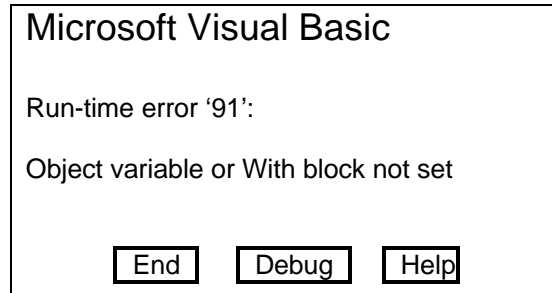


Learning to Debug in Visual Basic 6

So, you just finished entering that last line of code and you hold your breath as you press the F5 key... The form appears...and then



You exhale in a loud sigh (or worse.) “It sure would be nice if just once the program ran the first time. I don’t have time for this! Let’s see, what do I do now to find this error?”

Visual Basic provides a number of tools to help, but the first challenge is figuring out which tool is best for each debugging problem. We need to start by listing all the things VB offers.

- Option Explicit*
- Intellisense*
- MsgBox*
- Debug Object*
- Immediate Window*
- App Object*
- Break Mode*
- Breakpoints*
- Watch Window*
- Controlled Execution*
- Locals Window*
- Call Stack*
- Err Object*

This really is a lot of choices. You might be thinking, “Why do I need all of these? Can’t you just show me the one or two best ways?” And the truth is, for the simple bugs you can get by just fine with *Debug* and *MsgBox*. But it doesn’t stay simple for long. It is also true that the more you understand the debugging tools and processes, the better a coder you will become.

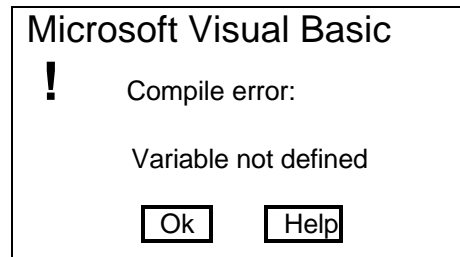
(Note: Throughout this tutorial, when you see something enclosed in brackets [], this indicates what you should literally see.)

Option Explicit

The first thing we need to look at is *Option Explicit*. While it is not an actual debugging tool, it is one of the best things you can use to catch bugs early. What is it? It is just a message to the VB runtime telling it that you intend to declare all variables with Dim, Private, Public or Global statements. So how can this help? Let’s say you have the following code:

```
Dim Mystring as String
Mystring = "This is a good debugging tool"
Mystring = Mid$(Mystring,11,4) 'Give me the word good
MsgBox Mystring
```

When the message box appears you expect to see the word [good] in it, but what you actually see is a blank message box. This is a simple example, and I am sure you can easily see what is wrong. Because you didn't have to declare Mystring, VB didn't care that you used Mystring also; it just initialized it to a nullstring and ran happily along. Now, let's see what happens when we put *Option Explicit* at the top of our module.



```
Mystring = Mid$(Mystring,11,4) 'Give me the word good
```

Hey, this is great. The compiler is telling us what is wrong and showing us where. The developers of Visual Basic were confident enough of the value of this statement that they provided a way to make it a default. Once you turn this option on, every new code module will automatically have it at the top. Unfortunately they didn't have enough confidence in the intelligence of programmers to make this the default automatically. To do this yourself, go to the Tools menu and select Options. On the first tab General, the second checkbox down on the left says, [Require Variable Declaration]. Check it and click OK. The next code module you create will reflect this.

Intellisense

Now that we know our poor, dyslexic typing won't be our downfall, what's next? *Intellisense* is one of Microsoft's better attempts at AI. It tries to anticipate what we want to type and shows all available options. This helps us three ways. First, as we are typing we don't have to worry about whether we have capitalized each variable, statement, function and object type correctly. It will do it for us. Also, if we have made a syntax error it will tell us. Next, as soon as we have typed in a function it starts showing us the parameters it expects. It tells us the type of the parameter and whether it is required or optional. It also shows us a list of defined (enumerated) constants, where applicable. The last way it helps is to show us all available properties and methods when we type the period at the end of an object variable, or all object types when we type the [As] in an object variable declaration. Actually, there is a fourth type of *Intellisense* help. We only see this when we have a program in *Break Mode*. We will look at this later on. By default all these options are turned on. If you don't see one or more of these things, select Options from the Tools menu. Check all of the checkboxes in the top frame on the General tab and click OK.

MsgBox

Now that we have optimized our development environment both to avoid bugs and to detect them early, it is time to look at the debugging process itself. We have entered our code and are trying to get it to run correctly. One of two things happens: It stops abruptly with an error, or the result is not what we were trying to achieve. Let's start with the incorrect result. One of the main methods in determining the point of failure is to view intermediate results. Three primary ways of doing this involve using message boxes, text output in a window, or text output to a file. If I simply want to be informed of some event happening or the value of some variable at a particular point I could insert a *MsgBox* function to do it. For example...

```
MsgBox "Click Event for Text1, strMyVar = " & strMyVar
```

Debug Object & Immediate Window

The advantage to using a *MsgBox* is that you get immediate notification. The disadvantage is that it disrupts program flow and changes how events occur and are handled. For those times when we need to let the program run smoothly, we can use the *Debug Object* to output text to the *Immediate Window*.

```
Debug.Print "KeyDown Event for Text1, KeyCode = " & CStr(KeyCode)
Debug.Print "KeyUp Event for Text1, KeyCode = " & CStr(KeyCode)
Debug.Print "KeyPress Event for Text1, KeyAscii = " & CStr(KeyAscii)
```

If we put each of these lines in its respective event procedure, we can follow the progress of events when we press one key, by looking at the output in the *Immediate Window*. Here is the output from pressing the letter 'a'

KeyDown Event for Text1, KeyCode = 65

KeyPress Event for Text1, KeyAscii = 97

KeyUp Event for Text1, KeyCode = 65

This shows us that the order of events is KeyDown, KeyPress, and then KeyUp. We could not have discovered this by sending these messages to a *MsgBox* because it would have interfered with the key events.

App Object

Let's say you have created the "Next Big Internet Game." It is based on a client program for each gamer that connects to a server program controlling the game. The client program has a user-interface, but the server program doesn't. A certain amount of debugging can be done while running the server program within the IDE (integrated development environment.) The real test is when you compile and load the program on the server. Even though it seemed to be working fine in the IDE, you now are seeing problems as the game progresses. How do you see what is happening in the server program? This is where you can use the *App Object* to write information to a log file. The syntax is easy. Somewhere at the start of your program you issue:

```
App.StartLogging App.Path & "\MyServerLog.log", vbLogToFile
```

Intellisense lets us down here. There are a number of options for the second parameter, but it doesn't tell us what they are. *VbLogToFile* is equal to 2. This means use the file name in the first parameter to write out log entries. Other options are available, including having it delete and recreate the log file every run.

The *App Object* is now primed for writing your log. At each point in the program where you might want to get some feedback you can use the *LogEvent* method.

```
App.LogEvent "Message Received from client#" & CStr(intClientnum), 4
```

When you run your compiled server program, each *LogEvent* will write one line to the log file. The number 4 in the second parameter just tells the *App Object* to mark the entries as informational. 1 would mean type error and 2 would mean type warning. It really doesn't matter as the text you specify will be there in any case.

Break Mode

There are times when these methods of watching the program execute just aren't enough. For one thing, the program may be aborting with an error before it even gets to our debugging output statements. This is when we need to bring out the heavy guns and take control. Most of the remaining debugging tools and procedures are used in *Break Mode*. *Break Mode* is when the program isn't really running, but it is still active. While in this mode we can do all sorts of things, including making minor changes to the code. When we want the program to continue, we can control whether it returns to normal run mode or just executes a small amount of code and returns to *Break Mode*. First, let's look at how we get into it.

Breakpoints

The way we all first become acquainted with *Break Mode* is when we are dumped into it with an ugly error box. Usually, that box has a button that says Debug. Well, that is what takes us into *Break Mode*. There are many other ways we can explicitly trigger *Break Mode*. The one I learned first (long before Windows, DOS, or even the PC) is Ctrl+Break. While I don't use this much any more, there are times when it is the only way to break out of an endless loop (without killing the VB IDE itself.) We can insert Stop statements into our code. We can use another member of the *Debug Object*, Debug.Assert. We can use the *Watch Window* to set an expression, and when that expression is True or the value of the variable we are watching changes, we will enter *Break Mode*. But the most common way of entering *Break Mode* (on purpose) is with *Breakpoints*. *Breakpoints* are simple; just pick a line of code you want to stop on and click the left window bar next to it. A large dot will appear where you clicked and the line of code will be highlighted. You can set many of them throughout your program. You can even set more of them or unset them while in *Break Mode*. To unset one, just click on the dot. To unset them all, go to the Debug menu and click Clear All Breakpoints. (There are some lines of code you can't set a *Breakpoint* on: Comments, Variable declarations, or anything in the General Section.)

Ok, we're in *Break Mode*; now what? Well, we could look at some variables. Or, we could type in some code in the *Immediate Window* and execute it. Or, we could continue running the program, line by line, a group of lines, a whole Subroutine or Function. Of course, we can always resume normal execution or just end the program. "So I want to look at some variables," you say. No problem. Remember what I said earlier about a fourth way *Intellisense* could help us. Just move your mouse cursor over some variable that is in scope. ("In scope" means it can be used as a variable in the particular section of code currently being executed.) Let's say the program is in *Break Mode* on the last line of this code segment.

```
strA = "This is a test line"  
intPos = 11  
intCnt = 4  
strB = Mid$(strA,intPos,intCnt)
```

As we move the mouse pointer over the variable strA, something like a tool-tip balloon appears, containing the text [strA = "This is a test line"]. The same thing happens when we move over the intCnt variable, [intCnt = 4]. When we move over the strB variable we see [strB = ""]. Why don't we see [strB = "test"]? It is important to remember that the line which is highlighted as the current line has not yet been executed. Maybe we want to test the execution of this line before it actually runs. We can go to the *Immediate Window* and type in Debug.Print Mid\$(strA, intPos, intCnt), then press Enter. On the next line we see [test]. This is an example of executing code in the *Immediate Window*. There is quite a bit that can be done this way. but I won't go into it now. By the way, in the *Immediate Window*, it is not necessary to type the Debug. in front of Print. In fact, it can be shortened down to just ?. But don't try to use ? in place of Debug.Print in code.

Watch Window

Well, that checks out; we are getting the desired result. While we are here looking at variables in *Break Mode*, let's inspect the *Watch Window*. We can go to the View menu and click Watch Window, or we can right click on one of the variables in the code. When we do that, a context menu appears where one of the options is Add Watch. When we click it, an Add Watch dialog window appears with the variable already in the Expression field. In fact, all the required fields have been given default values that will work well if we just want to watch the variable. So, we click OK, and the *Watch Window* appears with the one line we just added. At this point, we can see in the value field what the current value is. When we start executing again, the value will be updated whenever the variable changes. Let's add another watch in a slightly different way. Using the mouse, highlight another variable. Then, with the mouse cursor over the highlighted variable, press and hold the left button while dragging the variable to the *Watch Window*, then release. There it is, another watch.

There is much more to a watch than just seeing the value of a variable. To get a better idea of all the things possible we need to look at the options in the Add/Edit Watch dialog (they are identical.) We will right click on the first watch we added and select Edit Watch from the menu.

The upper frame concerns the context (scope) of the watch. If we have a variable that has greater scope (can be used programmatically in more than just one subroutine), we can have its value shown in the *Watch Window* for just one subroutine, a single module, or the whole project. If you try to watch a variable in a context of greater scope than it actually has, you will only see [<Expression not defined in context>] in the value field.

The lower frame lets us define the type of watch we want. A Watch Expression just shows us the value of the expression. Break When Value is True says we want to enter *Break Mode* when the *Watch Window* sees that the value of the expression is equal to Boolean True. Break When Value Changes means we want to enter *Break Mode* when the value of the expression changes. All of this begs the question, "What is an expression?" An expression is as simple as a single variable or as complex as a mathematical or Boolean equation involving many variables. Some examples of valid expressions are:

```
strB
strB = "test"
intPos1 + intPos2
intPos1 + intPos2 > 5 And strB = "test"
intPos1 = 5 Or (intPos1 = 3 And intPos2 > 6)
```

If you create a watch with a Boolean equation and set the type to Watch Expression, all you will see in the value field is True or False.

Controlled Execution

Having created our watches, we now need to set the program into motion using *Controlled Execution*. If we have created watches that will break on True or when the value changes, we may just want to press F5 to continue normal execution. Many times, however, what we want to do is single-step through each line of code, checking our watches after each line has executed. To do this we can select Step Into from the Debug menu. You will notice the shortcut key is F8. This is much easier to use. When we press F8 we see the current line highlight and arrow move to the next line. We can look at our watches and see what effect this one line of code had. Sometimes, we want to execute a group of lines and then stop. A good way to do this within a subroutine is to click on the line where we want to stop and then select Run to Cursor from the Debug menu. Once again notice the shortcut

key, Ctrl+F8. If the code we are stepping through contains calls to other Subroutines or Functions, we will automatically move there as we execute the code line containing the call. Sometimes we don't need to see each line in the called routine execute. For this we can select Step Over from the Debug menu or just press Shift+F8. This executes a single line of code like F8, but with the benefit of treating a subroutine call as a single statement. Maybe we have already single-stepped into a subroutine and then realized the problem is past this call. Do we have to continue to F8 through each line to get back to the calling routine? No, we can select Step Out from the Debug menu (Ctrl+Shift+F8).

We can do one more thing to control execution for the purpose of tracking bugs. We can actually change what is considered the current line of execution. There are two different ways to accomplish this. First, we can click on the line we want to become the current line, and select Set Next Statement from the Debug menu. Second, we can just grab the current line arrow on the left window bar and move it to the desired line. This is done by placing the mouse cursor over the arrow, pressing and holding the left mouse button, and moving the mouse cursor. The new current line must still be within the same routine. When would we want to do this? One example would be if we discovered that we had left out one or more necessary lines of code right before the current line. We can, in most cases, add these lines while in *Break Mode*, and then move the current line of execution back and execute these lines. Pretty amazing stuff, wouldn't you say?

Locals Window

There are times, especially when working with anything object related (forms, controls, collections, etc,) when it would be nice to be able to see the value of all variables and object properties, within the current scope. The *Locals Window* is just what the doctor ordered. It can be opened from the View menu by selecting Locals Window. What we get in this window is a hierarchical tree view similar to the Project Explorer Window. Instead of Forms and Modules, what we see are variables and objects. Simple variables just have a value, but more complex variable structures (objects and arrays) have a plus sign that we can click to expand the view. If we are in *Break Mode* in Sub MySub() we will notice that the only variables we see are variables with local scope in Sub MySub() and [Me]. Me is the application, and when we expand it we see all variables and objects in module or global scope. This includes every property of the form and every control on the form. We can continue to drill-down, viewing the properties of controls and other objects and the values of individual elements of arrays. One good use for this window is when we get the dreaded [Run-Time Error 91, Object variable or With block variable not set]. When we click Debug, we see the variable it is complaining about, but we can't see why it is complaining. When we open up the *Locals Window* and find the offending variable, (look under Me if it was declared outside the current routine), it reveals the problem. Its value is Nothing, probably because we never instantiated it. Take some time, exploring the various things exposed in this window. It can tell us much.

Call Stack Window

You might notice the small button with the [...] in the upper right corner of the *Locals Window*. This button takes us to the *Call Stack* window. The other way to view this window is from the View menu, *Call Stack* (Ctrl+L.) When one subroutine calls another and that subroutine in turn calls yet a third, VB has to keep careful track. This is necessary so that as each subroutine finishes, control can be passed back to the calling subroutine. This series of nested calls can be seen in the *Call Stack*. How can this help us in debugging? A simple example is a function that you call from many different places in your project. You set a *Breakpoint* in this function because it doesn't always seem to return the correct value. As you execute the project, you enter *Break Mode* each time this function is called. You discover that the times when it returns an incorrect value correspond to an incorrect passed parameter. The *Call Stack* tells you which calling subroutine is passing the invalid value. In fact, if you select the calling subroutine in the window and click Show it will point out to you the exact line

that made the call. Have you ever seen this error [Run-time error '28': Out of stack space]? This happens most frequently when the program is trapped in an endless loop of calls. Subroutine1 calls Subroutine2, which calls Subroutine3, which calls Subroutine1 again. The *Call Stack* clearly shows us what calls are happening and in what order.

Err Object

When your program encounters a situation it can't make logical sense of, it has no choice but to raise an error. Sometimes these errors are the result of the user trying to do something invalid (like saving a file to a floppy drive without inserting a floppy), but most of the time you have asked, in your code, for the impossible. VB uses the *Err Object* to inform us of the problem. When the error message box appears, the *Err Object* has been initialized with all the known information about the error, including the number and description we see. Based upon the type of error and what we are trying to accomplish, we have to decide if we are fixing a program bug, or if the error is the result of incorrect user interaction. In that case we need to add error handling code. Once we have clicked Debug and are in *Break Mode*, we can go to the *Immediate Window* and ask for the error information again.

```
? Err.Number & ", " & Err.Description & " - " & Err.Source
```

While we are testing our error handling code, we can artificially force an error to occur by using the Err.Raise method.

```
On Error GoTo ErrorHandler:
```

```
Err.Raise 9
```

```
. . .
```

```
ErrorHandler:
```

```
Select Case Err.Number
```

```
Case 9
```

```
MsgBox "You are trying to access a element of an array outside the  
dimensions"
```

```
. . .
```

```
End Select
```

The IDE gives us three options that control how we enter *Break Mode* when an error does occur. These settings can be found by going to the Tools menu and selecting Options. On the General tab we see a frame entitled "Error Trapping." The three options are: Break on All Errors, Break in Class Module, and Break on Unhandled Errors. As a short cut we can right click in a code pane, and on the menu select Toggle. The three options are shown with a check by the currently selected one. Once we have implemented error handling via "On Error GoTo" statements, our error handler will trap all errors. We may still have some bugs generating errors but we aren't seeing them. This is when we use Break on All Errors. If we have not yet implemented any error handling or have implemented it in some routines but not all, we can use Break on Unhandled Errors to find additional bugs or routines where error handling is still required. Class module programming and testing is a special case. Under normal program execution, setting or retrieving a class object's properties or using its methods will only cause an error break in the module holding the instance of the object. To actually see the errors inside the class we need to use Break in Class Module.

Putting it all together

We use computers because of the blazing speed at which they store and retrieve our data and perform our mathematical and logical calculations. When errors occur, they happen just as fast. The process of debugging is one where we "slow" the computer down. We want to see what the computer sees and understand how it is "thinking." If someone says to you, "I've lost my wallet (or purse) and I have looked everywhere and I just don't know what to do," the first thing you ask is,

“When was the last time you saw it?” The next thing you would probably say is, “You need to go back to the last place you remember having it and retrace your steps forward.” This describes the process of debugging too. The last time and place are the point in the program where we know everything is okay. Sometimes this is at the very start. Using *Breakpoints*, *Watches*, and *Controlled Execution* we trace the path through the program execution, stopping at every important point. Using MsgBox, Debug Object/Immediate Window, App Object/log files, and Watches/Locals/Call Stack windows, we can search for clues. If we search carefully, we will always find our bug.

Debugging can be a difficult and frustrating process, but knowing what tools are available and how to use them can greatly speed up the process and minimize the pain. I hope this tutorial has given you a glimpse of the possibilities while debugging in Visual Basic 6.