Reporting Errors

Doug Hennig

This month's article presents an error reporting class that prompts the user to fax or email you information about errors that occur. Doug also describes improvements he's made to the error handling scheme he presented in articles two years ago.

"What?", you're probably thinking if you're a long-time FoxTalk reader. "Hasn't this guy covered error handling enough by now?" True, I have covered error handling in four different articles over the years. However, this article addresses something that wasn't covered earlier: reporting errors to the developer.

I don't know what your users are like, but if they're anything like mine, you typically get a phone call like this: "There's an error in your application" (notice that when it's working well, it's *their* application and when something goes wrong, it's *your* application).

"What was the error message?", you ask.

"I don't remember."

"OK, what were you doing when the error occurred?"

"I wasn't doing anything!"

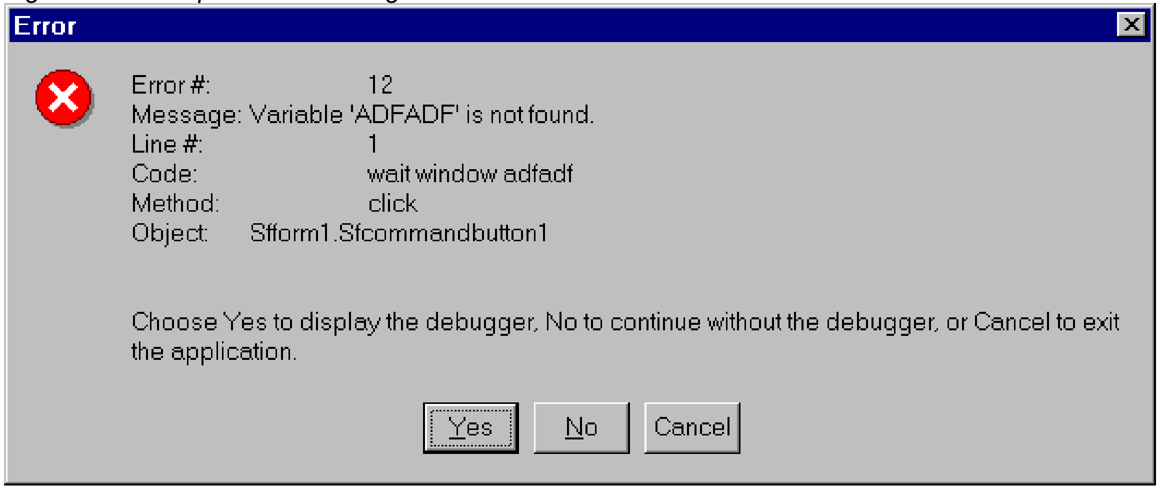
"No, I mean which function or screen were you using?"

"I don't remember. But I need you to fix the error right away."

This assumes they even report the error to you. Once I watched a user go through a process and when an error message came up, they went on by the error dialog without even reading it. "What was that message?", I asked. "Oh, it always does that. We just close the message and carry on, and everything works fine. Say, could you make that message go away? It'd make the process go quicker." It turned out there was a minor bug in the code, but no one had ever reported it. The "close the message dialog" step was just part of the process as far as they were concerned!

To make it easier for technical support and development staff to figure out what went wrong when an error occurs, we need a way to prompt the user to send us an error report. The dialog shown in Figure 1 is suitable for a developer when an error occurs because it gives them the information they need and a means of debugging the error immediately. However, it isn't suitable for a typical user because the information presented isn't useful to them (in the same way the "details" information in a GPF dialog isn't useful to most of us).

Figure 1. Developer error message.

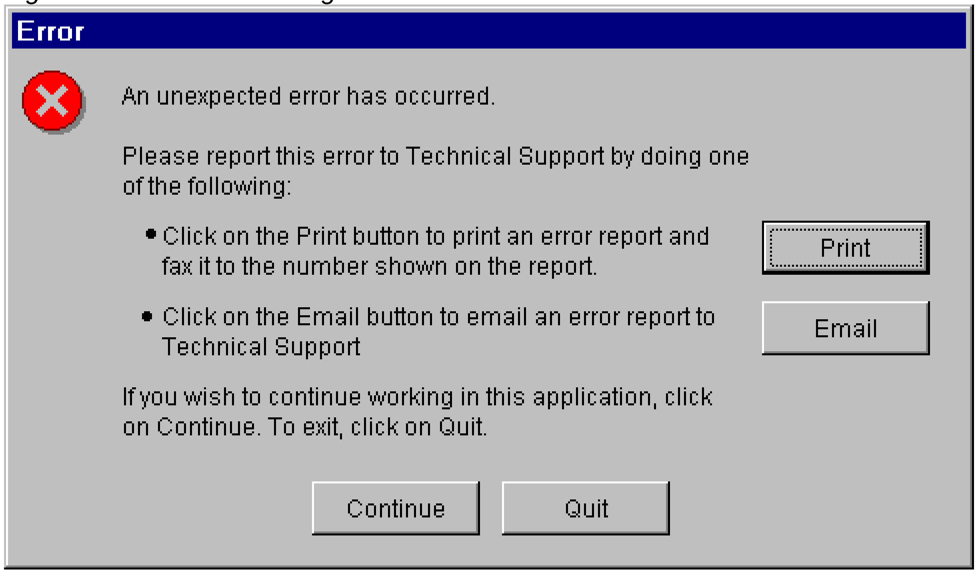


```
Error #:        12
Message: Variable 'ADFADF' is not found.
Line #:         1
Code:           wait window adfadf
Method:         click
Object:    Sfform1.Sfcommandbutton1
```

Choose Yes to display the debugger, No to continue without the debugger, or Cancel to exit the application.

Instead, I prefer to use a dialog like that shown in Figure 2. This tells the user that something unexpected has occurred (so it's not their fault) and prompts them to report it to their technical support staff (such as you). All they have to do is click one of two buttons (to either print a document they can fax or to send an email) to report it. They can then choose one of two actions: stay in the application (but not return to the method that caused the error, since that would likely cause another error to occur) or quit. This month's article will look at how this dialog was produced.

Figure 2. User error message.



Error

An unexpected error has occurred.

Please report this error to Technical Support by doing one of the following:

- Click on the Print button to print an error report and fax it to the number shown on the report.
- Click on the Email button to email an error report to Technical Support

If you wish to continue working in this application, click on Continue. To exit, click on Quit.

Error Handling Refresher and Enhancements

We'll briefly revisit the error handling mechanism to refresh your memory of how it works. For details, see my articles in the January and February 1998 issues of FoxTalk.

When an error occurs in a method of an object instantiated from a class in SFCTRLS.VCX (or a subclass of one of these classes), the Error method fires. The Error method can try to handle the error itself, or it can pass the error up the class hierarchy (from the object instance to its class, to the class' parent class, and so on) until we hit the top of the hierarchy (the class in SFCTRLS). The error is then passed up the containership hierarchy, from the object to its container, to the container's container, and so on until we hit the form the object is on. At any step in this error passing, something could decide how to handle the error and then the passing stops. Once we get to SFForm (the top class in the class hierarchy and top container in the containership hierarchy), there's no where else in the class or containership hierarchies to go, so if nothing has handled the error by now, it gets passed to a global error handler, the ErrorHandler method of an instance of the SFErrorMgr class. This error handler is also called if an error occurs in non-object code (such as a PRG) because ON ERROR calls it. This type of message passing mechanism is called a Chain of Responsibility design pattern.

SFErrorMgr doesn't really know how to handle specific errors. It just provides error handling services, like logging information about the error to a file (either a table or a text file), displaying a message to the user, and resolving the error. By "resolving", I don't mean fixing the problem; I mean what to do next: shutting down the application, returning to the method that caused the error, retrying the statement that caused the error, or staying in the application by dropping the method that caused the error off the call stack and returning to the READ EVENTS statement.

Since I described this error handling scheme more than two years ago, I've made a number of improvements to it. Some of these were things I've thought up and others were suggestions from FoxTalk readers.

Avoiding a Bogus Error Message

You may have already seen this but didn't know what caused it: when your application tries to open a non-existent file like ABC.TXT, the resulting error message is something like "File oError does not exist". But that doesn't make sense: you weren't doing anything with a file called oError; the message should've been "File ABC.TXT does not exist". The reason the error message is wrong is due to the TYPE() function; using it while in an error condition can change the error message. For example, the Error method of all classes in SFCTRLS.VCX uses TYPE('oError') to determine if a global error handling object exists. If it does, great. If not, the “parameter” part of the error message (“ABC.TXT” in this example) is overwritten with the expression evaluated by TYPE() (“oError” in this case).

The way I handled this is to save information about the error into an array (using the AERROR() function) before using TYPE(). Then, if Error determines that it’ll pass the error on to another object (such as the global error handler), it first calls the SetError method of that object (if it exists), passing it the array. Here’s an excerpt from the Error method that shows this:

```
aerror(laError)
* other code here
do case
* other cases here
  case type('oError.Name') = 'C' and ;
    pemstatus(oError, 'ErrorHandler', 5)
    if pemstatus(oError, 'SetError', 5)
      oError.SetError(lcMethod, tnLine, @laError)
    endif pemstatus(oError, 'SetError', 5)
    lcReturn = oError.ErrorHandler(tnError, lcMethod, ;
      tnLine)
```

The SetError method of SFErrorMgr saves the passed information into the aErrorInfo array property and sets the lErrorInfoSaved property to .T. When the ErrorHandler method is called to handle the error, it checks lErrorInfoSaved; if it’s .T., SetError has already been called and aErrorInfo has been filled with the correct error information. If not, it uses AERROR() to get information about the error and then calls SetError to store it in aErrorInfo.

This change (calling SetError with information gather before TYPE() is used) means that the error information is accurate rather than possibly containing a bogus message that would confuse the developer and lead them on a wild goose chase for a while.

Handling Contained Objects in Subclasses

The Error method of the classes in SFCTRLS.VCX prefixes the method the error occurred in (passed in the second parameter to Error) with the name of the object and a period so it looks like “ObjectName.Method”. As the error is passed up the containership hierarchy, each object adds its own name and a period to the start of this string. This makes it easier to track down where an error occurred because the string will ultimately be something like “frmCustomer.cntAddress.txtCity.Valid” rather than just “Valid” (which is what we’d see if we just passed the method name unchanged from object to object). This also gives us a side benefit: if the method string passed to the Error method contains a period, that means the error didn’t occur in a method of this object; instead, Error was called from a contained object as part of the Chain of Responsibility. In that case, Error won’t handle any clean up or resolution stuff but will instead RETURN to send execution back to the method that called it.

However, there’s one problem with this scheme: if you create a container class that contains several controls, then subclass that container class and drop an instance of the subclass on a form, when an error occurs in a method of one of the contained controls, the method name passed to the Error method contains the name of the control (for example, “cmdSave.Click” is passed rather than just “Click” as is normally the case). This not only messes up the method name (since we’d add the name of the object to the passed method name, we’d end up with “cmdSave.cmdSave.Click”) but confuses the code that decides if it has to process the error resolution or return it to the calling object (a period in the method string means we have to return, but in this case, we shouldn’t).

The solution is to see if the name of the object is already in the method name being passed. That won’t happen except in the case I’ve described here, so if it does, we’ll strip the object name off (because it might not be in the case we expect; sometimes, it’ll appear as “CMDSAVE.Click”), then add it back on. Here’s the appropriate code from the Error method:

```
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
```

Having the Debugger Show the Correct Method

If an error occurs in development mode, one of the options available in the error dialog is to display the debugger. This makes it very easy to track down the cause of the problem, and perhaps even get around it temporarily (for example, by declaring a missing variable in the Command window or assigning the correct value to a variable, then using the Set Next Statement function in the debugger to return to the line of code that failed). However, one issue with the previous version of SFErrorMgr is that we might be a long way from the method that caused the error (because of the Chain of Responsibility, the Error method of many objects may be on the call stack), so we had to choose the Step Out option in the debugger many times before we'd get to the correct method. In the new version, SFErrorMgr.ErrorHandler returns the string "debug" (defined in the constant ccMSG_DEBUG in SFERRORS.H, one of the include files used by all my classes) and every Error method but the first one returns this string (earlier, I mentioned that an Error method knows it has to return if there's a period in the method name it was passed). The originating Error method then displays the debugger if it receives this return string.

One final issue: because the debugger is called from Error, it'll display the code for that method, not the one that caused the error, so you have to choose Step Out once to get back to the original method. Being a lazy, er, efficient person, I prefer to avoid extra work if possible. The way to get the debugger to step out programmatically is to stuff the keypress for that function (Shift-F7) into the keyboard buffer and then suspend; as soon as execution returns to the Command window, the keypress will fire and the debugger will step out of Error and back to the original method. Due to the way the debugger is implemented (or it may be a bug), this only works if the debugger exists in its own window (the "Debugger frame" option in the Tools Options dialog, as opposed to the "FoxPro frame" choice). So, if the debugger exists in its own window, we'll do this; otherwise, we won't. Here's the applicable code from Error:

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

Error Message Dialog Classes

Now let's look at how to implement the error dialog shown in Figure 2. The ErrorHandler method of SFErrorMgr calls DisplayError to display the error message to the user. DisplayError instantiates an object from the class specified in the cMessageClass property; this property contains SFErrorMessage by default. In the previous version, SFErrorMessage was a form-based class that displayed the error message in an editbox and had command buttons so the user could select the appropriate action (Cancel, Quit, etc.). In the new version, SFErrorMessage is used only to display error messages to a developer running in development mode, so now it's based on Custom and uses MESSAGEBOX() instead.

A few changes were made to SFErrorMgr to make it more flexible. DisplayError now passes a reference to SFErrorMgr to the class it instantiates, calls a new CreateErrorMessage method to build the error message string, and calls a SetDialogProperties hook method (which has no code in this class) to possibly do other things with the message object before calling its Show method. This makes it easier to change the message class (change the values of the cMessageClass and cMessageLibrary properties); you can even subclass SFErrorMgr and put code in the SetDialogProperties method to use a message class with different needs. For example, in one application, I subclassed SFErrorMgr and had SetDialogProperties get other application-specific information and add it to the error message. Here's the new version of DisplayError:

```
loMessage = MakeObject(.cMessageClass, ;
  .cMessageLibrary, '', This)
loMessage.cTitle        = This.cTitle
loMessage.cErrorMessage = This.CreateErrorMessage()
This.SetDialogProperties(loMessage)
loMessage.Show()
lcChoice = iif(vartype(loMessage) = 'O', ;
  loMessage.cChoice, 'Cancel')
```

```
return lcChoice
```

In addition, two new properties were added to SFErrorMgr: cAppName (the name of the application the user is running) and cVersion (the application's version number). These properties, which should be set when SFErrorMgr is instantiated, help you figure out what application the user is experiencing an error in. For example, they may report a problem in version 6.1 that you've fixed in version 6.2 (they obviously haven't upgraded to that version), so you don't have to spend time tracking down the problem.

Two new classes, SFErrorMessageDialog and SFErrorMessageDialogEmail, were added to SFERRORMGR.VCX. SFErrorMessageDialog looks like Figure 2 except it doesn't have an email option. This class can be used to display error messages to a user when they don't have email capabilities (or you don't want to receive emails from them <g>). The Init method accepts a reference to the SFErrorMgr object that called it and stores it in its oErrorMgr property; we'll see how this gets used in a moment. The Click method of the Print button calls the PrintError method of the form. PrintError puts the cErrorMessage property and the cAppName and cVersion properties of the oErrorMgr object into variables (declared private rather than local so they'll be visible to the report we'll print), creates a temporary cursor with a single record (because a cursor must be open in the current workarea for an FRX to work correctly, even if the FRX doesn't use anything from the cursor), then prints the ERROR.FRX report. ERROR.FRX is a simple report; it doesn't print anything from any cursor, just some hard-coded values (like the fax number) and the lcMessage, lcAppName, and lcVersion variables. Here's the code for PrintError:

```
local lnSelect
private lcMessage, ;
  lcAppName, ;
  lcVersion
with This
  lnSelect = select()
  create cursor TEMP (FIELD1 I)
  append blank
  lcMessage = .cErrorMessage
  lcAppName = .oErrorMgr.cAppName
  lcVersion = .oErrorMgr.cVersion
  report form ERROR next 1 noconsole to print prompt
  use
  select (lnSelect)
endwith
```

SFErrorMessageDialogEmail is a subclass of SFErrorMessageDialog that adds an email option. SFErrorMessageDialogEmail has custom properties called cRecipient (who to send the email to), cMessage (the body of the message), and cSubject (the subject of the email). cRecipient is hard-coded in this example, but a better way to handle it is to subclass SFErrorMgr and in the SetDialogProperties method, set cRecipient to the appropriate email address (perhaps looking it up from an application setting, configuration file, or Registry setting). cSubject is set in Init to the string "Error in " plus the application name and version, but could also be changed in SetDialogProperties as desired. cMessage is set to cErrorMessage, but again can be changed in SetDialogProperties.

The Click method of the email button calls the SendMessage method of the form. At first, I was tempted to do something simple like use the Windows API ShellExecute function (discussed in my February 1999 column) and "mailto" to send the message using code such as:

```
lcFile = fullpath(sys(3) + '.TXT')
strtofile(This.cErrorMessage, lcFile)
lcMessage = 'mailto:support@xxtechsupport.com' + ;
  '?Subject=' + This.cSubject + ;
  '&Attach="' + lcFile + '"' + ;
  '&Body=See attachment for error details'
ShellExecute(lcMessage)
```

Unfortunately, I couldn't get this to work correctly; although the email was sent, it never had an attachment. Also, it didn't actually send the message; it popped up the Send Message dialog with everything filled in, but the onus was on the user to click on the Send button. So, instead, emailing is performed with the

SFMAPI class I presented in my column in the February 2000 issue of FoxTalk, although you could use another tool such as West Wind Technologies' wwIPStuff instead if you wish.

SendMessage calls the AddRecipient method of the SFMAPI object (named oMail on the form) to set the recipient, and sets the subject and message properties as well. Then it does something interesting: it tells the SFErrorMgr object that instantiated this form to log the error information to a temporary text file and uses that text file as an attachment to the email. This results in a lot more information being sent to you, since the log file contains stuff like the current values of variables (using LIST MEMORY), trigger information if a trigger failed, etc. Finally, SendMessage tells SFMAPI to send the message.

```
local lcAttachment, ;
  llLog, ;
  lcLog
This.cMessage = iif(empty(This.cMessage), ;
  This.cErrorMessage, This.cMessage)
with This.oMail
  .AddRecipient(This.cRecipient)
  .cSubject = alltrim(This.cSubject)
  .cMessage = alltrim(This.cMessage)
endwith

* Have the error handler create a text file with error
* information that we'll attach to the message.

lcAttachment = sys(3) + '.TXT'
with This.oErrorMgr
  llLog          = .lLogToTable
  lcLog          = .cErrorLogFile
  .lLogToTable   = .F.
  .cErrorLogFile = lcAttachment
  .LogError()
  .lLogToTable   = llLog
  .cErrorLogFile = lcLog
endwith
This.oMail.AddAttachment(lcAttachment)

* Send the message.

This.oMail.Send()
erase (lcAttachment)
```

Trying it Out

TEST.PRG demonstrates how to use the error handler and select which message class is used. This program instantiates SFErrorMgr, set the necessary properties, then calls the TEST form. Click on the "Click Me" button on this form; it has two invalid statements in its Click method, so you'll see the error message dialog.

```
public oError
oError = newobject('SFErrorMgr', 'SFErrorMgr')
with oError
  .cAppName           = 'Test Application'
  .cVersion           = '1.0'
  .cReturnToOnCancel = 'test.prg'
  .cMessageClass      = 'SFErrorMessageDialogEmail'
  .cUser              = 'DHENNIG'
  .lShowDebug         = .F.
endwith
do form TEST
read events
clear all
```

Try setting cMessageClass to SFErrorMessage or SFErrorMessageDialog to see the differences. Also, if you use SFErrorMessage, set lShowDebug to .T. to see how the debugger option works.

Conclusion

Providing an easy way for your users to report unexpected errors to you can make fixing these errors simpler and assure the user that it wasn’t their fault. I hope you find these classes useful!

Doug Hennig is a partner with Stonefield Systems Group Inc. in Regina, Saskatchewan, Canada. He is the author or co-author of Stonefield’s add-on tools for FoxPro developers, including Stonefield Database Toolkit, Stonefield Query, and Stonefield Reports. He is also the author of “The Visual FoxPro Data Dictionary” in Pinnacle Publishing’s “The Pros Talk Visual FoxPro” series. Doug has spoken at the 1997, 1998, and 1999 Microsoft FoxPro Developers Conferences (DevCon) as well as user groups and regional conferences all over North America. He is a Microsoft Most Valuable Professional (MVP) and Microsoft Certified Professional (MCP). He can be reached at dhennig@stonefield.com.