July, 2006

Learn to Use the Debugger

This powerful tool can save you time and gray hairs

by Tamar E. Granor, technical editor

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 is a good tool, offering power and flexibility.

Chocolate or Vanilla?

The Debugger can run two different ways. You can put it in the FoxPro frame, so it shares the main FoxPro window with the code you're testing. Alternatively, it can run in its own frame--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.

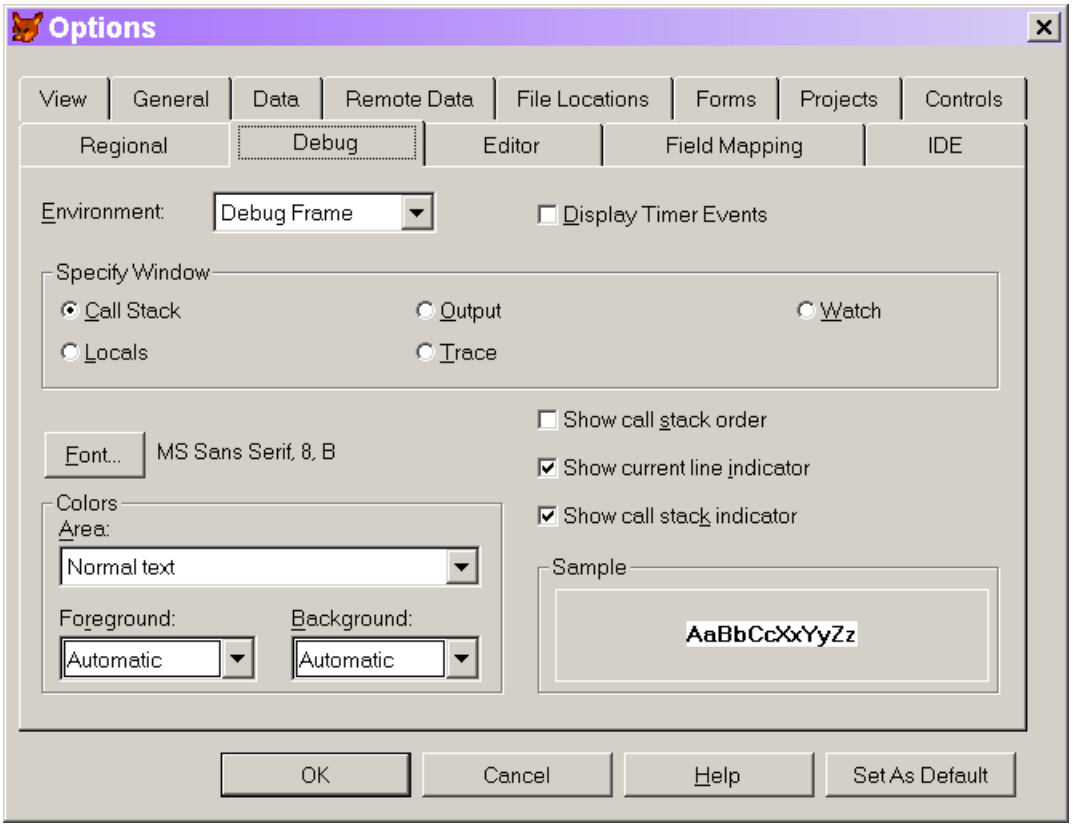

Figure 1 Debugger options—This page of the Options dialog lets you configure the Debugger. Use the Environment dropdown to specify whether Debugger windows appear in the main FoxPro frame or in a separate frame.

To a great extent, which frame to use is a matter of personal preference. There are several menu option (Fix, Load Configuration, Save Configuration) that's available only in the Debug frame, but otherwise the functionality is equivalent.

Starting the Debugger

Starting the Debugger varies a little depending whether you're using the FoxPro frame or the Debug frame. With the Debug 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, the Debug command opens the Trace and Watch windows (see Debugger Windows below) 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 Debug frame, choosing Suspend from the dialog that appears when an error occurs opens the Debugger.

Debugger Windows

The five main windows of the Debugger are Trace, Watch, Locals, Call Stack and Debug Output. The are three additional tools: Breakpoints, Event Tracking and Coverage Logging. (Event Tracking and Coverage Logging are beyond the scope of this article.) In VFP 7 and later, the five main Debugger windows are all dockable when working in the FoxPro frame.

Here's a look at each of the main windows and, where it contains something interesting, the corresponding context menu.

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.

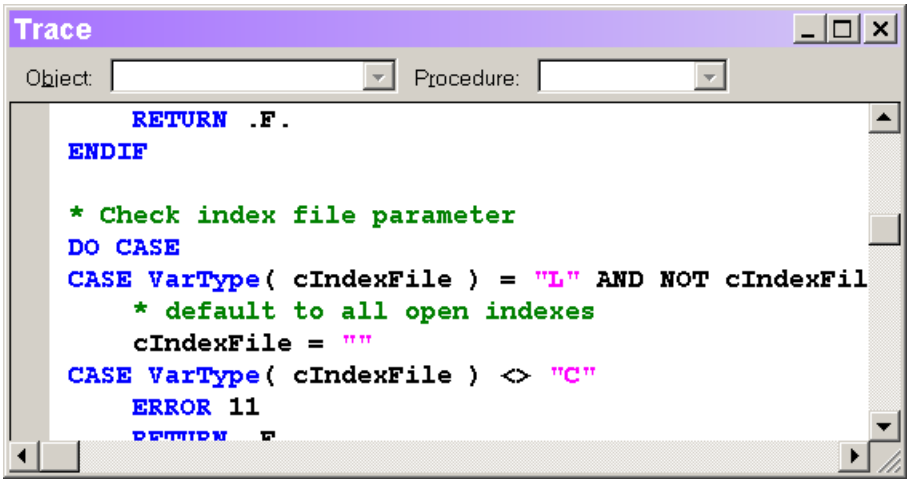

Figure 2 The Trace window—This window shows you the code that's currently running and allows you to step through it, as well as set breakpoints.

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 constant (#DEFINE) 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.

You can drag code out of the Trace window into the Watch window or any code editing window.

The Trace window's shortcut menu (Figure 3) includes choices for navigating within the running program. The navigation items are described in Stepping Through Code later in this article.

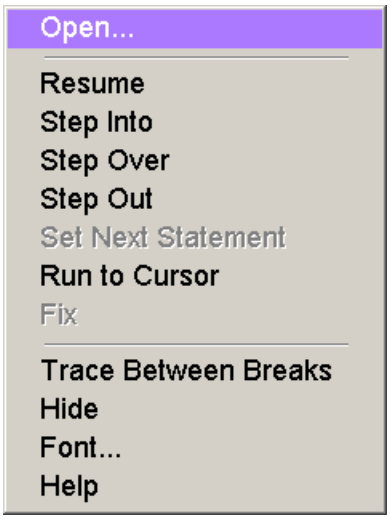

Figure 3 Context menu for tracing code—This context menu is available from the Trace window, and lets you control the running code.

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). 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.

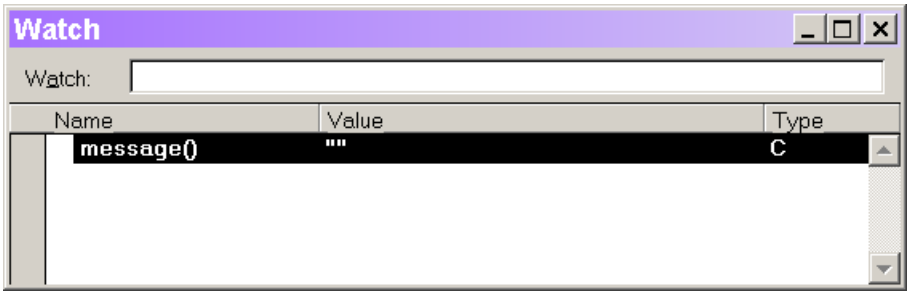

Figure 4 The Watch window—This window lets you see the values of any expressions you choose. The Name column can contain any expression, not just variables.

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.

Both the name and the value of each item listed in the Watch window can be edited in place. Editing the value actually changes it in the running program. This can be handy when you realize 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 that item, or click in the appropriate column when the item is highlighted.

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 expressions from the Locals window into the Watch textbox for editing.
- drag expressions 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.

The Call Stack window

The Call Stack window (Figure 5) displays the list of routines currently in the call stack, with the currently executing routine on top. This window interacts with the Trace window, to allow you to examine code from any routine in the call stack.

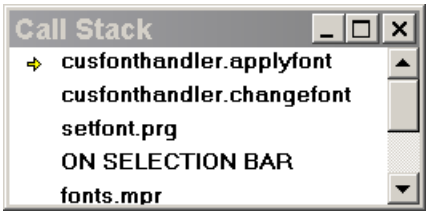

Figure 5 The Call Stack window—You can see what routine is currently running, as well as what routine called it, all the way down the line.

The context menu for the Call Stack (Figure 6) lets you configure the window, determining whether the routines are numbered and whether icons indicate the currently executing routine and the routine currently displayed in the Trace window.

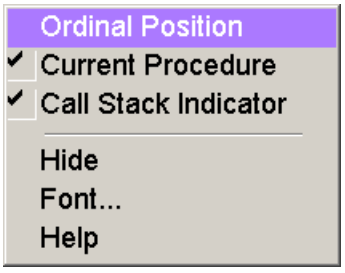

Figure 6 Context menu for the call stack—You can decide which indicators appear in the call stack.

The Debug Output window

The Debug Output window (Figure 7) has a number of 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; 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.

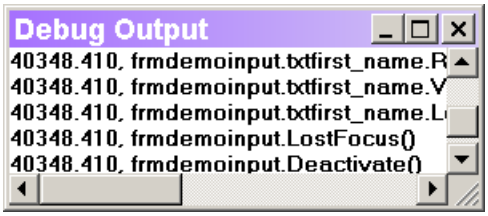

Figure 7 The Debug Output window—You can send information to this window with the DebugOut command. By default, Event Tracking output (like that shown here) goes to this window.

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

The Locals window

The Locals window (Figure 8) shows the values of the variables in scope. You can look at the variables for any routine in the call stack; use the dropdown list to choose. The context menu (Figure 9) lets you narrow down what's displayed.

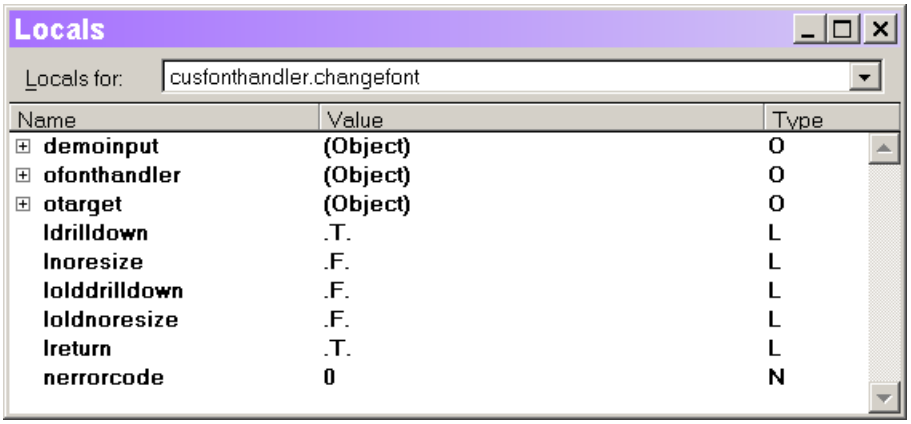

Figure 8 The Locals window—All the variables in scope in the chosen routine are displayed. You can drill down into objects and arrays.

Figure 9 Context menu for the Locals window—You can narrow down what's displayed in the Locals window.

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 10) offers you a number of ways to customize the Debugger's behavior. Making a choice from the Specify Window option group changes the bottom half of the page. Figure 1 shows the set-up for customizing the Call Stack window. Figure 10 shows the options for the Trace window.

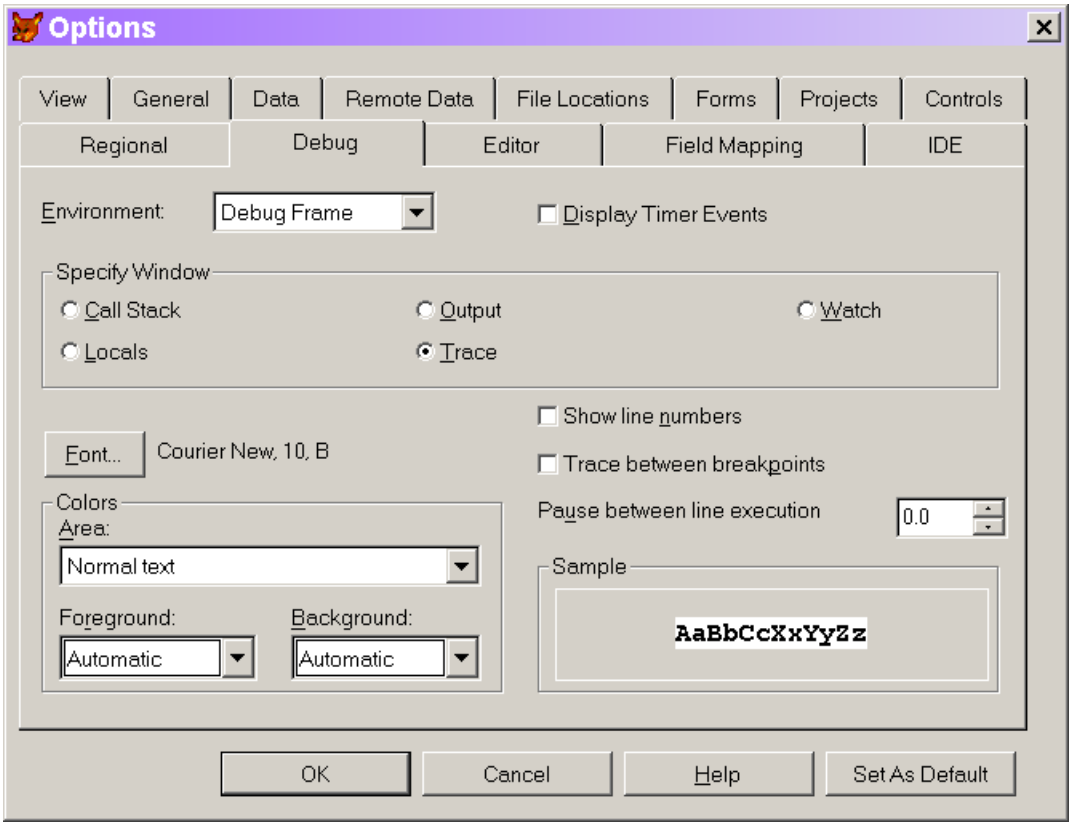

Figure 10 Customizing the Debugger—The option group switches between the various windows. Each has its own settings.

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 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.

As figure 10 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 between each executed line.

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.

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.

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 11) contains buttons and checkboxes for a variety of debugging operations. In the Debug 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 Debug frame also has a dedicated menu that contains the same items.

Figure 11 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 Stepping through code below.

Debugger windows-this group contains five graphical checkboxes, one for each of the Debugger windows. 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.

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.

You can do set breakpoints 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. 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 by using the Breakpoints dialog (Figure 12). 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.

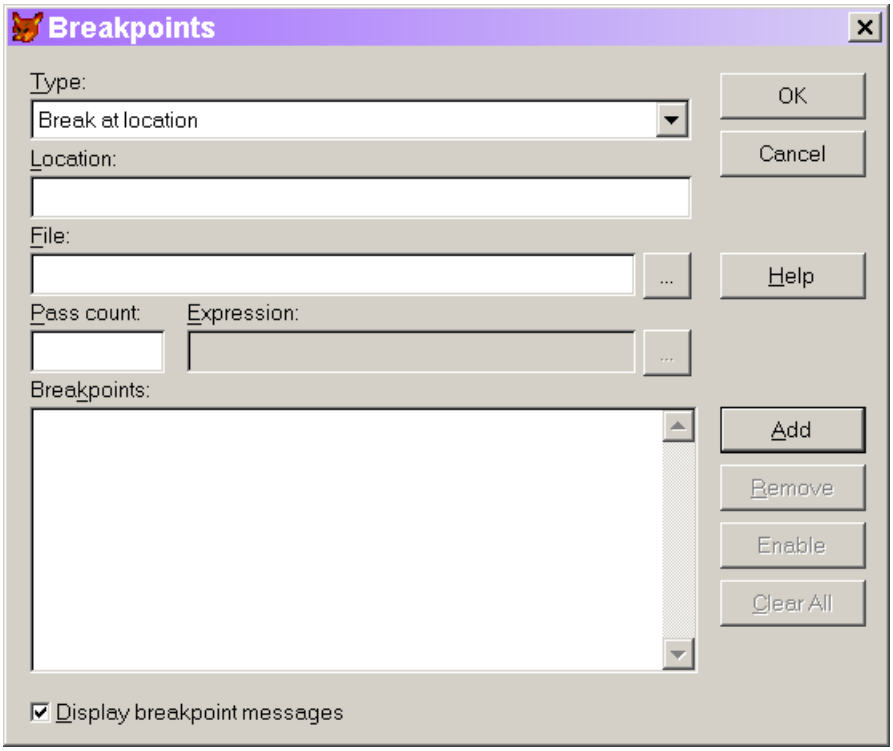

Figure 12 Setting breakpoints—The Breakpoints dialog lets you set all kinds of breakpoints.

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.

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 257th item in a loop, set a breakpoint at the beginning of the loop and set pass count to 256 or 257.

The Breakpoints dialog lets you disable a breakpoint definition, but keep it available for future use. Uncheck it in the list of breakpoints. While this would seem especially useful for working on multiple projects, there's a better solution in that case – see Saving Configurations below.

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.

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 edit it there, as well. (Actually, you can't edit a breakpoint, but you can create a new breakpoint based on the definition of an existing breakpoint.)

Setting breakpoints in the development environment

It's possible to set breakpoints without switching to either the Breakpoints dialog or the Debugger.

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, but there's no visual indication of the breakpoint in the code being edited.

VFP 7 and later feature a significant improvement. 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. 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 in discussed.

The simplest options 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. (There have a been a few reports of problems with the Fix option. Some developers choose to avoid it.)

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. Use this 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. Use this when you know the problem is in the current routine, not in the one it's calling.

Step Out (Shift-F7) executes the remainder of the current routine, returning to the calling routine. Use this when you find yourself inside a routine you don't want to trace.

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 even many) lines of code.

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 two

different ways. You can skip over code you know won't work by setting the next statement ahead of the current position. You can also 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 or forgetting to initialize a variable).

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 expression and send it 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 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
```

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 is displayed, and if the Debug Output window is open, 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.

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.

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.

To solve 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.

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".

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).