execution.

  case lcReturn = ccMSG_CANCEL
    cancel
```

```
* Go to the line of code following the error.

  otherwise
    return
endcase
```

The first thing this method does is use **aerror()** to capture the information about the error. This is important because the **type()** function, which will get used later in this routine, can mess up some of the error information, especially the name of a property or variable in a "Variable not found" error. This is discussed in further detail near the end of this document.

This code then checks to see if the form the control is sitting on has a FindErrorHandler method, and if so, calls it to locate the first parent of the control with code in its Error method (we won't bother looking at this code; you can check it out yourself in the supplied source code). This prevents the problem of error handling stopping on base class Page, Column, or other containers because they have no code in the Error method. It also calls the form's SetError method to save the previously gathered error information rather than letting the form do it because, as I mentioned earlier, any error information obtained after using **type()** may no longer be accurate.

If a parent prepared to handle the error is found, its Error method is called with the same parameters this Error method received, except the name of the object is added to tcMethod so our error handling services can know which object the error originated in. If a parent isn't found but a global error handler exists (we'll look at the global handler later), its SetError method is called to save the captured error information and then its ErrorHandler method is called. If an **on error** routine exists, we call it (first adjusting any parameters it might expect to match the values we have), either as a function or as a procedure. If we have nothing to pass the error on to, we'll use **messagebox()** to display an error message.

The return value from the error handler is then used to decide how to resolve the error. First, we must return the resolution message rather than handling it ourselves if the error is not our own. We'll check for this by looking for a period in the name of the method where the error occurred; since VFP passes just the method name if the error occurred in a method of the class but member objects pass the name of the object and the method, this provides a quick way to distinguish errors caused by the object itself or a member. If this isn't the case, this is our error, so we'll display the debugger, retry, cancel, or return.

Because they are the "top-level" containers (I don't use Formsets), the Error method for the SFForm and SFToolbar classes are different than other objects. This method uses the custom SetError method to populate some custom properties with information about the error; SetError doesn't do much if the custom lErrorInfoSaved property is .T., which is set within this method, to prevent the error information from being overwritten after it was populated by another class. Error then calls the HandleError method to handle the error. It then processes the return value, either taking an action itself (such as closing the form) or returning it to the object that called this method. Notice it doesn't return a value if the object is the DataEnvironment but instead handles those errors itself. You may wish to change this behavior. Also notice the use of **return to**; we'll discuss this in more detail later.

```
lparameters tnError, ;
  tcMethod, ;
  tnLine
```

```
local laError[1], ;
  lcReturn, ;
  lcReturnToOnCancel, ;
  lnPos, ;
  lcObject
with This

* Use SetError() and HandleError() to gather error
* information and handle it.

  aerror(laError)
  .SetError(tcMethod, tnLine, @laError)
  .lErrorInfoSaved = .F.
  lcReturn = .HandleError()

* Figure out where to go if the user chooses "Cancel".

  do case
    case left(sys(16, 1), ;
      at('.', sys(16, 1)) - 1) = 'PROCEDURE ' + ;
      upper(.Name)
      lcReturnToOnCancel = ''
    case type('oError.cReturnToOnCancel') = 'C'
      lcReturnToOnCancel = oError.cReturnToOnCancel
    case type('.oError.cReturnToOnCancel') = 'C'
      lcReturnToOnCancel = .oError.cReturnToOnCancel
    otherwise
      lcReturnToOnCancel = 'MASTER'
  endcase
endwith

* Handle the return value, depending on whether the
* error was "ours" or came from a member.

lnPos    = at('.', tcMethod)
lcObject = iif(lnPos = 0, '', ;
  upper(left(tcMethod, lnPos - 1)))
do case

* We're supposed to close the form, so do so and return
* to the master program (we'll just cancel if we *are*
* the master program).

  case lcReturn = ccMSG_CLOSEFORM
    This.Release()
    if empty(lcReturnToOnCancel)
      cancel
    else
      return to &lcReturnToOnCancel
    endif empty(lcReturnToOnCancel)

* This wasn't our error, so return the error resolution
* string.

  case lnPos > 0 and not ;
    (lcObject == upper(This.Name) or ;
    'DATAENVIRONMENT' $ upper(tcMethod))
    return lcReturn

* Display the debugger.

  case lcReturn = ccMSG_DEBUG
    debug
    suspend

* Retry.

  case lcReturn = ccMSG_RETRY
    retry

* If Cancel was chosen but the master program is this
* form, we'll just cancel.
```

```
  case lcReturn = ccMSG_CANCEL and ;
    empty(lcReturnToOnCancel)
    cancel

* Cancel was chosen, so return to the master program.

  case lcReturn = ccMSG_CANCEL
    return to &lcReturnToOnCancel

* Return to the routine in error to continue on.

  otherwise
    return
endcase
```

Here's the code for the SetError method:

```
lparameters tcMethod, ;
  tnLine, ;
  taError
local lnRows, ;
  lnCols, ;
  lnLast, ;
  lnError, ;
  lnRow, ;
  lnI
external array taError
with This

* If we've already been called, just update the method
* information.

  if .lErrorInfoSaved
    .aErrorInfo[.nLastError, cnAERR_METHOD] = tcMethod
  else

* Flag that an error occurred.

    .lErrorOccurred  = .T.
    .lErrorInfoSaved = .T.
    lnRows = alen(taError, 1)
    lnCols = alen(taError, 2)
    lnLast = iif(empty(.aErrorInfo[1, 1]), 0, ;
      alen(.aErrorInfo, 1))
    dimension .aErrorInfo[lnLast + lnRows, cnAERR_MAX]

* For each row in the error array, put each column into
* our array.

    for lnError = 1 to lnRows
      lnRow = lnLast + lnError
      for lnI = 1 to lnCols
        .aErrorInfo[lnRow, lnI] = taError[lnError, lnI]
      next lnI

* Add some additional information to the current row in
* our array.

      .aErrorInfo[lnRow, cnAERR_METHOD]   = tcMethod
      .aErrorInfo[lnRow, cnAERR_LINE]     = tnLine
      .aErrorInfo[lnRow, cnAERR_SOURCE]   = ;
        iif(message(1) = .aErrorInfo[lnRow, ;
        cnAERR_MESSAGE], '', message(1))
      .aErrorInfo[lnRow, cnAERR_DATETIME] = datetime()
    next lnError
    .nLastError = alen(.aErrorInfo, 1)
  endif not .lErrorInfoSaved
endwith
```

Here's the code for the HandleError method:

```
local lnError, ;
  lcMethod, ;
  lnLine, ;
```

```
  lcErrorMessage, ;
  lcErrorInfo, ;
  lcSource, ;
  loError, ;
  lcMessage, ;
  lcReturn, ;
  lcError
with This
  lnError        = .aErrorInfo[.nLastError, ;
    cnAERR_NUMBER]
  lcMethod       = .Name + '.' + ;
    .aErrorInfo[.nLastError, cnAERR_METHOD]
  lnLine         = .aErrorInfo[.nLastError, ;
    cnAERR_LINE]
  lcErrorMessage = .aErrorInfo[.nLastError, ;
    cnAERR_MESSAGE]
  lcErrorInfo    = .aErrorInfo[.nLastError, ;
    cnAERR_OBJECT]
  lcSource       = .aErrorInfo[.nLastError, ;
    cnAERR_SOURCE]

* Get a reference to our error handling object if there
* is one. It could either be a member of the form or a
* global object.

  do case
    case vartype(.oError) = 'O'
      loError = .oError
    case type('oError.Name') = 'C'
      loError = oError
    otherwise
      loError = .NULL.
  endcase
  lcMessage = ccMSG_ERROR_NUM + ccTAB + ;
    ltrim(str(lnError)) + ccCR + ccMSG_MESSAGE + ;
    ccTAB + lcErrorMessage + ccCR + ;
    iif(empty(lcSource), '', ccMSG_CODE + ccTAB + ;
    lcSource + ccCR) + iif(lnLine = 0, '', ;
    ccMSG_LINE_NUM + ccTAB + ltrim(str(lnLine)) + ;
    ccCR) + ccMSG_METHOD + ccTAB + lcMethod
  do case

* If the error is "cannot set focus during valid" or
* "DataEnvironment already unloaded", we'll let it go.

    case lnError = cnERR_CANT_SET_FOCUS or ;
      lnError = cnERR_DE_UNLOADED
      lcReturn = ccMSG_CONTINUE

* We have an error handling object, so call its
* ErrorHandler() method.

    case not isnull(loError)
      lcReturn = loError.ErrorHandler(lnError, ;
        lcMethod, lnLine)

* A global error handler is in effect, so let's pass the
* error on to it. Replace certain parameters passed to
* the error handler (the name of the program, the error
* number, the line number, the message, and SYS(2018))
* with the appropriate values.

    case not empty(on('ERROR'))
      lcError = strtran(strtran(strtran(strtran( ;
        strtran(strtran(upper(on('ERROR')), ;
        'SYS(16)',   '"' + lcMethod + '"'), ;
        'PROGRAM()', '"' + lcMethod + '"'), ;
        'ERROR()',   'lnError'), ;
        'LINENO()',  'lnLine'), ;
        'MESSAGE()', 'lcErrorMessage'), ;
        'SYS(2018)', 'lcErrorInfo')

* If the error handler is called with DO, macro expand
```

```
* it and assume the return value is "CONTINUE". If the
* error handler is called as a function (such as an
* object method), call it and grab the return value if
* there is one.

      if left(lcError, 3) = 'DO '
        &lcError
        lcReturn = ccMSG_CONTINUE
      else
        lcReturn = &lcError
      endif left(lcError, 3) = 'DO '

* We don't have an error handling object, so display a
* dialog box.

    otherwise
      lnChoice = messagebox('Error #: ' + ;
        ltrim(str(lnError)) + ccCR + ;
        'Message: ' + lcErrorMessage + ccCR + ;
        'Line: ' + ltrim(str(lnLine)) + ccCR + ;
        'Code: ' + lcSource + ccCR + ;
        'Method: ' + lcMethod + ccCR + ;
        'Object: ' + .Name + ccCR + ccCR + ;
        'Choose Yes to display the debugger, ' + ;
        'No to continue without the debugger, or ' + ;
        'Cancel to cancel execution', ;
        MB_YESNOCANCEL + MB_ICONSTOP, _VFP.Caption)
      lcReturn = ccMSG_CONTINUE
      do case
        case lnChoice = IDYES
          lcReturn = ccMSG_DEBUG
        case lnChoice = IDCANCEL
          lcReturn = ccMSG_CANCEL
      endcase
  endcase
endwith
lcReturn = iif(vartype(lcReturn) <> 'C' or ;
  empty(lcReturn) or ;
  not upper(lcReturn) $ upper(ccMSG_CONTINUE + ;
  ccMSG_RETRY + ccMSG_CANCEL + ccMSG_CLOSEFORM + ;
  ccMSG_DEBUG), ccMSG_CONTINUE, lcReturn)
return lcReturn
```

HandleError tries to pass the error to a global error handling object, referenced either through a global oError variable or through an oError property of the form. This scheme allows you to have a customized version of the global error handler associated with a specific form if desired. If an **`on error`** routine exists, we call it (first adjusting any parameters it might expect to match the values we have), either as a function or as a procedure. If no global error handler can be found, **`messagebox()`** is used to display an error message. The return value from the error handler is then passed back to the Error method.

Global Error Handler

SFErrorMgr is a non-visual class based on SFCustom. It's contained in SFMGRS.VCX and uses the SFERRORMGR.H include file for the definitions of several constants. It's instantiated into the global variable oError at application startup (see SYSMAIN.PRG). We won't look at all the code for this class, only those methods which help illustrate the overall scheme of error handling services. Feel free to examine any other methods yourself.

The Init method accepts three parameters: the title to use for the dialog displayed when an error occurs (stored in the cTitle property), a flag indicating whether Init should save the current **`on error`** handler and change it to its ErrorHandler method, and the name of the

object the class is being instantiated into (this is needed for the **`on error`** command, because we can't use **`This`**).

As is usually the case, the Destroy method cleans up things the class has changed; in this case, it resets VFP's error handler to the one that was in effect before the object was instantiated.

The ErrorHandler method is called both directly by objects as the last object in the chain of responsibility and indirectly since it's also the **`on error`** handler. Here's the code for this method:

```
lparameters tnError, ;
  tcMethod, ;
  tnLine
local lcCurrTalk, ;
  laError[1], ;
  lcChoice, ;
  lcProgram
with This

* Ensure TALK is off.

  if set('TALK') = 'ON'
    set talk off
    lcCurrTalk = 'ON'
  else
    lcCurrTalk = 'OFF'
  endif set('TALK') = 'ON'

* First, save the error information.

  aerror(laError)
  .SetError(tcMethod, tnLine, @laError)
  .lErrorInfoSaved = .F.

* If errors aren't being suppressed, display the error
* and get the user's choice of action.

  lcChoice = ccMSG_CONTINUE
  if not .lSuppressErrors

* Log the error if necessary.

    if .lLogErrors
      .LogError()
    endif .lLogErrors

* Display the error and get the user's choice if
* desired.

    if .lDisplayErrors
      lcChoice = .DisplayError()
      do case

* Cancel or Quit in development environment: remove any
* WAIT window, revert all open cursors and issue a CLEAR
* EVENTS (in the case of Quit), and then return to the
* top-level program.

        case lcChoice = ccMSG_CANCEL or ;
          (lcChoice = ccMSG_QUIT and version(2) <> 0)
          wait clear
          if lcChoice = ccMSG_QUIT
            .lQuit = .T.
            .RevertAllTables()
            clear events
          endif lcChoice = ccMSG_QUIT
          lcProgram = .cReturnToOnCancel
          return to &lcProgram
```

```
* Retry programmatic code: we must do the retry here,
* since nothing will receive the RETRY message (as is
* the case with an object).

        case lcChoice = ccMSG_RETRY
          lcMethod = upper(tcMethod)
          if at('.', lcMethod) = 0 or ;
            inlist(right(lcMethod, 4), '.FXP', '.PRG', ;
            '.MPR', '.MPX')
            if lcCurrTalk = 'ON'
              set talk on
            endif lcCurrTalk = 'ON'
            retry
          endif at('.', lcMethod) = 0 ...

* Quit: revert all open cursors, then quit.

        case lcChoice = ccMSG_QUIT
          .lQuit = .T.
          .RevertAllTables()
          on shutdown
          quit
      endcase
    endif .lDisplayErrors
  endif not .lSuppressErrors

* Restore TALK.

  if lcCurrTalk = 'ON'
    set talk on
  endif lcCurrTalk = 'ON'
endwith
return lcChoice
```

When an error occurs, three parameters are passed to ErrorHandler: the error number, the name of the routine in which the error occurred, and the line number where the error occurred. Like SFForm.Error, ErrorHandler uses the SetError method to set the lErrorOccurred property to .T. and put information about the error into the aErrorInfo property; it only does this if the error information hasn't already been saved. If the lSuppressErrors property is .T., the error isn't logged and no error message is displayed (this is used when you want an error to be trapped but not logged or displayed to the user). Otherwise, the LogError method is called to log the error and the DisplayError method is used to display a message about the error and get the user's choice about what action to take. The choices are:

- Debug: this option, which is only available if the lShowDebug property is .T. and we're running a development version of VFP, brings up the Trace and Debug windows. lShowDebug should be set to .T. only for developers (this can be looked up in a user table or the Windows Registry).

- Continue: returns to the command following the one that caused the error.

- Retry: retries the command. This gets a little tricky: if ErrorHandler was explicitly called (that is, from the Error method of an object), it can't just issue the **`retry`** command, since that would simply return control to the method which called it, rather than the method which caused the error. In that case, we'll just return the message "retry". However, if the error occurred in programmatic code (a PRG or MPR), ErrorHandler got called as the **`on error`** routine, so just returning this message won't work. In this case, ErrorHandler must directly issue the **`retry`** command itself.

- Cancel: a frequent question about error handling is: how do you prevent the rest of the code in the method or program that caused the error from executing? You can't use

cancel, since that cancels the entire application. **return** returns to the same method so that doesn't help either. The solution is to return to the routine containing your application's **read events** statement (which is normally where you're sitting when method code isn't executing). Since this routine may not be the first program in the application, we'll return to the program specified in the cReturnToOnCancel property rather than **return to master**. When you instantiate SFErrorMgr, you can set this property appropriately (you can set it to "MASTER" if the first program in the application contains the **read events**). If your **read events** statement is in a method of an object, leave off the object name; for example, if your **read events** statement is in oApp.EventHandler, put "EventHander" into the cReturnToOnCancel property.

- Quit: quits the application. We have different needs here depending on whether we're running a development copy of VFP or not. It'd be a pain if you had to restart VFP every time you got an error in development mode, so in that case, this option should **clear events** and return to the main program so the application can shut down in an orderly manner and then return to the Command window. If this is a runtime version of VFP, we'll just clean up and quit. In both cases, we'll use a custom RevertTables method to perform a **tablerevert(.T.)** on all cursors in all datasessions so we don't get additional errors (such as the infamous "uncommitted changes" error) on the way out.

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, but 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 at look the SFMaintForm class (in SFFORMS.VCX), a subclass of SFForm 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.

Since the Error method of SFForm calls the HandleError method, which simply passes the error to the SFErrorMgr's ErrorHandler method, we'll override the behavior of HandleError in SFMaintForm to handle specific data-based errors. Here's the code for HandleError:

```
local lnError, ;
  lcMethod, ;
  lcReturn, ;
  loObject
with This

* Get the error number and method.

  lnError  = .aErrorInfo[.nLastError, cnAERR_NUMBER]
  lcMethod = upper(.aErrorInfo[.nLastError, ;
    cnAERR_METHOD])
  do case

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

    case lnError = cnERR_DE_UNLOADED
      lcReturn = ccMSG_CONTINUE
```

```
* Handle a problem in the DataEnvironment.

    case 'DATAENVIRONMENT' $ lcMethod
      lcReturn = .ErrDataEnvironment(lnError)

* Handle a trigger failed.

    case lnError = cnERR_TRIGGER_FAILED
      lcReturn = .ErrTriggerFailed()

* Handle a field rule failed by calling
* ErrFieldRuleFailed(). If it returns an object
* reference, we'll set focus to that object so the
* user can correct the problem.

    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_OK + MB_ICONSTOP, _VFP.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 we 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 we 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 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. Interestingly, the DE is the one object that automatically calls its form's Error method if its own Error method has no code. Thus, we'll handle DE errors in the SFMaintForm's ErrDataEnvironment method. These include "table or database not found", "table access denied", "table in use", and "primary key invalid" 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.

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

Triggers

If a trigger fails, the Error method of the object causing the trigger to be called is fired, not the `on error` routine set up by the trigger (the RIError procedure). This is a big problem: RIError sets the public variable pnError to a non-zero value, which is then used by other routines to know 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. 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 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
```

As an aside, you can get around this and other bugs in the code generated by VFP's RI Builder if you use the RI code written by Steve Sawyer and published in the book he and Jim Booth wrote called "Effective Techniques for Application Development With Visual FoxPro 6.0", published by Hentzenwerke Publishing (www.hentzenwerke.com).

Field Rule Violation

A bug in the **`sys(2018)`** and **`aerror()`** functions in VFP 5 and 6 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 that 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>) in VFP 5.0 (not 5.0a), we need to do one of two things: if the string is "Field <field> validation rule is violated", we'll dig the field 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. Fortunately, in VFP 5.0a, an undocumented feature of **`aerror()`** is that element 5 of the array it creates contains the field number, from which we can determine the field name.

The SFMaintForm.ErrFieldRuleFailed method handles this behavior; here's the code:

```
local loCurrObject, ;
  lnWorkArea, ;
  lcAlias, ;
  lcTable, ;
  lcField, ;
  loObject, ;
  lcMessage

* If the field rule was checked and failed because the
* user clicked on a button with the Cancel property set
* to .T. or if the button has an lCancel property (which
* is part of the SFCommandButton base class) and it's
* .T., don't bother doing anything else.

loCurrObject = sys(1270)
if lastkey() = 27 or ;
  (type('loCurrObject.lCancel') = 'L' and ;
  loCurrObject.lCancel)
  return .NULL.
endif lastkey() = 27 ...
with This

* Figure out which field failed and find the object
* whose ControlSource is the field.

  lnWorkArea = .aErrorInfo[.nLastError, cnAERR_WORKAREA]
  if type('lnWorkArea') = 'N' and ;
    between(lnWorkArea, 1, 32767)
    lcAlias  = alias(lnWorkArea)
    lcTable  = cursorgetprop('SourceName', lcAlias)
    lcField  = lower(lcAlias + '.' + ;
      field(.aErrorInfo[.nLastError, cnAERR_TRIGGER], ;
      lcAlias))
    loObject = .FindControlSourceObject(lcField)
  else
    loObject = .NULL.
  endif type('lnWorkArea') = 'N' ...

* Display the error message

  lcMessage = .aErrorInfo[.nLastError, cnAERR_MESSAGE]
  messagebox(lcMessage, MB_OK + MB_ICONSTOP, ;
    _VFP.Caption)

* Set the lFieldRuleFailed flag to .T.

  .lFieldRuleFailed = .T.
```

```
endwith
return loObject
```

One of the first things ErrFieldRuleFailed does is check if user clicked on a “cancel” button in the form, which tries to set focus to that button and thus fires the field validation rule for the field bound to the current control. Since it’d be dumb to display an error message under these conditions, ErrFieldRuleFailed simply returns without doing anything else.

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.

ErrFieldRuleFailed also sets a form property called lFieldRuleFailed to .T. This can be used by the Valid method of a control to prevent the control from losing focus when a field rule fails. The reason this is needed is because the field validation rule violation occurs before the Valid of the control fires. After the error has been handled, the Valid method fires and normally returns .T., allowing the control to lose focus even though it contains a bad value. Here’s some code from the Valid method of SFTextBox that prevents this:

```
if type('Thisform.lFieldRuleFailed') = 'L' and ;
  Thisform.lFieldRuleFailed
  Thisform.lFieldRuleFailed = .F.
  return 0
endif type('Thisform.lFieldRuleFailed') = 'L' ...
```

Examples

To see examples of how both specific and unforeseeable errors are handled in this error handling mechanism, run MYAPP.APP from the sample files provided. 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. Blank out the Company field. Move to the City field and enter “Regina”, then choose Save from the File menu. 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 SFMaintForm avoids giving a field rule failure message when a "cancel" button is pressed, enter "Test" for the Company but click on the "Revert" button before trying to exit the field.

To see how DE problems are handled, exit the application, 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 Error method of the object that caused the error, since that's where the DEBUG 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.

- 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.

- Prior to VFP 6, VFP Automation servers couldn't properly return an error condition to their callers. That's changed with the addition of the **`comreturnerror()`** function. This function populates the COM exception structure with information about the error that occurred. It accepts two parameters: cExceptionSource, which is the name of the server, and cExceptionText, any message you want to return. The Error Form 1 form in MYAPP.APP demonstrates this by calling the Test() method of an Automation server called MyServer (the MyServer project is included with the sample files).

- As I mentioned earlier, if **`type()`** is used anywhere in the error handling chain, you may not always get the correct error message. The reason is that **`type()`** uses some of VFP's error handling itself, and as a result, the contents of **`message()`** can be overwritten if the variable or property checked with **`type()`** doesn't exist. This explains "variable not found" messages your error hander may sometimes report when the error was in fact caused by something else. See the code in the Error method of each class for a way to handle this.

- In VFP 5 and later, 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.

- 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.

- The previous point has an interesting side effect: if your **`read events`** statement is in a method of a global object (such as an application object), the Error method of that object in effect becomes a global error handler since the object is always in the call stack. You could do away with **`on error`** completely in this case, since it's only needed to handle errors in programmatic code not called from objects.

- You can't normally trap the case where the user enters an invalid date into a TextBox; VFP displays "Invalid Date" itself and doesn't fire the Valid method of the control. You can turn off the "Invalid Date" message with **`set notify off`**, but if you want the Valid method of the control to fire (for example, to give a different message or bring up a calendar), you'll have to set the StrictDateEntry property of the control to 0-Loose. One caveat: if the user enters an invalid date, it'll get blanked, so if you want to redisplay the former value, you'll have to save it in the GotFocus method of the control and restore it in the Valid.

- **`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.

Conclusion

In this session, 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 applications, 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.

Acknowledgements

I would like to acknowledge the following people who directly or indirectly helped with the information in this session: Philip Kelley, Darrel Miller, Lisa Slater Nicholls, Jeff Pace, Mac Rubel, Brad Schulz, and Paul Slate.

Doug Hennig
Partner
Stonefield Systems Group Inc.
1112 Winnipeg Street, Suite 200
Regina, SK Canada S4R 1J6
Phone: (306) 586-3341
Fax: (306) 586-5080
Email: dhennig@stonefield.com
World Wide Web: www.stonefield.com

Biography

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 and Stonefield Query. 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 the 1997 and 1998 Microsoft FoxPro Developers Conferences (DevCon) as well as user groups and regional conferences all over North America. He is a Microsoft Most Valuable Professional (MVP).