



# Using the Visual FoxPro Debugger Effectively

*Tamar E. Granor*

*Tomorrow's Solutions, LLC*

*Website: [www.tomorrowssolutionsllc.com](http://www.tomorrowssolutionsllc.com)*

*Email: [tamar@tomorrowssolutionsllc.com](mailto:tamar@tomorrowssolutionsllc.com)*

*Debugging is a part of every developer's day. Use the right tools for the job and you can spend more time coding and less time tracking down bugs. Visual FoxPro's powerful debugger gives you many tools for figuring out why your code doesn't work or doesn't produce the right results. In this session we'll dig into the debugger and learn how to make the most of its tools to speed up development, improve your code, and understand Visual FoxPro better.*

## Introduction

If life were perfect, we wouldn't need a Debugger. Alas, life isn't perfect and neither is our code. So good tools to help us figure out what's happening and why are invaluable.

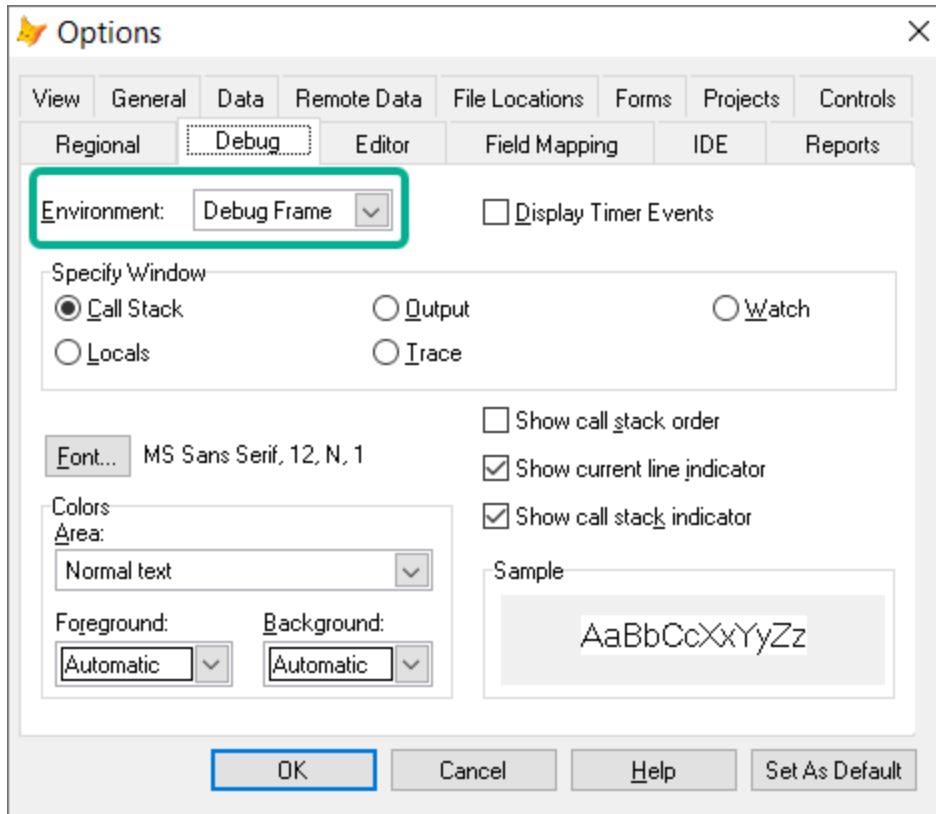
The Visual FoxPro Debugger fits into the good tools category. It's both powerful and flexible. Each new version of VFP has improved the debugging tools available.

The Debugger in its current form was introduced in Visual FoxPro 5.0. Prior to that, there were many fewer debugging tools available in FoxPro.

## Chocolate or Vanilla?

Before even starting the Debugger, you have one major choice to make-where to run it. The Debugger can run in the FoxPro frame, that is, sharing the main FoxPro window with the code you're testing. Alternatively, it can run in its own frame, that is, in a separate window that has its own task bar existence. Even when the Debugger is running in its own frame, however, its life is the same as VFP's. Closing VFP closes the Debugger.

You specify the frame for the Debugger on the Debug page of the Options dialog (available from the Tools menu and shown in **Figure 1**). As with other items in this dialog, changes apply to the current session only, unless you press the Set As Default button.



**Figure 1.** Debugger options: This page of the Options dialog lets you configure the Debugger. Your first choice is specified with the Environment dropdown: it lets you decide whether Debugger windows appear in the main FoxPro frame or in a separate frame.

How do you decide which way to set things up? To a great extent, it's a matter of personal preference. There are several menu options (Fix, Load Configuration, Save Configuration) that are available only in the Debug frame, but otherwise the functionality is equivalent. Your best bet is to try it both ways and see which one feels more comfortable for you.

## Starting the Debugger

There are lots of ways to get the Debugger running, but they vary a little depending on whether you're using the FoxPro frame or the Debugger frame. When you're set for the Debugger frame, issuing the Debug command or choosing Tools-Debugger from the menu opens the Debugger as you last had it configured. With the FoxPro frame chosen, the Debug command opens the Trace and Watch windows (see [Debugger Windows](#) below) where you last left them and the Tools menu lists each of the Debugger windows, so you can open them individually.

Issuing SET STEP ON opens the Trace window (for the FoxPro frame) or the Debugger window (for the Debugger frame). Setting a breakpoint in a code window (see [Breakpoints](#) below) has the same effect as the Debug command.

Finally, when you're using the Debugger frame, choosing Suspend from the dialog that appears when an error occurs opens the Debugger.

## Debugger Windows

The Debugger has five main windows, and several auxiliary tools. The windows are Trace, Watch, Locals, Call Stack and Debug Output. The tools are Breakpoints, Event Tracking and Coverage Logging.

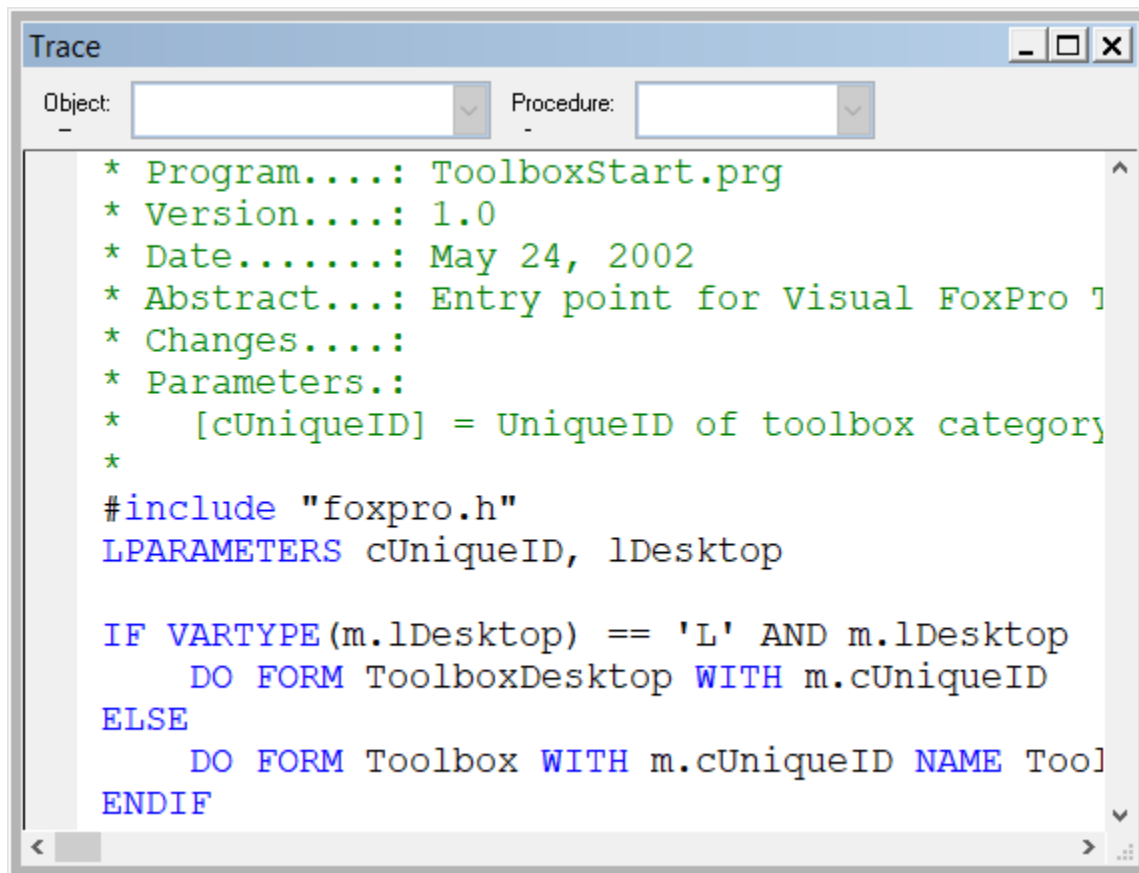
The five main Debugger windows are all dockable. In VFP 7 and later, when working in the FoxPro frame, the windows can be tab docked and link docked, as well.

VFP's Find dialog works in all of the Debugger's windows, except for Call Stack. You can open it with CTRL+F or from the Edit menu; if you're using the Debug frame, you can open it from VFP's Edit menu or the Debugger's Edit menu.

Here's a look at each of the main windows and their corresponding context menus.

### The Trace window

Trace (**Figure 2**) allows you to see the code that's being executed, and to step through it. Using the [Call Stack window](#), you can actually show the code from any routine in the call stack. You can also open any program and display its code in the Trace window.



The screenshot shows the 'Trace' window in Visual FoxPro. At the top, there are two dropdown menus labeled 'Object:' and 'Procedure:'. Below them is a text area containing the following code:

```
* Program.....: ToolboxStart.prg
* Version.....: 1.0
* Date.....: May 24, 2002
* Abstract...: Entry point for Visual FoxPro 7
* Changes.....:
* Parameters.:
*   [cUniqueID] = UniqueID of toolbox category
*
#include "foxpro.h"
LPARAMETERS cUniqueID, lDesktop

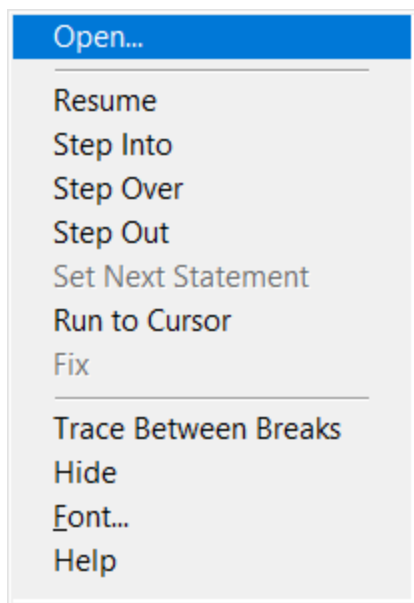
IF VARTYPE(m.lDesktop) == 'L' AND m.lDesktop
    DO FORM ToolboxDesktop WITH m.cUniqueID
ELSE
    DO FORM Toolbox WITH m.cUniqueID NAME Tool
ENDIF
```

**Figure 2.** The Trace window: This window shows you the code that's currently running and allows you to step through it, as well as set breakpoints.

You can hold the mouse over any variable (including an array) in the code to see that variable's current value. Starting in VFP 9, you can also hold the mouse over a defined (#DEFINE) constant to see the value of that constant. This feature is a little strange. What you actually get is the value of all constants in that line of code; apparently, distinguishing defined constants from actual values in the code is a difficult problem.

You can drag code out of the Trace window into the [Watch window](#) or any code editing window.

The Trace window has a context menu (**Figure 3**) that includes choices for navigating within the running program. The navigation items are described in the section [Stepping Through Code](#) below.



**Figure 3.** Context menu for tracing code: This context menu is available from the Trace window, and lets you control the running code.

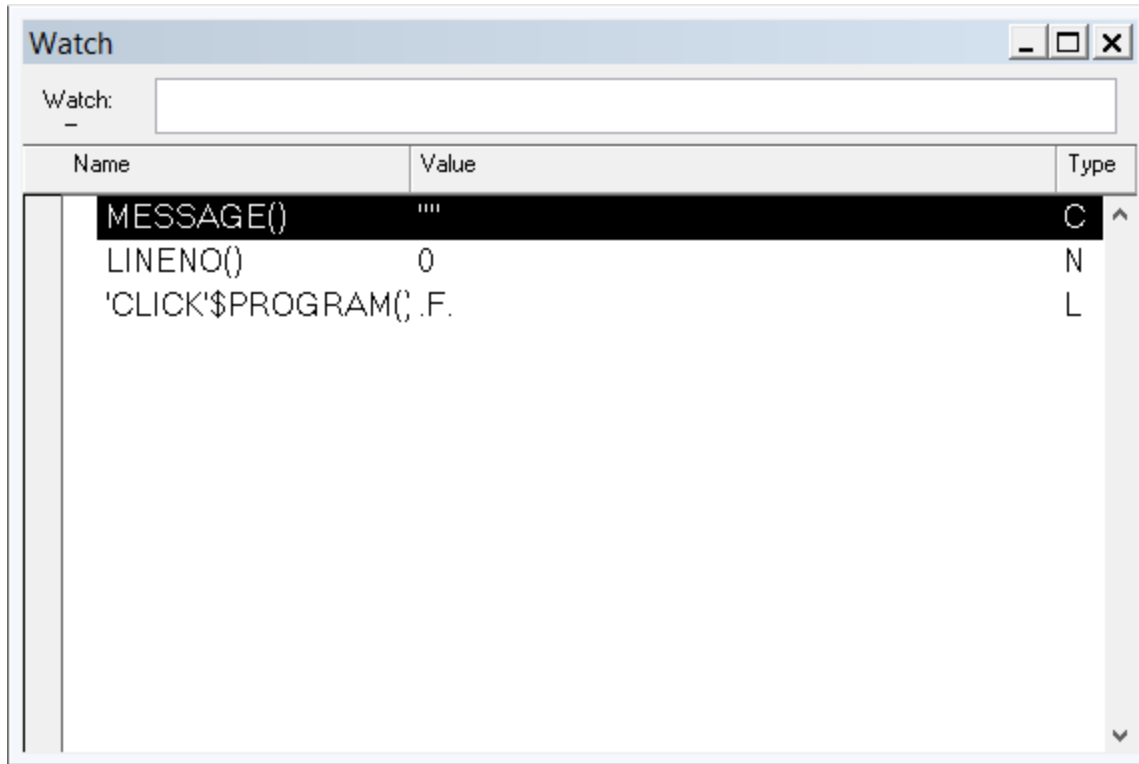
The Open command handles only textual programs (PRG, MPR, etc.). To get a form or class to show up in the Trace window, you need to set a breakpoint in it (or include SET STEP ON or SUSPEND in the code). See [Breakpoints](#) below. However, once a form or class is open, you can get to any of its methods using the Object and Procedure dropdowns in the Trace window.

The Trace Between Breaks setting determines whether you can watch code execute in the Trace window. When it's off, both the Trace and Call Stack windows are updated only at breakpoints. (See [Configuring the Debugger](#) for more information on tracing between breakpoints.)

### The Watch window

Watch (**Figure 4**) lets you track the values of variable and expressions, as well as set breakpoints based on them. It provides drilldown capability for objects and arrays. The

simplest way to add an expression to the list of watched expressions is to type it into the Watch textbox and press Enter.



**Figure 4.** The Watch window: This window lets you see the values of any expressions you choose. The Name column can contain any expression, not just variables.

Recently changed values show up as red in the Value column. (Actually, the color for changed values can be set in the Options dialog. Red is the default foreground color.)

Hold the mouse over a value that's too long for the Value column and a tip appears showing the complete value.

The Watch window is far more capable than it might appear. Both the name and the value of each item listed can be edited in place. Editing the value changes it in the running program. This can be a very handy technique when you realize that you've failed to initialize or update a particular variable, or when you want to jump right to the data of interest. Editing the name lets you modify what you're watching, for example, to look at the second element of an array rather than the first, or to drill farther down into an object hierarchy. To edit either the expression or the value, double-click on that item, or click in the appropriate column when the item is highlighted.

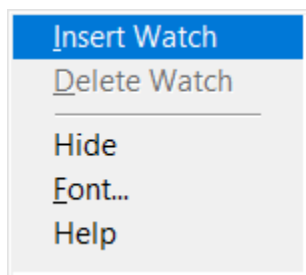
Starting in VFP 8, the textbox portion of the Watch window has IntelliSense, making it easier to find the property you want to watch.

The Watch window has various drag-and-drop capabilities. Among them are:

- Drag expressions from the [Trace window](#) into the Watch textbox for editing.
- Drag expressions from the Trace window into the list of watched expressions. That is, you don't have to go to the Watch textbox first.
- Drag variables from the [Locals window](#) into the Watch textbox for editing.
- Drag variables from the Locals window into the list of watched expressions.
- Drag expressions from the list of watched expressions into the Watch textbox for editing. This is an easy way to add an expression similar to one you're already watching.
- Drag expressions from the Watch textbox into the list of watched expressions.
- Drag expressions from the [Debug Output window](#) into the Watch textbox for editing.
- Drag expressions from the Debug Output window into the list of watched expressions.
- Drag expressions from any code editing window in VFP, including the Command Window, into the Watch textbox for editing.
- Drag expressions from any code editing window in VFP, including the Command Window, into the list of watched expressions.
- Drag expressions from the list of watched expressions into any code editing window, including the Command Window.

If you prefer the keyboard to the mouse, you can copy expressions from the Trace window or any code editing window to the Watch window using cut-and-paste. Unfortunately, there doesn't appear to be a way to cut-and-paste from the Locals window to the Watch window without first selecting the expression with the mouse. Similarly, there's no keyboard mechanism for accessing the list of watched expressions.

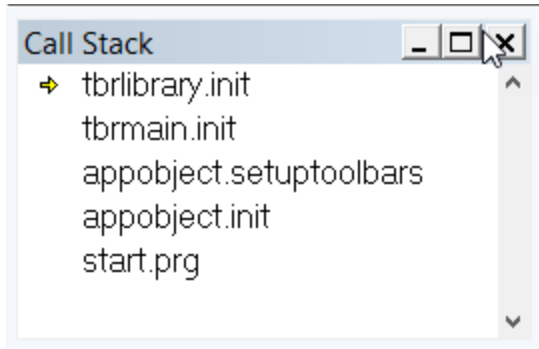
The Watch window's context menu (**Figure 5**) doesn't offer much that's unique to this window. Insert Watch positions the text cursor in the Watch textbox. Delete Watch deletes the currently highlighted item; you can do that by just hitting the Delete key, instead.



**Figure 5.** The Watch window's context menu: There's not much added value here.

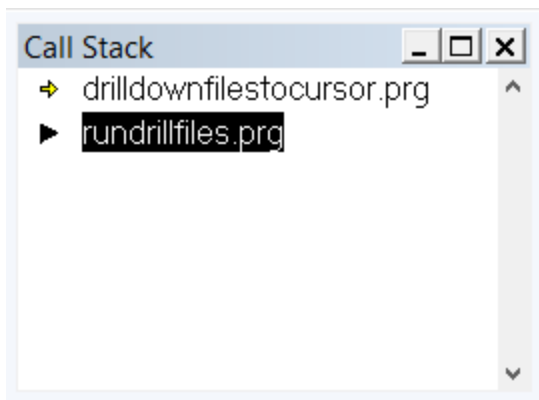
## The Call Stack window

The Call Stack window (**Figure 6**) displays the list of routines currently in the call stack, with the most recent on top. This window interacts with the [Trace window](#), to allow you to examine code from any routine in the call stack.



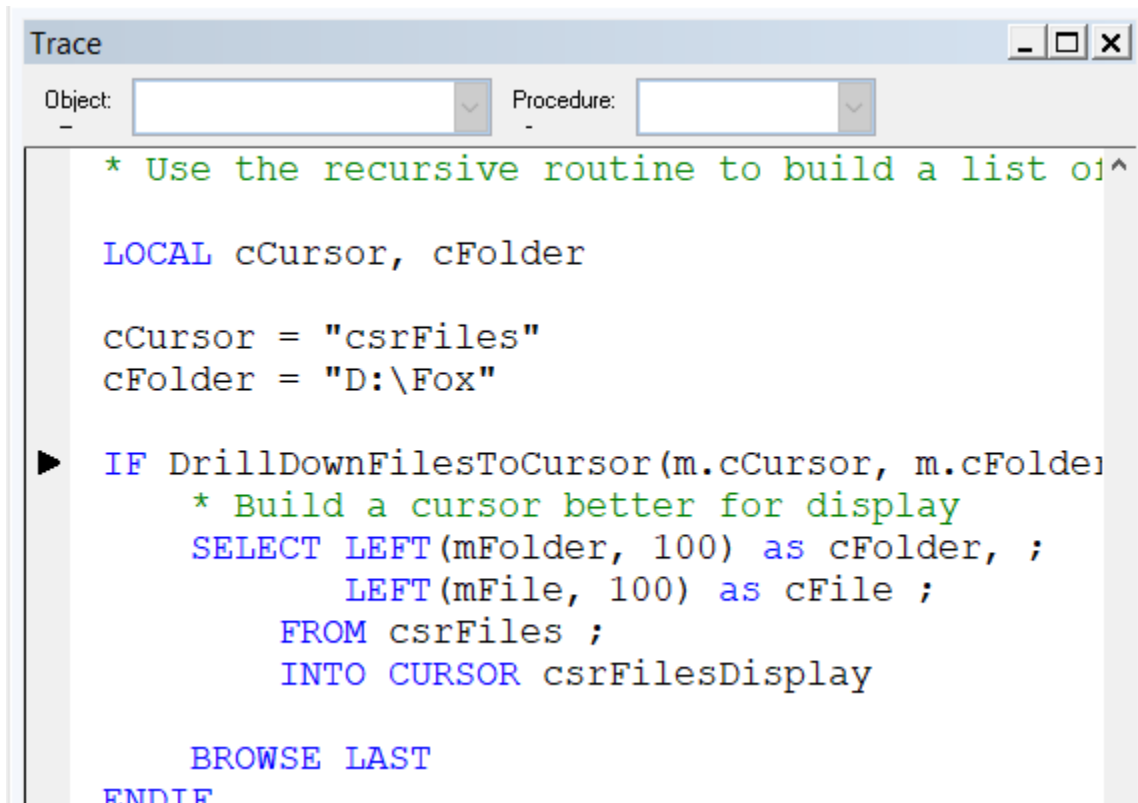
**Figure 6.** The Call Stack window: You can see what routine is currently running, as well as what routine called it, all the way down the line.

The yellow arrow indicates the currently running routine. If you click on another routine in the Call Stack, that routine displays in the Trace window and a black triangle appears next to that routine in the Call Stack window, as in **Figure 7**. In the Trace window, a black triangle appears next to the line containing the call that caused you to leave that routine, as in **Figure 8**.



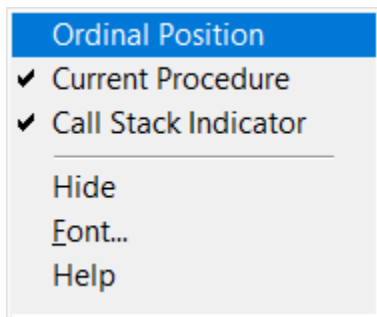
**Figure 7.** When you navigate to a different routine, a black triangle in the Call Stack shows you which routine you're looking at.





**Figure 8.** In the Trace window, the black triangle indicates the line that caused you to leave this routine.

The context menu for the Call Stack (**Figure 9**) lets you configure the window, determining whether the routines are numbered and whether icons indicate the currently executing routine (the yellow arrow) and the routine currently displayed in the Trace window (the black triangle).



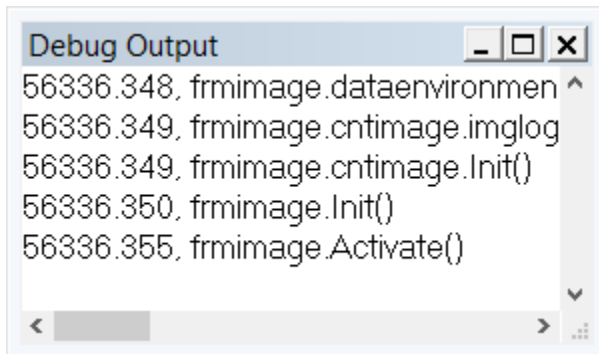
**Figure 9.** Context menu for the call stack: You can decide which indicators appear in the call stack.

The Trace Between Breaks setting (see [Configuring the Debugger](#) below) determines whether the Call Stack window is updated dynamically or only at breakpoints.

### The Debug Output window

The Debug Output window (**Figure 10**) has multiple uses. It's the default location for output from [Event Tracking](#). You can also send output directly to the window with the

DEBUGOUT command (discussed in [Sending Output to the Debugger](#) below). Messages from failed assertions (see [Testing conditions](#) later in this document) appear in Debug Output. In VFP 9, if Debug Output is open when you build a project, messages from the build process are echoed there; these messages can be very helpful when something goes wrong while building. You get similar output when a program file is compiled with the Debug Output window open. Also, in VFP 9, some errors in creating Watch expressions appear in the Debut Output window.

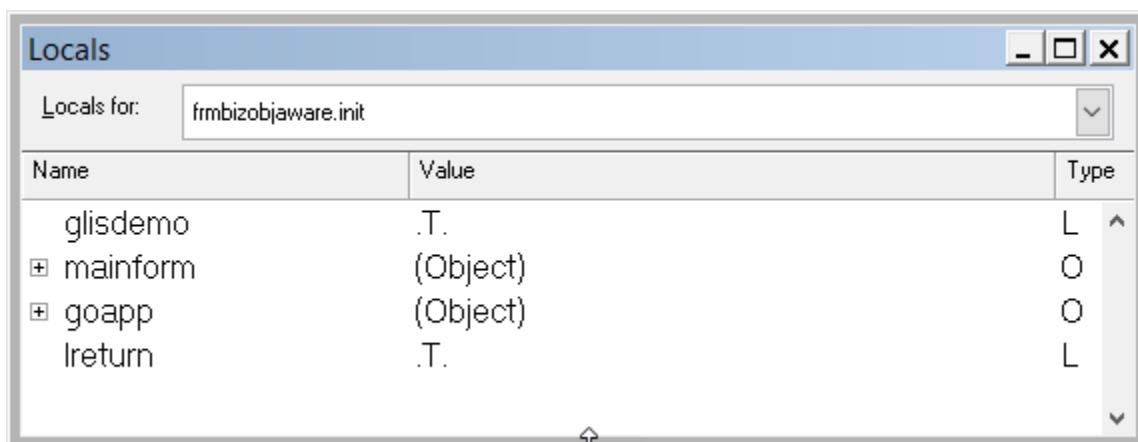


**Figure 10.** The Debug Output window: You can send information to this window with the DebugOut command. By default, Event Tracking output (like that shown here) goes to this window.

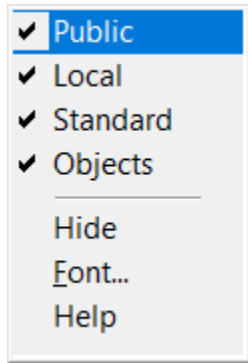
The contents of the Debug Output window can be saved to a text file (using the context menu). This is handy for sharing results with others or to allow you to compare different event sequences.

## The Locals window

The Locals window (**Figure 11**) shows the values of the variables in scope. Using the dropdown at the top, you can look at the variables for any routine in the call stack; the context menu (**Figure 12**) lets you narrow down what's displayed.



**Figure 11.** The Locals window: All the variables in scope in the chosen routine are displayed. You can drill down into objects and arrays.



**Figure 12.** Context menu for the Locals window: You can narrow down what's displayed in the Locals window.

When you choose a different routine in the Call Stack, the Locals window changes to show the variables from the chosen routine.

As in the Watch window, you can hold the mouse over a long value to see the complete result.

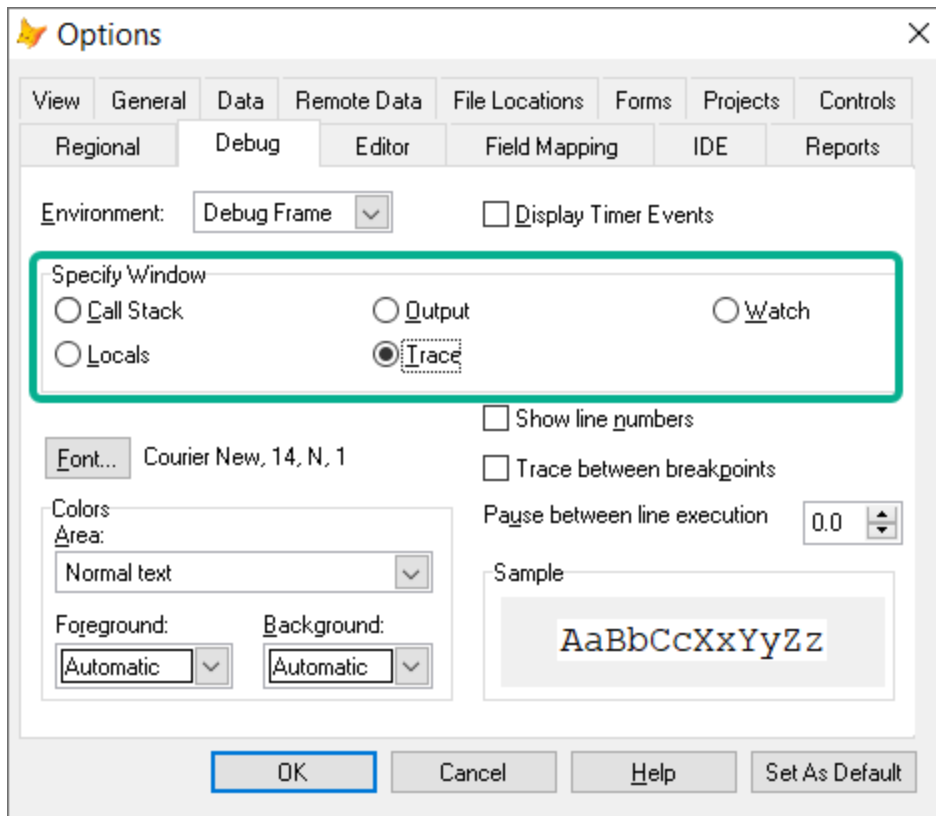
Also as in the Watch window, you can change the value of a variable or property in the Locals window. Double-click in the Value column or click in the Value column when the item is highlighted. This capability, combined with the Set Next Statement option on the Debugger menu (discussed in [Stepping Through Code](#)) makes it possible to correct small mistakes and continue running a test rather than having to fix the code and start over.

You can navigate the Locals window by keyboard. When the window is activated, the dropdown list of routines gets focus. Use Ctrl+Tab to move focus to the list of values. As you'd expect, the arrow keys navigate within the list. Press Tab to make the value of the highlighted item editable.

You can drag the name of a variable out of the Locals window into the Watch window or any code editing window. If the highlighted item is a property, the complete object reference to the property is dropped.

## Configuring the Debugger

In addition to choosing whether to run in the Debug frame or the FoxPro frame, the Options dialog (Figure 1 and **Figure 13**) offers you a number of ways to customize the Debugger's behavior. The Debug page of the Options dialog is organized in a unique way that makes it easy to miss some of your choices. Making a choice from the Specify Window option group (highlighted in Figure 13) changes the bottom half of the page. Figure 1 shows the set-up for customizing the Call Stack window. Figure 13 shows the options for the Trace window.



**Figure 13.** Customizing the Debugger: The option group switches between the various windows. Each has its own settings.

You can set the font and the colors used for the various components of each window. Each window has its own settings. The color choices you make for the Trace Window may not behave exactly as you'd expect because the syntax coloring settings from the Editor page of the dialog appear to override settings you make on the Debug page, except for Normal text. The context menu for each window also allows you to set its font, but not its colors.

The unique characteristics for each window appear in the upper right quadrant of the window-specific section. That is, they're on the right-hand side below the Specify Window option group. Many of these items are repeated in the context menu for the corresponding window.

The Locals and Watch windows don't have any unique choices.

As **Figure 13** shows, for the Trace window, you can specify whether line numbers should appear, whether to trace between breakpoints, and the "throttle" speed, that is, how long VFP should pause after each executed line when tracing between breakpoints.

Tracing between breakpoints is a mixed blessing. It's a good way to see exactly what's going on, but it slows down the code considerably. Usually, it's best to turn this setting off. This item can also be set using the Trace window's context menu.

Changing the throttle speed is handy when you want to watch some process occur, but [stepping through](#) would change the process. In that case, you can turn on tracing between breakpoints and set the throttle speed (which corresponds to the `_THROTTLE` system variable) to a value high enough to let you see what happens, but low enough that you don't have to wait all day. The throttle setting is also available on the Debug menu in the Debug frame and on the Trace window's context menu in the FoxPro frame.

The choices for the Call Stack (as shown in **Figure 1**) are the same as offered in its context menu, and determine which markers appear.

For the Debug Output window, you can specify a file to which output is echoed, and determine whether to overwrite the existing log file or add the new data to it. If you set this item, then choose Set as Default, only the file name carries over to future VFP sessions. You still have to turn on logging of this window explicitly.

## The Debug menu and toolbar

The Debugger toolbar (**Figure 14**) contains buttons and checkboxes for a variety of debugging operations. In the Debugger frame, it's docked under the menu by default. When the Debugger is set to the FoxPro frame, the toolbar appears (by default, docked at the top) whenever any Debugger window is opened. The Debugger frame also has a dedicated menu that contains the same items.



**Figure 14.** The Debugger toolbar: This toolbar is available in both frames. In the FoxPro frame, it appears whenever a Debugger window is opened.

The controls on the toolbar are divided into six groups, from left to right as follows:

- **Open program:** this group contains a single "Open" button that lets you open a program in the Trace window. Once a program is open there, you can set breakpoints, run it, or simply examine the code.
- **Run program:** this group contains two buttons. The "Resume" button begins or continues execution of the current program; it's equivalent to the RESUME command. The "Cancel" button ends execution of the current program; it's equivalent to the CANCEL command.
- **Step through program:** the four buttons in this group let you step through an executing program, running as little as one command or as much as you want. From left to right, these buttons stand for "Step Into," "Step Over," "Step Out" and "Run to Cursor." They're discussed in detail in the section [Stepping through code](#) below.
- **Debugger windows:** this group contains five graphical checkboxes, one for each of the Debugger windows. From left to right, they are "Trace," "Watch," "Locals," "Call Stack" and "Debug Output." When checked (pushed in), that window is open.

- Breakpoints: the three buttons in this group manage breakpoints. From left to right, they are "Toggle Breakpoint," "Clear All Breakpoints" and "Breakpoint dialog." See [Breakpoints](#) below for details.
- Other tools: the final group of two buttons provides quick access to [Coverage Logging](#) and [Event Tracking](#). Both tools are discussed below.

### Breakpoints

Breakpoints are one of the key tools in debugging. They allow you to stop executing at a designated point, so that you can check the current status and, if desired, proceed one command at a time.

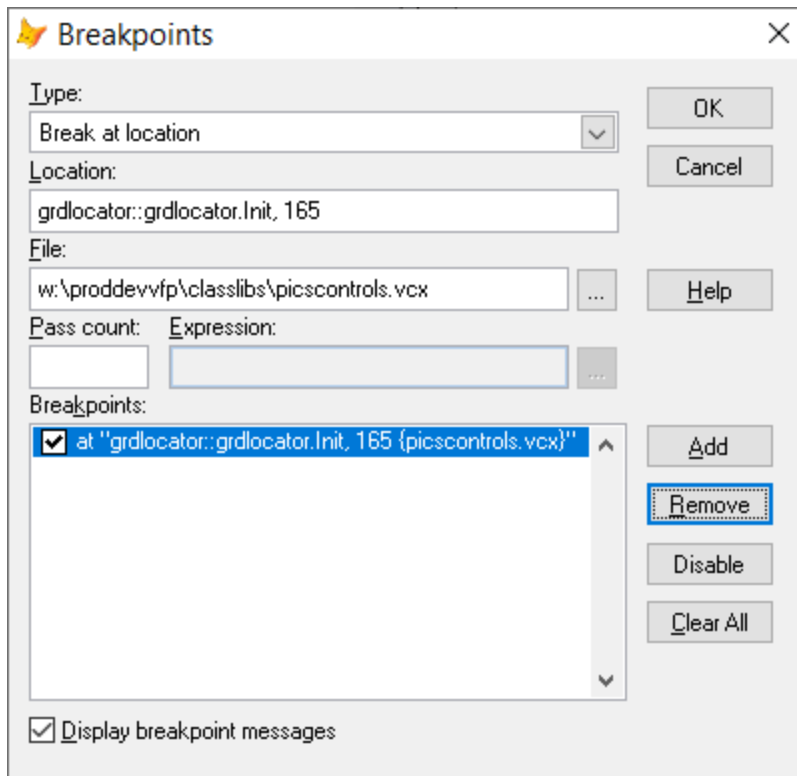
There are a lot of ways to set breakpoints in VFP. You can do so from both the Debugger and the development environment. Breakpoints can be based on a particular line of code, an expression, or a combination of the two.

Breakpoints fire only when the Debugger is open. This means that you can leave breakpoints set while developing code without having them interrupt your test simply by closing the Debugger. (You can also turn off individual breakpoints explicitly without closing the Debugger. See [The Breakpoints dialog](#) below.) However, even when the Debugger is closed, having breakpoints defined can slow down code execution, so be sure to remove all breakpoints before doing any timing tests.

Once a breakpoint fires and code execution is paused, the Command Window is available to you. You can issue any commands at all. (Be careful not to unintentionally destroy the variables and objects you're working with.)

### The Breakpoints dialog

The most obvious way to set breakpoints is with the Breakpoints dialog (**Figure 15**). The dialog is available on the Debugger toolbar, in the Tools menu of the Debugger, and, beginning in VFP 7, in the Tools menu of the VFP development environment.



**Figure 15.** Setting breakpoints: The Breakpoints dialog lets you set all kinds of breakpoints.

The dialog lets you create four kinds of breakpoints:

- Break at a specified location;
- Break at a specified location when a specified expression is true;
- Break when a specified expression is true;
- Break when a specified expression has changed.

You choose the breakpoint type using the Type dropdown in the dialog, then fill in some or all of the other items, depending which type of breakpoint you're creating. Be sure to hit the Add button to move a breakpoint from the top part of the dialog into the list of breakpoints at the bottom.

Be aware that breakpoints based on a line of code fire *before* that line executes. Breakpoints based on an expression, however, fire *after* the expression has changed. When you combine the two using the "Break at location if expression is true" option, the breakpoint fires before the specified line, but only if the expression is true before executing that line.

There are a number of things you can do with this dialog to get just the breakpoints you want.

Set Pass Count to have a breakpoint fire on a specific pass through a line of code. For example, if your program fails when processing the 257<sup>th</sup> item in a loop, set a breakpoint at the beginning of the loop and set pass count to 256 or 257.

In VFP 5, 6 and 9, you can specify a method name without a particular file or object to have a breakpoint fire each time any method of that name is executed. For example, set a breakpoint on Click to stop execution as soon as any Click event occurs. To fire only the Click methods within a particular form, put the name of the form file (the SCX) in the File textbox. Unfortunately, in VFP 7 and 8, this technique works only if you add a line number to the breakpoint (for example, "Click, 3"), and that line contains executable code.

The Breakpoints dialog lets you disable a breakpoint definition, but keep it available for future use. Uncheck it in the list of breakpoints. This is handy when a particular breakpoint was complex to define. While it would also seem useful for working on multiple projects, there's a better solution in that case; see [Saving Configurations](#) below.

Interestingly, you can't edit a breakpoint, but you can create a new breakpoint based on the definition of an existing breakpoint. Highlight the breakpoint in the list and then edit in the top portion of the dialog and then click Add to save the new breakpoint.

### Setting breakpoints in the Debugger

In addition to the Breakpoints dialog, when you have the Debugger open, you can set breakpoints in both the Trace and Watch windows. In the Trace window, you set a breakpoint by double-clicking in the shaded bar at the left-hand side. Double-click on the same breakpoint to remove it. Breakpoints set this way are "break at a specified location" breakpoints.

It works the same way in the Watch window. You can set a breakpoint on any expression by double-clicking next to it in the shaded bar. Again, double-click the breakpoint to remove it. Breakpoints set this way are "break when a specified expression has changed."

Once you create a breakpoint, it appears in the Breakpoints dialog and you can disable it or remove it there, as well.

### Setting breakpoints in the development environment

It's possible to set breakpoints without switching to either the Breakpoints dialog or the Debugger. In VFP 7 and later, it's very easy.

In VFP 6, the context menu for editing windows contains "Set Breakpoint." Choosing that option sets a breakpoint at the cursor location. The breakpoint appears in the Breakpoints dialog (which is available only through the Debugger, in that version), but there's no visual indication of the breakpoint in the code being edited. Like breakpoints set in the Trace window, this creates a "break at a specified location" breakpoint.

VFP 7 and later feature a significant improvement. By default, all code editing windows have a "selection margin," a shaded bar at the left hand side. As in the Trace and Watch



windows, you can set a breakpoint by double-clicking in the selection bar. In addition, the context menu contains "Toggle Breakpoint." Choosing it sets a breakpoint on the line or, if there's already a breakpoint on that line, removes it. A red dot in the selection margin indicates a breakpoint. Either method creates a "break at a specified location" breakpoint. Also, as with the Trace and Watch windows, such breakpoints are added to the Breakpoints dialog. When a breakpoint is disabled in the Breakpoints dialog, it shows as an open red circle in the code window.

### Stepping through Code

Once you stop execution where you think a problem is occurring, you're set to use one of VFP's most powerful debugging tools—the ability to step through code and see what happens. VFP provides a number of options for stepping through code. Most of the options are available on the Debugger toolbar, in the Debug menu of the Debugger, in the context menu of the Trace window, and via keyboard shortcuts. Since using the shortcuts can speed up tracing significantly, the various keyboard shortcuts are shown below, as each option is discussed.

The simplest options (which aren't really about "stepping" through code) are resuming execution (F5) and canceling the current program. These are good choices when you've figured out what's wrong, or in the case of resume, when you want to go to the next breakpoint.

The Debug menu, but not the toolbar, features another option to use when you've figured out what's wrong and are ready to repair it. In that case, choose Fix and the program or method currently displayed in the Trace window opens with the current line highlighted. (It's worth mentioning that there have been a few reports of problems with the Fix option. Some developers choose to avoid it. I haven't run into these issues.)

There are four options for executing code in order from the current position:

- Step Into (F8) executes the next line of code; if it calls another routine, tracing continues inside that routine. This is the choice to use when you have no idea what's gone wrong and you want to see exactly what's happening.
- Step Over (F6) executes the next line of code, but if it calls another routine, that routine is executed without tracing. This is the one to use when you know the problem is in the current routine, not in the one it's calling (or, more generally, when you're sure the routine to be called isn't the problem).
- Step Out (Shift-F7) executes the remainder of the current routine, returning to the calling routine. Use this one when you find yourself inside a routine you don't want to step through.
- Run to Cursor (F7) lets you execute several lines at a time and cuts down on the number of breakpoints you need to set. Click where you want execution to stop, then choose this option to execute several (or many) lines of code.

There's another (seriously underused) option that lets you tweak execution. Set Next Statement lets you change the execution path. Click on the line you want to run next, then choose this option. You can use this one in several ways. You can skip over code you know won't work by setting the next statement ahead of the current position. You can also use it to go back and run one or more lines again after you've fixed whatever caused a problem (such as being in the wrong work area, a file not existing or forgetting to initialize a variable). Finally, sometimes, you can use it when you realize that you should have stepped into a line you just stepped over. Whether that works depends on the actual code, of course.

In most tracing situations, you'll use a combination of all of these choices.

### Sending output to the Debugger

The `DEBUGOUT` command sends whatever you give it to the Debug Output window. This is a much better way to keep an eye on what's going on than using `WAIT WINDOW`. The `WAIT WINDOW` command interrupts execution (even with the `NOWAIT` clause), which means that it can affect the process you're tracking. `DEBUGOUT` doesn't involve a wait state, so the only change it makes is the tiny time slice needed to evaluate the specified expressions and send them to the Debug Output window. In addition, it won't do anything when the Debugger is closed, or in the runtime environment, so if you forget to remove a `DEBUGOUT` from your code, the user is no wiser.

What might you do with `DEBUGOUT`? The possibilities are virtually endless. Here are a few:

- Track methods and programs being run by putting `DEBUGOUT PROGRAM()` as the first line.
- Track the value of a variable or expression during processing by using code like:  

```
DEBUGOUT "Before assignment, count = " + TRANSFORM(nCount)
```
- Track the steps of a process by issuing `DEBUGOUT` with a description before each step.

The first item on the list is particularly useful because the Event Tracker, described in the next section, works only with events, not non-event methods or code outside objects.

In VFP 8 and later, `DEBUGOUT` accepts multiple expressions. Each expression is evaluated and the results are placed on a single line with a space between each pair. This ability means that the second example above can be changed to:

```
DEBUGOUT "Before assignment, count =", nCount
```

Note, also, in that line that `DEBUGOUT` doesn't require the expressions you give it to be character. It handles conversions as needed.

### Testing conditions

Like `DEBUGOUT`, the `ASSERT` command lets you check things during development with no impact on the runtime environment. `ASSERT` evaluates an expression; if it's true, all is well.

If the expression is false, a message displays, and if the Debug Output window is open, is copied there as well. The syntax is:

```
ASSERT lCondition [ MESSAGE cMessage ]
```

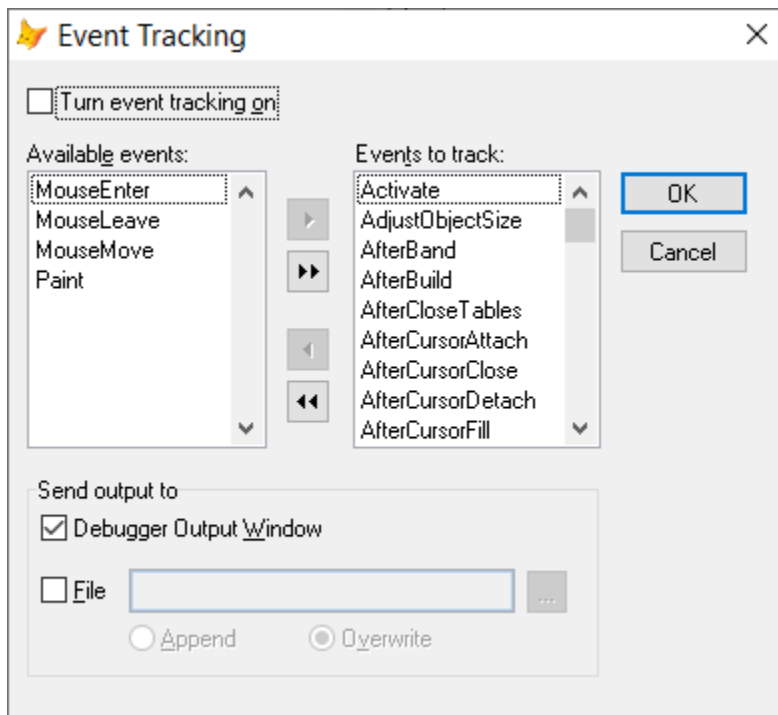
If you specify a message, it's displayed when the assertion fails; if you omit the MESSAGE clause, a default message appears. Assertions are evaluated only when SET ASSERTS is ON and only at development time.

Assertions are a powerful tool for conveying code expectations while testing, but don't take the place of testing parameters and inputs in code.

### Tracking Events

In event-driven, object-oriented programming, one of the most important pieces of information when debugging is which events are firing and in what order. VFP includes a tool to provide that information.

Event Tracking is available on the Debugger toolbar and from the Tools menu of the Debugger. However, the button on the Debugger toolbar is misleading. It's a combination button and checkbox. When Event Tracking is off, pushing the button brings up the Event Tracking dialog (**Figure 16**). Checking "Turn event tracking on" turns on tracking and the toolbar button stays depressed, like a checkbox. If you click the button again, it turns off tracking without bringing up the dialog.



**Figure 16.** Tracking events: This dialog lets you determine which events to track. It also lets you turn tracking on and off.

When Event Tracking is on, a line of text like the following is generated for each event that fires:

```
51722.101, screen.MouseMove(0, 0, 982, 789)
```

The first item is a timestamp. It's followed by the name of the object and event that fired, along with any parameters received by the event's method.

The Event Tracking dialog gives you control over which events are tracked. You can track one, a few, many or all of the VFP events. Most of the time, you won't want to track those that fire frequently. `MouseMove`, `MouseEnter`, `MouseLeave` and `Paint` are the ones you're most likely to remove from the list. (Figure 16 shows those four excluded.)

Beginning in VFP 7, you can also track a few events at the Windows level. The `SYS(2801)` function determines which events are tracked. Issuing `SYS(2801, 1)` tracks only VFP events. `SYS(2801, 2)` tracks only Windows events, and `SYS(2801, 3)` tracks both. The only Windows events tracked are mouse and keyboard events. You don't see system events such as those involving storage devices. In addition, you don't have control over which Windows events are tracked; once you decide to track them, it's an all-or-nothing proposition. Note that you need to turn event tracking off before changing `SYS(2801)`, then turn it back on for the change to take effect.

By default, Event Tracking output goes to the Debug Output window. You can also choose to send it to a file, either instead of the window or in addition to the window.

The Event Tracker can be controlled programmatically, as well as through the dialog. Use `SET EVENTTRACKING` to turn tracking on and off, and to direct output to a file. For example, the following commands turn on event tracking and send output to `events.txt`, as well as the Debug Output window. (There's no programmatic way to specify that Event Tracking output goes only to a file.)

```
SET EVENTTRACKING TO Events.TXT  
SET EVENTTRACKING ON
```

`SET EVENTLIST` controls the list of events tracked. Separate the individual events with commas. For example, the following command limits tracking to a few events:

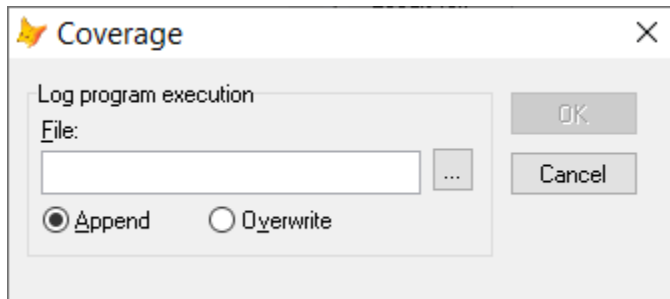
```
SET EVENTLIST TO Click, DblClick, RightClick, Keyboard
```

The corresponding `SET()` functions tell you the current settings. `SET('EVENTTRACKING')` returns ON or OFF; `SET('EVENTLIST')` returns a comma-separated list of tracked events.

### Coverage Logging

Sometimes the problem with an application isn't that it doesn't work, but that you're not sure whether it works, or it works, but it's too slow. Coverage Logging and the associated Coverage Profiler tool are designed to address these issues.

Like Event Tracking, Coverage Logging is available from the Debugger toolbar and the Tools menu of the Debugger. Also, as with Event Tracking, the toolbar button is a combination button and checkbox. When Coverage Logging is off, pressing the button displays the Coverage dialog (**Figure 17**). When Coverage Logging is on, pressing the (depressed) button turns it off without displaying the dialog. However, there's one key difference—you don't have to check a checkbox in the dialog to turn Coverage Logging on. Just specify a log file and click OK.



**Figure 17.** Coverage Logging: Use this dialog to specify a coverage log and turn on Coverage Logging.

You can turn Coverage Logging on and off programmatically using the SET COVERAGE command. SET COVERAGE TO a file name to turn Coverage Logging on. Issue SET COVERAGE TO with no file name to turn it off. SET('COVERAGE') returns the name of the current coverage file or the empty string, if none is specified.

Beginning in VFP 9, Coverage Logging is available at runtime as well as at design-time, so you can create a coverage log on a user's machine to track down a problem that doesn't occur in your testing.

Once Coverage Logging is turned on, each line of code that executes generates a line of text in the log file. The log indicates what line executed and how long it took. Here's an example:

```
0.000095,tastrade,tastrade.init,8,c:\apps\fox\vfp7\samples\tastrade\libs\main.vct,2
```

The items in each line are:

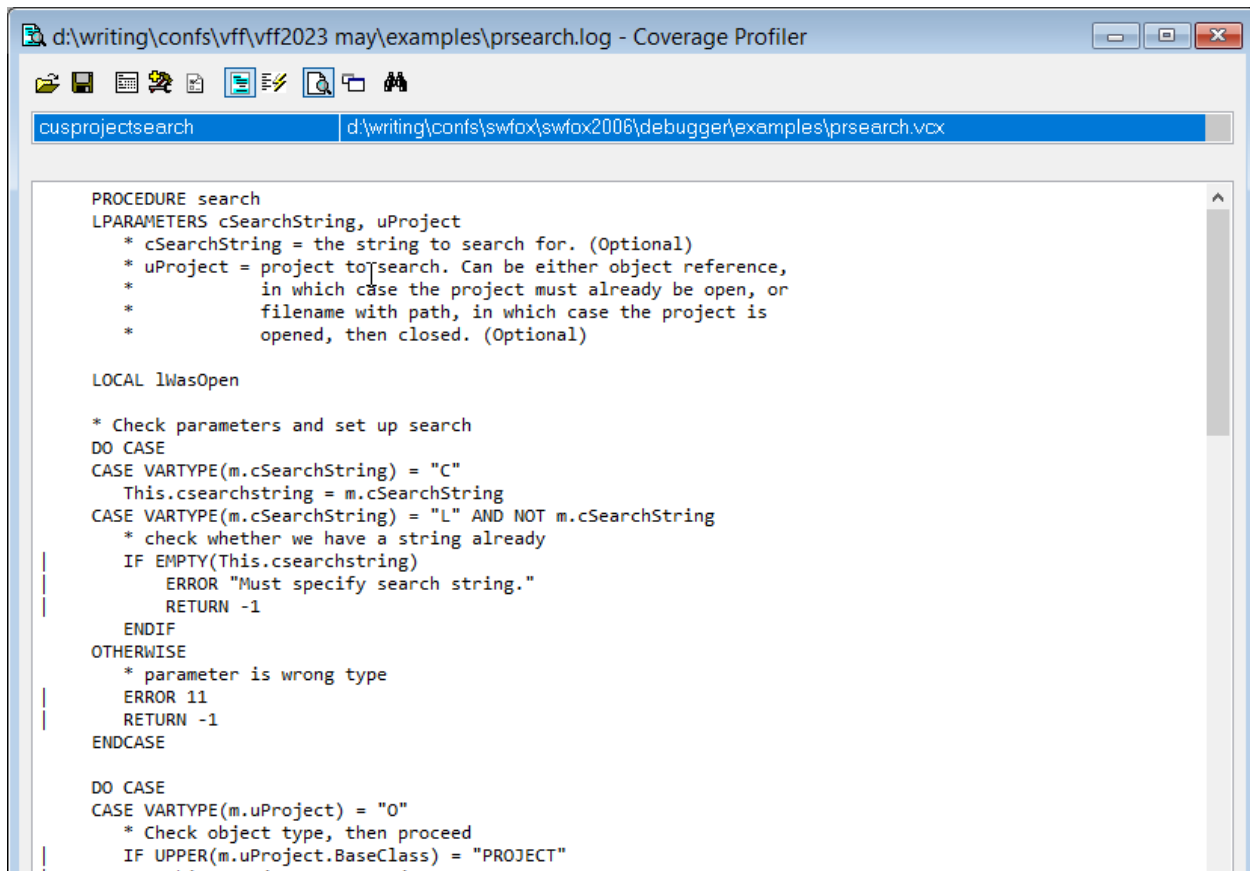
- Execution time;
- Name of the class executing the code;
- Method or procedure containing the code executed;
- The line number within the method or procedure;
- The file name, including path;
- The call stack level for this routine.

You can see that the log isn't something you'd want to parse and comprehend manually. Fortunately, VFP 6 and later come with the Coverage Profiler tool that processes coverage logs, providing a variety of information.

You can start the Coverage Profiler in several ways. From the Tools menu (in VFP, not the Debugger), choose Coverage Profiler. Alternatively, from the Command Window or a program, issue:

```
DO (_COVERAGE)
```

If Coverage Logging is active when you start the Coverage Profiler, the current log is used. If not, you're prompted to specify a coverage log. In either case, the log is processed and the Coverage Profiler interface (**Figure 18** and **Figure 19**) appears. As the log is processed, you may have to point to the source files; this is far more likely when processing a file created on a different machine.



The screenshot shows a window titled "d:\writing\confs\vff\vff2023 may\examples\prsearch.log - Coverage Profiler". The window contains a toolbar with icons for file operations and a text area displaying source code. The code is for a procedure named 'search' with parameters 'cSearchString' and 'uProject'. It includes comments and conditional logic for handling different parameter types and search strings.

```
PROCEDURE search
LPARAMETERS cSearchString, uProject
* cSearchString = the string to search for. (Optional)
* uProject = project to search. Can be either object reference,
*           in which case the project must already be open, or
*           filename with path, in which case the project is
*           opened, then closed. (Optional)

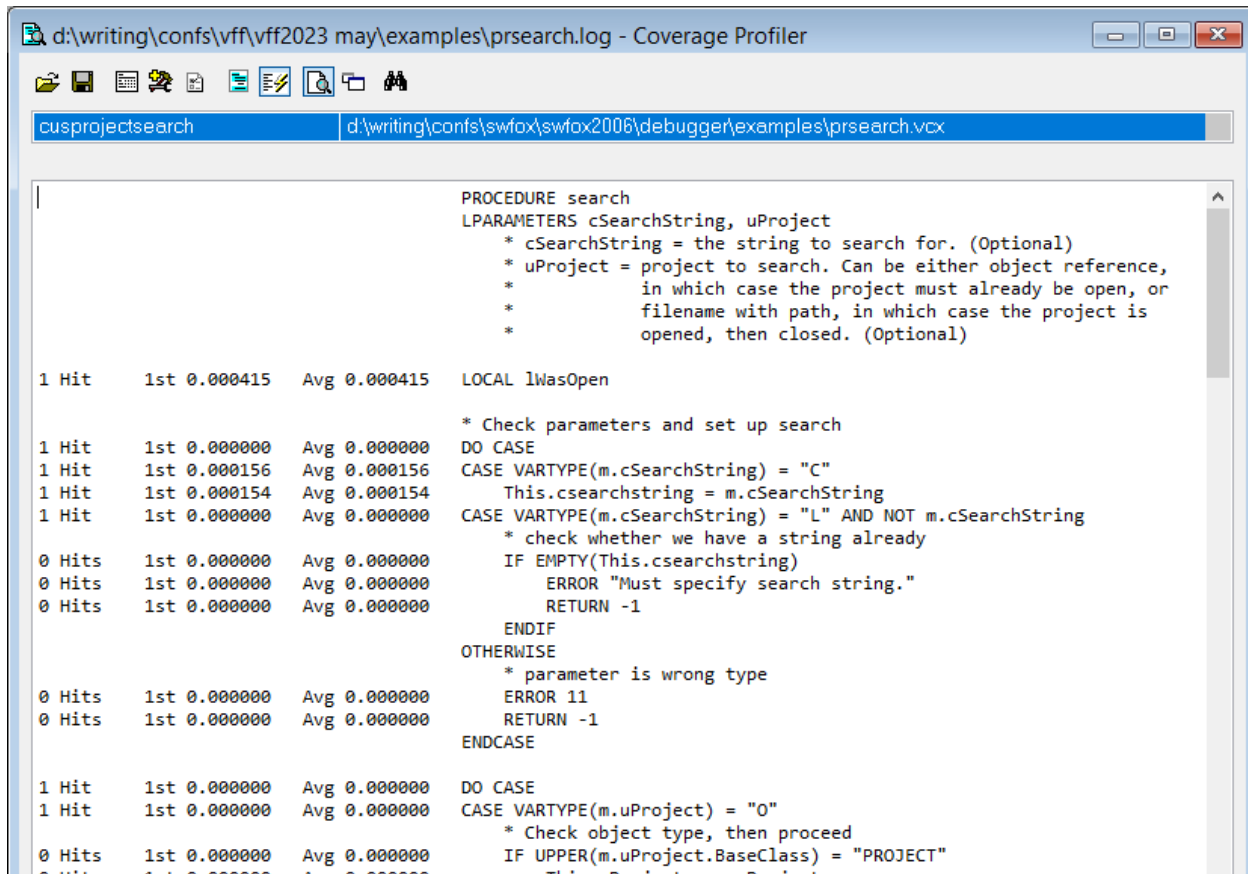
LOCAL lWasOpen

* Check parameters and set up search
DO CASE
CASE VARTYPE(m.cSearchString) = "C"
  This.csearchstring = m.cSearchString
CASE VARTYPE(m.cSearchString) = "L" AND NOT m.cSearchString
  * check whether we have a string already
  IF EMPTY(This.csearchstring)
    ERROR "Must specify search string."
    RETURN -1
  ENDIF
OTHERWISE
  * parameter is wrong type
  ERROR 11
  RETURN -1
ENDCASE

DO CASE
CASE VARTYPE(m.uProject) = "O"
  * Check object type, then proceed
  IF UPPER(m.uProject.BaseClass) = "PROJECT"
```

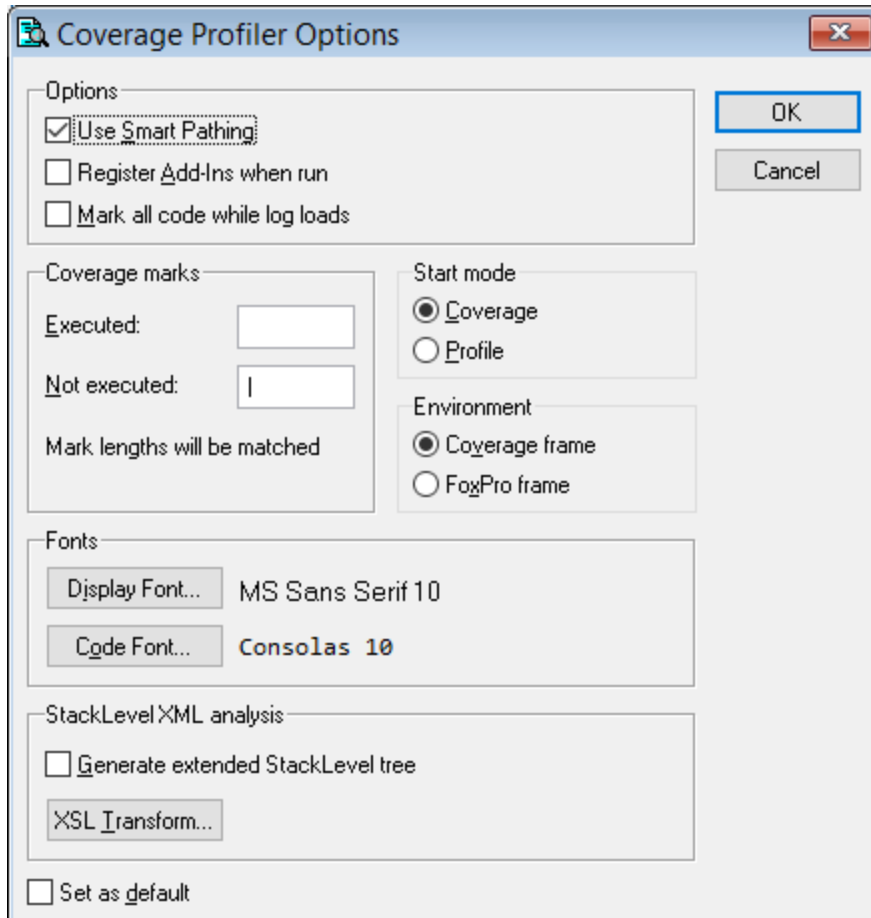
**Figure 18.** Coverage Profiler in Coverage mode: The Coverage Profiler shows you the routines that were called and which lines were executed. Use this option to ensure that everything gets tested.

The Coverage Profiler has two main modes. **Figure 18** shows Coverage mode, which indicates which lines were executed and which were not. **Figure 19** shows Profile mode, which provides information about how many times a line was run and how long it took.



**Figure 19.** Coverage Profiler in Profile mode: In this mode, you can see how many times each line was executed and how long it took. Use this option to find code that's bogging down your application.

The Coverage Profiler offers a number of customization options, most of them in the Options dialog (**Figure 20**) available from the Coverage Profiler's toolbar. The options range from the fonts used to which mode (coverage or profile) the tool should start in. You can specify the set of marks used in coverage mode. By default, executed lines are unmarked, while unexecuted lines are preceded by a vertical bar. You may prefer to mark executed lines or to use different characters. Like the Debugger itself, you choose whether the Coverage Profiler appears in its own frame (as in the two figures) or in the main FoxPro frame.



**Figure 20.** Configuring the Coverage Profiler: The Coverage Profiler's Options dialog lets you set things up the way you want them.

You can replace the Coverage Profiler with an alternate version, if you wish. It's written in VFP and the source code comes with the product. (Unzip XSource.ZIP in Tools\XSource, then look in Tools\XSource\VFPSource\Coverage.) If you prefer your own Coverage Profiler or one provided by another source, set `_COVERAGE` to point to the application you want to use.

You can also customize the Coverage Profiler by creating add-ins, custom code executed on your command. You can write add-ins that perform additional processing of the raw coverage log or the processed log.

I discussed several add-ins and alternative profilers in my white paper on Optimization: <https://tinyurl.com/m6aebpwb>. The materials for that paper (including some of those tools) are available at <https://tinyurl.com/25wvva7>. That paper also discusses a tool from Rick Schummer to fix paths in coverage logs that make it easier to use coverage at runtime; with Rick's permission, the downloads for that paper include the tool. VFPX includes a newer alternative profiler called CvgViewwr at <https://github.com/VFPX/CoverageViewer>; I haven't worked with it yet.



Although Coverage Logging is intended primarily for the two tasks handled by the Coverage Profiler, sometimes it's handy to use a coverage log as a debugging tool. You can review the log to see exactly which lines of code were executed in which order, much as you'd examine Event Tracking output to see which events fired. This technique is especially useful in situations where tracing the code would interfere with the results.

### Saving configurations

By default, the Debugger remembers a number of items between VFP sessions. That includes breakpoints, tracked events, and watch items, as well as the way you've positioned the Debugger's windows. All of this information is stored in the Resource file (FoxUser.DBF).

However, in a couple of situations, that may not be good enough. If you're working on multiple projects, you may get the Debugger set up just right for whatever you're working on, but then you have to move on to something else. The other case is when you crash VFP. The Debugger settings appear to be stored at the time you exit normally, so when VFP crashes, you lose any changes you've made in that session.

Fortunately, the Debugger has a solution for these problems. You can save information about breakpoints, items in the Watch window, and Event Tracking into Debugger configuration files (with a DBG extension). To return things to a previous configuration, load the appropriate file.

Loading and saving configurations is available only in the Debugger frame. The options are on the Debugger's File menu.

The configuration file uses plain text in a format similar to an INI file. Here's an example:

```
DBGCFGVERSION=4
WATCH=message()
WATCH=x
WATCH=y
WATCH=This
BPMESSAGE=ON
BREAKPOINT BEGIN
TYPE=0
PROC=init
LINE=3
DISABLED=0
EXACT=0
BREAKPOINT END
EVENTWINDOW=ON
EVENTFILE=
EVENTLIST BEGIN
Click, Activate
EVENTLIST END
```

This configuration has four items in the Watch window, a single breakpoint (on line 3 of the Init method) and sets the Event Tracking list to Click and Activate.

## Debugger Tips & Tricks

There are a number of little things you can do to increase your debugging productivity. Here's a collection of tips.

### Clever breakpoints

Setting the right breakpoints can make debugging much easier. One handy trick for expression-based breakpoints is to set them in the Watch window without opening the Breakpoints dialog. This keeps your breakpoints visible, and makes it easy to edit them and to turn them on and off.

To set a breakpoint when a particular routine or method is called, put an expression in the form "<PROGNAME>"\$UPPER(PROGRAM()) in the Watch window and set a breakpoint on it. This is especially handy if you want to stop every time a particular method is called, not just in a specified object. For example, the following expression stops execution whenever any Click method fires:

```
"CLICK"$UPPER(PROGRAM())
```

To make it easy to stop execution whenever you want, put LINENO() in the Watch window. When your application is in a wait state and you want to step through code after your next action (like clicking a button), set a breakpoint on LINENO(). That'll stop execution as soon as any code fires.

(The Watch window in **Figure 4** shows my minimum Watch window set-up, with these two items, as well as the watch on MESSAGE() described below. Adding these three is one of the first things I do when configuring VFP on a new machine.)

By default, a breakpoint on a variable fires when the variable goes in and out of scope. That can be annoying. To avoid the problem and break when a variable takes on a certain value, use an expression like TYPE("<Variable>") = "U" OR <Variable> = <Value>. For example, to break when nCount = 100, use:

```
TYPE("nCount") = "U" OR nCount = 100
```

Unfortunately, this technique isn't helpful when you want to break any time the variable changes.

To find out when a table is being opened or closed with a particular alias, set a breakpoint on USED("<alias>"). Use similar breakpoints to detect other changes to tables, such as EOF("<alias>"), RECCOUNT("<alias>") and RECNO("<alias>"). For example, to find out when Customer is being closed, you could use:

```
USED("Customer")
```

If a setting is getting changed and you don't know why or when, set a breakpoint on SET("<Whatever>"), such as:

```
SET("DELETED")
```

### Making life easier

Keep MESSAGE() in the Watch window so you don't have to remember what error message you're currently debugging. Be aware, though, that it always reflects the last error, so if you cause another error while exploring, that's the one you'll see.

The SYS(2030) function added in VFP 7 enables and disables debugging of system tools written in VFP. When you're debugging your own code that uses one of these tools, you probably don't want to trace through the Class Browser's or Coverage Profiler's code. Issue SYS(2030, 0) and you'll skip right past that code. On the other hand, if you're testing a replacement or enhancement for one of the system tools, you'll need to be able to debug it, so set SYS(2030,1).

Use \_SCREEN.ActiveForm to refer to the currently executing form in the Watch window. It's much easier than figuring out the name of the form. You can put any property of any item on the form into the Watch window using this construction. For example, to watch whether a textbox named txtDescription gets enabled or disabled, put this in the Watch window:

```
_SCREEN.ActiveForm.txtDescription.Enabled
```

You can use Event Tracking and DEBUGOUT output to help you figure out what difference a particular code change makes. Turn on Event Tracking and/or seed your code with DEBUGOUT PROGRAM() statements. Run the original code and save the Debug Output results to a file (or set it up to save them to a file in the first place). Change the code, run it again and save those results. Then, look at the two text files side by side to figure out the effects. (A tool like Beyond Compare is really handy for this kind of comparison.)

When code is paused, the Data Session window is available (as are other aspects of the VFP IDE). You can see what tables are open and Browse them to understand what's going on. Be sure to restore the original work area before continuing execution.

Also when code is paused in a method, you can use the keyword "This" in the Command Window to refer to the object whose method is executing. In VFP 7 and later, the addition of IntelliSense makes this technique particularly powerful for figuring out what's going on.

Check the Locals window after testing code to see what variable references are still floating around after your application cleans up. You may be surprised.

### Debugging reports

Starting in VFP 9, reports can interact with the Debugger. That makes it possible to debug calls to your code that appear in report expressions and in report band events, as well as code in report listeners. In VFP 8 and earlier, attempts to use the Debugger from code called in a report triggered an error.

To make it easier to debug reports, there's a special report listener subclass called `DebugListener` included in the FoxPro Foundation Classes. When you use it, all the steps of report processing are logged to a file. Look in `HOME() + "FFC\_ReportListener.VCX"`.

### Debugging with objects and collections

The Debugger can be frustrating when working with COM objects (such as those created via Office Automation). In the Locals and Watch window, the Debugger can't see all the PEMs of those objects. It can only show you the ones that have already been referenced. However, there is IntelliSense for those objects in the Watch textbox, so you can find the relevant properties and show their results in the Watch window. Even if you delete that item from the Watch window, you'll still be able to see that property when you expand the object in the Locals window (and if you put the object itself into the Watch window and expand it).

The Debugger also has issues showing collections (except for some that are part of VFP's object model). When you expand a collection in the Locals or Watch window, you can see the count, but there's no way to drilldown and see the individuals (the way you can with elements of an array). You can work around this by specifying the particular member you want to see in the Watch window or you can assign a member to a variable and examine that variable in the Locals window or put it into the Watch window.

There are a couple of VFPX tools that make it easier to debug collections. I created the Object Inspector for this purpose many years ago. More recently, Jim Nelson and Matt Slay combined that tool with some others they'd worked on to create the Object Explorer. It's worth having one or the other (or both) in your toolkit. You'll find both in the list of VFPX tools at <https://vfp.x.githu.b.io/projects/>.

### Debugging when the Debugger gets in the way

Now and then, you may run into cases where a piece of code doesn't work, but when you try stepping through it or maybe even just have the Debugger open, it's fine.

Sometimes, the issue is focus; taking focus off whatever you're debugging and giving it to the Debugger changes the order in which code runs or the context for a particular line of code. For situations like that, `DebugOut` is your friend. Lace your code with `DebugOut` commands that report on what's going on at different points in the code, and then read the log you've created to figure out where things are going wrong. Sometimes, it takes multiple passes and you add and remove `DebugOuts` until you narrow the problem down. Bugs where tracing gets in the way because it messes up the timing of the code can often be solved the same way.

The harder case is code where even having the Debugger open prevents the bug from showing. In those cases, you may have to do it the old-fashioned way, by logging not to the `DebugOut` window, but to an actual text file. A simple text logger that uses `StrToFile()` generally works in such cases; the [materials](#) for my [VFP Tips and Tricks session](#) include a basic logging class.

## Put the Debugger to work

The VFP Debugger is truly powerful. Once you understand how it works and what its capabilities are, you can cut debugging time down considerably.

Spend some time experimenting with the Debugger, trying a variety of breakpoints, digging down into the Locals and Watch windows, tracking a variety of events, and so forth. The more time you spend with it, the more productive you'll learn to be.

Finally, for another detailed look at the Debugger as well as a broader view of debugging, check out Nancy Folsom's book "Debugging Visual FoxPro Applications" ([www.hentzenwerke.com](http://www.hentzenwerke.com)).

## Acknowledgments

Several people contributed to these session notes. Thanks to Barbara Peisch, Ted Roche and the late David Frankenbach, who offered their debugging tips; and to Nancy Folsom, who reviewed the notes and offered improvements.