# But It Works For Me!

*Doug Hennig*

**Figuring out why a user is getting different results than you are can be a frustrating, time-consuming task. This month, Doug looks at a way of instrumenting your application so you can quickly determine what is being executed and how long it's taking.**

Recently, I got an email from a customer, complaining that a particular process in our application was "taking forever" (it actually turned out to be 30 seconds, showing that time is definitely relative). Of course, on my system, that process only took two seconds, which led to my favorite answer to support questions: "That's weird, it works for me."

There can be a lot of reasons why a process on a customer's system takes much longer than it does on yours: data size, hardware issues, network performance, server performance, existence of anti-virus software, etc. Because it's very time-consuming to look at all of the variables, especially ones outside my control, I decided to instrument the application so I can determine what is happening when and how long various tasks take.

What is "instrumenting"? It means to log the execution of various points ("milestones") in your application. In the September 1997 issue of FoxTalk, Rod Paddock wrote an article ("Instrumenting FoxPro Applications") on instrumenting applications, albeit to determine which parts of the application are being used and which aren't. (You can read this article online at http://www.pinnaclepublishing.com/FT/FTmag.nsf/Index/3563D5B0F068BBAA852568E800769C91?opendocument.)

VFP actually comes with a built-in instrumenting tool: the SET COVERAGE command. This command creates a log file showing information about every statement that's executed. Here's an example:

```
0.018225,sfdataengine,sfdataengine.executesqlselect,
  24,d:\tools\sfreports\sfreports.vct,8
0.000144,sflogger,sflogger.logmilestone,
  2,d:\tools\sfcommon\sflogger.vct,9
```

This shows the execution of two lines of code. The first value on each line is the number of seconds the statement took to execute, the second is the name of the class, the third is the procedure or method (in the case of a class, it shows Class.Method), the fourth is the line number, the fifth is the name of the file containing the class or procedure, and the last is the stack level. So, we can see that the first statement, which was on line 24 of SFDataEngine.ExecuteSQLSelect, took 0.018225 seconds, and the second, which was on line 2 of SFLogger.LogMilestone, took 0.000144 seconds.

Of course, you wouldn't want to look at the raw data in the log file and try to figure out what's going on. There's too much data (a log file for even a brief run of your application can have thousands of lines), making it hard to see the forest for the trees. Fortunately, starting with version 6, VFP comes with a tool called the Coverage Profiler that can analyze the data in the log file and tell you what portions of the application were run and which weren't, how many times each line of code was executed, and how long it took.

While it's a great tool for pinpointing bottlenecks in your code and determining which code you've tested, I decided not to use this feature when trying to track down the cause of my problem. There were a couple of reasons for this decision. First, the log file is too finely-grained. I wasn't interested in how fast each line of code executed at this point; I only wanted to know how long each method took. Second, and more importantly, SET COVERAGE is ignored in a runtime environment, so it wouldn't help me track down the problem at my customer's office.

So, I implemented my own code instrumentation. Like SET COVERAGE, I decided to output the results to a text file rather than a DBF, because it's easy to email and I can open it in my email client without having to save it to disk if I don't want to. I started putting STRTOFILE() statements into various places in my code, but quickly realized that a helper class would be a better way to go.

## SFLogger

The class I use for instrumenting an application is SFLogger, contained in SFLogger.VCX. SFLogger is a subclass of SFCustom, my base Custom class contained in SFCtrls.VCX. SFLogger has four custom properties: cLogFile, which contains the name of the file to log to; lLoggingEnabled, which is .T. if logging should occur; tLastMilestone, a protected property containing the DateTime of the last milestone; and aStartTime, a protected array containing the starting times of various processes.

At the start of your application, instantiate SFLogger into a globally visible container, such as a PRIVATE or PUBLIC variable or a property of an application object. Set the cLogFile property to the name of the log file and the lLoggingEnabled property to .T. to enable logging. Since you don't normally want logging enabled (you only want to turn it on to track down a problem, then turn it off again), one easy way to do this is to check the existence of a file that normally doesn't exist. For example, I use code like the following:

```
oLogger = newobject('SFLogger', 'SFLogger.vcx')
oLogger.cLogFile = 'DIAGNOSTIC.TXT'
oLogger.lLoggingEnabled = file('LOG.TXT')
```

If I want to enable logging, I simply create a file called LOG.TXT and run the application, then look at the log results in DIAGNOSTIC.TXT. When I'm done logging, I delete LOG.TXT.

To log that you're at a certain place in the code, call the LogMilestone method, passing it a message with information about the milestone. Here's the code for that method:

```
lparameters tcMilestone
local lcMessage
with This
  if .lLoggingEnabled
    lcMessage = .GetHeaderMessage() + tcMilestone
    .LogToFile(lcMessage)
  endif .lLoggingEnabled
endwith
```

GetHeaderMessage is a protected method that returns the header text to use in the log file for the milestone. (ccCRLF is a constant defined as CHR(13) + CHR(10), a carriage return and linefeed.)

```
local ltTimeStamp, ;
  lcMessage
ltTimeStamp = datetime()
lcMessage   = '===>' + ttoc(ltTimeStamp) + ;
  iif(empty(This.tLastMilestone), '', ;
  ' (' + transform(ltTimeStamp - This.tLastMilestone) + ;
  ' seconds since previous milestone)') + ccCRLF
This.tLastMilestone = ltTimeStamp
return lcMessage
```

LogToFile does the actual work of writing to the log file, appending the new text to any existing text in the file.

```
lparameters tcMessage
strtofile(tcMessage + ccCRLF + ccCRLF, This.cLogFile, ;
  .T.)
```

Here's an example that calls LogMilestone:

```
oLogger.LogMilestone('SFDataEngine.PerformQuery: ' + ;
  'about to retrieve data')
```

This adds an entry to the log file that looks like this:

```
===>07/29/2003 12:45:56 PM (2 seconds since previous
milestone)
SFDataEngine.PerformQuery: about to retrieve data
```

Note the number of seconds since the previous milestone. This helps determine how long a process takes to run. However, if you have several milestones and want to track the total time of all of them, use the StartProcess and LogElapsedMilestone methods. StartProcess simply records the starting time by adding it to the end of the aStartTime array:

```
local lnItem
with This
  lnItem = iif(empty(.aStartTime[1]), 1, ;
    alen(.aStartTime) + 1)
  dimension .aStartTime[lnItem]
  .aStartTime[lnItem] = seconds()
endwith
```

LogElapsedMilestone is similar to LogMilestone, but pops the last starting time off the bottom of aStartTime and outputs the elapsed time for the milestone. It normally doesn't add a header line to the log file, since you may have logged the start of the process by calling LogMilestone. However, pass .T. for the second parameter if you do want a header.

```
lparameters tcMilestone, ;
  tlIncludeHeader
local lnEnd, ;
  lnItems, ;
  lnStart, ;
  lcMessage
with This
  if .lLoggingEnabled
    lnEnd   = seconds()
    lnItems = alen(.aStartTime)
    lnStart = .aStartTime[lnItems]
    if lnItems = 1
      dimension .aStartTime[1]
      .aStartTime[1] = 0
    else
      dimension .aStartTime[lnItems - 1]
    endif lnItems = 1
    lcMessage = iif(tlIncludeHeader, ;
      .GetHeaderMessage(), '') + tcMilestone + ;
      ccCRLF + 'Time to run: ' + ;
      transform(lnEnd - lnStart) + ' seconds'
    .LogToFile(lcMessage)
  endif .lLoggingEnabled
endwith
```

Here's an example that calls StartProcess to record the starting time and LogElapsedMilestone to log the end of the process:

```
oLogger.LogMilestone('SFDataEngine.PerformQuery: ' + ;
  'about to retrieve data')
oLogger.StartProcess()
* some code here
oLogger.LogElapsedMilestone('Data retrieval ' + ;
  'complete: ' + transform(reccount()) + ;
  'records retrieved')
```

Here's the result in the log file:

```
===>07/29/2003 12:45:56 PM (2 seconds since previous
milestone)
SFDataEngine.PerformQuery: about to retrieve data

Data retrieval complete: 7396 records retrieved
Time to run: 0.020 seconds
```

## Instrumenting your application

Now that we have a class to help us with instrumentation, how do we use it? In the case of my application, I added calls to SFLogger methods in the most important parts of the application, places where something could go wrong if environmental conditions aren't right or where processing can be slow under some conditions. That way, if a customer is having a problem, all I do is ask them to create a text file called LOG.TXT, run the application, and email me DIAGNOSTIC.TXT.

After adding instrumentation to my application, sending it to my customer, having them run the application, and analyzing the resulting log file, I discovered the reason that process was taking so long on their system: it was doing some unnecessary work, which doesn't really affect the performance until you have a large data set (they had over 300,000 records).

Here are some other ideas for using SFLogger:

- If you're worried about the effect of calls to SFLogger.LogMilestone on your application's performance, don't be: it takes 0.018161 seconds on my system when logging is enabled and 0.000829 seconds when logging is disabled.
- Wondering why a SQL SELECT statement takes too long? Use SYS(3054) to show what VFP's doing behind the scenes:

```
sys(3054, 12, 'lcShowPlan')
oLogger.StartProcess()
* do SQL SELECT here
oLogger.LogElapsedMilestone('SQL SELECT statement:' + ;
  chr(13) + chr(10) + transform(reccount()) + ;
  ' records retrieved' + chr(13) + chr(10) + ;
  strtran(lcShowPlan, chr(13), chr(13) + chr(10)), .T.)
sys(3054, 0)
```

- Part of our application dynamically generates SQL SELECT statements. Sometimes, customers will report to us that they aren't getting the results they expect. By turning on logging and having the log file include the SQL SELECT statement, the name of the database being queried, and other important values, we usually discover that the user is pointing to the wrong database, that they didn't create their filter properly (for example, forgetting to use parentheses when mixing AND and OR conditions), or even that they have small amounts of corrupted data.

## Summary

Instrumenting my applications has allowed me to quickly discover why a user sees different results than I do. Sometimes, I have to fix an obscure bug or fine-tune some code. Other times, I can point out that the user hasn't set something up correctly. Either way, this mechanism has saved me a lot of time and frustration in providing support to my customers. I hope you find it as useful as I have.

*Doug Hennig is a partner with Stonefield Systems Group Inc. He is the author of the award-winning Stonefield Database Toolkit (SDT) and Stonefield Query, author of the CursorAdapter and DataEnvironment builders that come with VFP 8, and co-author of "What's New in Visual FoxPro 8.0", "What's New in Visual FoxPro 7.0", and "The Hacker's Guide to Visual FoxPro 7.0", from Hentzenwerke Publishing. Doug has spoken at every Microsoft FoxPro Developers Conference (DevCon) since 1997 and at user groups and developer conferences all over North America. He is a Microsoft Most Valuable Professional (MVP) and Certified Professional (MCP). Web: www.stonefield.com and www.stonefieldquery.com Email: dhennig@stonefield.com*