ErrorHandler

# Introduction

This document provides documentation for ErrorHandler, an error handling utility that’s part of VFPX (<http://vfpx.org>).

ErrorHandler provides a highly configurable and customizable error handler for any VFP application. It supports logging error information to a table, displaying an easy-to-understand dialog to the user (Figure 1), notifying support staff about the error via email or support ticket, and recovering from the error (either continuing in the application but not returning to the method that caused the error or terminating the application).

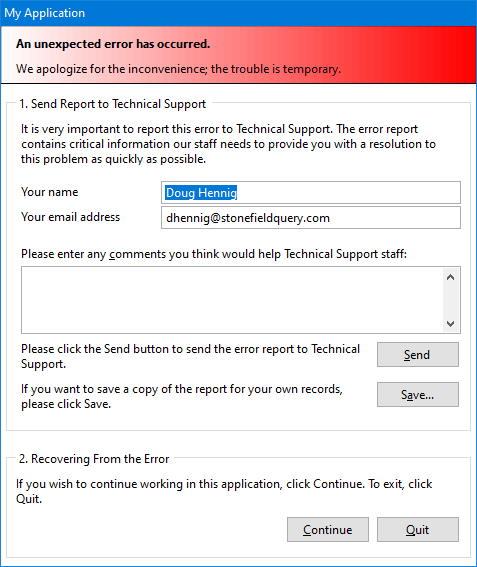


Figure 1. The error dialog seen by end-users.

The error dialog has the following features:

* It tells the user 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 only about 10% of the time but sometimes it’s helpful.
* It asks them to send the error to our technical support staff by pressing the Send button. This either creates a support ticket or sends an email to a particular address.
* It optionally provides a Save button that creates a text file 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.

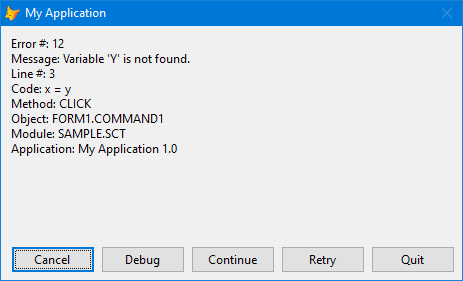


Figure 2. The error dialog displayed for developers gives options a developer needs to help debug the problem.

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 fixes 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 Continue button essentially does a RETURN, sending execution back to the line of code following the one that caused the error. Retry does a RETRY. Neither of these are that useful unless the error was something environmental you can correct, such as a missing file.
* The Quit button doesn’t QUIT but terminates the application so you can start fixing the problem. This is the second most-used button.

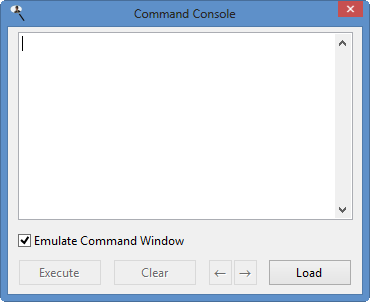


Figure 3. The Command Console window emulates the VFP Command window in a runtime environment.

# Testing ErrorHandler

Several sample files are included with this project. Edit Sample.prg and set the cRecipient, cMailServer, nSMTPPort, cUserName, and cPassword settings to the appropriate values for your email server. You can also change the values for cContact and cEmail if desired (if you don’t, any email you send from the error handler will look like it came from me).

DO Sample.prg and click the Cause an Error button in the form that appears. This causes an error by executing X = Y without defining a variable named Y. Click the Send button in the error dialog that appears to email an error report to the address you specified in cRecipient, click the Save button to save the error report to a text file, and click Continue to handle the error without exiting the “application.” Then click the Cause an Error Inside TRY button in the sample form and notice the error dialog no longer has a Continue button; the only option is to terminate the program. I’ll discuss each of these features later in this document.

Set llDebugMode to .T. in Sample.prg and run it again. This uses the developer version of the error dialog.

# Implementing ErrorHandler

ErrorHandler consists of the following files:

* SFErrorMgr.vcx and vct: contains the error handling classes.
* SFConsole.vcx and vct: a runtime replacement for the VFP Command window.
* SFLocalize.vcx and vct: contains a localizer class.
* Resource.dbf, fpt, and cdx: a language resource table.
* SFCtrls.h and SFErrorMgr.h: include files used by SFErrorMgr.
* ErrorHeader.gif: an image used in SFErrorMessageDialog, one of the classes in SFErrorMgr.vcx.
* ClrHost.dll, wwDotNetBridge.dll, and wwDotNetBridge.prg: components of Rick Strahl’s wwDotNetBridge (<https://github.com/RickStrahl/wwDotnetBridge>) which provide the ability to call .NET code from VFP.
* InTry.dll: a .NET DLL used by SFErrorMgr.
* SFMail.prg, SMTPLibrary2.dll, MailKit.dll, MimeKit.dll, and BouncyCastle.Crypto.dll: components of the SFMail VFPX project (<https://github.com/DougHennig/SFMail>) used to send an email to support staff when an error occurs.
* System.app: the main component of the GDIPlusX VFP project (<https://github.com/VFPX/GDIPlusX>), used by ErrorHander to take a screen shot.
* VFPEncryption71.fll: an encryption library created by Craig Boyd (<https://tinyurl.com/5b7xzdmv>).

Add these files to your application’s project:

* SFErrorMgr.vcx
* SFConsole.vcx
* SFLocalize.vcx
* Resource.dbf (marked as included)
* ErrorHeader.gif
* wwDotNetBridge.prg
* SFMail.prg

Deploy these files along with your application’s executable:

* ClrHost.dll
* wwDotNetBridge.dll
* InTry.dll
* SMTPLibrary2.dll
* MailKit.dll
* MimeKit.dll
* BouncyCastle.Crypto.dll
* System.app (only needed if you want screen shots taken)
* VFPEncryption71.fll

# Using ErrorHandler

Early in application startup, execute code like the following:

oError = newobject('SFErrorMgr', 'SFErrorMgr.vcx', '', *AppName*, .T., 'oError')

*AppName* is 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, as it is in this case, the Init method of SFErrorMgr points ON ERROR to its own ErrorHandler method (that’s why the variable name had to be passed).

SFErrorMgr has a lot of properties that control its behavior; these properties are shown in Table 1. Most 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 the errors that have occurred since the application was started. nLastError is the index into the array for the most recent error. |
| cAlias | The selected alias. |
| cAppName | The name of the application; this is used in the subject of an email sent to developers. |
| cAttachments | A comma-separated list of attachments for the email or support ticket (in addition to attachments SFErrorLog creates). |
| cContact | The user's display name. |
| cDefaultAction | The default error recovery action to take. |
| cEmail | The user's email address. |
| cEmailLogFolder | The folder to use for email logging; ignored if lEmailLogging is .F. |
| cEncryptionLibrary | The name of the encryption library to use. Defaults to “VFPEncryption71.fll.” |
| cErrorLogFile | The name and path of the table to log errors to if lLogToTable is .T. Defaults to “Errorlog.dbf.” |
| cErrorLogTextFile | The name of the text file to log errors to. |
| cImageFile | The name and path of the image file for the screen shot. |
| cLanguage | The language to use for messages. |
| cMailClass | The class to use for emailing; cMailLibrary specifies the library containing the class. |
| cMailErrorMessage | The message of any error from the mail object. |
| cMailLibrary | The library containing the class specified in cMailClass. |
| cMailServer | The mail server. |
| cMessage | The body of the error message. |
| cMessageClass | The class to use for an error dialog; cMessageLibrary specifies the library containing the class. |
| cMessageLibrary | The class library containing the class specified in cMessageClass. |
| cPassword | The email password. |
| cRecipient | The recipient of the email. |
| cResourceTable | The name of the resource table to use. |
| cReturnToOnCancel | What to RETURN TO if the user chooses the Cancel option in the error dialog. |
| cReturnToOnQuit | What to RETURN TO if the user chooses the Quit option in the error dialog. |
| cSenderEmail | The sender's email address. |
| cSenderName | The name of the sender. |
| cSubject | The subject of the email. |
| cTitle | The default title for the error dialog. |
| cUser | The name of the user. |
| cVersion | The application version number; this is used in the subject of an email sent to developers. |
| lAutoCreateTicket | .T. to automatically create a ticket. |
| lCanCancel | .T. if the user can "cancel" (that is, stay in the application). |
| lCanContinue | .T. if the user can "continue" (that is, RETRY or RETURN) |
| lDisplayErrors | .T. if we’re supposed to display errors and get the user’s choice. |
| lDisplayingErrorDialog | .T. if we're displaying the error dialog. |
| lEmailLogging | .T. to do email logging to the folder specified in cEmailLogFolder. |
| lGetMemVars | .T. to include memory variables in the error report; .F. if an ActiveX control would cause a buffer overrun. |
| lGettingMemVars | .T. if we're using LIST MEMORY or LIST OBJECTS; an Access method can check it and do nothing if .T. |
| lHandlingError | .T. if we’re handling an error; the Timer method of a timer could use this to decide to do nothing, for example. |
| lInsideTry | .T. if code was executing inside a TRY structure when the error occurred; for backward compatibility with an earlier version. |
| lLogToTable | .T. if the error should be logged to a table. Defaults to .T. |
| lMAPI | .T. to use MAPI for the email or .F. (the default) for SMTP. |
| lQuit | .T. if we're quitting (public so other objects can check on the way out). |
| lScreenShot | .T. to take a screen shot. |
| lSetOnError | .T. to set ON ERROR to point to the ErrorHandler method. |
| lShowDebug | .T. if “debug” should be an option the user can choose to recover from the error. |
| lTicketCreated | .T. if a ticket was created. |
| lUserCanSaveDialog | .T. if the user can save the error log. |
| nLastError | The index to the last error that occurred in aErrorInfo. |
| nMaxErrors | The maximum number of entries in aErrorInfo; 0 = no limit. |
| nReturnCode | The return code for ExitProcess. |
| nSMTPPort | The SMTP port to use. |
| nSecurityOptions | The SecureSocketOptions setting to use. |
| nThrottleErrorCount | The number of errors that must occur within the timeframe specified in nThrottleThreshold to cause error suppression. |
| nThrottleThreshold | The number of seconds nThrottleErrorCount errors must occur within for error suppression to be enabled; the default is 0, meaning error suppression isn’t in effect. |
| nThrottleWindow | How long to wait when error throttling before errors suppression is disabled. |
| tThrottleTime | When error suppression started. |

# 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 can also called directly, such as from the Error method of an object.

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).

The first thing ErrorHandler does is get information about the error using AERROR() and MESSAGE(1). It then logs the fact that an error occurred if a global logging object called oLogger exists (I use that for instrumentation purposes, which is beyond the scope of this document).

To recover from the error (discussed later in the “Recovering from an error” section), ErrorHandler needs to know if there is a TRY structure anywhere on the call stack. This 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 1; if you run this code, 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 1. 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 it might do to consolidate all error handling into one place, 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, ErrorHandler calls CheckInTry, which uses wwDotNetBridge to call a .NET component (InTry.dll) written by Christof Wollenhaupt that determines whether a TRY structure is on the call stack. It does this in an ingenious way: it tries to execute \_vfp.DoCmd("return to master") within a .NET TRY structure. If it fails, there’s a TRY structure on the VFP call stack. The source for InTry.dll is in the InTry folder.

ErrorHandler next calls SetError to add the error information to the aErrorInfo array and CreateErrorMessage to package the various pieces of information about the error (error number, message, procedure or method where it occurred, and so on) into a message string.

# Error suppression

Rick Schummer once told me a humorous story. His company’s error handler sends an email to a service which automatically creates a support ticket. One day, a network problem caused error after error after error to occur, so they suddenly had a thousand support tickets created for what was ultimately a single problem.

SFErrorMgr handles this by supporting error suppression. nThrottleThreshold specifies the number of seconds a certain number of errors (specified in nThrottleErrorCount) must occur within for error suppression to be enabled. If nThrottleThreshold is set to something other than zero, ErrorHandler checks if nThrottleErrorCount or more errors have occurred within nThrottleThreshold seconds and if so, sets lSuppressErrors to .T. so no errors are logged, displayed to the user, or passed on to support staff. Suppression remains in place until the number of seconds in nThrottleWindow has passed without another error occurring.

For example, suppose nThrottleThreshold is set to 10, nThrottleErrorCount is set to 3, and nThrottleWindow is set to 3600. That means if three or more errors occur within a span of ten seconds, error suppression occurs for an hour, after which error handling goes back to normal.

You can also manually set lSuppressErrors if you don’t want errors logged, displayed to the user, or passed on to support staff.

# Logging the error

If errors aren’t being suppressed, ErrorHandler logs the error by calling LogError. LogError logs the error to the table specified in cErrorLogFile if lLogToTable is .T. (that table is created if it doesn’t exist). 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.
* Under some conditions, LIST MEMORY causes a buffer overrun error. When she encountered this problem, Tamar Granor found that using LIST MEMORY LIKE A\*, LIST MEMORY LIKE B\*, etc. helped. If you encounter this when an error occurs in a particular section of code, you can set oError.lGetMemVars to .F.; SFErrorMgr won’t use LIST MEMORY in that case.
* 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. I’ve also run into issues with the Access method of properties causing a problem, so GetMemVars sets lGettingMemVars to .T. before issuing the LIST OBJECTS command; problematic Access methods can then check oError.lGettingMemVars if necessary.
* 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>).

LogError also writes the same information to a text file which is used as an attachment to an email sent to support staff or a support ticket.

After calling LogError, ErrorHandler calls ScreenShot to take a screen shot if lScreenShot is .T. ScreenShot uses GDIPlusX to do the actual work.

# Notifying support staff

If lAutoCreateTicket is .T. or if the user clicks the Send button in the error dialog (discussed later in the “Displaying the error to the user” section), the CreateTicketOrEmail method is called to create a support ticket or send an email to support staff.

CreateTicketOrEmail creates a list of attachments: the text file created by LogError (the name of which is in cErrorLogTextFile), the screen shot created by ScreenShot (the name of which is in cImageFile if one was taken), and any additional attachments specified in cAttachments. You could set cAttachments, for example, to the path of an import if the error occurs while importing from that file. CreateTicketOrEmail then calls CreateTicket, which is an abstract method returning .F. in SFErrorMgr. If you want to create a support ticket, subclass SFErrorMgr and fill in CreateTicket with the necessary code. Be sure to return .F. if the code fails for some reason. For example, Stonefield Software uses HESK as our support ticket software, so our SFErrorMgr subclass calls the HESK API to create a support ticket, including attachments. Another popular support ticket system is FreshDesk; the code in Listing 2 creates a support ticket using the API for that system.

Listing 2. Code for CreateTicket to create a support ticket using the FreshDesk API.

lparameters tcMessage, ;

tcEmail, ;

tcContact, ;

tcAttachments

local lcAPIKey, ;

lcURL, ;

lcFolder, ;

lcLogFile, ;

lcDescription, ;

laFiles[1], ;

lnFiles, ;

lcAttachments, ;

lnI, ;

lcParameters, ;

lnResult, ;

llResult, ;

lnCount, ;

lcTicket, ;

loException

\* Get the FreshService API settings.

lcAPIKey = 'API key to use'

lcURL = 'URL to use'

\* Define where to write the output to.

lcFolder = addbs(sys(2023))

lcLogFile = lcFolder + 'CurlLog.txt'

\* Create the parameters for CURL.

lcDescription = strtran(strtran(tcMessage, chr(13), '<br/>'), chr(10))

lnFiles = alines(laFiles, tcAttachments, 4, ',')

lcAttachments = ''

for lnI = 1 to lnFiles

lcAttachments = lcAttachments + ' -F "attachments[]=@' + laFiles[lnI] + '"'

next lnI

text to lcParameters noshow textmerge

-u <<lcAPIKey>>:X -H "Content-Type: multipart/form-data" <<lcAttachments>> -F "description=<<lcDescription>>" -F "subject=<<This.cSubject>>" -F "email=<<tcEmail>>" -F "priority=1" -F "status=2" -X POST "<<lcURL>>" -o "<<lcLogFile>>"

endtext

\* Execute the command to call the FreshService API. If it succeeded, get and

\* display the ticket number.

declare Sleep in Win32API integer nMilliseconds

try

erase (lcLogFile)

lnResult = ShellExecute(0, 'Open', 'curl.exe', lcParameters, lcFolder, 0)

llResult = lnResult > 32

if llResult

lnCount = 0

do while not file(lcLogFile) and lnCount < 10

lnCount = lnCount + 1

Sleep(500)

enddo while not file(lcLogFile) ...

if file(lcLogFile)

lcTicket = strextract(filetostr(lcLogFile), '"id":', ',')

if empty(lcTicket)

llResult = .F.

else

messagebox('Support ticket ' + lcTicket + ' was created.', ;

64, \_screen.Caption)

endif empty(lcTicket)

else

llResult = .F.

endif file(lcLogFile)

erase (lcLogFile)

endif llResult

catch to loException

endtry

return llResult

If CreateTicket returns .F., that means either there is no code to create a support ticket or the process failed. In either case, CreateTicketOrEmail calls SendEmail send an email to support staff instead. SendEmail expects the email-related properties (cRecipient, cMailServer, cContact, cEmail, lMAPI, nSMTPPort, nSecurityOptions, cUserName, and cPassword) to be filled in with the correct values and uses SFMail (see <https://github.com/DougHennig/SFMail> for details about SFMail) to send the email. If you want to use something other than SFMail, subclass SFErrorMgr and override SendEmail with the appropriate code.

# Displaying the error to the user

After logging the error, ErrorHandler displays an error dialog to the user if it’s supposed to (lSuppressErrors is .F. and lDisplayErrors is .T.) by calling DisplayError. DisplayError instantiates the class specified in cMessageClass and cMessageLibrary, sets some properties, and calls Show.

The dialog shown in Figure 1 comes from SFErrorMessageDialog in SFErrorMgr.vcx while the dialog in Figure 2 comes from SFErrorMessage, also in SFErrorMgr.vcx. I use the following code in my application startup:

if llDebug

oError.cMessageClass = 'SFErrorMessage'

oError.cMessageLibrary = 'SFErrorMgr.vcx'

oError.lShowDebug = llDebug

else

oError.cMessageClass = 'SFErrorMessageDialog'

oError.cMessageLibrary = 'SFErrorMgr.vcx'

endif 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.

If you don’t like either of these dialogs, feel free to create your own and set cMessageClass and cMessageLibrary to the appropriate values. Note that the class doesn’t have to be a form class; all it needs is the properties written to by DisplayError, a Show method, and optionally a cChoice property indicating how to recover from the error.

If the user clicks the Send button, the same mechanism described in the Notifying support staff section is used.

# Localizing the error dialog

By default, all messages are displayed in English. However, ErrorHandler uses a localization object, SFLocalize in SFLocalize.vcx, to display messages in any language you wish. Set the cLanguage property of the error object to the desired language; for example, oError.cLanguage = 'French' specifies that messages should be displayed in French.

SFLocalize uses a table named Resource.dbf (see Figure 4) to hold strings displayed to the user. If you wish to change any string, edit the appropriate memo field. For example, edit the contents of French to change the French message for a string. To add a new language, add a new Memo (Binary) field to the table named for the language (for example, “Italian”), fill in that field with the appropriate string for all records, and set oError.cLanguage to the name of the field to use.

Table

Description automatically generated

Figure 4. Resource.dbf contains the strings displayed in the error message dialog.

# 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 4.
* 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 the DEBUG command is executed.
* For “Retry”, RETRY is executed.
* 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 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.

As discussed earlier, RETURN TO can’t be used if there’s a TRY structure on the call stack. In that case, the only thing we can do is cancel or quit the program.

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

* 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 (when running in the IDE) or the ExitProcess Windows API function (when running an executable), 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 EVENTS statement.
* 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 it calls ExitProcess after all the cleanup tasks are done. It passes nReturnCode as the return value, so you can set that to any value you wish in a subclass of SFErrorMgr.

# Error log analysis strategies

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

* I tend to look at the text file generated by SFErrorMgr 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 an FLL written by Craig Boyd (VFPEncryption71.fll), so obviously that FLL 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 (Thor is another VFPX project: <https://github.com/VFPX/Thor>), 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 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.

# 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>