Debugging Clarion Programs

Presented by

Russell B. Eggen

RADFusion, Inc.

CHAPTER 1 - DEBUGGING

Why Debug?

It does not matter how bright or clever you are or how well one can study and apply a topic. You will encounter some undesirable behavior when running or testing your code. Bugs exist in all sorts of forms, not just from your code either.

Your code may indeed be at fault. You may have incorrectly coded something or the code works 100% yet still the application does not work. That situation is fairly common, yet the cause is hiding in plain sight – missing code!

It may be the cause of the problem is the vendor's fault. Documented behavior does not match what you are observing, or it just plain does not work.

The problem may stem from a 3rd party product inserted into your code. That can be tricky to isolate. Common in this group is the 3rd party vendor says its Softvelocity's bug and vice versa. There are reasons to come to either conclusion.

Bugs do arise from data errors too. How many times have you tested code, proved it works, yet when installed at a client's site, it does not? Ever try to hook up a relation only to find it does not see related records or even worse, the wrong ones? Usually, but not always clients introduce these bugs, most commonly from conversion processes.

Don't Code Bugs!

In a perfect world, you would just not code bugs. Some Clarion developers claim they don't need debuggers as they don't code bugs. These few state they use templates to generate perfect code every time. Granted, that is a great tool to prevent common bugs. But those that state they don't code bugs are not believable. When they wrote those templates, did it work the first time they used them? The more complex the template, the more ridiculous their claim is.

The best advice on coding I ever received stated simply; "If you are not coding bugs, you are not coding hard enough."

That does not mean one ships bug ridden code! It means that no matter how careful you are, you *should* be seeing a bug here and there. Coding, compiling, testing and fixing are all part of the development cycle. Just writing code is a small part of the effort.

Just like a writer who never makes grammar and spelling errors, no developer is immune from syntax errors and typos (which the compiler catches), or a short burst of temporary bad logic.

Computers Never Lie, But They are Often Wrong

One common, yet frustrating debug process is the "there is nothing wrong with this code, but my application does not work". The temptation here is to jump to conclusions. By that I mean not a proper investigation as to *why*.

I was once working on a bit of code that had a cosmetic issue that was driving me nuts. A second pair of eyes is often useful. This very talented programmer jumped to a conclusion by stating it

must be C6 and thread timing. Turned out that a SUSPENDed thread never gets a chance to cycle through it's ACCEPT loop. The solution required redesign of my code.

Of course the old favorite that always comes up when a new release arrives. "My code no longer works, went back to the previous version, it works again. Bug in new release!" Hate to break this to you, but that is *not* a proper investigation! It is an *incomplete* investigation as observation of a bug is the first step. If you can still ask "Why?", you need to dig more. How many times has something in Clarion been fixed, when no one knew about the bug existed only to cause a slight change in behavior? You usually see this around the drivers or statements that may affect them. You need to find out if something changed in the statement or driver. If so, you may need to adjust your code.

With Clarion 6.x, and the threading changes from earlier releases, expect behavior changes. How you deal with version issues is a good reason to become good at debugging, especially when upgrading your existing "stable" applications. Clarion 7 is just around the corner.

One infamous example of this was an undocumented change between Clarion 5 and 5.5. About 3 SQL tables stopped giving us a unique ID. This was passed to a function that required a unique ID, which promptly GPF'd. Turns out one of the driver strings were case sensitive in 5.5. This raised a new connection and the expect ID did not match, thus the function failed with a GPF. Of course, the function was fixed to not GPF as well.

What is a good Debugger?

To answer that, it depends on what you need at the time you notice non-optimum behavior in your application. For the rest of this session, I intend to cover the following topics:

- The Clarion Debugger
- Driver debugging
- API Debugging
- Third party offerings

Each of the above you should know. You may not need them all for most things, but you will need them eventually.

CHAPTER 2 - THE CLARION DEBUGGER

Introduction

The Clarion debugger has been around since it made its debut back with *Clarion for DOS 3.0*. That is the first debugger Clarion developers ever had. I actually liked it as it was remarkably similar to the *MicroFocus Level II COBOL* debugger. But that is all I am going to say on the matter as few Clarion developers if any, still use DOS debuggers.

When Clarion for Windows arrived on the scene, it too had a debugger; two debuggers actually: a 16-bit and 32-bit version. Today, as of Clarion 6, there is only the 32-bit debugger as Clarion 6 cannot make 16-bit applications. If you need 16-bit applications, you must stay at Clarion 5.5 or earlier.

The 32-bit debugger is my focus today as I don't support 16-bit applications anymore. Also, Clarion 6 is the version I do all my development with.

Threading

Clarion 6 gave us true threading. This impacted the debugger as some variables live on threads. What this means is that the Clarion 6 debugger does *not* behave the same as the earlier version. It couldn't if it were to support threading. I'll be covering this in more detail shortly.

Launching the Debugger

There are two ways to do this, via the IDE or via shortcut/command line. The popular way is via the IDE. More on this shortly.

Setting up Your Project

If your project is not set up properly, you can't use the debugger effectively, not matter how you start it.

Debugger project options

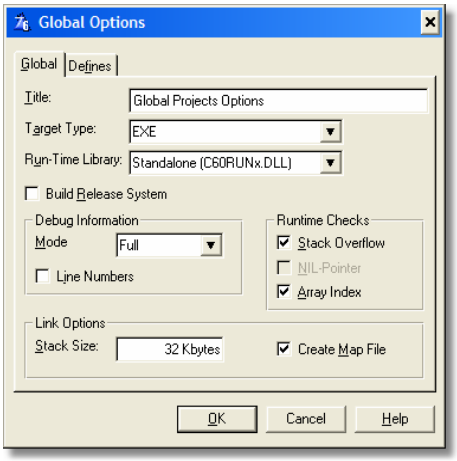

Figure 1 – Project settings for debugging.

The Application Generator by default has several options set to allow automatic access to the Debugger. The project editor is the same if you hand code, so these settings have interest for the hand coders too. These are the required settings:

The **Global Options** sheet contained in the Project Editor should have **Debug Information** set to **Full**.

Build Release System should be disabled in the **Global Options**. (If this step is omitted, have fun debugging your program in assembler mode!)

While not required to enable the debugger, the **Runtime Checks** should all be set. This provides extra debug checks.

The application now includes the information the Debugger needs. You can also turn on debugging information for a single module in the project. The advantage of this is that it reduces the overhead for the debugger. To do so, select only the source module you need to debug, and press the **Properties** button. When the Compile Options dialog appears, set the options as described above. You can also selectively disable debug information in specific modules by setting debug information off. However, you can also control *which* modules to activate when you first enter the Debugger.

32-Bit Debugging

Clarion 6 provides support for 32-bit project targets.

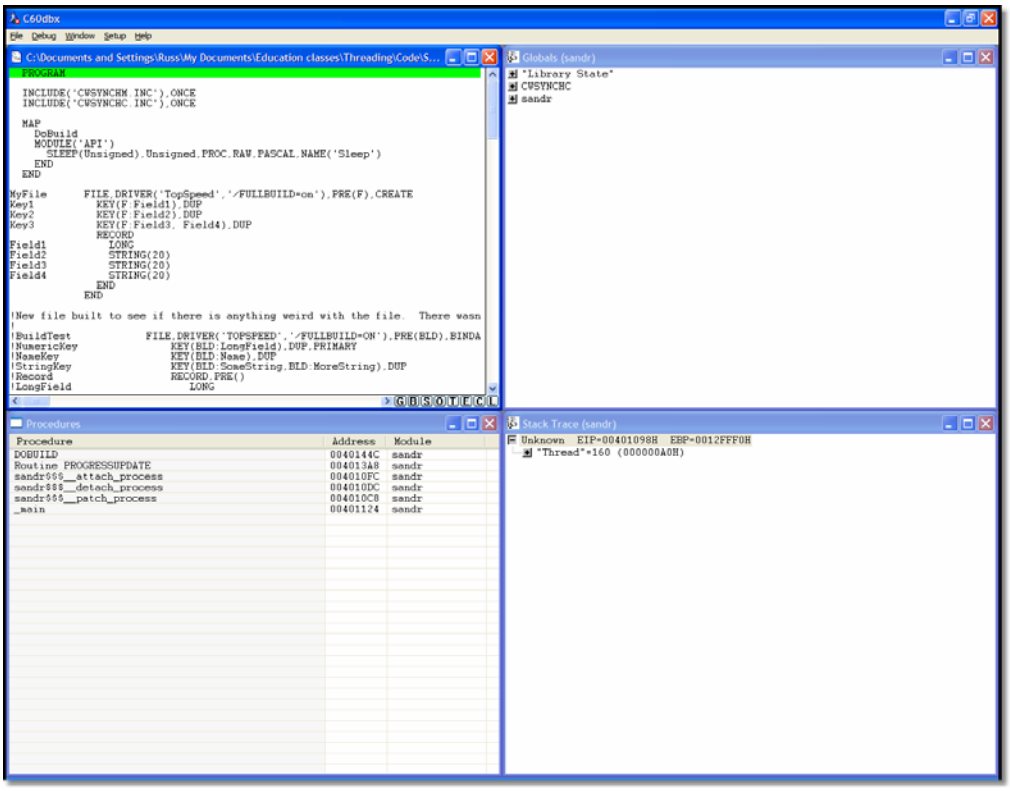

Figure 2 – The Debugger when first started.

The initial view is four windows as shown above. Starting at the upper left and going clockwise, you have the source, global, stack trace and procedures windows. The contents vary on the project being debugged.

There may be times the debugger shows a lot of source windows opening and then losing focus. Don't worry about it. The only source window you are interested in is the one where the problem procedure lives. The Clarion 6 debugger has 3 ways of finding what you are looking for. Look at the following figure:

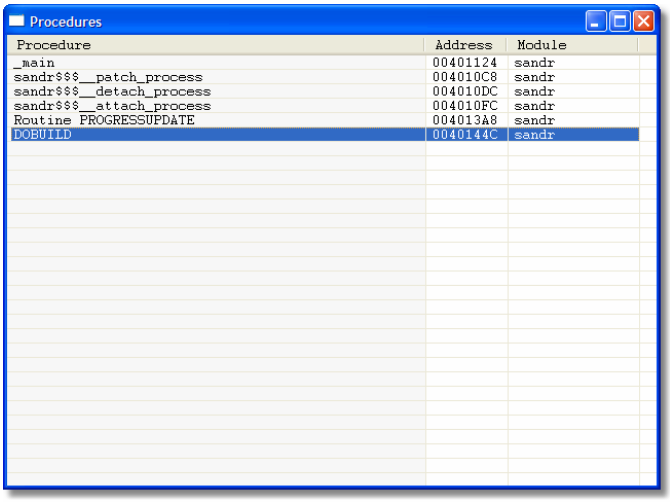

Figure 3 – Sorted by procedure name.

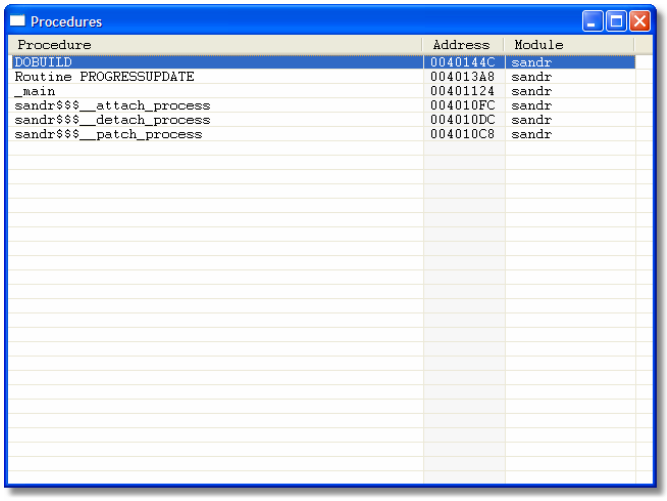

Figure 4 – Sorted by Address

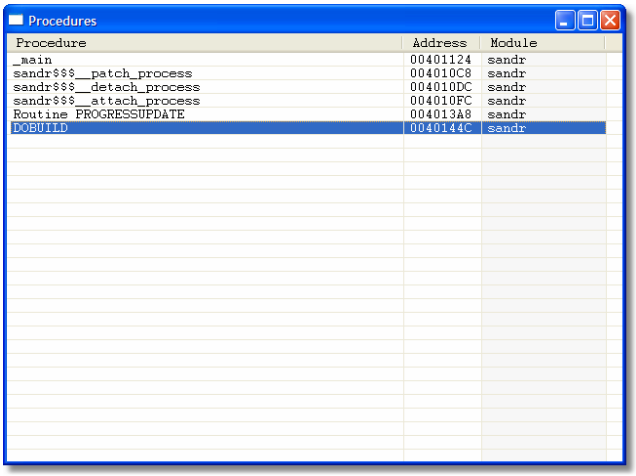

Figure 5 – Sorted by Module.

If you know the name of the Procedure, **click** the PROCEDURE HEADER to sort in ascending order. **Click** it once more to sort descending. Same for the other columns if you have the address information or you know the module name the procedure resides.

Each column has a step locator to ease finding what you are looking for as well. Best results in my opinion are when the sort is by procedure name.

To open the source, simply **click** on the PROCEDURE NAME and the source file instantly opens for you (which is why you ignore any and all open source windows up until this point).

Break Points

Break points are lines of source code causing the debugger to suspend executing the application at that line of source. When this happens, the debugger takes over.

You need to understand break points cold. If not, your debug session consists of wasteful stepping through underlying source code (such as ABC, Handy Tools or other class code). If you are really bored, you can do this, but I'd rather not. I'll get to stepping through source soon.

But why use them? First, if you don't set at least one, when will the debugger stop running your application? Second, you need to pause the code execution by break points so you can inspect what the application is seeing at a specific point. I mean variable contents.

Where do you put breakpoints? That is a very good question. Remember, debugging is an investigation to find something wrong. If you can reproduce a problem, then it's easy. You have a good idea that the bug lives somewhere between the earliest place in the code's execution and the latest place. That is two breakpoints. Place one midway between those two points if you wish. You are now set to let the debugger run the application now.

Source Stepping

At the lower right of each source window is a button toolbar that looks like this:

Figure 6 – Source stepping navigation button.

The buttons mean the following actions:

Button	Action
G	**G**o
B	Set **B**reakpoint
S	**S**tep Assembler
O	Step **O**ver Assembler
T	S**t**ep Source
E	St**e**p Over Source
C	Go to **C**ursor
L	**L**ocate Line

Table 1 – Button commands

Tool tips easily identify each button as you hover the mouse over them. You may also right-click anywhere in the source window to show the following pop menu:

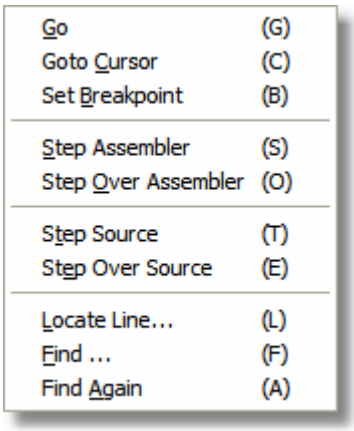

Figure 7 Pop-up menu.

You have two additional commands to find text strings in your source. The letter in parenthesis (or "brackets" as our friends across the pond like to say) is the keystroke needed to carry out the same action. The keystrokes are always there, even if the pop-up menu is not.

So you have break points set and you are ready to go. Here is a common mistake. I've seen many people who hate the debugger start stepping or stepping over source at this point. If I had to do this action, I would never use the debugger again as this is far too much work. So what is really going on here? Your application is not yet executing the module (most likely) where the break points are!

So the execution starts at some source about global run and you find yourself deep in ABC in two mouse clicks. Who would not be lost there?

You have break points set, *use* them! Break points are to stop your application at that point. So execute your application normally! You do this with the GO command. The instant your application hits the break point the execution halts right there. *Now* you can start stepping through and stepping over code!

The Data Windows

Its one thing to step through code, but what does the data look like at the exact point a break stops execution of your program? You have two types of data to deal with, global and stack data. The data is arranged as a tree structure. The default is a collapsed state, so to see any details; you must press the plus sign to expand the tree, minus sign to collapse a tree.

The Global Window

This is the window where you find global data. This includes library data such as the last error, which control has focus, first field, last field, etc.

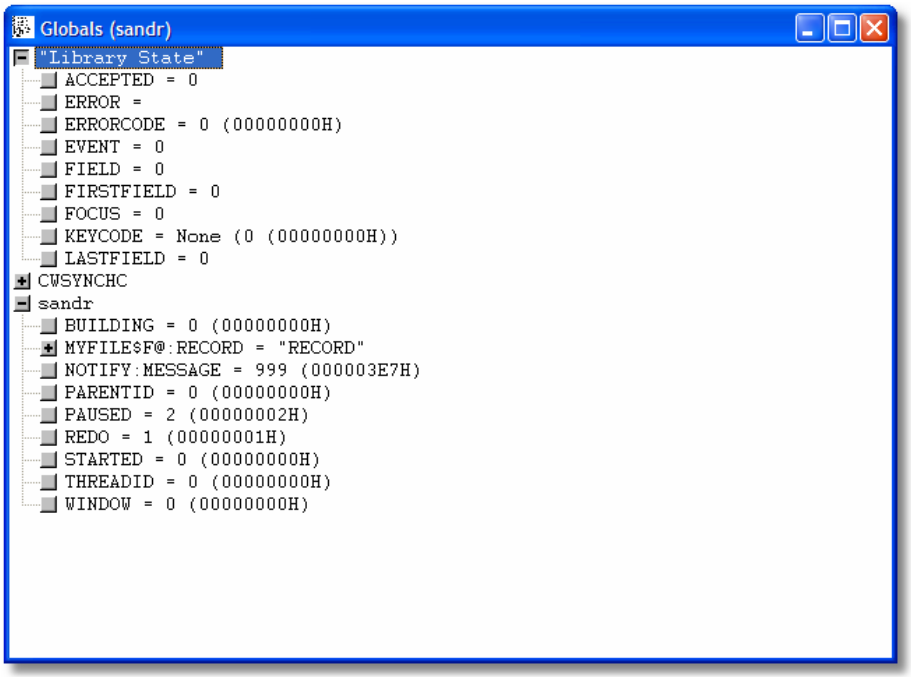

Figure 8 – Global Data Window with expanded elements.

What populates in this window and their values varies by project and where the program has suspended running.

Stack Trace (Local) Window

This is the window for variables that are local to the current *executing* procedure. You won't see valid data in it until the debugger enters the procedure:

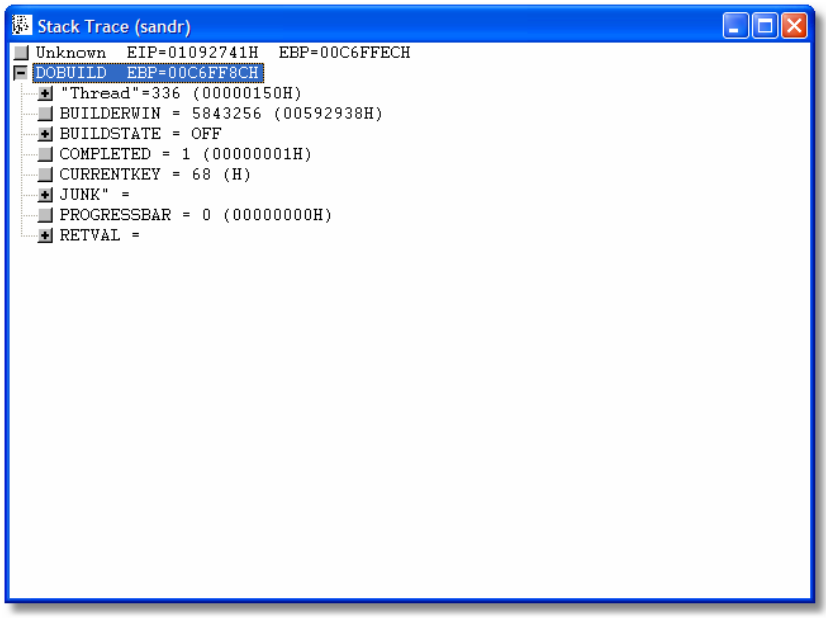

Figure 9 – Debugger in a local procedure.

The display in this window changes as you execute the debugee. There is more to cover with these windows, which I'll address later.

A Simple Debug Session

To illustrate a typical debug session, Clarion 6 ships with a broken application. The name of the application is *Orders* and it requires the ABC templates registered. It is located in the *Clarion6\Examples\Tutor\Debug* folder. I've included a modified version of this application. Open the application and compile it. When you run it, you notice the highlight bar can page down and page up. It can scroll up, but does nothing when scrolling down. That is a bug. Quit the application.

Press the DEBUG button to launch the debugger from the IDE and start executing the *Orders* application. Your initial view of the debugger is this:

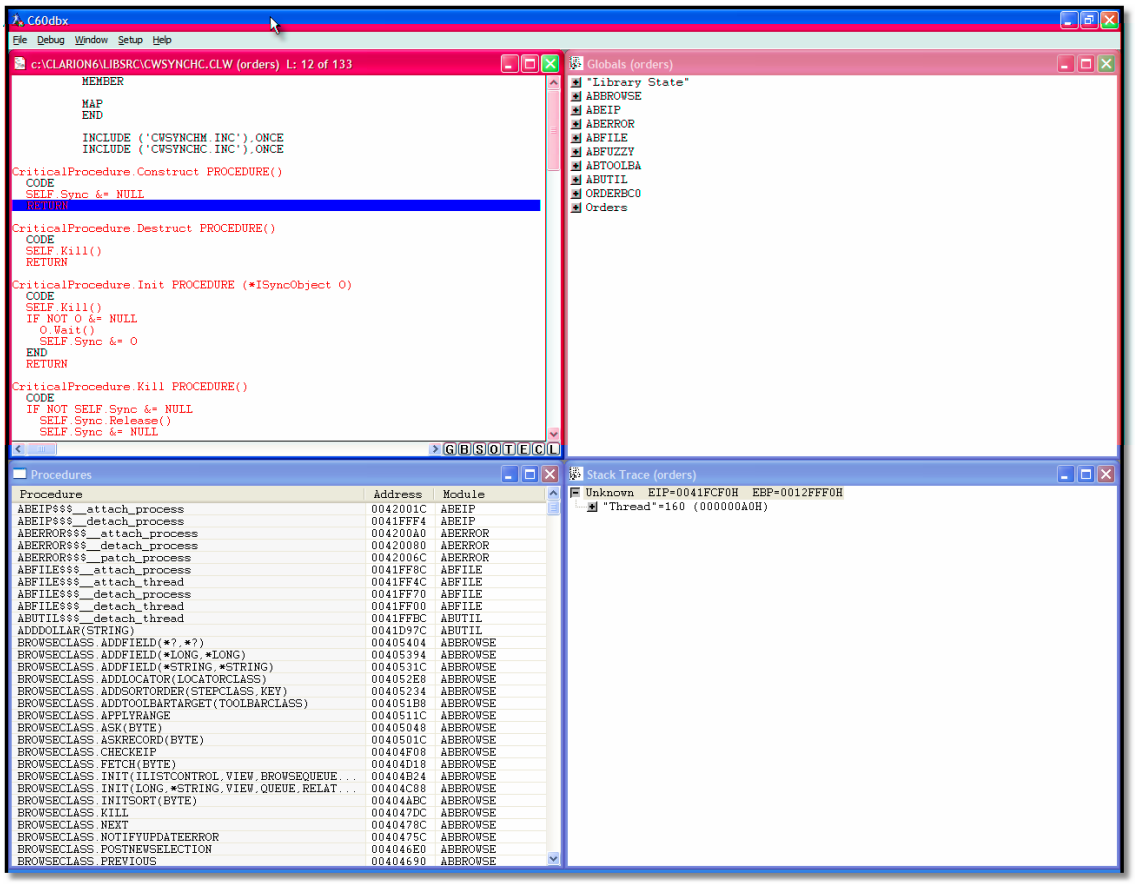

Figure 10 – Initial view of debugger desktop.

Even though this is a simple, 2 procedure application, the Procedures window appears quite busy. This is because all the classes and their source files appear in the list. They are there because they are used by this application, directly or indirectly.

I've shown the bug is in the *BrowseCustomer* procedure, so you can locate that procedure by any means you like. As soon as you select *BrowseCustomer* from the procedure list, the source for it opens and the highlight is on the first source line.

The bug was the inability to scroll down one line. Where is scrolling controlled? With the source window in focus, **press** F to open the Find dialog. **Enter** *scroll* and **press** OK. The first stop is in the window definition, so that is unlikely to be the source of the bug. **Press** A to find again.

Now the highlight is on a method labeled *ScrollOne*. Makes sense as browse lists scroll. **Press** F and **enter** *scrollone* in the entry. The highlight lands on *BRW1:ScrollOne*.

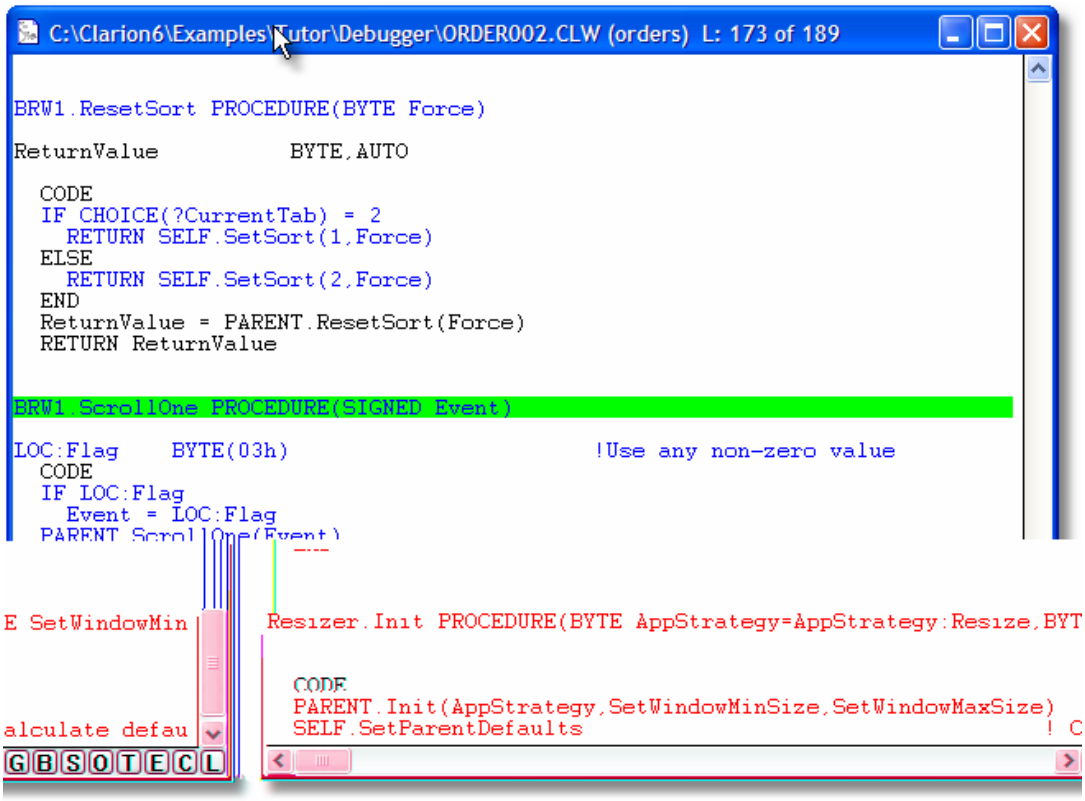

Figure 11 – The Scroll One method.

A small method, so highlight the *IF LOC:Flag* and **press** B or **double-click** to set a break point. Break points are indicated by the red color. If it looks yellow, you are not going color blind. Green and red produces yellow. Move the highlight up or down one line to see that.

You are set up to debug. Remember, the program is really not yet executing this procedure, it is suspended at the initial start (and the global and stack trace windows reflect this). You do not want to step trace each line of code to this spot. Use the GO command as I previously covered.

The program starts normally. Open the *BrowseCustomer* window and **press** SCROLL DOWN on the toolbar. The debugger now takes control as it hit the break point. You could look in the stack trace to see if you see anything interesting at this point:

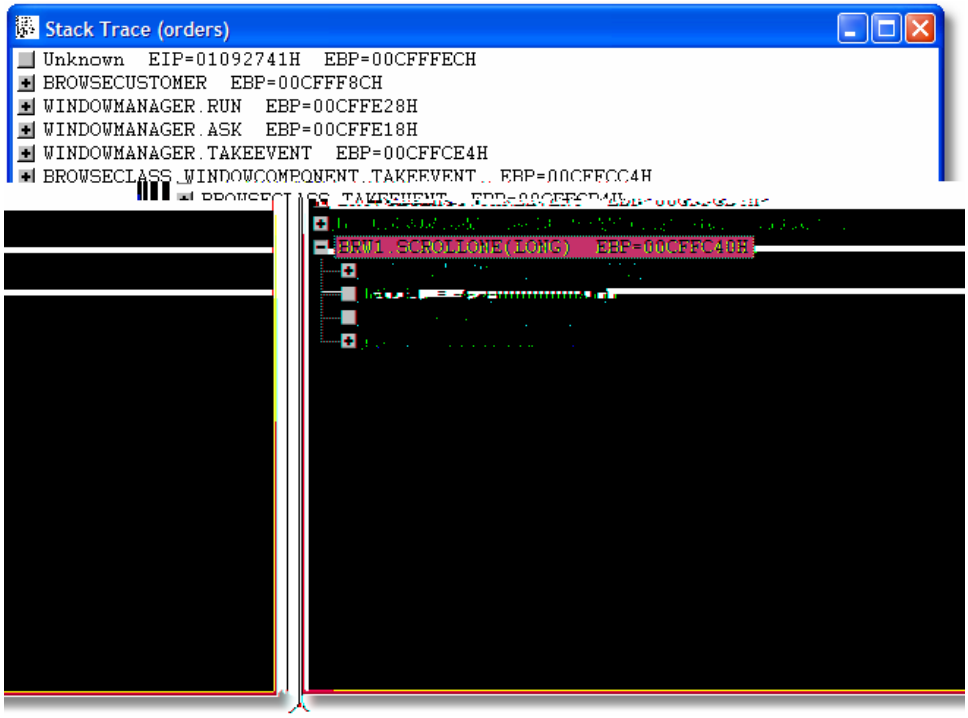

Figure 12 – Stack Trace of BrowseCustomer

The highlight is the current stack (top of stack, bottom item in window). Nothing looks that odd. Step through the code is a good option now. **Press** T to step trace and one line of code executes, and the highlight moves down to the next line of code to execute. Watch the stack trace and **press** T again. The Event variable changed from 4 to 3. Still, nothing obvious here.

The next line of code is *PARENT.ScrollOne(Event)*. This means the next stop is an ABC method as that is what is meant by "Parent". You can step trace or just step. If you just step, then the debugger allows the program to run normally until it returns from the parent call. It will then suspend execution again. Let's try both methods. **Press** E for *Step Over Source* (execute and stop upon return).

The highlight bar locates to the END statement. Nothing really changed, so **press** G. The debugee is now in control. OK, now what? Did not find any bugs. So the code in your application revealed nothing.

Remember, you still have a break point set. **Press** SCROLL DOWN on the toolbar again. The debugger is in control and the stack trace is like we found it before. This time, step through the *PARENT.ScrollOne(Event)* method. When the highlight bar reaches that line of code, **press** T to step trace. You now see this source window:

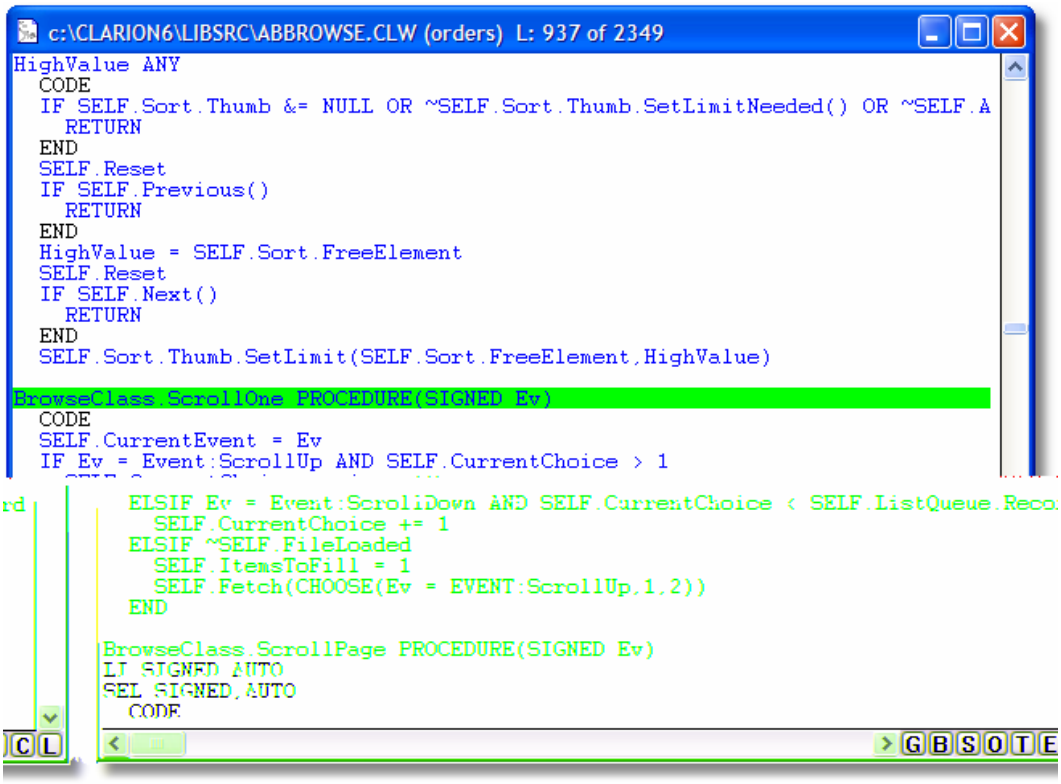

Figure 13 – Debugging ABC methods.

Now look at the stack trace window. Different values loaded and you can see the passed *Event* value (in variable *Ev* here). In order to ensure you don't get lost, when stepping through code in this method, always use the STEP OVER SOURCE command as many ABC methods call other methods. Unless you need and want to step trace other ABC methods.

SELF.CurrentEvent = Ev is the line of code executed once you **press** E. If you look at the stack trace, you can see a node of the current tree is called SELF. Expand it. SELF will always have a "record", so expand that too. Whoa! Lots of items!

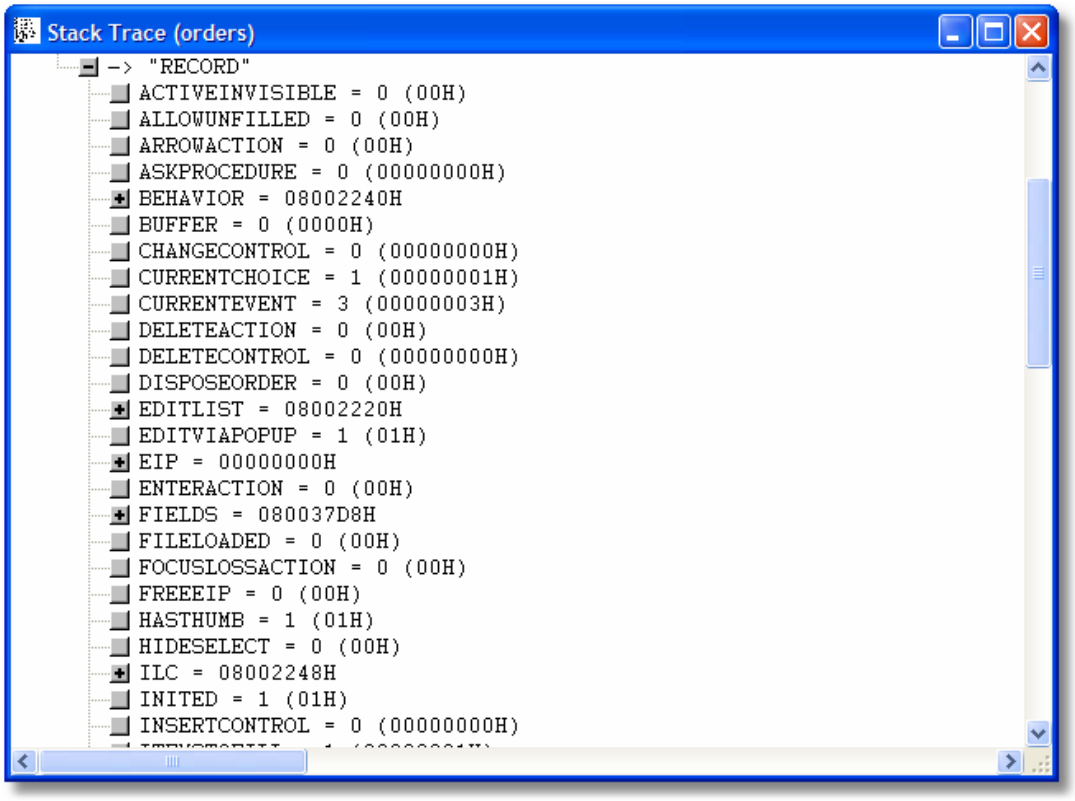

Figure 14 – The BrowseClass SELF node expanded.

This is a list of properties of the browse class. “SELF” means “whatever the current object is”. Since classes don’t know what their instance name is until runtime, “self” is merely a placeholder for whatever it may be. You can see the *CurrentEvent* is set to 3, just as the line of code just stepped did.

The next line of code tests for EVENT:ScrollUp. Here is where the global window comes into the picture. Events are in the runtime library, so the current event is shown there.

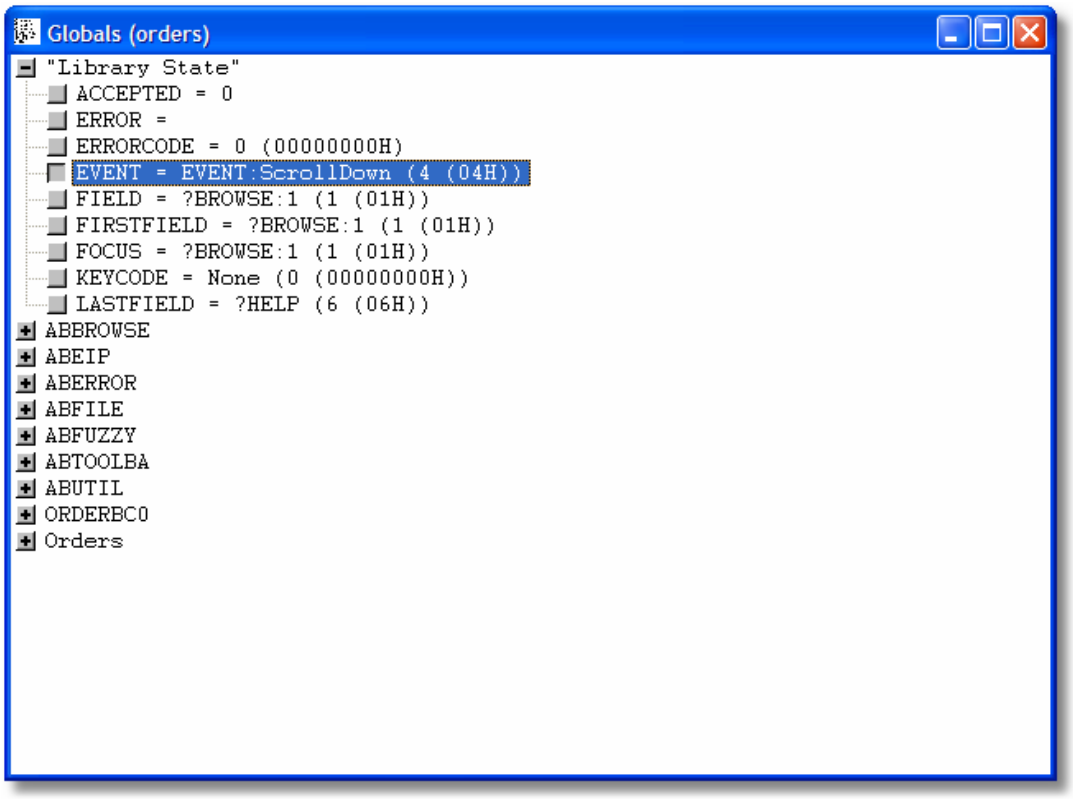

Figure 15 – The last event is EVENT:ScrollDown.

You can see the event is correct as you pressed the button to scroll down. So far, so good. The code is testing for EVENT:ScrollUp so you know this condition code will fail the test. **Press** E to see if it does.

It not only dropped through the first IF, but the ELSIF line too. This means the logic failed both IF conditions. Yet, the second condition is what should have executed! The execution point is now past the point where you are interested. So **press** GO.

The program is now running again. The breakpoint is still there, so **press** SCROLL DOWN on the toolbar again. **Press** T 4 times to get back to the *BrowseClass.ScrollOne* method. You should be on the *SELF.CurrentEvent = Ev* line. *Ev* has the event number passed to this method. It is 3. Remember the *Event* value when the debugger first took over? It was 4. What if you could change the 3 to a 4 before it gets passed to *SELF.CurrentEvent*?

In the stack trace, **highlight** *Ev* and **right-click**. A pop-up menu appears with one of the choices being "edit variable". **Select** it and then **enter** 4 in the entry provided and **press** OK.

Now **press** E and watch the execution. Now the second condition is true! This is different behavior than before! **Press** GO and when the application runs normally, this is what you see:

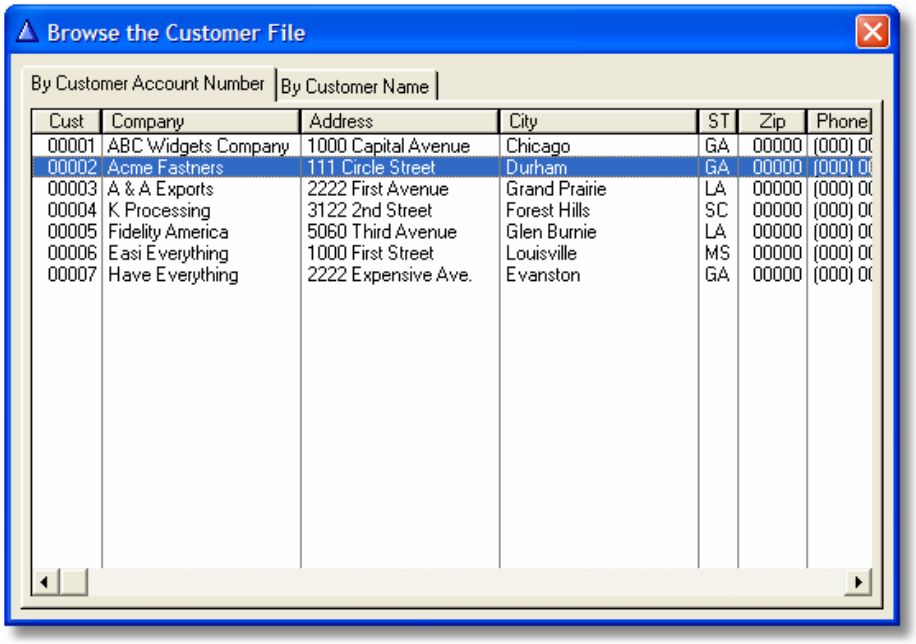

Figure 16 – Highlight moved down one line.

Therefore, the bug is not with ABC, it worked perfectly. But it was passed the wrong event number. The bug is with the local method. You can verify if that is indeed the case. **Press** SCROLL DOWN again to trigger the break point.

To test the theory the wrong event is passed, STEP SOURCE until the highlight is on the parent call. **Right-click** *Event* and change it back to 4 (its original value). Once you've done this, **press** GO. If all goes well, the highlight should be on the 3rd customer down:

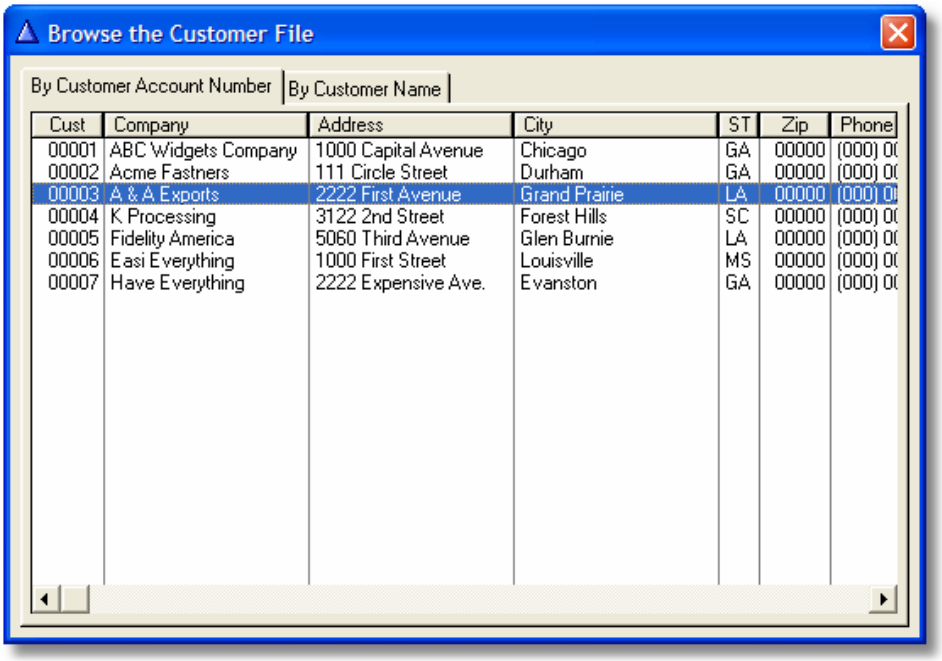

Figure 17 – Proving the bug.

You now know what the bug is and can prove it. The debugger showed you.

While this may be a simple 2 procedure application, the typical debug session is very similar to the steps taken above. This is true even with very complex and involved programs. The difference is you may need more breakpoints and a few "loops" around to narrow your search for a bug.

When bugs can be reproduced, a typical debug session should last about 2-6 minutes, including set up time.

Note: When you are finished with the debugger, shutdown the debugee first. Sometimes that is not always possible with badly behaving programs.

Restarting the Debugee

Sometimes, you might have shut down the program by mistake. Leave the debugger running! **Choose** FILE ▸ FILE TO DEBUG. A file dialog opens looking for your executable. Navigate to the correct folder if needed and find the EXE. The debugger starts the programs and suspends at the first line of execution, just as if you called it from the IDE.

The debugger sort of remembers your break points from the last run in such cases. They are still visible, but they really don't work anymore. The workaround is to simply **press** B twice.

Post mortem debugging

The Clarion debugger may be installed as the system debugger. From the debugger menu, **choose** SETUP and the following option:

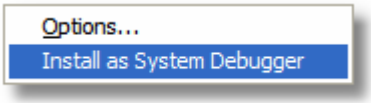

Figure 18 – Installing the Clarion debugger as the system debugger.

If you install the Clarion debugger as your system debugger, and a program causes a GPF, you then get this dialog:

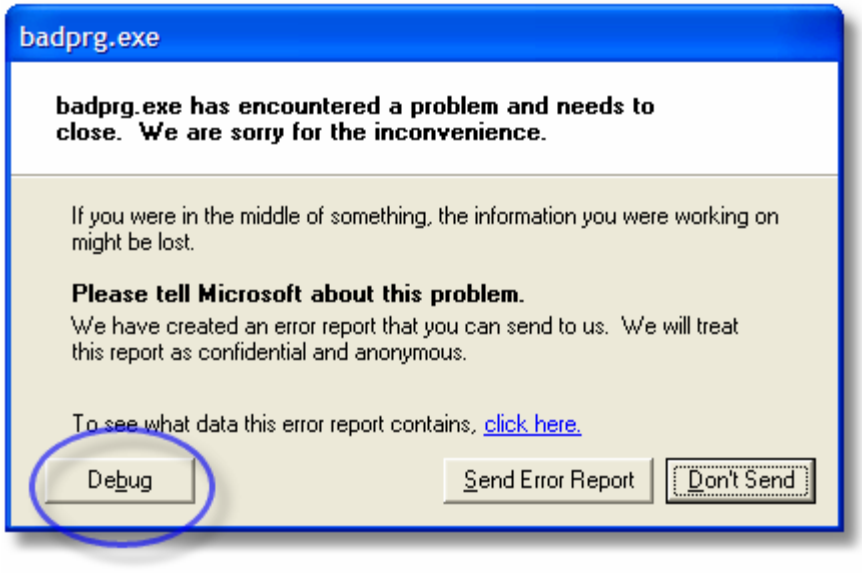

Figure 19 – GPF dialog as seen in Windows XP.

Notice the DEBUG button. If you **press** this button, this will start the debugger with the dead program (it is not running). You may be prompted to load a source file. I've never needed that feature, so I close it. You should get an access violation message that would match the address if you inspected the details on the GPF window above.

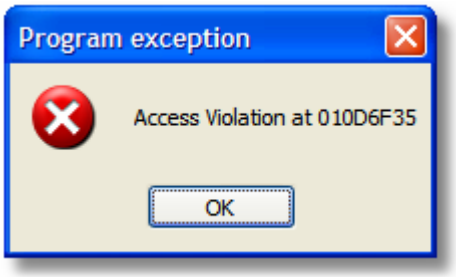

Figure 20 – Debugger displaying the access violation.

Simply **press** OK to close the message. The stack trace window is really the only window that is important.

The last item should be labeled "unknown". That is the point where the GPF happened, but rarely is that the *cause* of the GPF. Look for the last good label before this point. In this example, it is *_main* (which is in every Clarion program). **Right-click** and you get this menu:

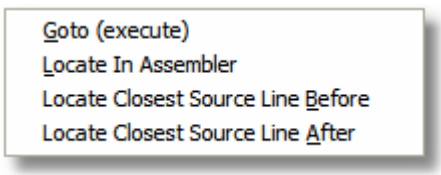

Figure 21 – Options to find that last bit of good code.

Choose *Locate Closest Source Line Before* if you have the source (and you should). The source window opens with the last line of code executed before the GPF:

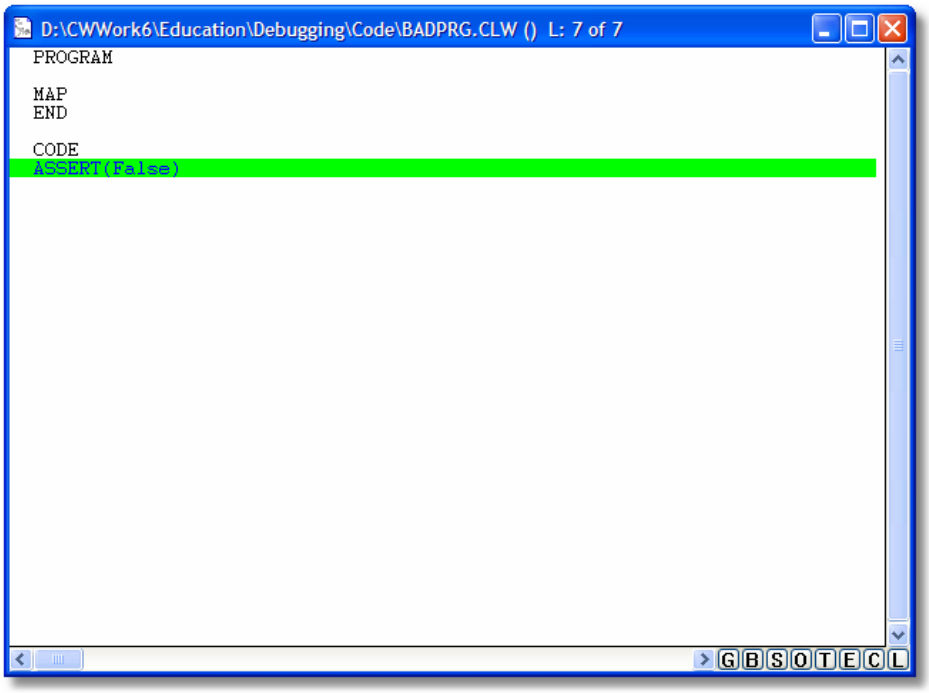

Figure 22 – Line of code that when executed, GPFs.

OK, that is cheating and this program's design is meant to GPF. However, I wanted to show you that this option does work and is no different than debugging a complex program that GPFs.

What you should take away from this is that a GPF is really your friend when finding bugs. Just like compiler errors in finding poorly written code.

What about more complex problems? Some applications GPF the moment you launch them. Remember, the GPF is not the problem. A GPF *reports* a problem. A program could GPF on perfectly valid code!

The next example, the program GPFs when you press a button. Going to post-mortem debug mode, the stack trace shows cryptic information:

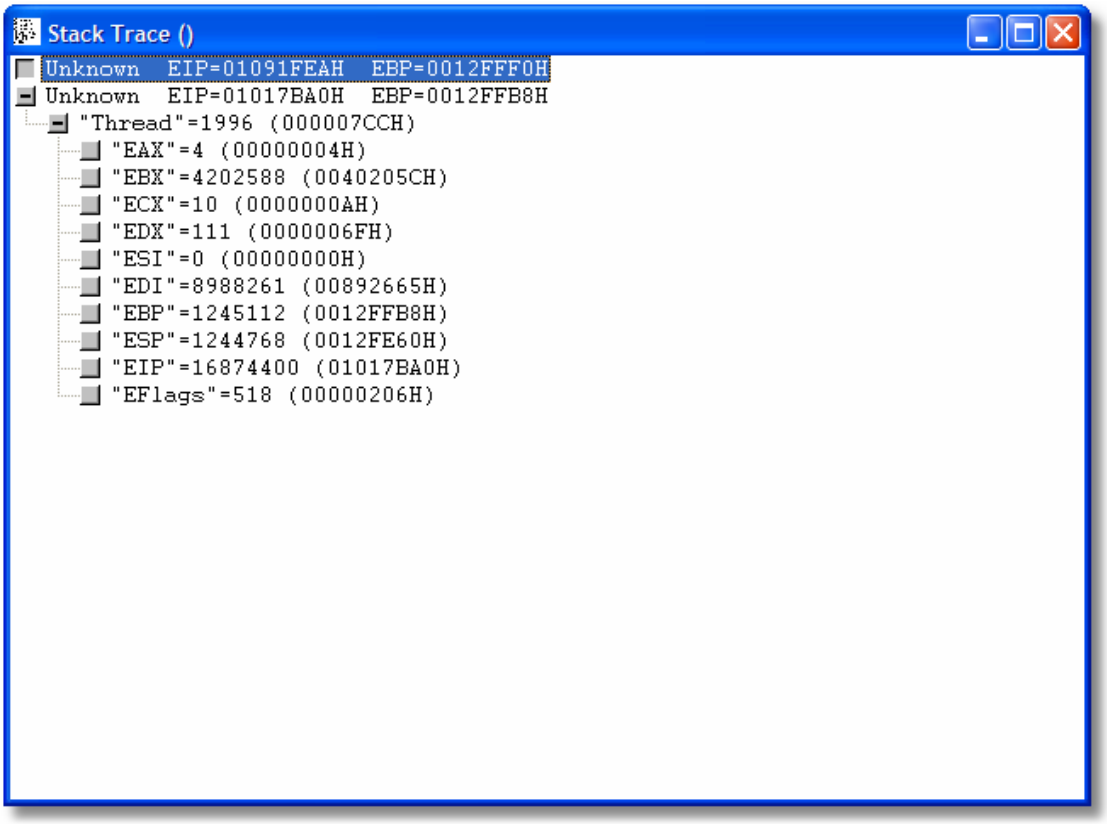

Figure 23 – Not much useful information finding the GPF.

The above may be useful if you are good with assembler. My assembler is out of date (8088 assembler) and I'm quite rusty with it. Up the proverbial creek without the proverbial paddle? What about the global window?

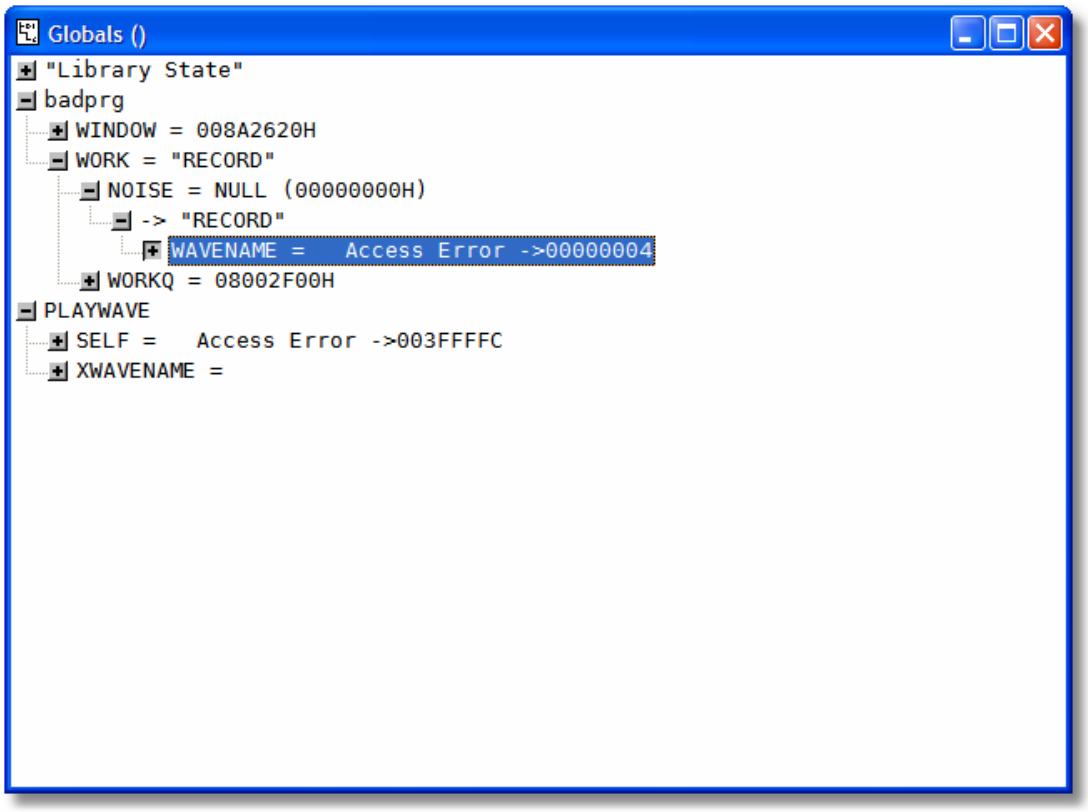

Figure 24 – Global window after a GPF.

When I expand *Playwave*, I see Access Errors. That is the error itself, but what about the cause? Going up one node, expand *badprg*. The "window" node looks fine as there is a value there. Expand *work* and I see *noise* has a null value. *WorkQ* appears fine. Expand *Noise* and you see *Record*. Expand that and you see Access Error. This is the earliest error. The only thing you do know at this point is that a problem exists with *Noise*.

Therefore, inspect the source code. When you press the button, you get a GPF, so look at the code for that:

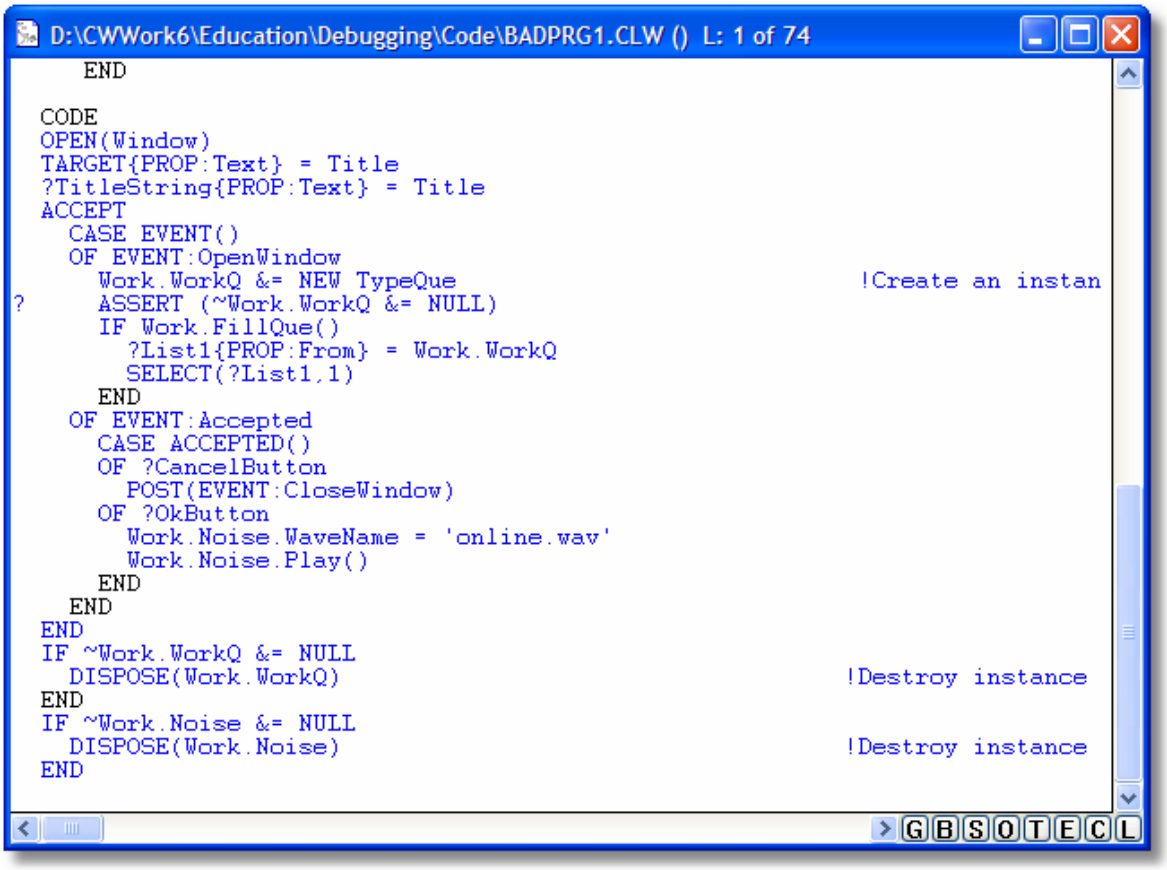

Figure 25 – The source code looks fine, should play a wave when pressed.

The above code as seen in the debugger looks fine, nothing wrong with it. But it crashes. So, something *is* wrong. You cannot resolve the problem here, so your only option in this investigation is to go earlier. Yet at first glance, there is nothing earlier that applies to the problem. When I say "earlier", the only option is before the CODE line.

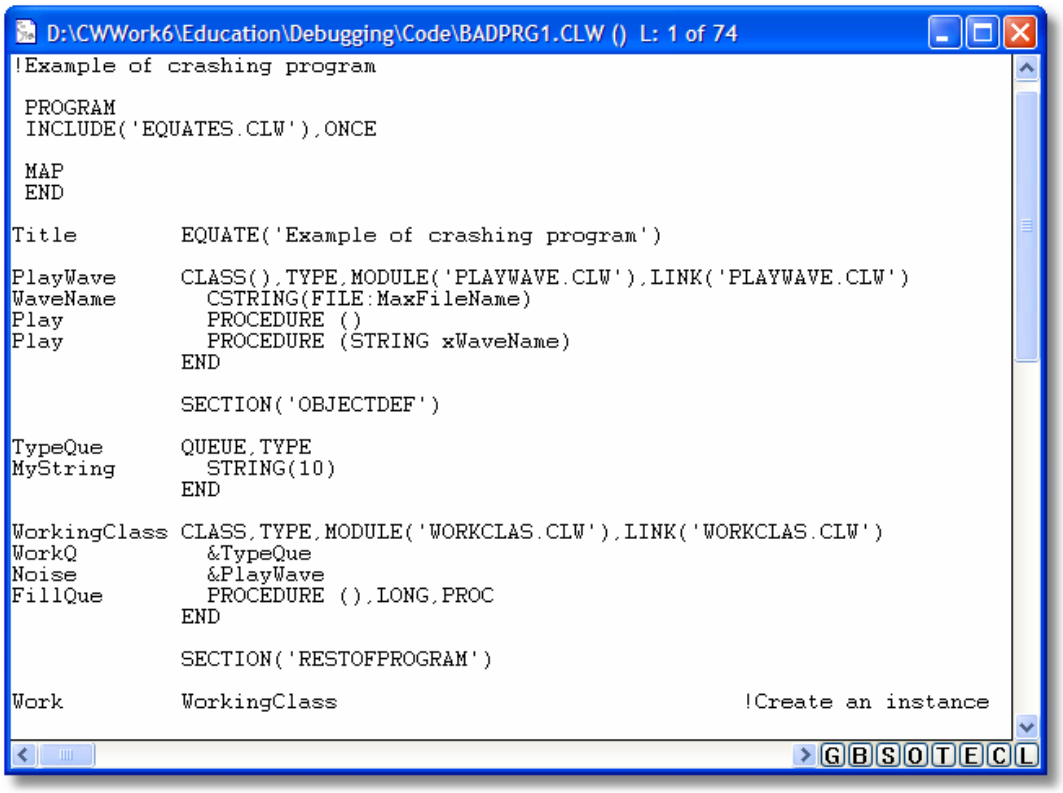

Figure 26 – The WorkingClass definition and instantiation.

There is nothing wrong with the class definition, nor its instantiation. This should work too (and it does). *WorkQ* and *Noise* are both references. *WorkQ* fills a browse list with data, *Noise* plays a wave file.

In figure 25, you can see a reference assignment to *WorkQ* and an ASSERT that asserts the reference is not null. But where is the reference assignment for *Noise*? What if you placed an ASSERT for *Noise*?

There is your problem, all the code is fine, yet it crashes. In this case, the problem is not the code that is there, it is code that is *not* there! To test that, put an *ASSERT(~Work.Noise &= NULL)* in the code. You could put it in the Open Window event or the button accepted event. If the assert fires, you proved your theory.

That is something you really cannot find with MESSAGE and STOP statements.

Remember, any object not instantiated GPFs when your code tries to use it. If this is the cause of GPFs in your application, MESSAGE and STOP cannot possibly help you. ASSERT does help you. Part of ensuring you write safe code is to ASSERT something is there. If not, the assert fires, alerting you that you forgot something.

One of the best “hide in plain sight” causes of GPFs is a class definition that is valid, yet it cannot work. Here is a conceptual example of this:

```
ABCClass        CLASS,MODULE('ABC.CLW'),LINK('ABC.CLW',_ABCLinkMode_),DLL(_ABCDLLMode_)
                END
```

Listing 1 - Conceptual ABC Class definition.

There is nothing really wrong with the above listing. Yet, if you try to use anything from this class (assume it has properties and methods), the program GPFs. The GPF happens about the time when you launch the program. You could post-mortem debug it and you will find that anything about the above class has "Access Error" next to it. That means it was not instantiated or could not be instantiated.

Notice the LINK and DLL parameters. You have those two *_ABCLinkMode_* and *_ABCDLLMode_* flags. If these are missing from your project defines (which defaults to zero) or they are set incorrectly, expect a GPF. This is the same cause if your app works as an EXE but GPFs as a DLL.

The same is true for 3rd party classes that follow this convention (but use their own flags). Fortunately, templates usually prevent this as they are responsible for putting those flags in the defines section of your project *and* setting them correctly. If not, speak to your vendor as they have a fatal bug.

Debugging and Threading

Before Clarion 6 came out, one of the useful things you can do with the debugger is place variables in the watch window. Doing this is child's play as all you need to do is **right-click** on a variable and **choose** WATCH. If this is your first watch variable, a new window appears. You can place anything in a watch window, stack trace and/or global variables.

The purpose is to watch what happens to variable contents as code executes in your program. As your code loops through ACCEPT for instance, you can see if the variable contents change, and if so, to what. Global variables never go out of scope in a watch window, but stack trace variables can (if you leave the procedure to run an ABC method for example). If a variable goes out of scope, it will redisplay correctly once execution returns to the procedure.

That is, until Clarion 6 came along. The threading model is a major paradigm shift for Clarion developers as much as when ABC first shipped.

What happens is that a variable placed in the watch window remains, but its value is gone from the watch window. Instances of threaded variables are allocated at run time, so the debugger cannot show these variables directly. Instead, stick to the stack trace window.

Conditional Breakpoints

The Clarion debugger does not support the concept of conditional breakpoints. It is not a feature of the debugger. This means debugging long processes that crash somewhere in the middle are next to impossible. Or are they? There is a way to use the concept of condition breakpoints, just not inside the debugger.

Let's say that you are running a long posting procedure. This is not uncommon in accounting type applications. Yet, when you get to record number 895,832, the program GPFs or hangs or just quits. What is worse, you use a key or index for the processing, and likely filtered in some fashion. This means the chances that the problem *is* record number 895,832 is next to nil. So inspection with a file viewer is useless.

Say the process code looks like this:

```
ACC:AccountNo = LOC:AccountChosen
SET(ACC:AccountKey,ACC:AccountKey)
IF ~Access:Accounts.Fetch(ACC:AccountKey)
  !Process code here
END
```

That's a simple enough concept, except it crashes after 5 minutes of processing. Just a simple edit and you have your own condition breakpoint:

```
ACC:AccountNo = LOC:AccountChosen
SET(ACC:AccountKey,ACC:AccountKey)
IF ~Access:Accounts.Fetch(ACC:AccountKey)
  Idx# += 1
  IF Idx# = 895832
    Dummy" = 'X marks the spot - Set break point here'
  END
  !Process code here
END
```

Say what you want about implicit variables, but this is a good use for them. This is one-off code, so you really don't care about the downsides of using them (but check your spelling!). Place the break point where indicated.

Note: The text is easy to locate using the Find function in the debugger. Search for "*X marks the spot*", a bit of text not likely to occur anywhere else in your application.

Now it is a simple matter of using the GO command. While you wait, go get a cup of coffee or take a short break. Once the condition becomes true, the debugger encounters the break point and takes control of the execution of your application. It now becomes a simple matter of step tracing and inspecting variables.

Debugging Runaway Processes

Imagine the horrifying scenario where you have a program that is caught in an infinite loop? But it gets worse, this looks like a hang. But more bad news, there is no window open, so nothing appears on your desktop. And worst of all, you ran this application without the debugger. How does one get the debugger inserted in this process?

The Clarion debugger has quite a few command line options. Here is the list:

C60DBX [redirection file] [INI file] [-s dllname [-s dllname ...]] program.EXE [parameters]
C60DBX [redirection file] [INI file] -p PID [-e EventNo]

The second form of the command line usage above is used to attach the debugger to an already running process.

redirection file A string constant that names an optional redirection file. The file name MUST have an RED extension. If present in the command line, the file name overrides the default redirection file specified in the Debugger Options dialog and replaces it with the new name. The file name must be enclosed in quotes if it contains spaces.

INI file A string constant that names an optional redirection file. The INI file MUST have an INI extension. If present in the command line, it overrides the appropriate default INI file name (C60EE.INI or C60PE.INI) located in the BIN directory. The file name must be enclosed in quotes if it contains spaces.

dllname The name of a DLL loaded by the program. When the system loader loads this DLL into process's virtual memory, the debugger suspends processing when the DLL entry point is encountered.

program.EXE The program name to be debugged (the debuggee).

parameters Optional parameters passed to the debuggee

PID The debuggee's process ID in decimal or hexadecimal format. In the latter case, the number must be ended with an H character (i.e., 27A3H).

EventNo The debug event number. This number is passed by the system if debugger is installed as a System Debugger and it is invoked because of a fault.

A runaway process is an application already running with no clear way to safely shut it down. You could use the task manager, but you may cause a memory leak, or risk leaving data files in an opened state.

To shut down a runaway process, open the task manager. Find the rogue program and one of the columns is a **P**rogram **ID**entification or PID number. Let's say its 8088. **Press** START ▸ RUN and then enter:

```
C60dbx –p 8088
```

The debugger then starts debugging the runaway process. You can then put a breakpoint in the code and when it hits it, do whatever you need to have the application shutdown on its own. What action you take there largely depend on the situation but one useful action is to edit variables so the condition to shutdown the application begins.

Break In

As an alternative, *Capesoft's BreakIn* tool can be handy. Just start it from Clarion's IDE (see Accessories menu) and load the runaway application. It will load needed DLLs then show you the stack trace. Unlike the debugger, this stack trace is "upside down", in other words, the most recent stack is on top, its caller is just below it and so on. While an app is spinning out of control, this display can seem a bit odd looking. Just **press** BREAK and *BreakIn* suspends the app.

You may then inspect the stack trace in detail and usually spot the block of code that is causing the application to run endlessly.

Summary

This section attempted to highlight the important options available to you when compiling and debugging your program.

I've shown how easy the debugger is to use and setup. I've also shown you do not (and should not) have to step trace every line of code in your program.

I've explained that debugging is not a tool per se, but an investigation. The debugger is a tool and a powerful tool, which is easy to use despite the power it provides.

CHAPTER 3 - USING API TO DEBUG

Introduction

I am resigned to the fact that no matter how much I demonstrate the Clarion debugger, there are Clarion developers that will dangerously continue to use MESSAGE and STOP for debugging reasons. MESSAGE and STOP are for displaying messages and should only be in your programs for their intended use.

The appeal using these statements for debugging is obvious, they are easy to insert into your code as a substitute for break points. But due to their own event handling, they can alter behavior of your applications, but not always. An ideal solution is to have something you can place in your code as easily as MESSAGE or STOP, yet be safe.

DebugView

This is a utility from Sysinternals. It is free. It is not part of the materials as the author of the utility likes to keep track of downloads from their web site. You may find it at http://www.sysinternals.com/ntw2k/utilities.shtml.

All *DebugView* does is simply trap events and report them. It is configurable so that you may apply certain filters and color code them if you want (to pick out certain events, or indicate severity levels is one use for filters). You use API calls to send messages that *DebugView* can trap and report. Those calls are in a class for your use.

Debuger Class

That is not a typo that is how it is spelt – deliberately. This class is ABC compliant in every way. It even has a template wrapper for use in apps. It is also handy for hand coded projects too. One of the best uses is a safe replacement for why people use STOP and MESSAGE. The class for this is included as part of this presentation.

Application Use

In any application you wish to use, be sure you register the Debuger template. You application merely needs the global extension. The settings of most interest to you are the following:

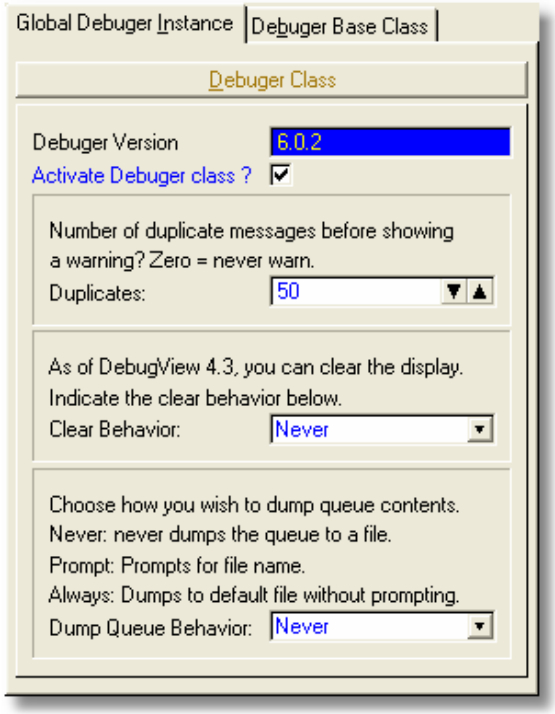

Figure 27 – Global Debuger settings.

Activate Debuger class?

Checking this box activates the Debuger class. Clear the check to deactivate it. If you have a DLL project and this app is not meant for debug use, then clear the check. All this does is prevent the startup messages and cluttering your display.

Duplicates

The duplicate messages happen in certain circumstances where the same event message is sent over and over. For example in a looping process, which causes the same message to be sent many times, you can limit how long this happens before a message statement pops up. Set it to zero to disable this feature.

Clear Behavior

The class can clear the display; this setting sets the default behavior.

Dump Queue Behavior

Indicates how the class behaves if you leave off the file name parameter of the *Init*() method. This happens only when you use the *DumpQue*() method.

That is all that is needed to make the class work. If you compile and run the previous broken Orders app, this is what you see:

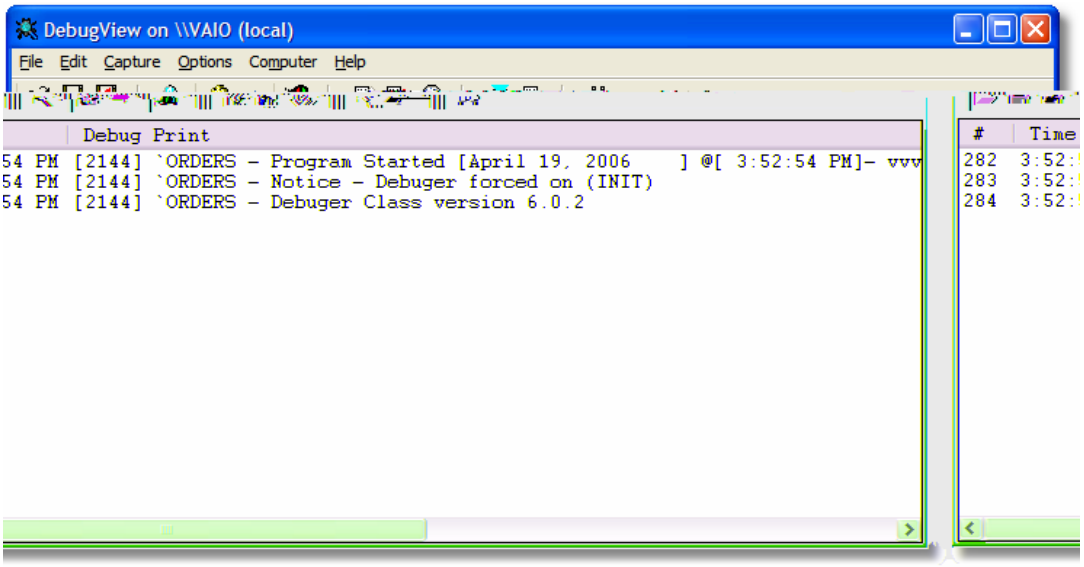

Figure 28 – Orders application started as seen in DebugView.

But that is all you see until you exit the application. Then you'll see a message that the program quit. This is a fast and simple test to ensure *DebugView* and the supporting class is working properly.

I should also note that a filter is in place. You may need to filter messages depending on what is going on with your machine. For example, if I remove all filters, *DebugView* shows these messages constantly:

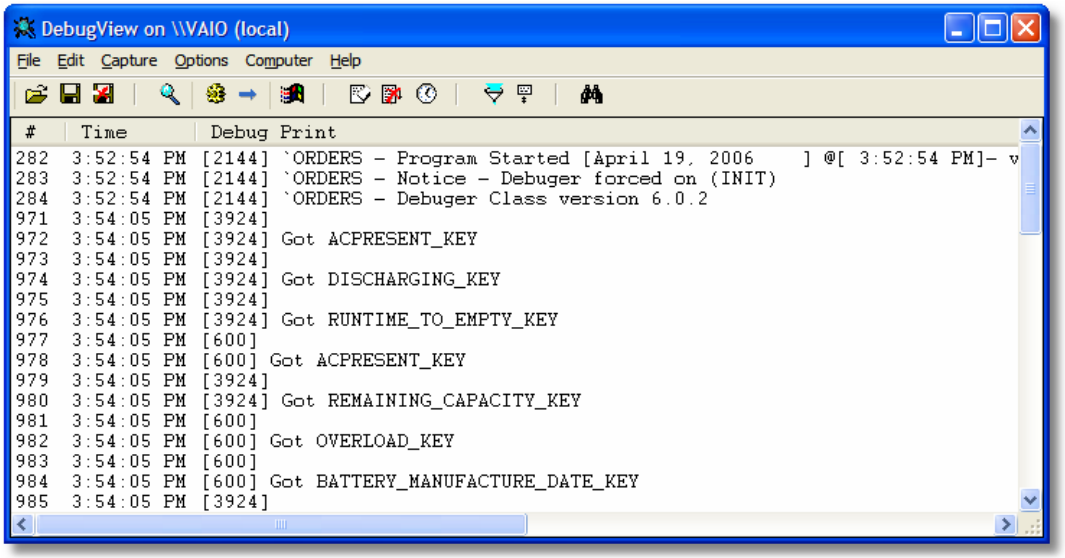

Figure 29 – Computer talking to UPS device.

This makes it hard to see what you need to see. The *DebugView* class uses an "unshifted" tilde character to "flag" messages sent by it. Thus open the filter dialog and fill it is as follows:

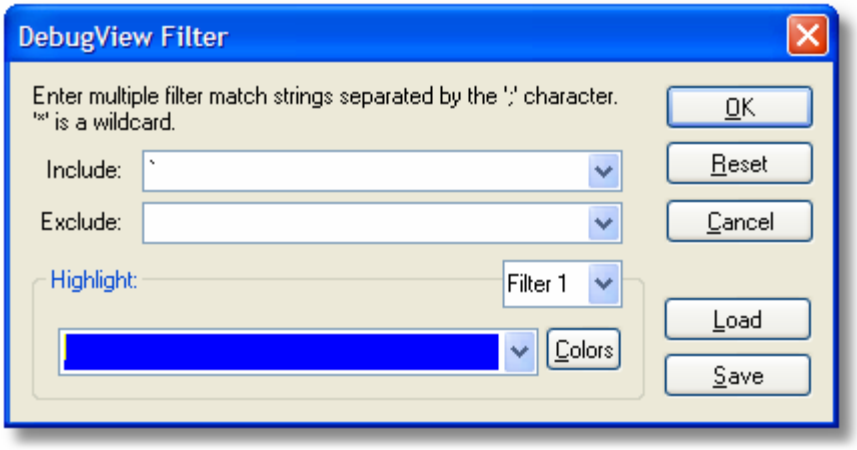

Figure 30 – Filter dialog.

To use the class in your application is quite simple. Use the *CallABCMethod* code template and call *DebugOut*. It takes similar parameter as MESSAGE(). It may look like this:

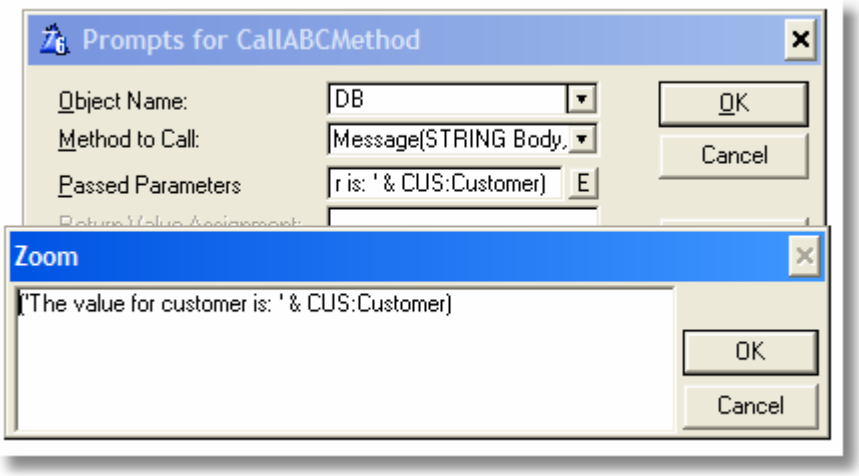

Figure 31 – Using CallABCMethod code template to call Debug method.

That template generates code as follows:

```
! Start of "Browser Method Code Section"
! [Priority 2500]
IF LOC:Flag
  Event = LOC:Flag
DB.Message('The value for the Customer is: ' & CUST:CustNumber)
! [Priority 3800]
```

Figure 32 – The DebugOut method.

When run and you trigger the buggy behavior, *DebugView* shows the following:

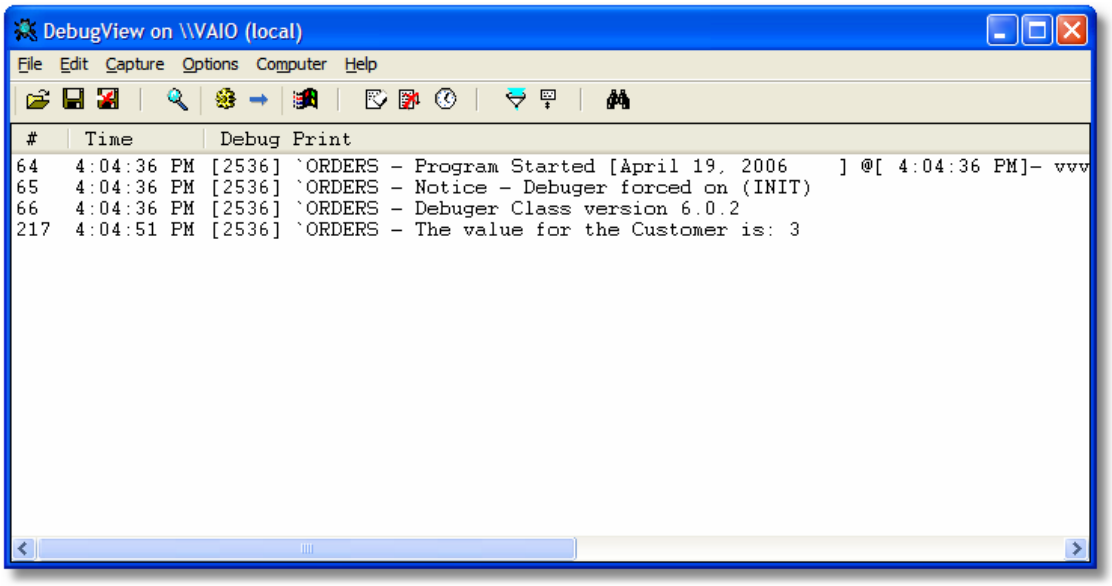

Figure 33 – Message showing you a variable's value.

Hand Coded Use

Using this class in a hand coded project is marginally more difficult. All you need to do is to manually set up your project that the template does for you.

The first step is to ensure the project defines variables are correctly set:

Figure 34 – Debuger flags defined and set.

Your project may or may not be an ABC project, so the flags for ABC use may or may not be in your defines list. The *_DebugerDllMode_* and *_DebugerLinkMode_* flags are required. In the figure above, they are set for EXE use. If your project is a DLL or LIB, then reverse the flag values. Zero is "off", one or missing is "on".

The second thing you need is to INCLUDE the class in the global section of your project:

```
INCLUDE('Debuger.INC'),ONCE          !Include the Debuger class
```

Instantiate the class is all that is left:

```
DB              Debuger
```

All you need now is to call the methods you need. An appendix describing the use of the popular and common methods is part of this document if you wish to use more methods.

The Fastest Way to Start the Debugger

In the previous section, I showed how to use the debugger. Some people don't like the debugger because they think it is too much work to set up. For those who agree with that sentiment, here is a little tip on how to launch the debugger with no effort on your part.

It would be very nice to simply have the debugger start and pause at a breakpoint in source, just like anyone with Visual Studio can do. To do this in Clarion is harder than Visual Studio, it requires one line of code. Oddly enough, this technique could be used in Visual Studio as well.

Say you have a block of code you would like to launch the debugger, open the correct source,

CHAPTER 4 - DRIVER DEBUGGING

Introduction

Regardless whether you use SQL or ISAM based files, there are times when you need to see what is happening between your application and the data files. Clarion provides file drivers so any Clarion statement is the same from one backend to the next (with minor documented differences).

Problem behavior manifests itself in many ways. You may get random error codes, regardless of backend. Your application may have become slower. Innocent changes on your part may escape notice of a major effect on your application. Or a setting on the data server.

Driver tracing

Clarion ships with a utility to trace file driver calls. This works with SQL or ISAM drivers. It is located in the bin folder under the Clarion root folder (default *C:\Clarion6\bin*). It is called *trace.exe*. When started, it appears similar to this:

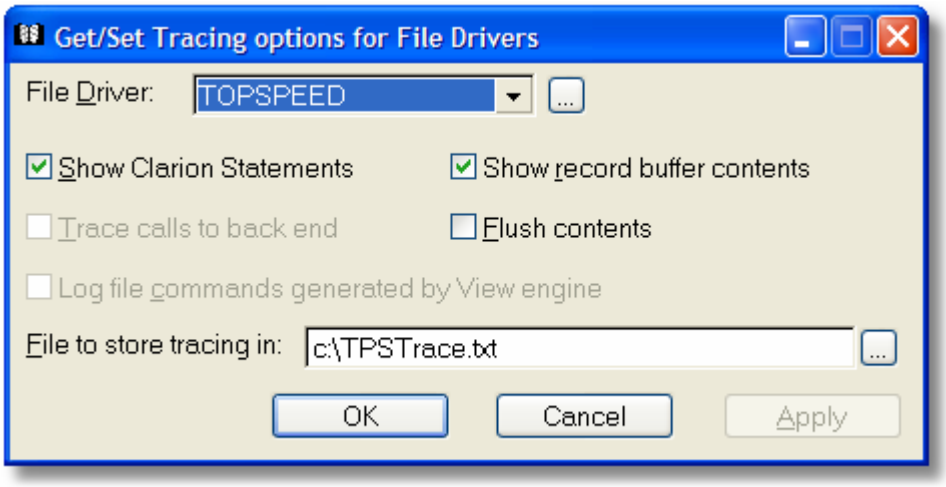

Figure 35 – Trace utility

Using the orders application from earlier, below is a typical log file from starting the app, opening the browse, and then closing the application:

```
06CCH(2) 16:28:59.031 CUSTOMER
FILE,DRIVER('TOPSPEED'),NAME('CUSTOMER'),CREATE,RECLAIM
06CCH(2) 16:28:59.031 CUST:BYNUMBER   KEY(+CUST:CUSTNUMBER),OPT,NOCASE,PRIMARY
06CCH(2) 16:28:59.031 CUST:BYNAME     KEY(+CUST:COMPANY),DUP,OPT,NOCASE
06CCH(2) 16:28:59.031                 RECORD
06CCH(2) 16:28:59.031 CUST:CUSTNUMBER   LONG
06CCH(2) 16:28:59.031 CUST:COMPANY      STRING(30)
06CCH(2) 16:28:59.031 CUST:FIRSTNAME    STRING(20)
06CCH(2) 16:28:59.031 CUST:MIDDLENAME   STRING(20)
06CCH(2) 16:28:59.031 CUST:LASTNAME     STRING(20)
06CCH(2) 16:28:59.031 CUST:ADDRESS      STRING(30)
06CCH(2) 16:28:59.031 CUST:CITY         STRING(20)
06CCH(2) 16:28:59.031 CUST:STATE        STRING(2)
06CCH(2) 16:28:59.031 CUST:ZIP          DECIMAL(5,0)
06CCH(2) 16:28:59.031 CUST:PHONE        DECIMAL(11,0)
```

```
06CCH(2) 16:28:59.031                  END
06CCH(2) 16:28:59.031                END
06CCH(2) 16:28:59.031
06CCH(2) 16:28:59.031 GET_PROPERTY(CUSTOMER:042C1D8H)  Time Taken:0.00 secs
06CCH(2) 16:28:59.031 GET_PROPERTY(CUSTOMER:042C1D8H)  Time Taken:0.00 secs
06CCH(2) 16:28:59.031 GETPROPERTYkey(CUSTOMER:042C1D8H,CUST:BYNUMBER[0])  Time
Taken:0.00 secs
06CCH(2) 16:28:59.031 GETPROPERTYkey(CUSTOMER:042C1D8H,CUST:BYNUMBER[0])  Time
Taken:0.00 secs
06CCH(2) 16:28:59.031 GETPROPERTYkey(CUSTOMER:042C1D8H,CUST:BYNUMBER[0])  Time
Taken:0.02 secs
06CCH(2) 16:28:59.046 GETPROPERTYkey(CUSTOMER:042C1D8H,CUST:BYNUMBER[0])  Time
Taken:0.00 secs
06CCH(2) 16:28:59.046 GETPROPERTYkey(CUSTOMER:042C1D8H,CUST:BYNUMBER[0])  Time
Taken:0.00 secs
06CCH(2) 16:28:59.046 GETPROPERTYkey(CUSTOMER:042C1D8H,CUST:BYNUMBER[0])  Time
Taken:0.00 secs
06CCH(2) 16:28:59.046 GET_PROPERTY(CUSTOMER:042C1D8H)  Time Taken:0.00 secs
06CCH(2) 16:28:59.046 GETPROPERTYkey(CUSTOMER:042C1D8H,CUST:BYNAME[1])  Time
Taken:0.00 secs
06CCH(2) 16:28:59.046 GETPROPERTYkey(CUSTOMER:042C1D8H,CUST:BYNAME[1])  Time
Taken:0.00 secs
06CCH(2) 16:28:59.046 GETPROPERTYkey(CUSTOMER:042C1D8H,CUST:BYNAME[1])  Time
Taken:0.00 secs
06CCH(2) 16:28:59.046 GETPROPERTYkey(CUSTOMER:042C1D8H,CUST:BYNAME[1])  Time
Taken:0.00 secs
06CCH(2) 16:28:59.046 GETPROPERTYkey(CUSTOMER:042C1D8H,CUST:BYNAME[1])  Time
Taken:0.00 secs
06CCH(2) 16:28:59.046 GETPROPERTYkey(CUSTOMER:042C1D8H,CUST:BYNAME[1])  Time
Taken:0.00 secs
06CCH(2) 16:28:59.046 GET_PROPERTY(CUSTOMER:042C1D8H)  Time Taken:0.00 secs
06CCH(2) 16:28:59.046 GET_PROPERTY(CUSTOMER:042C1D8H)  Time Taken:0.00 secs
06CCH(2) 16:28:59.046 GETNULLS(CUSTOMER:042C1D8H) Error: File Not Open  Time
Taken:0.00 secs
06CCH(2) 16:28:59.062
OPEN(C:\Clarion6\Examples\Tutor\Debugger\CUSTOMER.TPS:042C1D8H)  Time Taken:0.01
secs
06CCH(2) 16:28:59.062
GET_PROPERTY(C:\Clarion6\Examples\Tutor\Debugger\CUSTOMER.TPS:042C1D8H)  Time
Taken:0.00 secs
06CCH(2) 16:28:59.062
SETNULLS(C:\Clarion6\Examples\Tutor\Debugger\CUSTOMER.TPS:042C1D8H) Error:
Unsupported File Driver Function  Time Taken:0.00 secs
06CCH(2) 16:29:01.343
CLOSE(C:\Clarion6\Examples\Tutor\Debugger\CUSTOMER.TPS:042C1D8H)  Time Taken:0.00
secs
06CCH(1) 16:29:01.515
DESTROY(C:\Clarion6\Examples\Tutor\Debugger\CUSTOMER.TPS:042C1D8H)  Time
Taken:0.00 secs
```

Listing 2 - Small driver trace log

There is a lot of detail saved to the log, even with a simple action of starting the application, opening a browse and then quitting. Using the trace utility, you can adjust to a certain extent, the amount of data in the log. If I did more than just open a browse, then memory dumps of the data are also placed in the log.

Note: Be sure to turn off driver tracing when you are finished! Leaving it on impacts performance and the log file grows quickly.

Special note: Whenever you are doing drive tracing, be sure to shutdown other applications using the same driver first!

This tool has its uses. The best is when you really don't have a good idea when trouble first starts. Or perhaps you need to see which files or tables open first (or attempt to open). If you need to know just the data when a particular procedure runs, then you must use other means.

Driver Tracing on Demand

If you really don't wish to turn on driver logging globally as you would with the trace utility, then consider turning it on in varying degrees. By that I mean you can turn it on when you enter a procedure and turn it off when you return from it. Or turn it on when a control is accepted and turned off when another specific event happens. All sorts of combinations.

PROP:Profile

Property of a FILE that toggles logging out (profiling) all file I/O calls and errors returned by the file driver to a specified text file. Assigning a filename to PROP:Profile initiates profiling, while assigning an empty string ('') turns off profiling. Querying this property returns the name of the current log file, and an empty string ('') if profiling is turned off.

PROP:Details

This tells the driver to include record buffer contents in the log file; however, if the file is encrypted, you must turn on both the Details switch and the ALLOWDETAILS switch to log record buffer contents (see ALLOWDETAILS). Details=0 tells the driver to omit record buffer contents. The Profile switch must be turned on for the Details switch to have any effect.

PROP:Log

This property writes a string to the log file.

As an example, the following is example code that writes what you want to a log file:

```
Customer{PROP:Profile} = 'Customer.log'
Customer{PROP:Log} = 'Event new selection. Current customer: ' & CUST:Company
Customer{PROP:Profile} = ''
```

Listing 3 - Simple file logging.

This listing illustrates that you may turn on logging by naming a file where the dump statements go. It is a text file. You may use a path and file name, otherwise the file is in the current folder. The second line writes whatever you need in the log file. The last line turns off the logging. A typical result would look like this:

```
0BE8H(2) 10:20:56.343 Event new selection. Current customer: ABC Widgets Company
0BE8H(2) 10:20:56.781 Event new selection. Current customer: Have Everything
0BE8H(2) 10:20:57.187 Event new selection. Current customer: Easi Everything
0BE8H(2) 10:20:57.578 Event new selection. Current customer: Fidelity America
0BE8H(2) 10:20:58.015 Event new selection. Current customer: K Processing
```

Listing 4 - Example dump log contents.

The advantages to using this method over the system wide method should be obvious. You may put anything you like in the dump file. Since you control the code, you may even use conditional logic for when something is dumped to a file.

As a final suggestion, you could place any code like this as debug statements. This is the form where column one contains a question mark and at least one space. This has a hidden second

benefit too. These debug statements are ideal places to locate code to set breakpoints when you do need to use the debugger.

Summary

Dumping data to a log file is easy and straightforward. You may turn on system wide driver tracing if you are not sure where the problem lies or suspect a data error, like a customer ID pointing to a wrong customer ID in the invoice file. Once you've narrowed down where the problem may be, you may use PROP:Profile to start the process of dumping data to a log file. PROP:Log writes whatever string expression you want to the dump file, thus you are in complete and total control over what goes in the dump file.

COMMON CODING PROBLEMS

Introduction

Knowing how to code a GPF may sound strange as we tend to spend our production time not coding errors. The theory here is that if you know how to code GPFs or produce them, then you are less likely to have these in production code. Same for other coding errors that the compiler does not complain about.

Program just quits suddenly

This is when you are running the application and it just shuts down with no warning, no errors. It's like pressing ALT-F4. How do you trace that? Check for these reported causes:

Recursive calls

Too many recursive calls can exhaust resources. While you can code these in an application (which Clarion does point out), one or two may not cause any issues other than just bad coding design. It can get tricky if these calls are across DLLs. Thanks to Mike Hanson for this tip.

Bad slicing

This is the condition where the begin slice is greater than the end. This can happen when one is thinking SUB but doing slicing.

```
xVal = 10; yVal = 3
sVal = SUB(rVal,xVal,yVal)      !This works
sVal = rVal[xVal : yVal]        !Ends program with no GPF
```

Thanks to Carl Barnes for this tip.

Delayed GPFs

This is when a bit of bad code corrupts the stack, but the GPF happens later. Some situations make using the debugger difficult and this is one of them. One reason the debugger won't be much use in this situation is that you don't have a good place to put some breakpoints. And if you think you have an idea, the next pass through the suspect area, nothing happens.

One sure fire way to make life miserable is to rely on an implicit RETURN. This can corrupt the stack and a GPF is not far away once this happens. Here is what it would look like:

```
IF SomeVar > 10
  RETURN LEVEL:Benign
END
```

OK, that looks harmless. But what happens if the condition is false? Where is the code for the false side? The compiler will place an implicit RETURN here. But what does it return? Anything? If you just place a RETURN by itself, at least the compiler would point out that a value is missing.

The same theory holds for ROUTINEs. Explicit EXITs are always a good idea. Plus you get a nice benefit – you can put a breakpoint on these lines and check the values as a procedure or routine returns to the calling code.

Another cause could be a bad slice buffer overrun. Using the slice example above, you could make the above code safe:

```
ASSERT(xVal > 0 AND xVal <= yVal AND |
       xVal <= SIZE(rVal) AND |
       yVal <= SIZE(rVal))     !Asserts before problem code below
sVal = rVal[xVal : yVal]       !Ends program with no GPF
```

Thanks to Carl Barnes and Gordon Smith for this tip. I would also like to add that a liberal use of ASSERTs in your code can trap problems early.

APPENDIX A – DEBUGER CLASS REFERENCE

Introduction

The following is a quick reference of the common methods of the *DebugerClass*. A full reference is available for free at www.radfusion.com.

Init

Prototype

```
PROCEDURE(STRING ProgramName,BYTE DebugAlways=FALSE ,SHORT DuplicateCount=50),VIRTUAL
```

Description

ProgamName – A string constant or expression stating the name of the application

DebugAlways – Either true or false (default) - if true then Debuger on regardless of project Debug setting. '/Debuger' on the command line will also turn on the Debuger functions for that launch as if this parameter is set to True.

DuplicateCount – The number of duplicate Debug messages in a row before warning message. If zero, then do not track duplicates. Default is 50.

Message

Prototype

```
PROCEDURE(STRING body,<STRING Header>,BYTE ShowMessage=FALSE,BYTE ForceDebug=FALSE),VIRTUAL
```

Description

Body – The body of the message, either a string constant or expression.

Header – An optional message header. A string constant or expression.

ShowMessage - if True then a MESSAGE() will be issued along with the DebugOutString

ForceDebug - If True then this overrides the debug setting.

DebugBreak

Prototype

```
PROCEDURE(),VIRTUAL
```

Description

This method is a wrapper for the *DebugBreak* API call. This has the same effect as setting a breakpoint. It is only called if the *DebugActive* property is true.

Kill

Prototype

```
PROCEDURE()
```

Description

Sends an output string to *DebugView* indicating the program quit. It also sets the *DebugActive* property to false. It is automatically called by the *Destruct* auto method. You could also call it and re-issue an *Init* if you wanted to "restart" the class.