Visual Basic for Windows Debugging Tips and Tricks

Overview

Bugs have always been part of software development, and probably always will be. This session outlines strategies for avoiding bugs, for correcting them once they manifest, and for dealing with the problems that crop up in the real world.

Avoiding Bugs

To paraphrase a cliché, a few minutes spent avoiding bugs is worth hours of debugger time. In general, the best plan is to document well and be consistent.

Coding Style

Comments Everywhere

Memorize the phrase "I never met a comment I didn't like." Repeat it as you shower. Repeat it as you drive to work. Repeat it five times before writing any code.

Many developers (myself included) have a tendency to write code as fast as possible, while promising themselves that they'll go back and comment later. After all, you wrote the code; you know it. Comments help others who may later modify your code, right? Wrong. Comments serve the following purposes:

1. Critical when working in teams
2. Critical for projects that can't be finished in one sitting
3. Critical for forcing you to note undone or partially done functionality
4. Critical when splitting time between several code areas (you'll forget what Function Foo does after spending a week with Function Bar)

You should establish guidelines for code commentary, especially when working in teams. For example, you might decide that all functions carefully document input and output values at the top of the function.

Modularity

Most of us work on one project at any given time. So we tend to put all routines required for that application in a few tightly woven modules. This tends to create a reliance on other code and global variables. It also makes it hard to find, let alone extract, a given routine for use in another application. For example, how many times have you written a generic "Center a form," "Min/Max," or "Center a line to text" procedure? If you're like most people, you end up re-inventing the wheel several times for small tasks because it takes too long to find a module that has the code you need, grab the code (which can't always be loaded into the current application because of procedure name conflicts), and munge it as necessary for the new application.

Rather, create one or more general-purpose libraries that you use in several applications. This file (or files) contains your Min, Max, FormCenter, and similar routines. If one or more of these "externally called" routines requires "internal only" routines, be sure to add the new `Private` keyword to the internal procedures to avoid name-space conflicts with other applications.

Global.BAS

With the advent of Microsoft Visual Basic 2.0, it's no longer necessary to put all global declarations in one module. Nonetheless, it's still a good idea to do so unless you have a compelling reason to do otherwise. There are at least two reasons for this:

1. It's easier to find global variables if you always know where to look.
2. The Microsoft Visual Basic editor allows one code window per module, although that window can be split. Many developers find it convenient to have a global declarations window open at all times. That is only possible if your globals and your "real code" are in different modules.

Likewise, you might consider not adding any procedures to the Global.bas, not even simple initialization routines. That's because initialization routines frequently assign values to global variables—something that's easier to do if the variable definitions are visible while writing code that refers to them.

Always Use Option Explicit

Since its inception, Microsoft Basic had always implicitly declared variables if you referred to them. While conducive to quick coding, it created subtle bugs in code and drove large development teams crazy ("Was the variable called InterestRate or IntRate or RateOfInterest."). Now you can add Option Explicit to the top of all code and form modules. Option Explicit tells Microsoft Visual Basic to reject all variables that are used without first being declared. As a side benefit, you'll notice that the Microsoft Visual Basic parser accepts lines with undefined variables but doesn't expand and colorize them the way it handles most code, giving you immediate feedback.

Option Explicit is done on a module-by-module basis so you can mix Option Explicit modules with legacy code written before Option Explicit. Nonetheless, you should add Option Explicit to old code if that code is being used again, so any modifications to that code can take advantage of Option Explicit.

Avoid the Variant Data Type

Microsoft Visual Basic 2.0 introduced the concept of a chameleon-like data type called *variant*. Unless otherwise declared, all variables are variants. The variant is extremely useful for working with data that needs to deal with both strings and numbers (such as database work), but can introduce subtle bugs when misused. For example:

```
Dim MyName, MyAge As Variant

MyName = "Steve"

MyAge = 25

Call NameAndAge (MyName, MyAge)
```

```
Sub NameAndAge (Age, Person)
     Print Person; " is"; Age; " years old today."
End Sub
```

The output would be:

```
25 isSteve years old today.
```

This particular code fragment violates several other rules (below), but it gives a good reason not to use variants. In this case, MyName was passed where NameAndAge was expecting a number, and MyAge where it expected a string. Of course, since both the Age and Person parameters were not defined, Microsoft Visual Basic assumed them to be variants. In this case, that allowed the routine to function apparently, yet give the wrong result.

Here are three suggestions:

1. Add Option Explicit everywhere (Select Environment from the Options menu to tell Microsoft Visual Basic to add that to all modules/forms for you automatically).
2. Add `DefInt A-Z` to the top of all modules and forms. That way, any Dim without an explicit type character or "As" clause defaults to an integer, which is more likely to be correct.
3. Always explicitly type variables instead of using default type or type symbols. Even using a more common default data type, such as integer, can be a problem if you ever cut and paste code, because the default data type in one module could be different than in another. Many people also find As clauses easier to read than type symbols ($, %, and so on.), especially those not steeped in Basic history, who may forget that # means Double and @ means Currency.

Avoid Multivariable Dim Statements

The preceding code fragment violated another rule: Two variables were declared with one Dim statement (`Dim MyName, MyAge As Variant`). Many programmers, especially ones inexperienced with Microsoft Basic, assume that the `As Variant` applies to both MyName and MyAge. It does not. Microsoft Visual Basic applied the `As Variant` just to MyAge and left MyName as the default data type. In this case, the default data type, since not stated, happened to be a Variant, so no apparent harm was done. But it's easy to think of examples where this type of code could wreak havoc.

Use the Tightest Scoping Possible

It's common to share variables between procedures. All too often, most of us add module-level or global variables to do this. While that might be the fastest solution to the particular problem at hand, it's a poor habit to get into because reliance on non-local variables makes it difficult to "genericize" code.

1. Use local `Const` statements. For example, you may have several routines that scan for commas. It's tempting to add a `Global Const COMMA = ","` somewhere and forget about it. But then your routine relies on that global constant, so it can't be used in other projects. Instead, add `Const COMMA = ","` to the procedures that need it. Or if that seems wasteful to you, add Const COMMA = "," to the module level of your code, instead of using Global Const.

2. Pass Parameters. If you need to share data between code in two different code modules, you have to choose between a global variable and passing parameters. In general, it's better to pass the data as a parameter to avoid cluttering up the global name space. It also makes the code easier to read, because you can see the definitions of all variables referenced, instead of trying to hunt down apparently magic variables. Unfortunately, there is no easy way to pass data to a form module because all variables and procedures are local to the form module. You can use a form's Tag property to share data, but that doesn't work if the data needs to be used during a form load event. For example:

```
'Form 1.  Form1 is the startup form

Sub Form_Click ()

     Form2.Tag = "Steve"

     Form2.Show

End Sub


'Form2

Sub Form_Load ()

     Print Tag

End Sub
```

Remember that the `Form2.Tag = "Steve"` line causes form2 to load itself. During Form2's load event, the Tag property is still null. Therefore, the `Print Tag` code in form2's load event *always* prints a null string.

Declarations at the Top of Procedures

Now that Option Explicit is with us, we see many more `Dim` statements in Microsoft Basic code. Technically speaking, you can put the `Dim` anywhere, before the variable is first used. But you might find that life is just slightly easier if you put all `Dim` statements at the top of the procedures. For example, one could argue that the `Dim` statements should come after the first `If` statement in the code below because there is no reason to declare the variable if it is not needed. Still, much like using one Global.Bas, you do less "Dim hunting" if you always know where the variables are defined.

```
Sub SetCombo (CurControl As Control, Text As String)

     Dim i As Integer

     Dim MaxItems As Integer


     If CurControl.ListCount > 0 Then

          MaxItems = CurControl.ListCount

          i = 0


          Do Until CurControl.List(i) = Text Or i = MaxItems
               i = i + 1
```

```
            Loop

            If i < MaxItems Then

                 CurControl.ListIndex = i

            End If

     End If

End Sub
```

Use Lots of White Space

All code in this session demonstrates a liberal use of white space. Blank lines, for example, are used as logical separators (leave a blank line after a block of `Dim` statements or after functionally related groups). Likewise, simple If/Then statements that can work as single code lines (e.g. `If X <> 0 and Y <> 0 Then SetControlPosition X, Y, Z, True`) are usually expressed in multiple lines, such as follows:

```
If X <> 0 and Y <> 0 Then

     SetControlPosition X, Y, Z, True

End If
```

This style helps avoid scrolling horizontally because the code fits into less horizontal space. It also mentally reinforces that you should consider the `Else` part of the `If` statement. Similarly, comments are usually placed just above the code described, rather than to the right.

Indenting is another key way to use white space. Most people indent loops and `If` statements, but you can consider using indents where you want to remember to restore states. This is most useful for file I/O, error handling, and setting flags. For example:

```
Open FileName For Input As 1

     Input #1, NumAccounts

Close 1


On Error Goto FileNotFound

     Open FileName For Input As 1

On Error Goto 0


GlobalFlag = TRUE

     Call FillListBox

GlobalFlag = FALSE
```

Keep Procedures Short

Frequently, a once simple routine grows to a remarkable length over time. This is especially easy to do in the age of Cut-Copy-Paste. Perhaps replicating a few lines is the most efficient solution to a quick problem, but it quickly degenerates into a maintenance problem when those few lines, replicated several times, need to be changed globally. Therefore, it's a good idea to keep an eye on the absolute procedure length and start chopping once it exceeds a screen or two. Reducing repetitive code also reduces code size, of course.

Naming Conventions

Other than the eternal "What language is best" debate, few topics draw so much attention as the merits of different naming conventions. These religious wars are beyond the scope of this session but I will attempt to summarize a few simple thoughts. The general rule, of course, is simply to be consistent, regardless of the chosen method.

Control Naming

Microsoft Visual Basic requires that controls have unique names (except control arrays). Therefore, Microsoft Visual Basic dynamically invents new names as new controls are created. Unchecked, forms end up with ten text boxes, conveniently named Text1 through Text10. While this might be fine for very small projects, it gets difficult to remember exactly which control does what. Therefore, the Microsoft Visual Basic on-line Help system suggests a simple naming convention in which all control names begin with a three-letter prefix signifying the control type (txt for a Text Box, cmb for Combo, cmd for Command Button, frm for Form, and so on) and includes a simple description. For example, txtAccountName is much more self-evident than Text5. Even better, attempt to keep control names and captions identical. For example, a label captioned "Account Name" should be named lblAccountName not lblAcctName. Nested controls, most notably menus, are also easier to remember if child names reference their parent name. For example, mnuEditDelete is more descriptive than mnuDelete, especially if you have a Delete item listed under several top-level menus.

Procedure Prefixes

Microsoft Visual Basic encourages you to write modular code because all forms, by definition, are self-contained. But, unless you use the new `Private` keyword, procedures and functions in all Microsoft Basic modules are global to the project. That isn't normally a problem, but name conflicts occasionally surface. Or given several Microsoft Basic modules, it isn't always clear where a given procedure lives. Therefore, some developers prefix procedure names with information about which module it lives in. For example, a module that manipulates bitmaps might have all public procedures prefixed with BMP, so BMPGetSize doesn't conflict with FileGetSize.

Consistent Procedure Conventions

Commonly, you have several procedures that take similar or identical arguments. Life becomes more sane if you always list parameters in the same order, preferably with the same name. For example:

```
Sub Foo (CurID As Integer, CurControl As Control)

Sub Bar (CurID As Integer, CurControl As Control)
```

is easier to work with than:

```
Sub Foo (CurID As Integer, CurControl As Control)

Sub Bar (CurControl As Control, CurID As Integer)
```

and certainly better than:

```
Sub Foo (CurID As Integer, CurControl As Control)

Sub Bar (Ctrl As Control, IDNum As Integer)
```

Use Case Effectively

Microsoft Basic, unlike Microsoft C, and some other languages, does not differentiate between case, but that doesn't mean that you can't use case to convey information. For example, it's common practice to use all upper-case letters and underscores to signify constants, such as WHITE or DARK_GREEN. On the other hand, variables might mix upper and lower case instead of using underscores (DarkGreen). But you'll have to be careful to keep case consistent, because Microsoft Visual Basic seems to have a tendency to change variable case. In reality, that's just a side-effect of the threaded p-code compiler technology under the hood. (As each line is typed, Microsoft Visual Basic incrementally compiles it into p-code. Microsoft Visual Basic interprets that p-code back into ASCII and lists the ASCII back to the screen. To save memory, Microsoft Visual Basic just stores the p-code representation rather than both the p-code and the ASCII text. So entering FooBar as "foobar" causes all references to FooBar to suddenly change case.)

Hungarian Notation

It's not uncommon to see the letter "g" prefixed to global variables or "i" prefixed to integers. The global prefix is generally useful but the data type prefix is most valuable with compilers that take some time to discover mismatched data types. Luckily, Microsoft Visual Basic has a fast compiler that flags passing incorrect data types to procedures almost instantly. Still, Hungarian notation is an excellent way to make code even more self-documenting. The literature on Hungarian notation is both exhaustive and well beyond the scope of this session, however.

Correcting Bugs

Unfortunately, even the best planned, most consistent code always leaves a few bugs to swat. Perhaps the best general advice is to know your debugger inside and out. That said, there are a few simple tactics to consider.

Breakpoints and Stepping

It's easy to get into the habit leaving one hand permanently glued to F8 while debugging. But there are some common ways to accelerate the process a bit.

Single Stepping vs. Procedure Stepping

Procedure Step (Shift+F8 or the double steps icon in the toolbar) is a great way to step over a procedure instead of meandering through the entire call. For example, learn to press Shift-F8 to avoid executing every line in a function when you're reasonably sure that the problem is outside of the function. Also note that pressing Shift+F8 acts precisely like F8, if there is no procedure to jump around.

Use Break and Step Together

Almost inevitably, you learn to put breakpoints as close to the potentially incorrect code as possible to avoid unnecessary single steps. But you might find yourself walking through dozens of lines of code if you have no idea where things go awry. In that case, you might consider using a series of breakpoints, equidistantly placed, to help you zoom in on bugs. For example, instead of single stepping through 100 lines of code, place five breakpoints 20 lines apart and test at each breakpoint. If the error occurs, you have it nailed down to a 20-line region that can be subdivided again.

Watch Expressions and Watchpoints

Of course, single stepping is most expedient when you have a simple way to detect when things go bad. In Microsoft Visual Basic, you can use watch expressions to get the current value of variables just by pressing Shift-F9 (or clicking the eye glass icon), while the cursor is over a variable. You can even evaluate complete expressions, such as "ScaleWidth - cmdOK.Left," with that same simple keystroke. Additionally, you can manually enter any valid Microsoft Basic expression and see the results. Logical expressions evaluate to True or False. Therefore "X > 500" produces either 0 (FALSE) or -1 (TRUE) in the debug (watch) window.

Watchpoints are even more powerful because they effectively use Microsoft Visual Basic to do the "set a breakpoint, set watch expressions, single step until the expression changes" sequence for you. For example, you can tell Microsoft Visual Basic to monitor a given expression continually until the expression changes to True. When the expression is true, Microsoft Visual Basic halts execution and brings you to the line of code that triggered the change.

Microsoft Visual Basic lets you set the scope of the evaluation. This is handy for several reasons. The simplest is that watching variables slows Microsoft Visual Basic down, so a global watch can affect performance significantly. More important, however, is the problem with the same local variable name being used in several procedures. In that case, Microsoft Visual Basic needs to know which procedure to monitor.

Use F9 and F5 to Move by Chunks

Frequently, you hit a breakpoint, single step a few times, then become convinced that the problem is further down the code path. You can either hit F8 many times or use an F9 (break)-F5 (Continue) combination. For example, to skip down to the bottom of a procedure do the following steps:

1. Set a break at the bottom of the procedure.
2. Press F5 to execute to that point.
3. Press F9 again to toggle off the breakpoint.

Use the Call Stack to Jump Out of Procedures

Similarly, many developers inadvertently step into a procedure and want to jump back out quickly. Try this trick:

1. Hit Ctrl-L to show the call stack. Jump to the procedure that you just left. The cursor is now on the line that made the call.
2. Move the cursor down one line and press F9 to set a breakpoint.
3. Press F5 to continue execution to that new breakpoint.
4. Press F9 again to toggle off the breakpoint.

Shift+F2 and F2 to Navigate Procedures

You can effectively hypertext through code by pressing Shift+F2 while the cursor is over the name of a procedure. But there is no "go back to where I was" key, so remember to make mental notes before hitting Shift+F2.

Using the F2 key to show a list of all procedures is new to Microsoft Visual Basic for Windows, but long time Microsoft Basic users will instantly recognize the resurrection of the View Subs shortcut from QuickBasic. F2 pops up a dialog box that shows all of the forms and modules in the project and then, in a lower list box, all of the procedures in the selected module. This is especially useful when you aren't sure which controls have events handled, so scrolling through the dual drop downs on a code window is time consuming.

Consider Using Debug Flags

On large projects, it's common to skip entire regions of code while debugging. Or perhaps you want to programatically set some variables instead of continually typing the same values into dialog boxes. Either way, you can use a global DebugMode flag to control temporary jumps through code. For example:

```
Global DebugMode As Integer

DebugMode = TRUE


If DebugMode Then

     FileName = "Test.txt"

Else

     FileName = GetFileName (CurDrive)

End If
```

Of course, you want to document these jumps carefully and possibly remove them from shipping code to save space. You can even write a code preprocessor to remove such code automatically before making retail builds.

Dealing with the Real World

Developers generally demand computer hardware in excess of what their customers use. While that makes sense in terms of minimizing development time, it makes it less common for developers to hit runtime errors like Disk Full, Out of Memory, and so on. Clearly, a solid project has to accommodate runtime errors, unlikely though they may seem to be.

Error Handling

Do It Early

Many programmers have a habit of treating error handling and comments the same way—saving it until last. The problem with saving these tasks until last is obvious: There is a fair chance that they will never be done.

All I/O, All Memory Allocation

Clearly, you want to encase Open, Print#, and Put# statements within On Error. By habit, many of us put these simple I/O statements in-line. But, in many cases, the code actually can be generalized into a standard file I/O procedure that has all of the error handling bells and whistles. For example, instead of:

```
FileNum = FreeFile()
Open "Steve.INI" For Input As FileNum
```

use something more like:

```
FileNum = stdSeqFileOpen ("Steve.INI")


Function stdSeqFileOpen (FileName as string) As Integer
     Dim FileNum As Integer
     On Error GoTo stdSeqFileOpenNoHandles
          FileNum = FreeFile ()
     On Error GoTo stdSeqFileOpenNotFound
          Open FileName For Input As FileNum
     On Error GoTo 0
     stdSeqFileOpen = FileNum
Exit Function


stdSeqFileOpenNoHandles:
     If Err = 67 Then
          'Too many files
          MsgBox "Please close a file before continuing."
     Else
          'Something else
          MsgBox "Assert: stdSeqFileOpen "+Str$(Err)+Error$
     End If
Exit Function


stdSeqFileOpenNotFound:
     If Err = 53 Then
          'File not found
          MsgBox "File not found. Ensure correct disk in drive."
     Else
          'Something else
          MsgBox "Assert: stdSeqFileOpen "+Str$(Err)+Error$
     End If
     Close FileNum
End Function
```

Indent To Show Trap Areas Clearly

The example above also demonstrated liberal indenting. It's easy to forget to turn error off handling and even easier to forget to redirect it to different places depending on the code. Indenting is simply one mechanism that some people use make it easier to see what is trapped and what isn't.

Only Use Where Necessary

Error handling is very powerful—sometimes too powerful. Unless care is taken, it's possible to inadvertently ignore a valid error. On Error Resume Next is especially dangerous because it tells Microsoft Visual Basic to continue on any trappable error. Your code can easily be generating a legitimate divide-by-0 error that you never see. On Error Resume Next is really best used when you are *very* sure about what errors a small area of code can generate. For example, if you mean to ignore a divide-by-0 error, it's valid to place one line of code between On Error Resume Next and On Error Goto 0.

Check Error Code Before Resume

The error-handling example also checked the error code (Err), instead of assuming what the error is. It's easy, when opening a file, for example, to assume that the only possible error is File Not Found and simply offer the user a chance to abort, retry, or ignore. Of course, there are several other possibilities—albeit unlikely. Therefore, you should check the error, do anything special for a small number of errors that you're prepared for, and create a generic error message for anything else.

Use Case Else

While not directly related to error handling, the Select Case statement offers another chance for error detection. Instead of assuming that you know all possible cases, always add a Case Else to catch the unexpected. For example:

```
Function GetClassName (Device As String) As String
     Select Case Device
          Case "Video"
               GetClassName = "AVIVideo"
          Case "Audio"
               GetClassName = "WaveAudio"
          Case "Bitmap"
               GetClassName = "Pbrush"
          Case Else
               MsgBox "Assert: GetClassName Device=" + Device     End
Select
End Function
```

Use MsgBox to Display Information

The examples above used the MsgBox to display unexpected errors. These messages should be explicit, not only to help you during debugging, but also to help you find errors that might occur after an .EXE is shipped to customers. You'll save considerable time if a customer who hits an unexpected error can give you very precise information. For example, you can put the module name, the procedure name, a possible explanation, and applicable data into the message.

Miscellaneous Tips

The following tips are random tidbits of debugging lore that might be helpful to you. No logical grouping is offered.

Mark and Date-Undone Sections

Create a standardized convention for marking undone code, such that you can use Search to quickly scan code later. At the same time, you should add any information to the comment that you might need later when finishing the code. For example:

```
'NYI 1/24/93.  Add handlers for CD-ROM / other removable media.
```

Window Positioning

Even in the age of the SuperVGA, there is never enough screen real estate to display everything you want to see at the same time. Here are a few suggestions:

1. Close the Tool box while writing code.
2. Keep the Project Window open (perhaps on the far right) and sized at all times so that all forms and modules are visible without scrolling. Custom controls can be below the current scroll, however, because you never need to double-click on a custom control to edit it.
3. Keep a GLOBAL.BAS window open all the time, but keep it somewhere off the screen.
4. When debugging a form, don't double click on the form's name in the Project window. This shows only the form itself, generating unnecessary screen clutter. Select the file name and hit the View Code button instead; or use F2 to jump between procedures.
5. Keep the Property window from overlapping forms so that clicking on either the form or the Property window doesn't obscure the other.
6. By default, all code windows pop into the same screen location at the same size. Stagger multiple, open-code windows slightly so you can jump modules by clicking on a window.

Use Multiple Modules to See Multiple Procedures Simultaneously

Microsoft Visual Basic for Windows shows one code window per module, although that window can be split. It might make sense to put commonly used procedures in different modules, simply so you can view several of them in their own windows at the same time.

Don’t Maximize Forms

You might want access to the Microsoft Visual Basic tool bar while debugging (hit Pause, Watch, Stop). If so, avoid maximizing forms at runtime because your form will obscure the tool bar.

Multiple .MAK Files

There is no rule that says that any one application can have only one .MAK file. In fact, you might have one or two modules of debugging information that you want to exclude from finished builds. Therefore, consider using a RETAIL.MAK and DEBUG.MAK, or use similar files.

Save Frequently in ASCII

It goes without saying that you should save your code fairly often in case the unexpected happens. Missing .DLLs or Windows API calls can make your computer especially queasy. You should also use ASCII file saves exclusively for several of the following reasons:

1. It's possible to reconstruct corrupted files .
2. Saving allows a source library manager, such as Microsoft Delta, to keep track of code changes.
3. Saving lets you use more advanced text editors to make global code changes.

Write Tests Before Writing Code

Before writing code, consider how you will test it. If possible, construct test suites before or during application development instead of waiting until the end. Even better, use different teams of programmers to develop and test code simultaneously since the developer frequently makes subtle assumptions that other people (especially those trained in testing methodology) won't make.

Be Careful with Replace

Replace is very powerful, but has a tendency to trounce on innocent code if you don't restrict the range well enough. In general, avoid global replace without at least using the Verify option. The Match Whole Words Only option is also useful when looking for short, relatively common phrases.

Multiple Level Undo is Your Friend

Just a quick reminder that Microsoft Visual Basic 2.0 added unlimited levels to the old Undo command. Of course, being reasonably sure that code works before you save it (and overwriting a working version of the same file doesn't hurt either). Serious development teams should also carefully consider using source library management tools such as the new Microsoft Delta.

OLE Automation

Visual Basic 3.0 supports OLE Automation of other OLE 2 applications, as well as In-place editing of objects through use of the OLE 2 control. The single most important thing to understand when using OLE Automation is the object model of the applications you are communicating with. Many errors that occur in OLE Automation code happen because you called and object and it didn't respond in the way you anticipated. When using CreateObject, keep in mind that you might be starting multiple instances of applications. This can quickly result in out of memory or out of resources errors. Instead, load multiple documents in the same instance of a particular application. Variable Scoping is key to working with OLE objects. If a variable looses it scope, the OLE object reference will also be lost. This doesn't mean that all OLE Objects should be declared as Global. But, if an object variable has been defined at the form level, when that form closes, the object reference closes also. One of the most common bugs is a Visual Basic application is to get an invalid object reference error because the object variable has gone out of scope.

Conclusion

Few people consider debugging to be enjoyable, so time spent avoiding bugs almost certainly pays dividends. Code style, although subject to stylistic interpretation, should at least be consistent to you—preferably consistent between others on your development team as well. Adding robust error handling is critical because customers and reviewers often see it as the difference between a well-developed product and a product seemingly assembled by one person in his basement. But the actual process of debugging is still more an art than it is a science. The best advice is simply to know your editor and debugger as thoroughly as possible and always be open to expanding your tools with additional debugging aids. In any event, it may comfort you to think of debugging skills as your job security.