July, 2005

Use BINDEVENT() for invisible logging

Binding logging code to application code lets you track your application without changing its code

by Tamar E. Granor, Technical Editor

When I first learned about the inclusion of BINDEVENT() in VFP 8, I had a hard time figuring out why I would use it. Why not just call the code I want when I want it to run? It took a while for the light to go on, but finally a couple of useful cases came to mind.

The first situation where BINDEVENT() is handy is when you can't change the source code. If you don't have the source or it comes from a third-party, binding may be the only way to hook your own code in.

This realization led me to the second case, times when you may have access to the source, but adding the necessary calls would be onerous or would make your code unwieldy. In particular, I think this applies to progress reporting and logging application activity. Both showing progress and logging are removed from the main purpose of your code and what you want to do is likely to change over time, especially for logging.

I've now used BINDEVENT() successfully for both tracking progress and logging application activity. In the case of logging, I've both tacked it on as a debugging measure and built it in from the start. In this article, I'll show you how to easily set up logging code without changing the actual application code.

A Logger base class

The whole thing starts with a class capable of creating a log. Since you might want to do different things with the log (create a text file, send it to the Debug Output window, create a web page, etc.), the base class is abstract and quite simple:

```
DEFINE CLASS Logger AS Custom

PROCEDURE LogIt(cString as Character)
* Abstract here
ENDPROC

ENDDEFINE
```

I've created two subclasses of the Logger class: LogToDebugOut sends log information to the Debug Output window, while LogToFile sends it to a text file. Here's the code for LogToFile; all three classes are included on this month's Professional Resource CD (PRD) in Logger.PRG.

```
DEFINE CLASS LogToFile AS Logger

cFile = ""
lStartOver = .T.

PROCEDURE Init(cFile as Character)

IF NOT EMPTY(cFile)
   This.cFile = m.cFile
ENDIF

RETURN
ENDPROC

PROCEDURE LogIt(cString as Character)
#DEFINE CRLF CHR(13) + CHR(10)

STRTOFILE(cString + CRLF, This.cFile, ;
          NOT This.lStartOver)
This.lStartOver = .F.

RETURN
ENDPROC
```

While you could simply instantiate one of the Logger subclasses and call the LogIt method directly in your application, BINDEVENT() means you don't have to. Instead, you use another class that instantiates the appropriate Logger subclass and has one or more methods you can bind to the application methods. This class instantiates one of the Logger subclasses and uses it for logging.

Logging the Toolbox

To demonstrate what you can do with logging, I'll use the Toolbox as an example. Introduced in VFP 8, the Toolbox is a very cool example of what you can do with VFP. You'll find the code for it in Tools\XSource\VFPSource\Toolbox, once you unzip the file Tools\XSource\XSource.ZIP. (To learn how to use the Toolbox, see Cindy Winegarden's articles in the January, April and June 2003 issues of FoxPro Advisor.)

The code for the Toolbox is complex and while trying to understand what's going on, you aren't likely to want to modify any of it. Logging

its events and methods provides the opportunity to see how your actions are processed.

The Toolbox has two class hierarchies, one for the user interface and one for the underlying engine. Each has methods you might want to log. When you run the toolbox, the main user interface class is instantiated and a reference stored in the variable _oToolbox. That object instantiates the engine object and stores a reference to it in its oToolboxEngine property.

We'll look at two different approaches to logging the Toolbox's events.

Generic logging

The first way to log an object's events is to have a generic method that uses AEVENTS() to find out why it was called and logs a fairly generic message based on the results. In this case, the class you need (LogGeneric.PRG on the PRD) looks like this:

```
DEFINE CLASS LogApplication AS Custom

oLogger = .null.

PROCEDURE Init(cLogClass as Character, ;
               cLogClassLib as Character, ;
               cLogParams as Character)

IF EMPTY(cLogParams)
   * No parameters to pass to Logger class
   This.oLogger = NEWOBJECT(cLogClass, cLogClassLib)
ELSE
   This.oLogger = NEWOBJECT(cLogClass, cLogClassLib, ;
                            "", cLogParams)
ENDIF

ENDPROC

PROCEDURE LogAny(p1, p2, p3, p4, p5, p6, p7, p8, p9, p10)
* Log any method or event
#DEFINE CRLF CHR(13) + CHR(10)

LOCAL aCallingEvent[1], cLogString, nParmCount

nParmCount = PCOUNT()

* Find out who called
AEVENTS(aCallingEvent, 0)

cLogString = aCallingEvent[1].Name + "." + ;
             aCallingEvent[2]
cLogString = cLogString + " was called with "
```

```
cLogString = cLogString + TRANSFORM(nParmCount) + ;
             " parameters:" + CRLF

FOR nParm = 1 TO nParmCount
   cLogString = cLogString + "  Parameter " + ;
                TRANSFORM(nParm) + ":"
   cParm = "p" + TRANSFORM(nParm)
   cLogString = cLogString + ;
                TRANSFORM(EVALUATE(cParm)) + CRLF
ENDFOR

This.oLogger.LogIt(cLogString)

RETURN
ENDPROC

PROCEDURE BindIt(oObject as Object)
* Bind methods of logged object to LogAny method
* of this class

LOCAL nObjCount, nMember, nMatch
LOCAL aObjMembers[1]

nObjCount = AMEMBERS(aObjMembers, oObject, 1)

FOR nMember = 1 TO nObjCount
   IF INLIST(aObjMembers[nMember, 2], "Event", "Method")
      BINDEVENT(oObject, aObjMembers[nMember, 1], ;
                This, "LogAny")
   ENDIF
ENDFOR

RETURN
ENDPROC

ENDDEFINE
```

The Init method receives the name and class library for the Logger subclass to use, along with any parameters for that subclass. It then instantiates the Logger subclass, saving the reference to a custom property, oLogger.

The LogAny method does the actual logging. It uses AEVENTS() to figure out why it was called and then builds a string to send to the log. LogAny can be bound to any method that accepts up to 10 parameters. If you have code with more than that, add more parameters to LogAny. (Actually, if you have methods with more than 10 parameters, consider redesigning them to use fewer parameters and to take more advantage of properties.)

The BindIt method accepts an object as a parameter and binds every method or event of that object to the LogAny method, so every time a method or event of that object fires, the action is logged.

To demonstrate the technique in use, here's a short program (BindToolboxGeneric.PRG on the PRD) that runs the Toolbox, instantiates the LogApplication class and then binds the events and methods from the toolbox's two key objects to the LogAny method:

```
PUBLIC oLog

DO (_toolbox)
oLog = NEWOBJECT("LogApplication","LogGeneric.PRG", "", ;
                 "LogToFile","Logger.PRG", "Toolbox.log")
oLog.BindIt(_oToolbox.oToolboxEngine)
oLog.BindIt(_oToolbox)
```

The oLog variable here is public because this program has no wait state; a local or private variable would go out of scope at program end, and logging would be turned off. If you're binding to objects in a normal application, you just need to make sure that your logging object gets created with appropriate scope. (You might choose to create a logging object as part of your application object, so that it's always available. In that case, to use it, you only need the calls to the BindIt method.)

After running the program, take some actions with the Toolbox. When you're done, if you examine the file Toolbox.log, you'll see entries like this:

```
ToolboxEngine.GETRECORD was called with 1 parameters:
  Parameter 1:microsoft.ffc

ToolboxEngine.RUNADDINS was called with 2 parameters:
  Parameter 1:microsoft.ffc
  Parameter 2:ONLOAD

TOOLBOX.PAINT was called with 0 parameters:

TOOLBOX.RESIZEALL was called with 0 parameters:

TOOLBOX.RENDERCATEGORY was called with 1 parameters:
  Parameter 1:(Object)
```

Of course, the exact contents of the log file depend on what actions you take in the Toolbox.

Application-specific logging

The second approach to logging is to create a separate logging class for each application and log only the events and methods you're interested in. An application-specific logging class needs a method for each method you want to track; name each method "Log" plus the name of the method to be logged.

We again start with an abstract class. It's similar to the LogApplication class above, but it doesn't include the LogAny method. In addition, the BindIt method here does a little more work. As in the first version, you pass it an object reference; in this version, it finds all methods of the object you passed for which there's a corresponding Log method in the logging object and binds them. Both this top-level class and the toolbox-specific class are included in LogToolbox.PRG on the PRD:

```
DEFINE CLASS LogApplication AS Custom

oLogger = .null.

PROCEDURE Init(cLogClass as Character, ;
               cLogClassLib as Character, ;
               cLogParams as Character)

IF EMPTY(cLogParams)
   * No parameters to pass to Logger class
   This.oLogger = NEWOBJECT(cLogClass, cLogClassLib)
ELSE
   This.oLogger = NEWOBJECT(cLogClass, cLogClassLib, ;
                            "", cLogParams)
ENDIF

ENDPROC

PROCEDURE BindIt(oObject as Object)
* Bind methods of logged object to methods of this class

LOCAL nObjCount, nLogCount, nMember, nMatch
LOCAL aObjMembers[1], aLogMembers[1]

nObjCount = AMEMBERS(aObjMembers, oObject, 1)
nLogCount = AMEMBERS(aLogMembers, This, 1)

FOR nMember = 1 TO nObjCount
   IF INLIST(aObjMembers[nMember, 2], "Event", "Method")
      * Does the log have a corresponding method?
       * Case-insensitive search, EXACT ON, return row
      nMatch = ASCAN(aLogMembers, "Log" + ;
                     aObjMembers[nMember, 1], ;
                     -1, -1, 1, 15)
      IF nMatch <> 0
```

```
         BINDEVENT(oObject, aObjMembers[nMember, 1], ;
                   This, aLogMembers[ nMatch, 1])
      ENDIF
   ENDIF
ENDFOR

RETURN
ENDPROC

ENDDEFINE
```

The logging code for each method to be logged is quite simple. The method you create must receive the same parameters as the method it's logging. Other than that, there's just one line in the method—a call to the LogIt method of the oLogger object. (Of course, you can use additional code to build the string to send to the log.) For example, here's the Log method for the ToggleAlwaysOnTop method of the _toolboxform class:

```
PROCEDURE LogToggleAlwaysOnTop(lAlwaysOnTop)

This.oLogger.LogIt("Toggling always on top setting: " ;
                   + TRANSFORM(m.lAlwaysOnTop))

RETURN
ENDPROC
```

This approach gives you more control over the log content. In the generic version, the message is the same for each logged method. With a specific logging class, you can customize the message based on the method called.

Having separate Log methods also lets you control which methods are logged. With the generic Log approach, you can end up with extremely large logs. With this approach, only the methods and events for which you create a corresponding Log method are logged.

The code to use the specific log class (BindToolbox.PRG on the PRD) is essentially the same as the code to use the generic log. You just need to specify the appropriate class. Like the earlier example, this code runs the toolbox, creates the logger and binds the same two toolbox objects. However, in this case, only methods for which there's a corresponding Log method are bound:

```
PUBLIC oLog

DO (_toolbox)
oLog = NEWOBJECT("LogToolbox","LogToolbox.PRG", "", ;
                 "LogToFile","Logger.PRG", "Toolbox.log")
oLog.BindIt(_oToolbox.oToolboxEngine)
```

```
oLog.BindIt(_oToolbox)
```

How the binding works

The BindIt method is the key to both versions. In each case, AMEMBERS() gets a list of PEMs for the object whose events and methods are to be bound. In the application-specific version, a second call to AMEMBERS() provides a list of methods of the logging object.

The code loops through the members of the object being bound. If the member is an event or method, the code acts. For the generic version, it simply binds that event or method to the LogAny method of the logging object, with this line:

```
BINDEVENT(oObject, aObjMembers[nMember, 1], ;
          This, "LogAny")
```

The application-specific code checks whether the logging object has a method with the same name as the logged object's event or method prefixed with "Log". If so, the event or method to be logged is bound to the corresponding Log method. This code does the search and the binding:

```
nMatch = ASCAN(aLogMembers, "Log" + ;
               aObjMembers[nMember, 1], -1, -1, 1, 15)
IF nMatch <> 0
   BINDEVENT(oObject, aObjMembers[nMember, 1], ;
             This, aLogMembers[ nMatch, 1])
ENDIF
```

The best of both worlds

You don't really have to choose either the generic or the application-specific approach; you can combine the two. One possibility is to check for a specific Log method corresponding to an object's event or method. If one exists, bind to it. If not, bind to a generic log method.

A second possibility is to stick with the generic approach, but limit it to logging only events and methods of interest. In this approach, you need to specify which items to log using, for example, an array or an XML string. The BindIt method would check each event and method of the logged object to see whether it should be logged.

You can even combine these two options to provide specific logging for some methods, generic logging for some others, and no logging at all for a third group.

Another way you could extend this code is to stick with the generic LogAny, but use a case statement to let you build more specific strings to send to the log in some cases. The best way to do this is to add a call in LogAny to another method, abstract in the LogApplication class, and then subclass as needed.

Put it to work

While setting up a two-level logging system like this takes a little effort, the payoff is that you can turn logging on and off without touching your application's code. That makes it much easier to track down problems, whether they occur while you're debugging or after an application is deployed.