Fix Problems Fast with Advanced Error Handling and Instrumentation Techniques

Doug Hennig
Stonefield Software Inc.
Email: dhennig@stonefield.com
Corporate Web site: www.stonefieldquery.com
Personal Web site : www.DougHennig.com
Blog: DougHennig.BlogSpot.com
Twitter: DougHennig

Your customer calls to report that their application crashes. Where do you start to figure out what's causing the problem? This document looks at techniques for troubleshooting application problems, including advanced error handling to provide complete state information and instrumenting your applications to determine exactly what steps led up to the crash.

Introduction

My company's main product is Stonefield Query. Stonefield Query is unusual for a VFP application: it doesn't have a fixed database structure that it works with or even a fixed database engine. It may have to work with a SQL Server accounting database, a MySQL customer relationship management database, or a VFP medical billing database. When a problem occurs, we sometimes ask the customer to send us a copy of their Stonefield Query configuration files (including a data dictionary) and application database so we can reproduce the problem, but that isn't always possible:

- We may not have a license for the database engine they're using.
- The database may be too large to send to us in a timely manner.
- The database may contain sensitive information protected by government laws or company policies.

So, many times, all we have to go on is what we can log at the time the error occurred. As a result, we've had to become very good at generating detailed log information and analyzing that information to determine and fix the problem. Thanks to these techniques, there are very few problems we can't track down quickly.

This document discusses the code we use for error handling and instrumentation as well as techniques for quickly tracking down and solving problems. Feel free to use this code as is or adapt it to your own applications as you see fit.

Error handling

Error handling 101

As you probably know, when an error occurs, it can be handled in one of four ways:

- Not at all; that is, none of the following mechanisms are used. This is a bad thing, because the dialog displayed to the user is scary and really doesn't provide any options that work.
- TRY structure: if an error occurs within a TRY, the code in the CATCH statement is executed. This isn't directly relevant to this discussion, because you anticipated the error and handled it yourself. However, TRY structures do throw a wrinkle into the generic error handling mechanism as I'll discuss later.
- Error method: if an error occurs in the method of an object that has code in its Error method (or one of its ancestors has code there), that method is called.
- ON ERROR: the command specified in the ON ERROR statement executes if the previous two mechanisms aren't applicable when an error occurs, such as an error in a PRG.

A solid application needs to use at least the latter mechanism, although a combination of all three works the best. TRY doesn't use any support code so I won't discuss it here. All of my

base classes (subclasses of VFP base classes, all located in SFCtrls.vcx and starting with "SF") have code in the Error method, so let's start there.

The error handling chain

Listing 1 shows the code in the Error method of my base classes. There's a lot of code there, but it's well-commented so should be easy to follow.

Listing 1. The Error method of my base classes.

```
lparameters tnError, ;
   tcMethod, ;
   tnLine
local lnError, ;
   lcMethod, ;
   lnLine, ;
   lcSource, ;
   laError[1], ;
   lcName, ;
   lcOrigMethod, ;
   loParent, ;
   lcReturn, ;
   lcError, ;
   lcMessage, ;
   lnChoice

* Use AERROR() to get information about the error. If we have an Exception
* object in oException, get information about the error from it.

lnError  = tnError
lcMethod = tcMethod
lnLine   = tnLine
lcSource = message(1)
aerror(laError)
with This
   if vartype(.oException) = 'O'
      lnError  = .oException.ErrorNo
      lcMethod = .oException.Procedure
      lnLine   = .oException.LineNo
      lcSource = .oException.LineContents
      laError[cnAERR_NUMBER]  = .oException.ErrorNo
      laError[cnAERR_MESSAGE] = .oException.Message
      laError[cnAERR_OBJECT]  = .oException.Details
      .oException = .NULL.
   endif vartype(.oException) = 'O'
endwith

* Determine which method of which object the error occurred in. If the error
* occurred in a child object, the method may already have our name on it, so
* handle that.

lcName   = upper(This.Name) + '.'
lcMethod = upper(tcMethod)
if lcMethod = lcName or '.' + lcName $ lcMethod
```

```
   lcOrigMethod = substr(tcMethod, rat('.', tcMethod) + 1)
else
   lcOrigMethod = tcMethod
endif lcMethod = lcName ...
lcMethod = This.Name + '.' + lcOrigMethod

* 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, lnLine, lcSource, @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(lnError, lcMethod, lnLine)

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

   case type('oError.Name') = 'C' and pemstatus(oError, 'ErrorHandler', 5)
      if pemstatus(oError, 'SetError', 5)
         oError.SetError(lcMethod, lnLine, lcSource, @laError)
      endif pemstatus(oError, 'SetError', 5)
      lcReturn = oError.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 = upper(on('ERROR'))
      lcError = strtran(lcError, 'SYS(16)',   '"' + lcMethod + '"')
      lcError = strtran(lcError, 'PROGRAM()', '"' + lcMethod + '"')
      lcError = strtran(lcError, ',ERROR()',  ',lnError')
      lcError = strtran(lcError, ' ERROR()',  ' lnError')
      lcError = strtran(lcError, 'LINENO()',  'lnLine')
      lcError = strtran(lcError, 'MESSAGE()', 'laError[2]')
      lcError = strtran(lcError, '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 ' or '=' $ lcError
```

```
            &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
         lcSource  = message(1)
         lcMessage = ccMSG_ERROR_NUM + ' ' + transform(lnError) + ccCR + ;
            ccMSG_MESSAGE + ' ' + laError[cnAERR_MESSAGE] + ccCR + ;
            iif(empty(lcSource), '', ccMSG_CODE + ' ' + lcSource + ;
            ccCR) + iif(lnLine = 0, '', ccMSG_LINE_NUM + ' ' + ;
            transform(lnLine) + ccCR) + ccMSG_METHOD + ' ' + lcMethod
         if version(2) = 0
            lnChoice = messagebox(lcMessage + ccCR + ccCR + ;
               'Choose OK to continue or Cancel to cancel execution', ;
               MB_OKCANCEL + MB_ICONSTOP, _VFP.Caption)
         else
            lnChoice = messagebox(lcMessage + 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)
         endif version(2) = 0
         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 '.' $ lcOrigMethod
      return lcReturn

* Display the debugger.

   case lcReturn = ccMSG_DEBUG
      debug
      if wexist('Visual FoxPro Debugger')
         keyboard '{SHIFT+F7}' plain
```

```
      endif wexist('Visual FoxPro Debugger')
      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
```

Here are the most important features of this code:

- If the control is sitting on a form (even in a container on a form) and that form has a FindErrorHandler method (which my SFForm base class does), that method is called to get the object to which the error should be passed by calling its Error method. This provides a Chain of Responsibility error handling mechanism described in my *Error Handling in Visual FoxPro* white paper listed in the References section of this document. This mechanism allows errors to bubble up the inheritance and containership chain until someone decides to specifically handle it or one of the other handlers discussed next is used.
- If nothing was found to pass the error to but a global object named oError exists, the ErrorHandler method of that object is called.
- If an ON ERROR statement is in effect, the procedure or function specified in that statement is executed.
- If the above conditions weren't selected, an error message is displayed using MESSAGEBOX(). Thus the error handling mechanism works in all environments, whether there's a global error handler or not.
- A value may have been returned from the handler indicating what to do next: return to the offending code, RETRY, CANCEL, or display the debugger. Note the KEYBOARD statement for the latter option; this causes execution to jump out of the current method so the debugger shows the line of code after the one causing the error in the appropriate method rather than the Error method.

Now let's look at SFErrorMgr, my generic error handling class.

SFErrorMgr

Early in application startup, my startup code does the following:

```
oError = newobject('SFErrorMgr', 'SFErrorMgr.vcx', '', lcAppName, .T., 'oError')
```

lcAppName contains the name of the application, which is used as the title for error message dialogs, .T. is passed so ON ERROR is set up, and the last parameter tells the Init method that the object reference is stored in a variable named "oError," which is a global variable in the application. If .T. is passed as the second parameter, the Init method of SFErrorMgr points ON ERROR to its own ErrorHandler method (that's why the variable name had to be passed):

```
lcError = lcErrorObj + '.ErrorHandler(error(), sys(16), lineno())'
on error &lcError
```

SFErrorMgr has several properties that control its behavior; these properties are shown in **Table 1**. Some of these properties are discussed in more detail in this document.

Table 1. SFErrorMgr's properties control its behavior.

Property	Description
aErrorInfo	An array containing information about all of the errors that have occurred since the application was started. nLastError is the index into the array for the most recent error.
cAppName	The name of the application; this is used in the subject of an email sent to developers.
cDefaultAction	The default error recovery action to take.
cErrorLogFile	The name and path of the file to log errors to: a table if lLogToTable is .T. or a text file if not. Defaults to "Errorlog.dbf."
cErrorMessage	The text to display in the error dialog presented to the user. Defaults to "An unexpected error has occurred."
cMessageClass	The class to use for an error dialog.
cMessageLibrary	The class library containing the class specified in cMessageClass.
cReturnToOnCancel	What to RETURN TO if the user chooses the Cancel option in the error dialog.
cTitle	The default title for the error dialog.
cUser	The name of the user; this is logged.
cVersion	The application version number; this is used in the subject of an email sent to developers.
lDisplayErrors	.T. if we're supposed to display errors and get the user's choice.
lInsideTry	.T. if code was executing inside a TRY structure when the error occurred.
lLogErrors	.T. if we're logging errors to a file; the default is .T.
lLogToTable	.T. if the error should be logged to a table; otherwise, it's logged to a text file. Defaults to .T.
lOverwriteFile	.T. to overwrite the log text file, creating a new one whenever an error occurs.
lShowDebug	.T. if "debug" should be an option the user can choose to recover from the error.
nLastError	The index to the last error that occurred in aErrorInfo.

Catching an error

Since oError.ErrorHandler is defined as the ON ERROR handler, it's automatically called when an error occurs in the application that isn't caught by the Error method of an object or in a TRY block. ErrorHandler is also called from the Error method of my base classes as we saw earlier.

ErrorHandler has three main tasks, the first two of which are optional: log the error, display the error to the user, and recover from the error (quit the application, RETRY, return to the offending method or somewhere else, and so on). **Listing 2** shows the code for this method. Again, it is well-commented and easy to understand, but let's break it down.

Listing 2. The ErrorHandler method of SFErrorMgr deals with errors.

```
lparameters tnError, ;
   tcMethod, ;
   tnLine
local lcCurrTalk, ;
   lcChoice, ;
   laError[1], ;
   llReturn, ;
   lcProgram, ;
   llReturnTo, ;
   lcMethod, ;
   loException as Exception, ;
   llInsideTry

* If we have a logging object, log the error.

if type('oLogger.Name') = 'C'
   oLogger.LogMilestone('SFErrorMgr.ErrorHandler: error ' + ;
      transform(tnError) + ' in line ' + transform(tnLine) + ' of ' + ;
      transform(tcMethod))
endif type('oLogger.Name') = 'C'
with This
   try

* Ensure TALK is off and set the default return value to This.cDefaultAction.

      if set('TALK') = 'ON'
         set talk off
         lcCurrTalk = 'ON'
      else
         lcCurrTalk = 'OFF'
      endif set('TALK') = 'ON'
      lcChoice = .cDefaultAction
      do case

* Ignore "DataEnvironment already unloaded", "Error loading printer driver",
* and "Collate sequence not found" errors.

         case inlist(tnError, cnERR_DE_UNLOADED, cnERR_PRINTER_DRIVER, ;
            cnERR_COLLATE_NOT_FOUND)

* Handle any other error. First, save the error information.

         otherwise
            aerror(laError)
            .SetError(tcMethod, tnLine, message(1), @laError)
            .lErrorInfoSaved = .F.
```

```
            llReturn  = .IsReturnToOnCallStack()
            lcProgram = .cReturnToOnCancel

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

            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()
               endif .lDisplayErrors
            endif not .lSuppressErrors
      endcase
      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
            llReturnTo = .T.

* Display the debugger (runtime environment): use a pseudo command window.

         case lcChoice = ccMSG_DEBUG and version(2) = 0
            .CommandShell()
            lcChoice = ccMSG_CONTINUE

* Display the debugger (development environment): ensure _SCREEN is visible and
* return ccMSG_DEBUG to let the calling routine display the debugger itself.

         case lcChoice = ccMSG_DEBUG
            _screen.Visible = .T.

* 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, CLEAR EVENTS, and return to the top-level
* program.

          case lcChoice = ccMSG_QUIT
             .lQuit = .T.
             .RevertAllTables()
             on shutdown
             clear events
             llReturnTo = .T.
       endcase

* Restore TALK and the default error message.

       .ResetErrorMessage()
       if lcCurrTalk = 'ON'
          set talk on
       endif lcCurrTalk = 'ON'
    catch to loException
    endtry
endwith

* Return to the appropriate location (we have to do it here rather than in the
* TRY structure). We may not be able to do that if we're inside a TRY structure
* somewhere, since RETURN isn't allowed.

llInsideTry = This.lInsideTry
This.lInsideTry = .F.
do case
   case inlist(_vfp.StartMode, 2, 3, 5)
      comreturnerror(This.cAppName, ;
         This.aErrorInfo[This.nLastError, cnAERR_MESSAGE])
   case not llReturnTo
   case llInsideTry or (version(2) = 0 and llReturn and lcProgram = 'STARTUP')
      This.ImmediateExit()
   case llReturn
      This.CleanupBeforeReturn()
      return to &lcProgram
   otherwise
      This.lQuit = .T.
      This.CleanupBeforeReturn()
      return to master
endcase
This.lInsideTry = llInsideTry
return lcChoice
```

Logging the error

The first thing ErrorHandler does is log the fact that an error occurred if a global logging object exists. We'll look at this object when we discuss instrumentation later in this document. Next, the code calls SetError to add the error information to the aErrorInfo array.

If errors are supposed to be logged (lSuppressErrors is the default of .F. and lLogErrors is the default of .T.), the error is logged by calling LogError. LogError logs the error to either a table or text file, depending on the setting of lLogToTable. Among the things logged are:

- The date and time the error occurred.
- The name of the user (taken from the cUser property).
- The error number, message, method, line number and source (the latter two of which may not be available in a runtime environment).
- Current alias.
- Trigger type if an error occurred in a trigger.
- The values of all relevant memory variables.
- The call stack.
- The values of public properties for all accessible objects.
- The text of LIST STATUS.
- The contents of all Windows environment variables.

The latter five items are handled by the by-now slightly misnamed GetMemVars method. Some interesting things about GetMemVars are:

- It uses an interesting capability of the LIST MEMORY function: it can "see" all variables in the application, including those declared LOCAL. This is very important because otherwise there'd be no way to know the values of those variables, which is a key thing when debugging an application.
- GetMemVars cleans up some quirks LIST MEMORY has in formatting.
- I don't care about variables used in certain functions and methods, including those in the error handler itself, so those are removed from the list of variables. You can adapt that list as necessary.
- Like LIST MEMORY, LIST OBJECTS can "see" objects contained in variables, even LOCAL ones, and it lists the values of all public properties of those objects. However, LIST OBJECTS can crash under some conditions, such as when accessing certain types of properties of COM objects, so the code using it is wrapped in a TRY.

- Like variables, the code cleans up some formatting quirks with LIST OBJECTS and because there are some objects I don't want listed, such as the error handler itself, it removes those from the listing. You can adapt that list of objects as necessary.
- GetMemVars retrieves the names and values of environment variables using WMI, thanks to code written by Sergey Berezniker (http://tinyurl.com/oe9ufdf, which sadly seems to be offline).

Displaying the error to the user

After logging the error, ErrorHandler displays an error message to the user if it's supposed to (lSuppressErrors is the default of .F. and lDisplayErrors is the default of .T.) by calling DisplayError. DisplayError instantiates the class defined in cMessageClass and cMessageLibrary, sets some properties, and calls Show. Note that the class doesn't have to be a form class; SFErrorMessage, which is the default class, uses a MESSAGEBOX() (although it's actually an extended MESSAGEBOX() with custom button captions, thanks to Cesar Chalom: http://tinyurl.com/qar5o6m). All the class needs is the properties written to by DisplayError and a Show method.

What type of information should be displayed in the dialog? The dialog shown in **Figure 1** displays when an error occurs for an end-user. This form comes from SFErrorMessageDialog in SFErrorMgr.vcx.

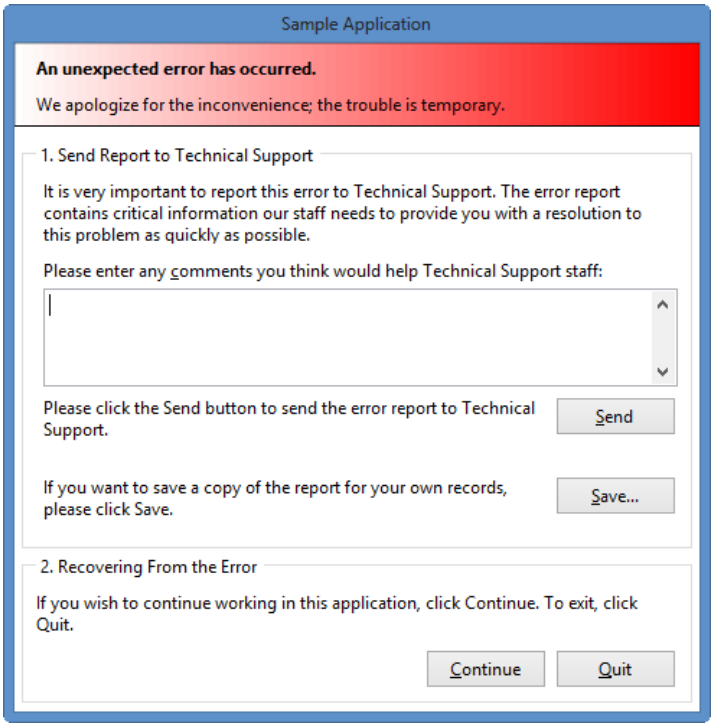

Figure 1. The error dialog seen by end-users prompts them for additional information.

This dialog has the following features:

- It tells them the problem is only temporary and that we'll fix it. I think this is important because many users panic when an error occurs, thinking that some permanent damage has happened by something they did.
- It prompts them for additional information about what they were doing when the error occurred. In my experience, end-users type something in about 10% of the time.
- It asks them to send the error to our technical support staff by pressing the Send button. What this actually does is discussed later.
- It provides a Save button that creates a file named Error.txt containing information about the error; we'll see the contents of this file later. This button is there for us when we're directly interacting with a customer or for when the Send button fails for some reason, such as no Internet connection.
- It may provide recovery from the error condition (not the cause of the error but being in an error state), depending on the type of problem that happened. When possible, the Continue button is visible, allowing the user to stay in the application and carry on as if nothing happened. When that isn't possible, only the Quit button, which terminates the application, is visible. We'll see the types of problems that allow recovery and how to actually perform the recovery later.

While this dialog is useful for end-users, it isn't for developers because it doesn't provide a way to debug the problem. So, when an error occurs, developers get the dialog shown in **Figure 2** instead.

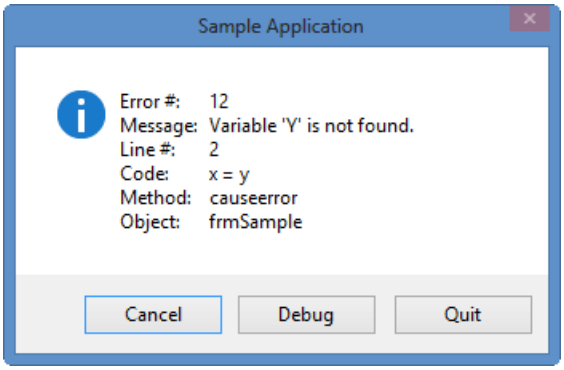

Figure 2. The error dialog displayed for developers gives options a developer needs to help debug the problem.

This dialog has the following features:

- It displays the error number, message, object and method name, and when running in the IDE, the line number and line of code where the error occurred (the latter two aren't usually available at runtime).
- The Cancel button acts like the Continue button in the end-user dialog: the application continues to run but execution is returned to the READ EVENTS, not the next line of code.

- If the application is running in the IDE, the Debug button opens the VFP Debugger at the line of code following the one that caused the error. I use this button most of the time, because I can often try something, like changing the value of a variable, and then use the Set Next Statement function to go back to the offending line and see if that fixed the problem.
- If the application is running at runtime, the Debug button opens the Command Console dialog shown in **Figure 3**. This dialog emulates the VFP Command window, allowing you to execute VFP commands at runtime, including opening and browsing tables, examining or changing the values of global variables, and so on. The Emulate Command Window checkbox determines whether code entered in the editbox is executed when you press Enter (like the Command window) or when you click the Execute button (allowing you to write multi-line code similar to a PRG before executing it). The Clear button clears the editbox, the arrow keys scroll through statements you previously entered, and Load loads a PRG or text file so you can execute it. Of course, none of this is magic; it just relies on the VFP EXECSCRIPT() function to execute code at runtime.
- The Quit button doesn't QUIT but terminates the application so I can start fixing the problem. This is the second most-used button.

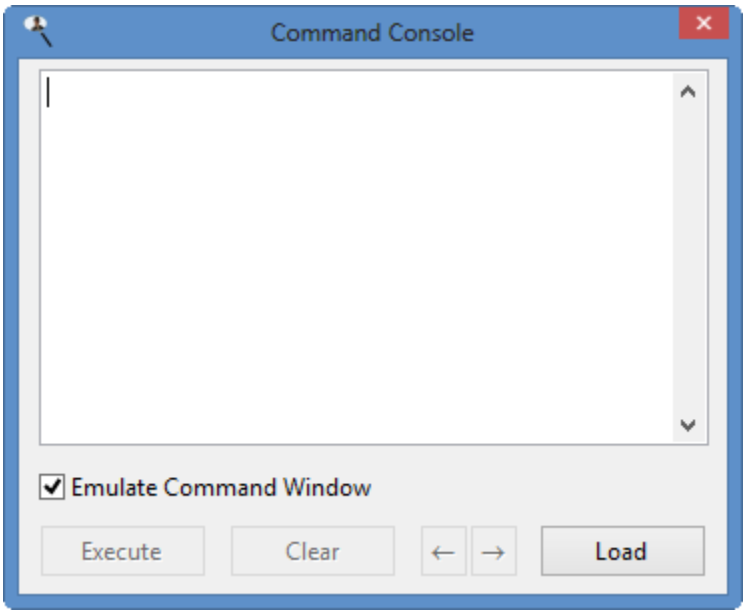

Figure 3. The Command Console window emulates the VFP Command window in a runtime environment.

How do you tell SFErrorMgr which dialog to use? I use the following code in my application startup:

```
if llDebug
   oError.cMessageClass   = 'SFErrorMessage'
   oError.cMessageLibrary = 'SFErrorMgr.vcx'
else
   oError.cMessageClass   = 'SFErrorMessageDialog'
   oError.cMessageLibrary = 'SFErrorMgr.vcx'
```

```
endif llDebug
oError.lShowDebug = llDebug
```

llDebug is set to .T. if we want to run in "debug" mode, meaning we get the developer's dialog and have a "debug" option.

Notifying development staff

The Send button in **Figure 1** sends a message to development staff that an error occurred. This is handled in the SendMessage method of SFErrorMessageDialog, shown in **Listing 3**. This code uses Craig Boyd's VFPExMAPI library (http://tinyurl.com/38gou7z) to send an email using MAPI. You can, of course, replace this with code using your favorite email mechanism such as West Wind's wwSMTP class.

Listing 3. SFErrorMessageDialog.SendMessages sends an email to development staff.

```
#define MAPI_TO 1
#define IMPORTANCE_NORMAL 1

local lcAttachment, ;
   lcBody, ;
   llOK, ;
   lcMessage
with This

* Create an attachment by having the error handler create a text file with
* error information.

   lcAttachment = sys(2023) + sys(3) + '.TXT'
   .CreateLogFile(lcAttachment)

* Email the error information.

   set library to VFPExMAPI.fll
   lcBody = alltrim(.cMessage) + iif(empty(.edtComments.Value), '', ;
      ccCRLF + ccCRLF + 'Comments:' + ccCRLF + .edtComments.Value)
   EMCreateMessage(.cSubject, lcBody, IMPORTANCE_NORMAL)
   EMAddRecipient(.cRecipient, MAPI_TO)
   EMAddAttachment(lcAttachment)
   if not empty(.cAttachment)
      EMAddAttachment(.cAttachment)
   endif not empty(.cAttachment)
   llOK = EMSend(.T.)

* Display any error that occurred during sending.

   if not llOK
      lcMessage = 'The email could not be sent.'
      messagebox(lcMessage, MB_ICONSTOP, _screen.Caption)
   endif not llOK

* If we created the attachment, delete it.

   if empty(.cAttachment)
```

```
      erase (lcAttachment)
   endif empty(.cAttachment)
endwith
```

This code creates a log text file by calling CreateLogFile, which asks SFErrorMgr to do it by setting the appropriate properties (lLogToTable .F., lOverwriteFile .T., cErrorLogFile to the name of a text file) and calling LogError.

The subject for the email is set in Init:

```
.cSubject = 'Error in ' + .oErrorMgr.cAppName + ' Version ' + ;
   .oErrorMgr.cVersion
```

The message for the email comes from cMessage, the recipient from cRecipient, and any additional attachment for the email (such as a document the application was processing when the error occurred) from cAttachment. Where are these properties set? The problem is that they may be application-specific. So, create a subclass of SFErrorMgr, override the SetDialogProperties method to set those properties, and use it as the class for oError.

Of course, instead of sending an email, you may want to create a support ticket on your system. We use HESK (www.hesk.com) as our support ticket system. Although it doesn't have an API we can call to programmatically create a support ticket, it uses a MySQL database for support tickets. So, with a little detective work, we determined the structure of the appropriate tables and created an ASPX web page that accepts several POST variables, such as the user's name, email address, and name of the error log file, writes them to a new record in the MySQL database, and emails the user with a link to the new support ticket. We use West Wind Client Tools (https://tinyurl.com/yavel3kg) to call the ASPX web page to create the support ticket. We also upload the error log file using FTP to the directory where HESK stores support ticket attachments. The bottom line is that when the user clicks Send, it automatically creates a support ticket for them and emails them so they can access the ticket. You, of course, can use any mechanism you wish to create a support ticket in your own system.

Recovering from an error

DisplayError returns a string indicating what the user chose to do in the dialog. Notice in **Figure 1**, there's an option to continue working in the application. How could that work? After all, the VFP CONTINUE command returns execution to the statement following the one that triggered the error, and since the one causing the error didn't execute, anything it was supposed to do, such as creating a variable, didn't happen. As a result, another error is very likely to occur (after all, if the statement not executing isn't important, what is it there for?). Let's discuss error recovery.

ErrorHandler decides what to do based on the action the user chose:

- If the action is "Cancel," a flag is set to return to the program in the call stack containing the READ EVENTS statement.

- If the action is "Quit" when we're running the application from the VFP IDE (in which case, we don't want to quit from VFP, just the application), the code calls RevertAllTables so all transactions are rolled back and all open cursors are reverted so we don't have issues with uncommitted changes, and then sets a flag to return to the top-most program in the call stack.
- For "Debug" in a runtime environment, the code calls CommandShell to display SFConsoleForm in SFConsole.VCX, the runtime "command window" shown in **Figure 3**.
- For "Debug" in the VFP IDE, _SCREEN.Visible is set to .T. in case you're running a top-level form application with _SCREEN hidden and "Debug" is returned to the caller. When called from the Error method of an object, this results in the VFP Debugger window opening.
- For "Retry" from procedural code or when ErrorHandler wasn't called from the Error method of an object, RETRY is executed. For "Retry" when called from Error, "Retry" is returned so the object can execute RETRY itself.
- For "Quit" in a runtime environment, the code calls RevertAllTables as described earlier, issues ON SHUTDOWN and CLEAR EVENTS to prepare for shutting down the application, and then sets a flag to return to the top-most program in the call stack.

The actual error recovery is a little complicated. Normally, we'd do one of a few things:

- If we're supposed to just return the action string to the caller, use RETURN lcChoice.
- If we're supposed to continue in the program, use RETURN TO *SomeMethod*, where *SomeMethod* is the name of the routine where READ EVENTS exists. This has the effect of unwinding the call stack that caused the error to occur and the application is sitting in the event loop, waiting for the next action by the user.
- If we're supposed to quit the application, use RETURN TO MASTER to unwind the call stack and go back to the top-level program so we can do an orderly shutdown.

The problem is due to a restriction in TRY: you can't issue a RETURN statement when a TRY is active. Typically, SFErrorMgr.ErrorHandler isn't called directly from a TRY since CATCH handles the error rather than an Error method or ON ERROR. However, if the method of an object is called from within TRY, an error occurs in that method or code called from that method, and the object has an Error method, then that Error method is executed rather than CATCH. You can see that in the code shown in **Listing 4**; if you run this code (taken from TryProblem.prg), you'll see that the Error method of SomeObject caught the error, not the CATCH statement. That's not a problem in this case, but try adding RETURN TO MASTER at the end of the Error method; you'll get an untrappable "RETURN/RETRY statement not allowed in TRY/CATCH" error.

Listing 4. CATCH doesn't always catch errors.

```
loObject = createobject('SomeObject')
try
```

```
   loObject.SomeMethod()
catch
   messagebox('CATCH caught the error')
endtry


define class SomeObject as Custom
   function SomeMethod
      x = y
   endfunc

   function Error(tnError, tcMethod, tnLine)
      messagebox('The Error method of SomeObject was called')
   endfunc
enddefine
```

So, the problem is that if the Error method of the object calls ErrorHandler, which all of my objects do, how do we know whether we can use RETURN TO or not? You might think SYS(2410), which tells you how an error will be handled, would help, but it can't tell whether there's a TRY somewhere in the call stack, so it returns 2 (Error method) or 3 (ON ERROR). Funny that VFP can't tell whether a TRY is involved until you try to do a RETURN TO!

To solve this issue, the code at the end of ErrorHandler has to decide whether or not to use RETURN TO. The only way it can know if a TRY is involved is if you tell it by setting the lInsideTry property to .T. before calling any method or procedure from inside a TRY and .F. again afterward. For example:

```
oError.lInsideTry = .T.
try
   This.SomeMethod()
catch
endtry
oError.lInsideTry = .F.
```

(I know this seems clunky but it's the only way I've found to resolve this issue.)

The code at the end of ErrorHandler does one of the following things:

- Simply return the string containing the user's choice if we're not using RETURN TO.
- If we're inside a TRY or the error occurred in the startup program at runtime, call ImmediateExit, which closes all forms, releases all global objects, closes all procedure and library files, does other cleanup, and terminates the program with either CANCEL (running in the IDE) or the ExitProcess Windows API function, which returns an error code (I'll discuss that in more detail later).
- If we're supposed to continue in the program, do some cleanup and then RETURN TO the routine containing the READ EVENT.

- If none of the above is done, do some cleanup and RETURN TO MASTER to shut down the program in an orderly manner.

Returning an error code

While a function or method can return a value to the caller, an executable can't. So, if your application was launched from another application, such as the Windows Task Scheduler, and you want to tell that application whether your application succeeded or terminated with an error, you need some way to send that to the caller. One way is to terminate the application using the ExitProcess Windows API function. The function returns an exit code the other application can use; 0 means the application succeeded.

ExitProcess is easy to use:

```
declare ExitProcess in Win32API integer ExitCode
ExitProcess(lnReturnCode)
```

However, since it terminates the application immediately, be sure to do this as the last thing in your application. If you look at the ImmediateExit method of SFErrorMgr, you'll see that ExitProcess is used after all the cleanup tasks are done.

Error handling summary

Here's a summary of the process flow when an error occurs in code using the techniques and/or code outlined in this document:

- If an error occurs in an object with code in its Error method, the error bubbles up through the inheritance and containership hierarchy until either some object handles the error or it goes to SFErrorMgr.ErrorHandler.
- If an error occurs in procedural code or in an object without code in its Error method, the error goes straight to SFErrorMgr.ErrorHandler.
- ErrorHandler logs the error to Errorlog.dbf then displays a dialog asking the user what to do.
- If the user chooses Continue (Cancel in the developer dialog), execution goes back to the READ EVENTS statement, unwinding the call stack up to that point.
- If the user chooses Quit, the application terminates, returning an error code.
- If the user chooses Debug, execution returns to the line of code following the one that caused the error, with the VFP Debugger open at that line.

Error log analysis strategies

Let's look at some strategies for analyzing the error log.

- I tend to look at Error.txt much more often than Errorlog.dbf, mostly because that's what's emailed to us or attached to a support ticket. It contains information about the most recent error. However, sometimes I ask the user to send me Errorlog.dbf

and fpt because they contain a history of all errors, and sometimes that's needed for more complex issues.

- The first things I look at are the error number, message, and method. Sometimes that alone tells you what the problem is. For example:

```
Error #:    2005
Message:    Error loading file - record number 3.  frmScheduleWizard <or one
   of its members>.  Loading form or the data environment : OLE error code
   0x80040154: Class not registered
Method:     sfqapplication.schedulereport
```

  The ScheduleReport method instantiates a form class, which fails when loading the class (otherwise the error would have occurring in a method of the class). "Class not registered" means an ActiveX control on the form isn't registered on the user's system. A quick check of the controls on the form tells me it's TaskScheduler.DLL that for some reason isn't registered, so using REGSVR32 fixed the problem.

  Here's another one:

```
Error #:    1
Message:    File 'decrypt.prg' does not exist.
Method:     sfutility.decrypt
```

  Looking at the Decrypt method, I see a call to a Decrypt function. However, that function is actually in a DLL written by Craig Boyd (VFPEncryption.DLL), so obviously that DLL wasn't loaded with SET LIBRARY TO. Code doing that is wrapped in a TRY structure, so a quick peek at the folder on the user's computer showed that file was missing. Replacing it solved the problem.

- After looking at the error number and message, I next scroll down to the variables defined in the method where the error occurred, looking for clues. If it's an object, I may also look at the values of the properties of that object. For example, this error occurred when trying to output a report to a file:

```
Error #:    202
Message:    Invalid path or file name.
Method:     sfroutputdelimited..createfile
```

  The cOutputFileName property of SFROutputDelimited was set to "Z:\Data\Export.csv." The user used to have a drive mapping for Z: but doesn't anymore, hence the error. The immediate solution is to tell the user to use a different path, but this also gives us an opportunity to improve the error handling of the application: rather than letting the user get the "red error dialog," we could test whether the specified path is valid and display an appropriate warning message if not.

  Here's another error:

```
Error #:    1526
Message:    [MySQL][ODBC 5.2(a) Driver][mysqld-5.5.28-log]FUNCTION
```

```
   central.GoMonthDay does not exist
Method:     SelectFromSingleTable
```

Looking at the variables for SelectFromSingleTable, I see that tcSelect, which contains the SQL statement to use, contains:

```
select CustID, SettlementDateTime, DollarAmountPaid, MergedSettlementID from
settlement where SettlementDateTime>=GoMonthDay(curdate, -5,  1)
```

The user created a filter that uses an expression the MySQL database doesn't understand.

- Unless you're running the application in the IDE or had Debug Info turned on when you built the EXE (which makes the EXE much larger), the line number and code statement in the error log are empty. To figure out exactly where an error occurred in that case sometimes calls for detective work. I look at the values of variables and determine where they were set in the code. For example, variables that contain .F. probably weren't touched (they're .F. because the LOCAL statement that declared them made them logical by default), so I know execution didn't get to the point where those variables were assigned values. Here's an example:

```
LCGROUPBY Local C "group by 14"  processcursor
LCGROUP_AVERAGE_FIELDS Local C ""  processcursor
LCGROUP_COUNTS Local C ""  processcursor
LCGROUP_COUNT_FIELDS Local C ""  processcursor
LLHAVECOUNT Local L .F.  processcursor
LCAVERAGE Local L .F.  processcursor
```

It looks like the block of code where the error occurred is between the initialization of lcGroup_Count_Fields and llHaveCount.

- Note that declaring variables in the LOCAL statement in the same order as they're first assigned values helps immensely. Because declaring them assigns them a value of .F., they'll appear in the order declared in the error log. If they're also assigned values in that same order, code assigned values to variables which appear in the error log with the default of .F. probably hasn't been reached yet. Rather than doing this manually, which would be tedious, I use Thor's Create LOCALs tool, which I've assigned a hotkey to.
- Because LIST OBJECTS only shows objects stored in memory variables, you can't see the values of properties of objects that are contained in properties of objects. For example, suppose you have a SFApplication object stored in the global variable oApp, and it has a reference to a SFUser object in its oUser property. All the error log will display for oUser is this:

```
OUSER  O SFUSER
```

The only way to see the values of the properties of oUser is to put it, even if just temporarily into a memory variable:

```
loUser = oApp.oUser
```

Now loUser will show up as its own object, complete with a listing of property values.

A bigger design issue is that you should favor objects as individual global variables rather than members of a single global application object.

- I don't look at the call stack very often. About the only time is when I'm trying to figure out where a particular method was called from when it can be called from multiple places.
- The environment variables section is occasionally handy. One puzzling VFP error that can occur before the application even starts is caused by an invalid PATH setting (the Windows, not VFP, path). Also, if the error occurs before the user logs into the application, the environment variables often contain values that hint at who the user might be in case you need to contact them for more information:

  ```
  TEMP (DHENNIGWIN8\dhennig): %USERPROFILE%\AppData\Local\Temp
  ```

- The other sections in the error log have only proven useful once or twice but it's worth having them there just in case.
- Sometimes the cause of an error can be far removed from where the error happens. Here's part of an error log we received recently:

  ```
  Error #:    31
  Message:    Invalid subscript reference.
  Method:     sfgetcondition.cboOperator.requery
  LNOPERATORS Local N 0   requery
  LNI Local N 29   requery
  LOOPERATOR Local O SFOPERATORISNO  requery
  LCOPERATOR Local C ""  requery
  LAITEMS Local A requery
  ( 1) L .F.
  ```

  Here's the relevant code from the Requery method:

  ```
  for lnI = 1 to alen(.aOperators, 1)
     loOperator = .aOperators[lnI, 1]
     do case
        case vartype(loOperator) = 'C'
           lcOperator = loOperator
        case .oValues.cFieldType $ loOperator.cDataTypes
           lcOperator = loOperator.cOperator
        otherwise
           lcOperator = ''
     endcase
     if not empty(lcOperator) and not lcOperator == lcLastOperator and ;
        not (lcOperator = '\-' and lnOperators = 0)
        lnOperators = lnOperators + 1
        dimension laItems[lnOperators, 2]
  ```

```
            store lcOperator to laItems[lnOperators, 1], lcLastOperator
            if vartype(loOperator) = 'O'
               laItems[lnOperators, 2] = lower(loOperator.Class)
            else
               laItems[lnOperators, 2] = ''
            endif vartype(loOperator) = 'O'
         endif not empty(lcOperator) ...
      next lnI

      * If the last item is a separator, remove it.

      if laItems[lnOperators, 1] = '\-'
         lnOperators = lnOperators - 1
         dimension laItems[lnOperators, 2]
      endif laItems[lnOperators, 1] = '\-'
```

This code creates an array of filter operators applicable to the data type of a field; for example, we don't want "begins with" for anything but character fields. The code goes through an array of all possible operators and only adds those where the data type of the field is in the list of types the operator is applicable to to the laItems array. We can see from the value of lnI that the loop must have finished yet laItems is empty. In fact, that's the cause of the error: the IF statement right after the loop blows up because lnOperators is 0 which isn't a valid subscript value for an array. That means the data type of the field wasn't found. Although we can't see the value of This.oValues.cFieldType in the log, code calling this method sets that property from loField.cFieldType. The log has this for loField:

```
Object: LOFIELD          Local    O  SFRFIELD

Properties:
CFIELDNAME               C  "CONTACT.Carte"
CFIELDTYPE               C  " "
```

CONTACT.Carte is a custom field added dynamically to the data dictionary. Checking with the customer, it turns out it's a picture-type field and our code adding custom fields to the data dictionary, which executed long before and far away from the error, didn't handle that data type, so cFieldType is empty. Fixing that resolved the error, but we also added defensive code in Requery to handle illegal subscript values in case another data type comes up in the future.

Instrumentation

Not all problems a user may experience are errors so there may be no error report for you to go on. For example, the application may have performance issues, reports may show incorrect results, and so on. For that reason, in addition to having a robust error handling strategy, your application also needs instrumentation.

Instrumenting an application means logging important aspects about it at various places in the code. What those aspects are depends on the application in general and the specific code being instrumented.

If an error report is a snapshot in time, like looking at the wreckage of a plane crash, the diagnostic generated by the instrumentation is like the black box in the plane: it shows what happened at various stages so you can backtrack and see how an event occurred. Both types of documentation are vital to tracking down and solving a problem.

How to instrument your application

Instrumenting an application is actually very simple: you log things that are important to you at various places in the application's code. How you write to the log can vary: it can be a table, a text file, a web service, etc. I like to log to a text file because it's easily opened and read even if the VFP IDE isn't installed. For that, I use a VFPX project called Log4VFP (https://github.com/VFPX/Log4VFP).

Log4VFP is a wrapper for Log4NET, a popular diagnostic logging library for .NET applications. Using Log4VFP is easy: instantiate the Log4VFP class in Log4VFP.prg to initialize it and wwDotNetBridge, a utility by Rick Strahl that allows a VFP application to use a .NET assembly, then call the Open method with the name of a text file to log to. For example, here's the code from the sample Main.prg that sets up logging:

```
lcLogFile = lower(fullpath('Diagnostic.txt'))
try
   erase (lcLogFile)
catch to loException
endtry
oLogger = newobject('Log4VFP', 'Log4VFP.prg')
oLogger.cConfigurationFile = fullpath('verbose.config')
oLogger.Open(lcLogFile)
oLogger.LogInfo('Startup: application started')
```

This code erases any existing log file (I usually include a path as well but omitted it in the samples that come with this document), instantiates the Log4VFP wrapper class into the oLogger global variable, tells it which configuration file to use (the configuration file defines how to perform the logging; see the Log4VFP and Log4NET documentation for details), starts logging by calling Open, and logs that the application has started. The reason for erasing the log file is to prevent it from growing very large and cumbersome to analyze over multiple application runs. You may decide to not delete it, only delete it under certain circumstances (I sometimes do that by checking for an entry in an INI file if I want to keep a history over several runs), or not delete it but include a timestamp in the filename so you keep the log files but have one for each application run.

Listing 5 shows the complete log of a run of the sample code that comes with this document, including clicking the Perform Process button in the form.

Listing 5. The complete log of a run of the sample code.

```
2018-10-02 15:48:08,702 Startup: application started
0.0599637 seconds since previous milestone
Total time to run: 0.0619629 seconds

2018-10-02 15:48:08,711 Startup: creating error manager
```

```
0 seconds since previous milestone
Total time to run: 0.0629623 seconds

2018-10-02 15:48:08,715 Startup: setting error manager properties
0.004996 seconds since previous milestone
Total time to run: 0.0679583 seconds

2018-10-02 15:48:08,718 Startup: running Test form
0 seconds since previous milestone
Total time to run: 0.0699579 seconds

2018-10-02 15:48:08,740 ReadEvents: starting event loop
0.0220129 seconds since previous milestone
Total time to run: 0.0919708 seconds

2018-10-02 15:48:11,943 SFErrorMgr.ErrorHandler: error 12 in line 2 of
frmSample.causeerror
3.226313 seconds since previous milestone
Total time to run: 3.2962709 seconds

2018-10-02 15:48:13,834 frmTest.cmdProcess.Click: starting process
0.0009981 seconds since previous milestone
Total time to run: 5.1867948 seconds

2018-10-02 15:48:13,835 frmTest.cmdProcess.Click: done process
0.0019971 seconds since previous milestone
Total time to run: 5.1877938 seconds

2018-10-02 15:48:15,321 Startup: application ended
1.4882191 seconds since previous milestone
Total time to run: 6.6740158 seconds
```

What should you log

When and what you log is up to you. We started very simply by just logging major events, like the user starting a report or running a query against the database. As issues arose where we needed more granular information, we added more logging. Here are some places you'll likely want to log:

- Application startup and some of the tasks performed during startup. In my experience, things can go wrong here, not necessarily because of errors in the code but due to environmental issues: tables you try to open don't exist or are corrupted, files you expect to write to are read-only due to folder permissions, and so on.
- Processes that take some time, such as posting an order or importing a customer list. These are likely places where the user may complain about performance, so knowing the amount of time each step in a process takes can help you find where the bottleneck is.
- Access to external resources like databases, the file system, web services, etc. Connecting to a database like SQL Server has many points of failure: the server can't be found, the login credentials are invalid, etc. Performing a query can be error-prone: the table or columns may not exist, the syntax of the query may not be

correct, and so on. Web services are often tricky to deal with. Logging the various steps in these processes makes it a lot easier to find out what's failing and why.

- Code where the user could do something destructive, such as deleting a record. Ever have a user deny deleting a record and accuse the software of being buggy? The log will prove what really happened.
- Tricky code. Let's face it: we've all written code that six months later is really tough to figure out what it's doing. Sprinkling logging statements throughout the code makes it easier to track exactly how the code is executing and in what order tasks are done.
- Code where you display a warning message of some type to the user but don't log it as an error. A lot of time, the user won't tell you they saw the warning message (or will even deny having seen it when asked) so having it in the log is useful.
- Code that's particularly error-prone. Some places in the application seem to cause more problems than others. We found, for example, that a couple of complex methods in Stonefield Query represented 10% of our support calls. Analyzing what path execution took when problems arose helped us to rewrite these methods to be more reliable.

What you log is also up to you and can vary from process to process. For example, in an import routine, you'll probably want to log the name of the import file, how many records are in the file, the progress of the import, how many records were rejected, and so on. For a method that performs complex calculations, you should log the values of the inputs and each intermediate result so if the final result is wrong, you can backtrack through the calculations to see what went wrong and where. One thing I always include in log entries is the name of the routine so I can quickly find where the code was executing.

What about overhead?

One topic that always comes up when I talk to other developers about instrumentation is whether it slows the application down or not. Obviously, the more work code has to do, the longer it takes. However, since Log4VFP really just appends to a text file, which takes a small fraction of a second, it doesn't impact performance in any noticeable manner. Of course, you may need to consider whether or not to use logging in time-sensitive code or extensive loops (not only for the performance issue: do you really want to read a log with 10,000 entries from a loop?).

Tying it all together

Let's look at some techniques that combine error and diagnostic log analysis for complete tracking of problems.

- Use the diagnostic log to see where the problem occurred and how it got there. This error log shows that the error occurred in Sysmain.prg but not where:

```
Error #:    108
Message:    File is in use by another user.
```

```
Method:     SYSMAIN.FXP
```

A glance at Diagnostic.TXT showed the last thing logged before the error occurred:

```
===============> 1/07/2013 6:25:45 PM (0.015 seconds since previous milestone)
Sysmain: about to login
0 seconds since previous milestone
Total time to run: 2.422 seconds

SFQErrorMgr.GetMemVars: terminating with error #108
```

Just before calling the Login method, the Users table is opened. That table is normally opened shared but in this case, another user was reindexing the table, so it was opened exclusively.

- Sometimes you need a combination of the diagnostic and error logs to determine where the error occurred.

```
LCMESSAGE Local C "Processing <Insert1>..."  fillcursorcollection
LCCURSOR1 Local C "_3UE0RY2QH"  fillcursorcollection
LCDIRECTORY Local C "C:\Program Files (x86)\Stonefield Query SDK\Data\"
   fillcursorcollection
LCTABLE Local L .F.  fillcursorcollection
LOCURSOR Local L .F.  fillcursorcollection
```

This shows that the last variable assigned is lcDirectory. However, there are numerous lines of code between the assignment of lcDirectory and the next variable, so that leaves a gap in which the error could have occurred. However, the last entry in the diagnostic log is "Processing tables," which is logged before lcDirectory is assigned a value, and there are several logging calls after lcDirectory, so the error must occur fairly soon after lcDirectory is assigned a value. In this case, the error was "File is in use" and right after lcDirectory is assigned a value, a database is created, so obviously that database was already open.

- Add more logging code to narrow down the location. Sometimes, especially in the beginning when you haven't added a lot of logging to the application, there can be wide gaps between log entries, which makes it difficult to determine exactly where the problem occurred. Don't be afraid to add additional logging calls to help pinpoint the location.
- Performance problems can be tracked down by putting logging calls around (or even inside) long-running or time-sensitive processes, such as SQL statements, import or export routines, year-end procedures, and so forth. For example, several customers complained that Stonefield Query hung when they started it. The log file showed:

```
====================> 1/24/2012 5:32:08 PM
Checking for update

====================> 1/24/2012 5:37:08 PM
Finished checking for update
```

```
Time to run: 302.015 seconds
```

It wasn't hung, it just took 5 minutes to check for an update. Adding additional logging to the update process showed that on some systems, connecting to the FTP server failed but it took 5 minutes to timeout. Changing to use passive FTP resolved the problem.

In another case, a customer's report took a long time to run. Sure enough, looking at the end of the log file, we could see it took 93.703 seconds. Looking at the start of the report run, we noticed it took less than a second to retrieve the data from the database, which is normally the most time-consuming task:

```
SFQDataSourceODBC.SelectFromSingleTable
cSelect = <SQL statement went here>
Retrieved 1110 records
0.079 seconds since previous milestone
```

However, after the data was retrieved, there was a long section of log entries of 1,110 additional hits to the database, each looking similar to:

```
===============> 2/28/2013 5:33:05 PM (0.063 seconds since previous milestone)
SFScriptMgr.Execute: about to execute DataEngine.PerformQuery

SFRDataSourceODBC.SelectFromSingleTable: sending SQL statement:
select EXTPRICE from dbo.ARTRAN ARTRAN where INVNO=?INVNO and ITEM='FREIGHT'
Retrieved 0 records
0.016 seconds since previous milestone
Total time to run: 11.094 seconds

===================> 2/28/2013 5:33:05 PM (0 seconds since previous milestone)
SFScriptMgr.Execute: finished executing DataEngine.PerformQuery

RunSQL: return value: $0.00000000000000000
RunSQL: variable values:
INVNO:     282159

0 seconds since previous milestone
Total time to run: 11.110 seconds
```

RunSQL is a function that executes a SQL statement. The user had created a calculated field with an expression of RunSQL("select EXTPRICE from ARTRAN where INVNO=?INVNO and ITEM='FREIGHT'") to determine the freight amount for each invoice. Evaluating the calculated field meant hitting the database once for each record, so that was 1,110 additional queries taking nearly 90 seconds. We showed them how to create a cursor of freight values, indexed by invoice number, with a single query and use SEEK to find the desired value, and the report run dropped to under five seconds.

- Create a special "debug" build. Sometimes, even lots of log entries don't narrow down the location quite enough. In that case, turn on Debug Info in the Project

Information dialog, build the EXE, install it in the production folder, and have the user reproduce the problem. The error log should now have the line number and line of code that caused the error. Don't forget to turn Debug Info back off again.

- Use SET COVERAGE if necessary. I rarely use this but sometimes it's really puzzling about how execution got to a particular place, especially if BINDEVENTS, ON KEY routines, Assign methods, or other things that change the execution flow are involved. In that case, add SET COVERAGE TO SomeFile just before the last place you know execution took place, and SET COVERAGE TO later in the code to minimize the size of the coverage log. As with Debug Info, don't forget to remove those statements afterward.

References

Error Handling in Visual FoxPro, http://www.doughennig.com/Papers/Pub/errorh.pdf

Error Handling in VFP 8,
http://www.doughennig.com/Papers/Pub/ErrorHandlingInVFP8.pdf

Error Handling Revisited, http://www.doughennig.com/Papers/Pub/Jan98.pdf

Summary

90% of fixing a problem is finding out where it happens and why. You can almost always find the cause quickly if you can trace the code, but many times you don't have that luxury. In those cases, all you have to go by is what your error and diagnostic logs tell you. The better and more information you log, the faster you can track down the problem. Comprehensive logs accompanied with good detective skills can usually help you track the problem down quickly.

Biography

Doug Hennig is a partner with Stonefield Software Inc. He is the author of the award-winning Stonefield Database Toolkit (SDT); the award-winning Stonefield Query; the MemberData Editor, Anchor Editor, and CursorAdapter and DataEnvironment builders that come with Microsoft Visual FoxPro; and the My namespace and updated Upsizing Wizard in Sedna.

Doug is co-author of *VFPX: Open Source Treasure for the VFP Developer*, *Making Sense of Sedna and SP2*, the *What's New in Visual FoxPro* series, *Visual FoxPro Best Practices For The Next Ten Years*, and *The Hacker's Guide to Visual FoxPro 7.0*. He was the technical editor of *The Hacker's Guide to Visual FoxPro 6.0* and *The Fundamentals*. He wrote over 100 articles in 10 years for FoxRockX and FoxTalk and has written numerous articles in FoxPro Advisor, Advisor Guide to Visual FoxPro, and CoDe.

Doug spoke at every Microsoft FoxPro Developers Conference (DevCon) starting in 1997 and at user groups and developer conferences all over the world. He is one of the organizers of the annual Southwest Fox conference (http://www.swfox.net). He is one of

the administrators for the VFPX VFP community extensions Web site (http://vfpx.org). He was a Microsoft Most Valuable Professional (MVP) from 1996 through 2011. Doug was awarded the 2006 FoxPro Community Lifetime Achievement Award (http://tinyurl.com/ygnk73h).