

Debugging Visual Basic 6 Code

Introduction

Debugging is the process of tracking down the cause of, and solving bugs and problems in your code. Bugs are usually caused by incorrect logic in your code, unexpected inputs or even just simple mistakes or typos. A bug can cause the program to behave incorrectly, crash, give incorrect outputs or runtime errors. We find most of the smaller obvious bugs by testing our code, passing test cases to functions, classes and other modules, and providing test inputs to the whole program and ensuring it gives the correct output. Later on, subtler bugs may be revealed through simply using the program. This tutorial will not go about telling you how to test your code and actually find bugs, but assumes you already have an incorrectly working or buggy program which needs to be debugged. The Visual Basic 6 IDE contains many valuable tools for debugging your code, and knowing how to use them all to their full power can allow you to quite quickly figure out what's causing a particular bug so you can fix the code.

The Immediate Window

One of the most important tools is the “immediate window”. You can open the immediate window using the Control-G shortcut, or by clicking the “View” menu and clicking “Immediate Window”. While still in design mode, you can evaluate expressions in the immediate window, making it quite handy for testing small snippets of code. You do this by prepending a question mark to the expression. For example, typing “?2^32” will result in “4294967296” being printed after it. The real power of the immediate window comes when you're running your code. You can type expressions involving variables that are currently in scope in your code. You will learn how this is useful once you learn how to set breakpoints and step through code. Another handy thing you can do with the immediate window is to print text to it from your code using the “debug.print” method. If you type “debug.print” in your code, it will be added to the immediate window, giving you a kind of “log” of what's happening in your program. For example you could do something like:

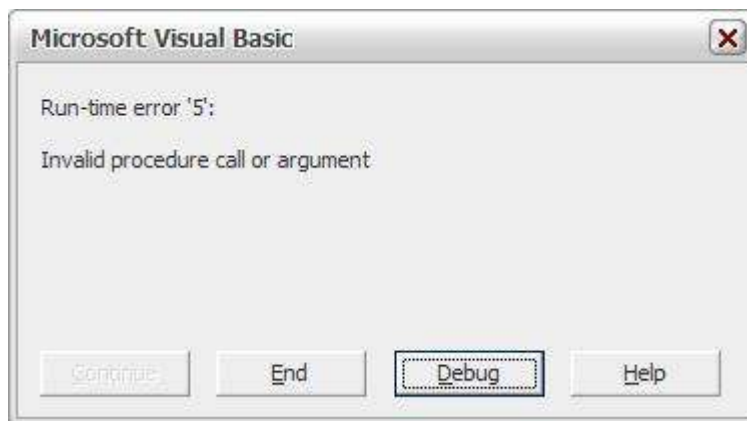
```
x = 1
debug.print "MyModule: MyFunction -> Value of X is " & x
debug.print "MyModule: MyFunction -> Adding 3 to x"
x = x + 3
debug.print "MyModule: MyFunction -> New value of x is " & x
```

Obviously your real code will be more complicated but adding debug output around suspected buggy code is a very valuable debugging technique that quite often will reveal bugs immediately. You can print out the values of variables at certain known stages to see if they're as expected. You could print out strings to indicate the flow of code. For example:

```
if x = 1 then
    debug.print "x = 1. Calling foo"
    foo
else
    debug.print "x <> 1. Calling bar"
    bar
end if
```

Debugging After a Runtime Error

Quite often most of the simpler more obvious bugs will take the form of a runtime error occurring. You will usually get a dialog like this appearing:



Go ahead and click the “Debug” button. You'll see that the offending line of code is highlighted in yellow. At this point your program is in a “paused” state. It's been stopped while running. All variables in scope at this point still have their values retained and so on. Here's my code:

```
Option Explicit

Private Sub cmdBug_Click()

    Dim s As String
    Dim Length As Long

    s = "hello"

    Length = SomeFunction(s)
    MsgBox Left$(s, Length)

End Sub
```

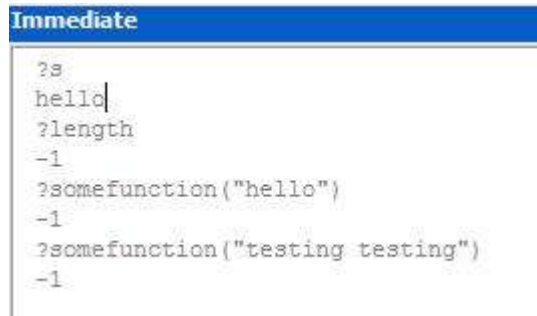
It would be very handy right now if you knew exactly why certain functions might raise certain runtime errors. I know that Left\$ returns a string, and that “MsgBox” takes a string so the MsgBox function is fine. The problem must be in the “Left” function. Let us look at the actual error message. It says “Invalid procedure call or argument”. Well let's examine the arguments. Hover the mouse over the variable, “s”.

```
Length = SomeFunction(s)
MsgBox Left$(s, Length)
s = "hello"
```

You can see it has the value, “hello”. There's nothing wrong there. So now let's look at the “Length” argument.

```
Length = SomeFunction(s)
MsgBox Left$(s, Length)
Length = -1
```

Here we can see that obviously a length of -1 is invalid. We've found the immediate cause of the runtime error. Now to continue debugging, we would find out why the Length was -1. If you examine my code above again, you'll see that the length is gotten from SomeFunction, so we can pinpoint the bug to somewhere inside that. You can also make full use of the immediate window while the code is in a paused state like that. Here are some examples of the kinds of things you can do:



```
Immediate
?s
hello
?length
-1
?somefunction("hello")
-1
?somefunction("testing testing")
-1
```

Deliberately Stopping the Program

Sometimes you'll have subtler bugs which don't cause runtime errors. You have to deliberately force your program to stop. There are several ways you can do this listed below:

The Stop Keyword

The “Stop” keyword will pause the program and step in to the debugger on the “Stop” line. Here's my code:

```
Length = SomeFunction(s)

If Length < 0 Then
    ' there's an obvious bug if you stop here
    Stop
End If

MsgBox Left$(s, Length)
```

I could then debug as normal from here, examining the variables and so on.

Assertions

An assertion is basically just a quicker way of doing the above. You write “debug.assert” then a condition which must be true for the program to continue running. If the condition is false, the debugger pauses the program on the assertion line and allows you to debug. Here's my changed code again:

```
Length = SomeFunction(s)

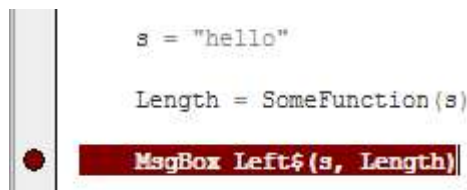
Debug.Assert Length > 0

MsgBox Left$(s, Length)
```

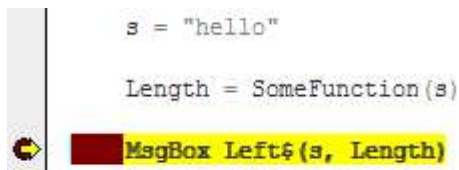
Here, I've asserted that the Length variable must always be greater than 0. However, since, as we know, it has the value -1, the check fails and the debugger breaks on the assertion line allowing you to debug. You can sprinkle assertions in to your code as you go, or you can add them to code you suspect of being buggy as a further debugging aid to check that things are going as they should. When they start going wrong, you'll hopefully hit one of the assertions allowing you to pinpoint roughly where things started to go wrong.

Breakpoints

Break points are one of the most valuable tools for debugging. A break point is a “flag” on a particular line of code which causes the program to stop on that line and break in to the debugger, just as it does when it hits a “Stop” line, when an assertion fails, or when a runtime error occurs and you click “Debug”. To add a breakpoint to a line of code click the line at design time and hit the “F9” key, or click the grey margin next to the code. This will add a circle in the margin and highlight the line in a different colour.



Now when the code is run, the debugger will pause the program as soon as it gets to the line with the breakpoint on it. Note that it doesn't actually execute that line. So now when I run the program, it breaks on that line without a runtime error occurring, allowing me to examine variables as usual.

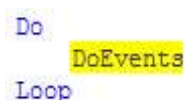


Hitting the Big Pause Button

Sometimes your code goes in to infinite loops and you want to stop the program and figure out why. Sometimes you want to just let it load then pause it and go in to debug mode to debug what happens when you click a button, for example. You can either press the “Control + Break” key combination. The pause button looks like this on your toolbar:



There should also be an option in the “Debug” menu. Here's what happens when you click it, or press control + break during an infinite loop:



The program just pauses on the current line, which in this case can only be the “DoEvents” line. This allows me to then debug why my loop isn't ending.

Stepping Through Code

Now that you know how to get in to debug mode in several ways and examine variables and evaluate some expressions in the immediate window, it would be useful to know how to go through lines of code one at a time. This is called “stepping” through the code. There are 3 types of steps you can make: Step into, Step over and Step out. “Step into” will cause the next line to be executed and if there's a function, you'll be taken inside that too. “Step over” will cause the entire function to be executed. This is useful if you know there's a problem in a particular procedure and you want to step through each line of code, but don't want to step through all 300 lines in a function it calls over and over in a loop. You can just “step over” this function, and it'll execute it, keeping you where you were. “Step Out” is for when you are inside a function, and you want to just finish the function and go back to where it was called. The current line of code being executed is highlighted in yellow with an arrow next to it in the column on the left. The 3 stepping functions should be in the “Debug” menu and also in the toolbar, but I advise you to learn the shortcuts, as you don't want to have to keep clicking the “Debug” menu or a button on the toolbar every time you want to go through some lines of code. The shortcuts are:

Step into – F8

Step over – Shift + F8

Step out – Control + Shift + F8

Here's an example of my code again:

```
Private Sub cmdBug_Click()  
  
    Dim s As String  
    Dim Length As Long  
  
    s = "hello"  
  
    Length = SomeFunction(s)  
  
    MsgBox Left$(s, Length)  
  
End Sub
```

As you can see here, I set a breakpoint on the s = “hello” line, ran the code, and clicked the button. The code “breaks” on this line. I'm now in debug mode. I'll press F8 to execute that line and move on to the next one:

```
Private Sub cmdBug_Click()  
  
    Dim s As String  
    Dim Length As Long  
  
    s = "hello"  
  
    Length = SomeFunction(s)  
  
    MsgBox Left$(s, Length)  
  
End Sub
```

Now the next line is highlighted in yellow. I can do one of 2 things here. I can either step in to “SomeFunction” and follow its execution, or I can step over it. I'll step over it for now to illustrate something else. So I press Shift+F8:

```

Private Sub cmdBug_Click()

    Dim s As String
    Dim Length As Long

    s = "hello"

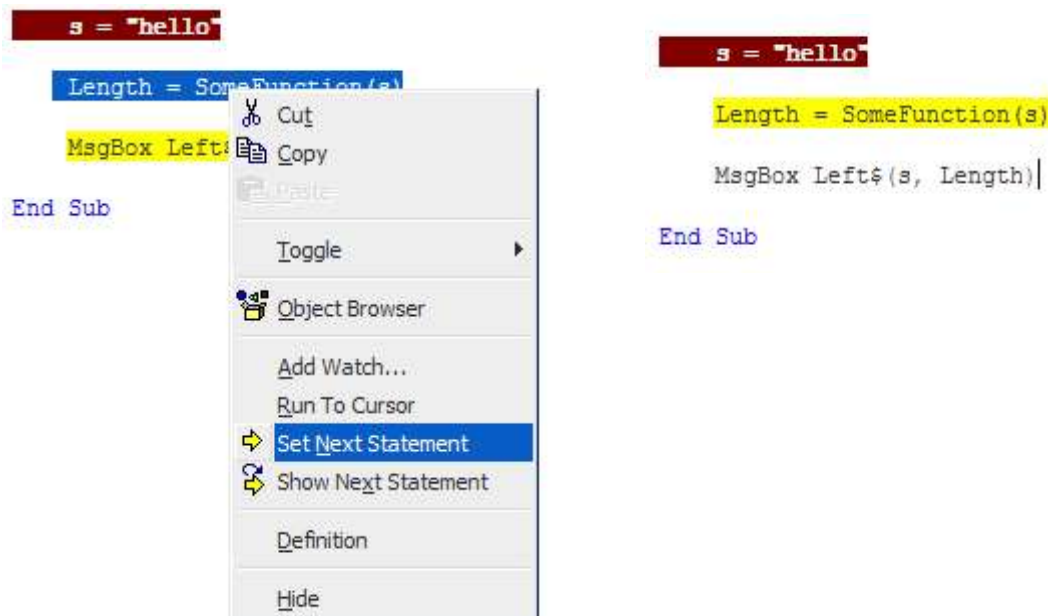
    Length = SomeFunction(s)

    MsgBox Left$(s, Length)

End Sub

```

Now the function is executed and the cursor moves on. It might be at this point I see “Length” has a negative number, and I realise that SomeFunction is wrong. I want to be able to go back and step in to SomeFunction. We can do this without restarting the program by right clicking on the line we wish to execute again and clicking “Set Next Statement”:



Isn't that handy. Now we're back to the line with the function on it. This time we want to step into the function, not over it. So we hit F8 instead of Shift+F8. Now we're taken over to “SomeFunction”:

```

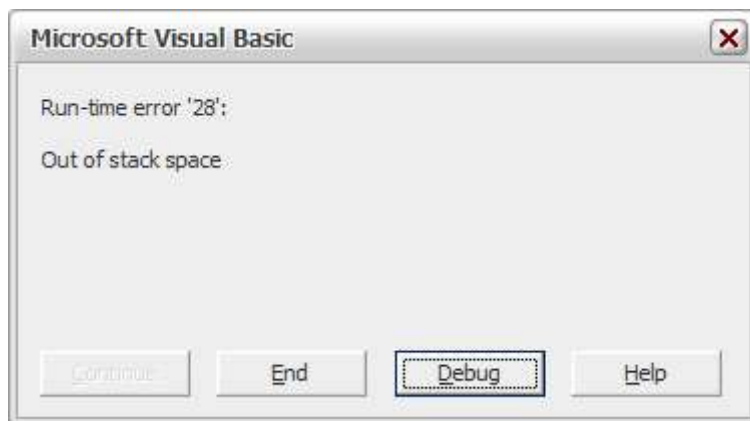
Public Function SomeFunction(s As String) As Integer
    SomeFunction = -1
End Function

```

And you caught me :) I just deliberately made it return -1 for the purposes of this tutorial. However you would then use the above debugging techniques to go through that function. You would execute each line and use the above techniques to examine the variables and so on.

The Call Stack

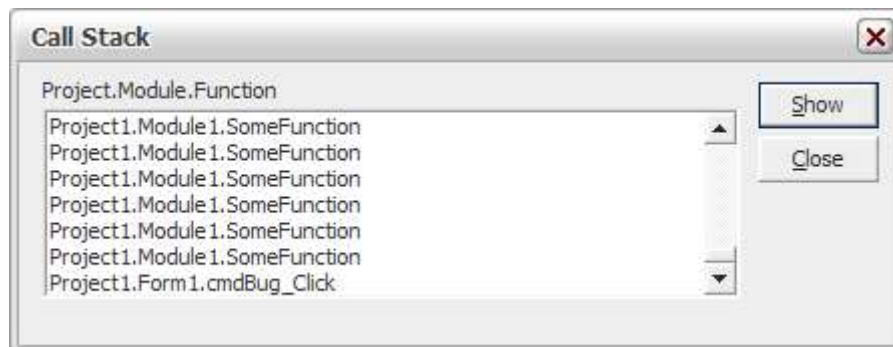
The call stack is a window which shows you the history of function calls currently on the stack. To show how this is useful, let us consider this new error:



We click debug, and we are taken here:

```
Public Function SomeFunction(s As String) As Integer
    SomeFunction = SomeFunction(s)
End Function
```

It's quite obvious what's wrong with the function. It just keeps on calling itself forever until it runs out of stack space, however, let's pretend it's much more complicated and we don't know that it's calling itself forever and ever. The first thing you should do when you get an Out of stack space error is bring up the call stack, so let's do this. Click the “View” menu and click “Call Stack”, or simply press Control + L.



You can see here at the bottom of the stack is cmdBug_Click. Then after that we have a whole bunch of calls to SomeFunction. Far too many than we should have, and enough to blow all the stack space. We now know the cause of the error, and need to debug the logic in the function to figure out why it's calling itself infinitely instead of stopping eventually (Note: Some recursive functions call themselves so much that they burst the stack space, even though they are in fact correct and would theoretically stop eventually – there is nothing logically wrong with those. It's just a memory limitation). You could also imagine how useful it would be to see exactly which functions called which at any given point in time. You can see this using the call stack.

The Locals Window

The locals window shows you all local variables at any one point in the program. Let us write some code and set a breakpoint in order to illustrate the usefulness of the locals window:

```
Option Explicit

Private Type Foo
    x As Long
    y As Double
    z As String
End Type

Private Sub cmdBug_Click()

    Dim s As String
    Dim Length As Long

    Dim a(1 To 3) As Long

    Dim x As Foo

    x.x = 123456789
    x.y = 0.12345
    x.z = "a string"

    a(1) = 123
    a(2) = 456
    a(3) = 789

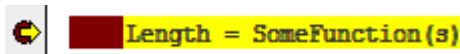
    s = "hello"

    Length = SomeFunction(s)

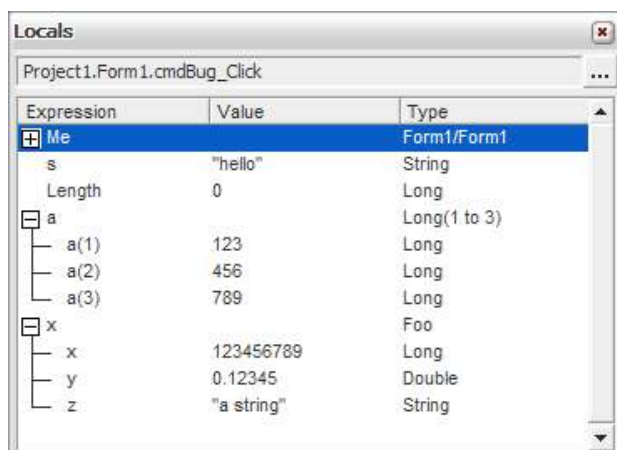
    MsgBox Left$(s, Length)

End Sub
```

As you can see we have arrays, user defined types, longs, strings and doubles. We run the code and click the button, and as usual the breakpoint is hit.



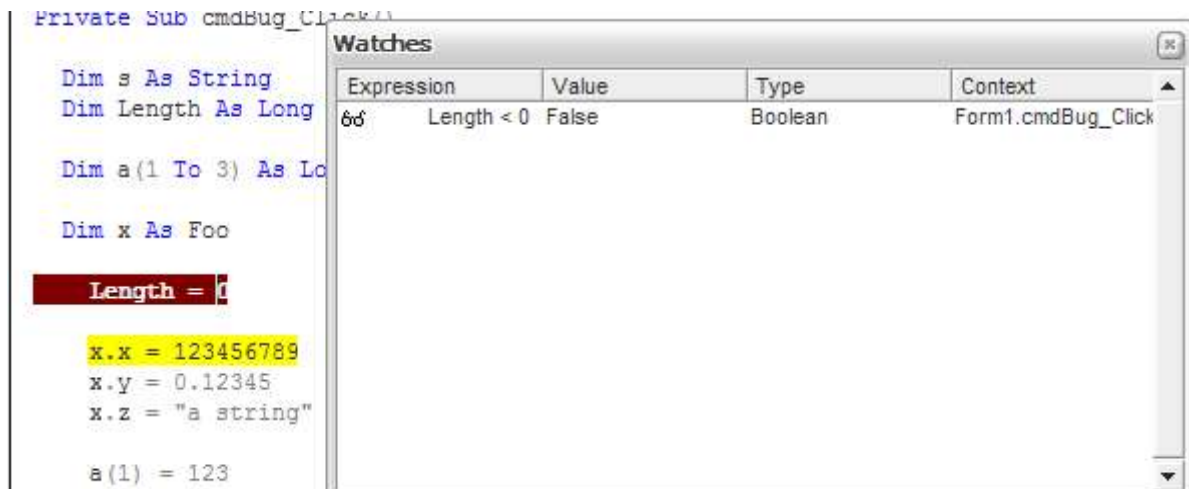
Now let's bring up the locals window. Click "View" then click "Locals". We see the following window:



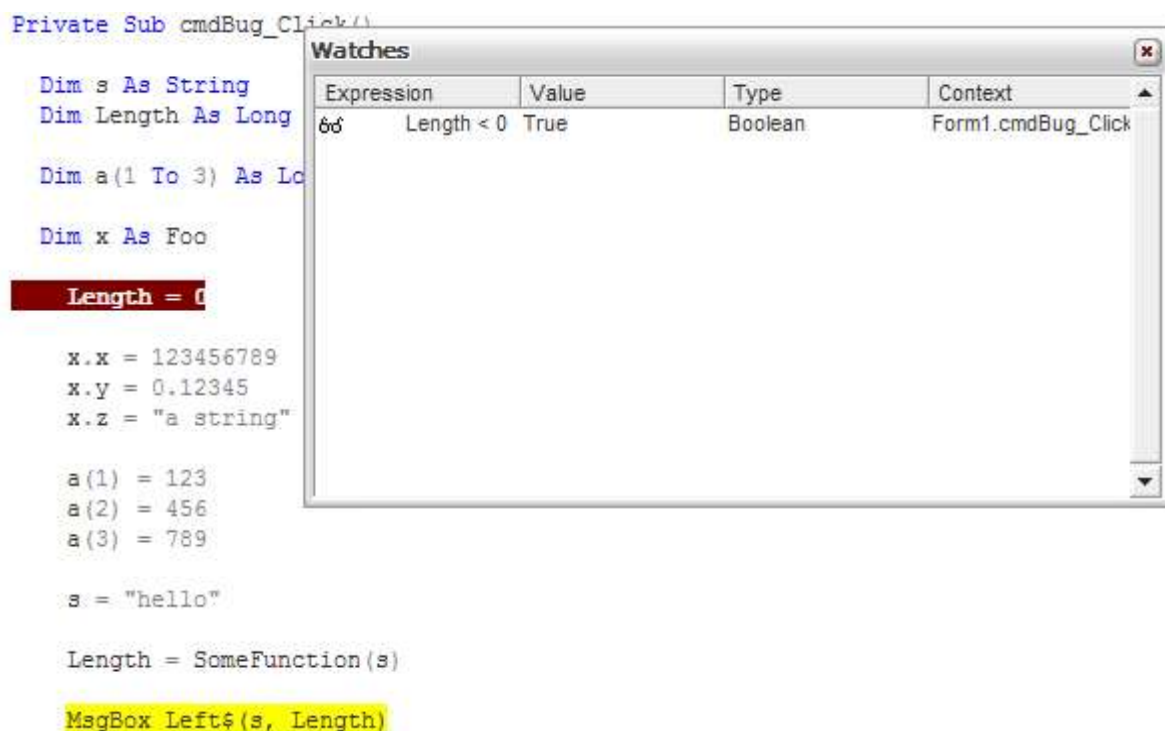
Isn't that useful? You can see all of the values of all the local variables, including more complicated structures such as arrays and user defined types.

The Watch Window

The watch window looks very useful, but I've never really used it, and I'm not quite sure how it works. But here's the results of some experimentation. Click “View” then click “Watch Window”. We're going to watch for the expression “Length < 0” in the above code. So right click it and click “Add Expression”. Type “Length < 0” and press OK. Now we run the code and immediately step in to it with a breakpoint.



As you can see, the expression is false at this point. Let's step over a few lines.



Notice now that we've executed the “Length = SomeFunction(s)” line, the value is now true? You can also set up the watch to break into the debugger as soon as the expression is true, or even when the value simply changes.

Conclusion

Now you know how to use the immediate window, write assertions, set breakpoints, step through your code and use the watch window, locals window and callstack to your advantage. You still need to use your superior logical skills in order to actually make sense of this data and know how to use these tools to your advantage. Unfortunately, these kinds of skills are very difficult to teach. You have to think hard and be patient. Debugging code also requires that you fully understand how your code ought to work, and what values variables should have at particular points in the program etc (Hint: make use the immediate window to calculate which values variables should have at certain points for more complicated programs). For production code, note that `debug.print` lines are taken out of the code. So for code which is released to the public, it may be a better idea to write debugging lines to a log file which users could send you for analysis should they encounter an error or bug. Good luck with your debugging.