

V

B

A

F

O

R

T

H

E

V

B

E

Acknowledgments

I would not be even a fraction of the VBA programmer I am without standing on the shoulders of giants. One such was Chip Pearson. Unfortunately, he died in a car accident in April 2018. The programming world and especially the Excel world, is very much poorer for him not being there. All communications I had with him showed him to be polite, thoughtful, very experienced, and willing to share that with others. RIP Chip.

My brother Dave has been invaluable in proofing and pointing out, amongst amongst other things, my extreme inconsistency and difficulties with punctuation. Thanks bro.

My talented other half, Lisa, who wrote most of the original code to the original CodeCode excel add in. She has had over 1000 downloads of it over 7 years.

The internet. Thank you to the gazillions of people who have posted code and other items there for people like me to download, copy, and abuse.

More specifically: Thank you to the Code Cage web site and the people on it. It's not there any more and I personally think that's a real pity. Thank you to Eileens lounge and the experts there and their many many years of experience and not least their humour.

UPDATE:

Directly after his death, Chips site was pulled in its' entirety. It's back up now. Thank you to those who are responsible.

Contents

Acknowledgments.....	2
Table of Code	7
Table of Figures.....	9
Introduction	10
Disclaimer.....	11
Some initial comments about VBA and the VBA IDE/VBE	13
Environment	14
Application specific code	18
REMEMBER REMEMBER	19
Project names	24
Code to SORT DIMS.....	27
SortDims Walkthrough.....	35
Functions used from the Extensibility library in subccSortDims.....	41
Why Sort the dims	44
Sort a selection of Dims	48
Splitting up Dims from a single line	54
What's in a name	61
Recap number one!	62
Where Are We	63
Procedures to Classes	68
A little about classes.	68
Class Advantages.....	68
Class Disadvantages.	68
Simple class layout.....	68
Building a class from a Sub.....	71
Code to Split Dims up.....	91
Debug Print variables.....	97

De debug.....	103
VBE programmers do it Immediately!	107
Button up	109
Messages.....	118
Debugging and Tracing	128
The Registry	145
Line Numbers	148
Recap number two.....	164
Time out!.....	170
Dims Bottom	174
Building A Compile Report and Multithreading.....	177
Debug Reprise	202
Continuations.....	209
I do Declare	213
Time Ladies and Gentlemen please!.....	221
Insert timing code	224
Some Timer examples.....	235
It's Complicated	241
No Comment!.....	254
Inserting comments	262
Key Words	244
HTML Report.....	249
Recap number three	250
What's to come?.....	252
Export and Import.....	254
Saving as an Add-in	254
Runtime Errors	254
Parameters in WhereAreWe.....	275
Considerations for Inserting, Cross referencing and Cleaning.....	283

Multilinguality.....	285
Get the strings.....	286
Finally.....	288
Appendix Notes.....	290
Appendix A Links.....	291
Chip Pearson.....	291
Deconstruction of a code module.....	291
Code for Menus and buttons in the VBE.....	291
Jan Karel Pieterse.....	291
Stephen Bullen.....	291
Multithreading with vbs.....	292
Cyclomatic Complexity.....	292
MSDN Discussion of high-performance time stamps.....	292
Defensive programming and reliability. Analysis of post mortem NASA software.....	292
Naming Conventions.....	292
Version Control.....	293
String optimization.....	293
Appendix B Software for the VBE.....	294
Pretty Code Print.....	294
MZ-Tools.....	294
Smart Indenter.....	294
WinMerge.....	296
Spy++.....	296
RubberDuck.....	296
CodeCleaner.....	297
Code Manager.....	298
vbWatchdog.....	298
VBA Code Compare.....	298
Appendix C My Personal Naming Convention and procedure layout.....	300

Variables	300
Procedures	301
Procedure Comments	301
Procedure layout.....	303
Module Names.....	304
Appendix D VBA as an OOP programming language.....	305
Appendix E Creating an Add-in	Error! Bookmark not defined.
Appendix F Clearing the immediate window.....	307
Appendix G P-Code	309
Appendix H Acronyms	310
Appendix I Comments and Contribut as of <>	311
Appendix J Latest cWhereAreWe Class Module	Error! Bookmark not defined.
Appendix K Cross Reference example.....	314
Appendix L High Definition Timer Class.....	319
Appendix M Word Doc stuff.....	322
Appendix N Contact.....	324
To Do	Error! Bookmark not defined.

Table of Code

1 subccSortDims.....	28
2 subMsgBox.....	33
3 subccSortDims walkthrough	35
4 subccSortSelectedDims.....	48
5 GetSelection parameters	52
6 Setting lines to sort	53
7 All Dims on a single line	54
8 One Dim per line	54
9 fncGuessVarType	56
10 INI File entries for my pre/suffixes.....	59
11 Declaration as Global	61
12 subWhereAreWe.....	64
13 Setting variables to VBE objects.....	66
14 Simple Class layout.....	68
15 Simple cNameExample Class code.....	69
16 Using the cNameExample example class.....	70
17 subWhereAreWe recoded to convert to a Class.....	71
18 subInsertGetProperties.....	74
19 subtestcWhereAreWe.....	84
20 subtestsubWhereAreWe.....	85
21 subWhereAreWe corrected for line numbers	86
22 My Procedure parameters layout.....	90
23 subccSplitAllDims	92
24 subInsertSelectionDebug	98
25 Sub before subInsertSelectionDebug	100
26 Sub after insertSelectionDebug	100
27 subccDeleteDebugPrint	103
28 Keep UserForm in the VBE	105
29 Clear immediate window with SendKeys.....	107
30 Clear immediate window. Debug.Print 1.....	108
31 Clear immediate window. Debug.Print 2.....	108
32 Button code for the CLASS module.....	110
33 Button code in STANDARD module.....	110
34 subBookMarkAndBreakpoint.....	113
35 Userform/class properties for subMsgBox	118
36 Using the MsgBox userform.....	123
37 Tracing example 1	128

38 subSpendTime.....	130
39 Tracing Example 2	130
40 subListProcsToImmediateWindow	131
41 subAddTraceLinesToAModule	134
42 subDeletDebugPrintForModule1	139
43 Registry key for VBA.....	145
44 VBA Registry calls	145
45 SaveSetting statement	146
46 subDeleteSettings example	149
47 subGoToLine	153
48 subccInsertModuleLineNumbersInProcedure	158
49 subccDeleteLineNumbersInProcedure	160
50 aamTopOfMacros	165
51 Ancient variable suffixes	170
52 VBA Converting example	172
53 subccInsertSingleDim	174
54 fncGetDeclarations	213
55 fncGetJoinedArray	214
56 fncGetJoinedArrayFromDeclarations.....	216
57 WhereAreWe Update	224
58 fncLookThroughDims 1	226
59 fncLookThroughDims 2	227
60 My method of dealing with errors.....	264
61 subccInsertErrorCodeAtLine	265
62 String length example	285
63 Sub Update SmartIndenter Tab Width	295
64 Some of my procedure comments.....	301
65 My procedure layout.....	303
66 Clear immediate window with API.....	307
67 Ancient type symbols.....	Error! Bookmark not defined.
68 Latest cWhereAreWe Class Module	313
69 High definition timer class	319

Table of Figures

1 My VBE Toolbars	15
2 Going grey	19
3 Can't enter break mode	20
4 Document References.....	23
5 WinMerge 1	52
6 WinMerge 2	53
7 Can't enter break mode	97
8 Insert Debug Options	102
9 My MsgBox UserForm.....	122
Figure 10 subMsgBox in use.	126
11 My InputBox UserForm.....	127
12 Registry after SaveSetting.....	147
13 Error on DeleteSetting	149
14 Error message from handler code	151
15 Module line number on code line.....	152
16 Error message with module line number	153
17 My mumble menu	168
18 My Arrange macros menu UserForm.....	169
19 Variable not defined	174
Figure 20 Variable not defined	178

Introduction

This book/document is for people who...

- Know what VBA is and use it
- Want to "enhance" the VBE
- Want to do that on their own terms
- Have some knowledge of the VBE interface and how to manipulate it

This book/document is **NOT** for people who...

- Are completely fresh to the VBE. Though, a lot of the processes are explained step by step

This is a **PRACTICAL** guide to using VBA to alter and debug code in the VBE. The emphasis is on **code** rather than object models and so on. There are some excellent sources that describe that stuff in lots of detail, some of which are listed in the appendices. Most of those sources offer excellent code as well. If you are looking to perform a specific action, then I'd advise you to google first.

However, I specifically **don't** want this to be a "ooh lets copy this" source. Sure, you can do that but most of it will need tweaking. I want you to think.

This offering is an attempt to show you **how** to code things in the VBE that other sources don't tend to do. An example is sorting Dims in a procedure.

I'm reminded by Hans of Eileens lounge fame, that in order for any of the procedures here to run they have to be allowed to run. See [Environment](#).

And finally, for the introduction anyway, none of the code here is C# C++ C Python Java Javascript or any other language except VBA.

This really is VBA for the VBE.

Disclaimer

This bit is going to be pretty long. Have you tried reading some of the microsoft disclaimers? HAH! And you have to agree to them! Not so here. I'm just saying it as it is for us so we don't get beat up or taken to court.

I make no claims for consistency here or anything else for that matter. There are variable names used as appropriate for their purpose in a piece of code, and some variables will have different prefixes for the same thing depending on when it was written. For example, I don't use integers now, long history there, and now use Long. I used to refer to long with a single l. I found that wasn't enough for me, and now use lng for all "integer" numbers. The result is some older procedures use l for integer then l for Long and now use lng for Long and also str for string and other oddities. Therefore, there are **NO** guarantees as to the **relative quality** of the code! And it's all deliberately open and free so anyone can alter it as they choose anyway! That's not an excuse. It's a fact. And a description of code here.

If you use any code from here, it would be nice if credit is given somehow, although that's **not** obligatory at all. I myself make use of other peoples' code unashamedly, but I'll always do my best to give credit where it's due. If you want to pretend you wrote it though, no one is stopping you.

If anyone wants to crit any of the code here, go ahead. **Love it!** Let me know if you would delete/add/change anything for what you think is better. If applicable I'll change it and you **WILL** get credit.

This book is mostly about **CODE**. There is a **SIGNIFICANT** amount of it here, and I wouldn't blame you for feeling cheated with the actual substance of the book. However, I feel it's worth including it so that it's possible to copy/paste and/or work through offline. Too often I've come across books that include snippets of code that are difficult to read because there's missing context and you need internet access to perhaps download.

There is a **LOT** of code here performing string manipulation.

All the code is downloadable from thinkz1.com as well as listed here. It's in the VBA for the VBE Docm.

This is all free. But hehehe, donations are welcome.

Some appendices may dissappoint you. Most notably [VBA as an OOP Programming language](#), and [P-Code](#). Both of these are covered far better than I could ever do so in the given links and I say exactly that.

I try desparately to not pretend to be what I'm not!

I do my best to explain the code given here to show how things work. It's up to you if you want to extract code sections into subs and so on. I **STRONGLY** advise you, at least at the start, and if you're not using

the MZ-Tools or Rubberduck code explores, to use a **LOT** of modules! And, as soon as you've inserted a new module to **RENAME IT!**

Much of the code here has been "refactored" specifically for this book. A lot of it is from code from an Excel add in built quite some time back almost entirely by Lisa. There are approximately 700+ subs and functions in that add in. I've said "a lot" because some of them are just class methods to return things. Lisa does a good job of maintaining a list of subroutines on the site using a procedure she wrote, which is also on the site. She is always updating it. Mad coder incarnate. The same stipulations apply. If you get in touch, she **WILL** reply.

The add in is downloadable from thinkz1.com. For free. Same conditions apply. My/our [email](#) is in the appendices. The equivalent code with this book though is a leeeetle bit better. Which is part of the definition of being refactored. But beware! **Some** of the code here is quite old, works, and hasn't been touched because of the legendary cliché... If it ain't broke, don't fix it. subSortDims, the piece of code we kick this journey of with is an example of that.

I make no excuses for my use of exclamation marks dots ellipses and question marks!!!! ... ??? Thanks again Dave!

And here's a punctuation warning! Depending on your MS Word setup, if you copy and paste code from this document then you may have to change the type of quotation marks after pasting into the VBE. I've done my best to allay that... But hey! It's your word setup.

To date, the only way around this that I've found is to do just that. Copy and paste into the VBE and then run code to alter the quotes as needed.

Now then. Don't expect a logical progression. This book wanders a bit. It's intentional sort of because that's the way I work. While looking at some code, we may see that we could change it in some way to make it better. And we go there and do it. In that way it doesn't follow a strict pattern. This is also hmmmmm, intentional. If you like it's a stream of consciousness, in that a door opens and we go there. But we always at least try to go back through the door and get back to what we were up to before. Climb back up the chain. This means it's not really a book of separate chapters. Most of them follow on and are connected. Sort of.

And finally, for the disclaimer, everything here and elsewhere is provided "As Is" and under the MIT licence, the GNU licence, and any other licence you can think of for public stuff.

Some initial comments about VBA and the VBA IDE/VBE

IMNSVHO:

- VBA ain't going away any time soon.
- It's free. Totally. No extra costs. No hidden costs. Nothing.
- The VBAIDE is common across applications. It's the same in ALL of them.
- There's a **LOT** of legacy VBA code in the wild.
- VBA has been much maligned, from being a "script kiddy" language to not being proper OOP. I point to a discussion of this in the [appendix](#). I do not enter into a discussion myself.
- A plus point is VBA comes with most of the desktop office applications: Excel, Word, Access, Powerpoint and so on, and others. Most famously AutoCad! It's even possible to add it to an application that doesn't have it!
- The VBAIDE has been more or less the same format with the same windows and so on since the beginning, and it looks very much as though it's going to stay that way with no further major updates. It's a constant.
- VBA is a great deal better than repetitive manual work, and a great deal better than nothing at all. Not my words and I'm not sure where from. ☺. But true.

UPDATE: Source Wikipedia

VBA itself has actually been updated quite a bit apparently to, amongst other things, try and take advantage of 64-bit processors. Not sure how or if that has changed the VBE though. Development on VB stopped, and the last version of VB, VB6, was delivered in 1998.

UPDATE: Source RubberDuck

The VBA IDE was last updated well over 20 years ago.

UPDATE: MZ-Tools

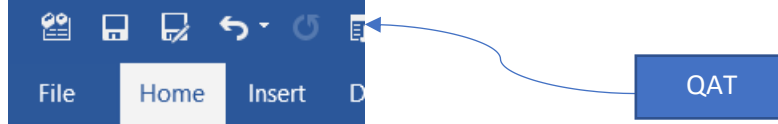
MZ-Tools V8 is a **major** update to a major set of tools. IMO There is some minor overlap with rubberduck though. While rubberduck is built in, I think, C++, MZ-Tools 8 is a total rewrite of MZ-Tools 3 with a **LOT** more, in .Net. Their philosophies though, are chalk and cheese.

Environment

We're going to practically live in the VBE for a while, so here are a few tips to make life **possibly** a bit easier. Note though, this isn't obligatory! If you work differently then vive la difference! So, open the VBE and let's get started. And remember, these are only suggestions!

- Put all the good icons you like on one custom toolbar.
 - Right click on any toolbar or menu item.
 - Click the bottom option which should be "Customize...". A small window opens with three tabs.
 - On the Toolbars tab, create a New empty toolbar. This is a floating toolbar. Drag it to the top where the standard toolbar is.
 - Go to the second tab, Commands.
 - Click the top left-hand category. This should be File.
 - Look at the Commands on the right-hand side and decide if you want it in your new custom toolbar.
 - If you want a command, drag it to your toolbar.
 - Repeat for the other categories. It's worth going through all of the commands to see if there's something you'd like to have an icon for. This may take as much as 10 minutes!
 - While the Customize window is open you can alter any of the toolbars by dragging the command icons to them or off them. If you want to alter the standard menu for example, just drag the icons off it and release the mouse.
- Get rid of toolbars you don't need, because you've put all of your icons together on one custom toolbar. Opening that toolbar then opens everything you want.
 - Untick the Toolbars that aren't needed in the Toolbars tab
- Alter the indent to two rather than the default four. This will **significantly** reduce sideways scrolling.
 - Tools
 - Options
 - Editor
 - Alter Tab width
 - OK
- Switch on Option Explicit for new modules.
 - Tools
 - Options
 - Editor
 - Tick, Require variable declaration.
 - OK
- Make switching to the VBE easier than going to the developer tab, by adding an icon for the VBE/VBA to the QAT (Quick Access Toolbar) in the application. Put it in the same place in all the applications you use. I suggest top left.

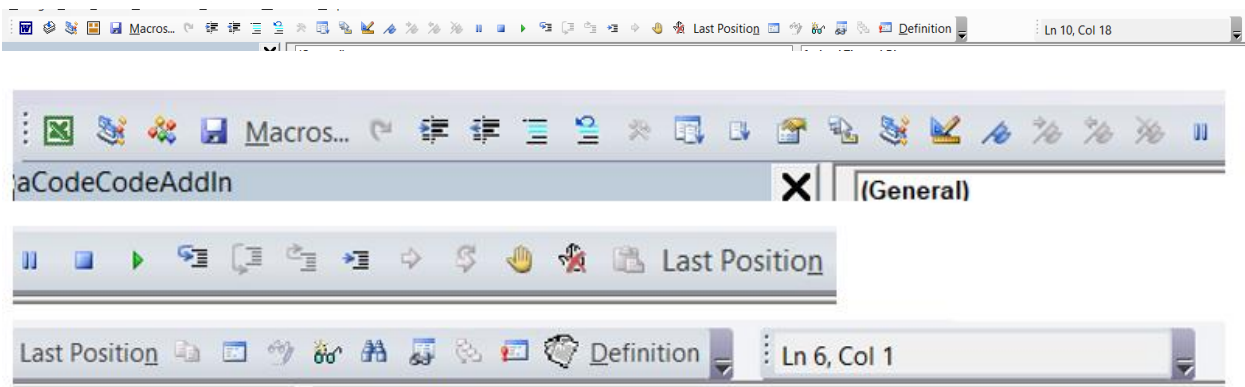
- Go to the Document. The QAT is not in the VBE. You will probably see a few icons at the top left of your screen. That's the QAT.



- Right click one of the icons.
- Click Customize quick access toolbar...
- A large window opens
- From Choose commands from, select All commands
- Scroll down the list on the left deciding which commands/icons you want on the QAT
- Use the Add button between the lists to add them to the QAT
- OK
- Trust access to the VBA project object model in the parent application.
 - File
 - Options
 - Trust Center
 - Trust Center Settings
 - Macro Settings
 - Choose Enable All macros
 - Tick Trust access to the VBA project object model
 - OK
 - OK

For the record, here's my VBE toolbar setup. This is from Excel as you can see by the far left icon. It's **IDENTICAL** in Word, Access and so on once you build it in any one of those applications.

1 My VBE Toolbars



I've overlapped the pictures so you can see that most of the icons are in a single custom toolbar.

Note there are only **TWO** toolbars.

Line/Column cannot be put into a different toolbar. It is in fact in the Standard toolbar, from where it just will not move! Stubborn bugger! So, I've removed the other icons from the standard toolbar. It's easy enough to reset if I need to.

The items I have in my custom toolbar are:

1. View Microsoft Excel (application)
2. Project explorer
3. Insert Module
4. Save
5. Macros
6. Redo
7. Indent
8. Outdent
9. Comment block
10. Uncomment block
11. Toolbox
12. List Properties/Methods
13. Parameter Info
14. Design Mode
15. Toggle bookmark
16. Next bookmark
17. Previous bookmark
18. Clear all bookmarks
19. Break
20. Reset
21. Run
22. Step into
23. Step over
24. Step out of
25. Step to cursor
26. Set next statement
27. Show next statement
28. Toggle breakpoint
29. Clear all breakpoints
30. Last position
31. Locals
32. Edit watch
33. Add watch
34. Watch window
35. Call stack
36. Immediate window

37. Go to Definition

38. Pretty code print

Yes, I know there's an edit toolbar and a debug toolbar and so on. I chose to do things this way because everything I want on a regular basis is in one place. It doesn't move and can be put on one line, so I save screen space, and there are no duplicates. If I want one of the floating toolbars, I can still activate them.

Pretty code print is an add-in and I mention about that and some other add-ins briefly in the appendices.

If you set this up in one application it will be the same in the other client applications as well because there's one VBA to bind them. Except Access.

And finally, to repeat, add a Visual Basic icon to the QAT so you don't have to go to the Developer tab all the time! Shove it over to the left most position so you know where it is.

I emphasise again, this is not obligatory. You do not have to do it! If you're happy with your set up then leave it alone!

UPDATE: RubberDuck.

Rubberduck is an open source COM add-in for the VBE in VBA. It's very definately worth checking out. Especially the code explorer! The first version was v1.0 (duh), and was published on Nov. 29th, 2014. The current version at the creation of this PDF is v2.2.0.3086 and was published on Apr. 9th 2018.

And.... Rubberduck will add an extra toolbar on a separate line reducing the available screen space. Apparently, this is by design. You can "hide" it. But at the moment, you have to do that every time you start VBA.

Update: MZ-Tools 8

Just want to mention there is a code explorer here as well.

Application specific code

There isn't any.

Weeeelllll I say that. I've done my best to make it so but of course I may have missed something or just plain made a mistake. **PLEEAASSE!** Let me know. You **WILL** get credit.

There's a reason for there not being any application specific code. We're not using the application. We're editing and so on, in the VBE.

There is **some** hard-coded stuff that refers to specific filenames for different applications in the section dealing with code for compiling. I'm working on that. The aim is to have different code doodads for/with the same extensions and so on so it's more generic.

Otherwise, this has been tested in Word Excel Powerpoint and Access. Honestly. I can show you the log files even!

Those are the big four. VBA is also native in Visio, Project, Outlook, FrontPage, Autocad, and even WordPerfect. I have no reason to suppose the VBA presented here will not work with those. This is because we are massaging code **in the VBE**, and not doing anything in the host application.

SharePoint and OneNote have missed out on VBA. Shame. OTOH, there's probably no need!

If you have issues then get back **ASAP**. We can even email each other and get into a dialog! Stuff will be altered appropriately to reflect any concerns with credit. There will also be a separate **ADDENDUM** file or some such, maybe just on the web site or even in an appendix here, that I will update with the latest and greatest as it comes in from every/anyone. The format is still up to debate but I think I'm going to go with a web page. Dunno. Opinions?

REMEMBER REMEMBER

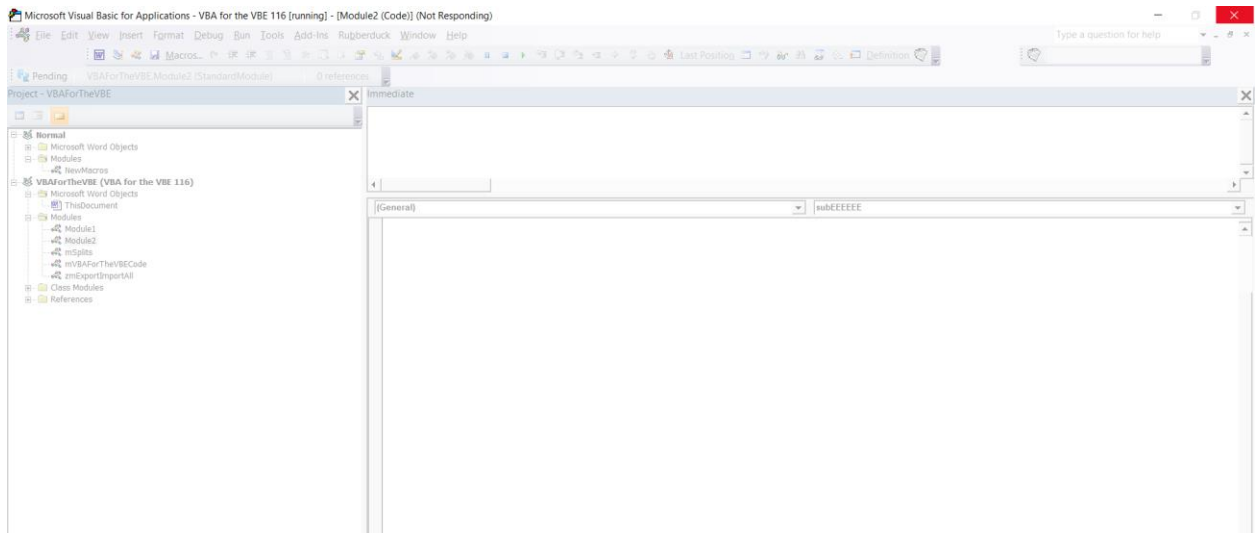
FIRST!

There are some big main, **MAJOR** tenets to using code to alter code in the VBE:

1. Don't do it to yourself!

VBA saves its code as "P-Code" as part of the VBA project you're in. There's a [link](#) in the appendices for an explanation, sort of, of p-code, though "it's complicated". If you screw up the pointers by adding a line to the code you're running, or worse, deleting lines, it throws its hands up, says I don't know where I am anymore, and everything goes grey. Then the **ONLY** recourse is to kill and restart the Application. In some cases you will add or delete a line and all will be good. This is because the pointers in the P-Code are still pointing to the same place. **Don't trust it!** If you don't know how P-Code works then there is no reason to think Oh, I can add lines below this, or I can delete lines above this. If you are certain that it's ok to do that then go ahead! Are you certain? Having said that, just reporting stuff like line numbers and so on doesn't alter anything. So running such against yourself in that case is fine.

2 Going grey



2. Save. A lot.

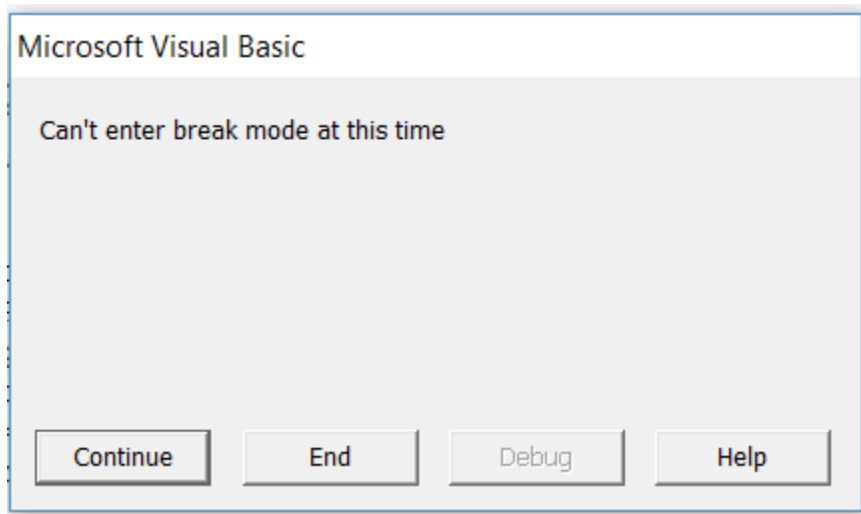
Apart from being good practice, there's also a good reason. Aaaaand it has a lot to do with the above. Number 2 **WILL** happen from time to time. When it does, you don't want to have to type in a bunch of code again to get back to the point where you got the problem.

3. Using Step/F8 is not always possible.

If you try to breakpoint while altering code, you'll often get the "Can't enter break mode at this time" message below, where the options are either Continue or End. While working with altering code in the VBE, get used to using Debug.Print to trace what's happening. You can stop the code sure, by pressing END. But you probably need runtime information. Debug.Print is your **only** option.

If you try to step through code that's bombed in this way you will get the same message wether you like it or not. Stepping through may give you the line before where this happens. This is a standard method in mainframe memory dumps. The dump will give you the memory address of the line before the problem. Then go back to the assembler and look at the next line. That's the one that's causing the problem. We're not dealing with mainframes or assembler here. But at least you get to know which line **might** be causing a problem.

3 Can't enter break mode



4. You need a reference to "Microsoft Visual Basic for Applications Extensibility 5.3".

You need this for **ANYTHING** you do with VBA in the VBE.

UPDATE: There are quite a few sites out there that say you do actually need this reference, but it turns out this isn't strictly true. You cannot use any of the VBIDE objects as VBIDE objects. But, you can set all of the objects as Object. Thank you, Doc.AElstein and HansV of Eileens Lounge.

The following three pieces of code are equivalent and will all give the same result.

The big difference between the first and second is that the second is much easier to read.

The big difference between the second and third is that you do not get any “intellisense” with the second and you do with the third.

There are some implications for early or late binding too but I don’t go into that.

My personal advice, FWIW, is to use a reference to the extensibility library.

1 Single line Without Extensibility lib Ref

MsgBox Application.VBE.ActiveVBProject.VBComponents(Application.VBE.ActiveCodePane.CodeModule.Name).Name

2 Multiple lines without Extensibility lib Ref using Dim... As Object

Dim vbplProject As Object
Dim vbclComponent As Object
Dim vbcmlCodeModule As Object
Dim vbcplCodePane As Object
Dim slModuleName As String
Set vbplProject = Application.VBE.ActiveVBProject
Set vbcplCodePane = Application.VBE.ActiveCodePane
Set vbcmlCodeModule = vbcplCodePane.CodeModule
slModuleName = vbcmlCodeModule.Name
Set vbclComponent = vbplProject.VBComponents(slModuleName)
MsgBox vbclComponent.Name

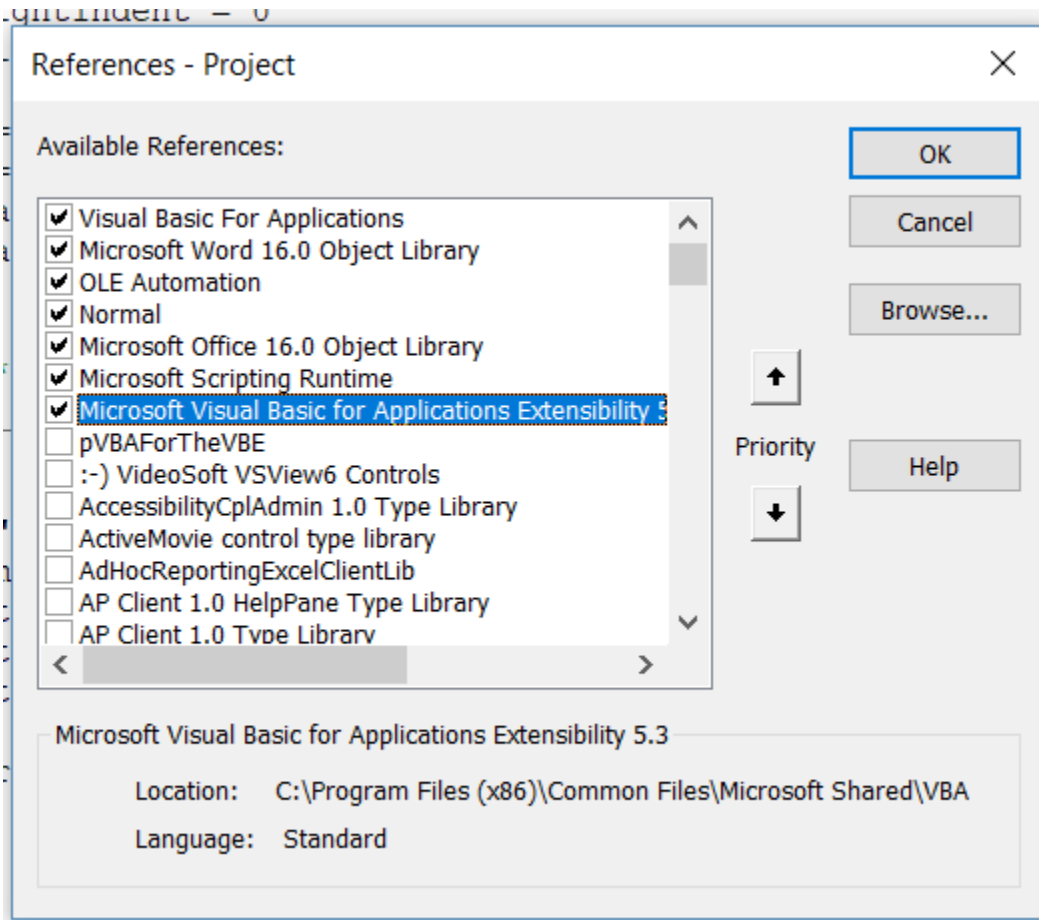
3 Multiple lines with Extensibility lib Ref using Dim... As VBIDE.item

Dim vbplProject As VBIDE.VBProject
Dim vbclComponent As VBIDE.VBComponent
Dim vbcmlCodeModule As VBIDE.CodeModule
Dim vbcplCodePane As VBIDE.CodePane
Dim slModuleName As String

	Set vbplProject = Application.VBE.ActiveVBProject
	Set vbcplCodePane = Application.VBE.ActiveCodePane
	Set vbcmICodeModule = vbcplCodePane.CodeModule
	sIModuleName = vbcmICodeModule.Name
	Set vbclComponent = vbplProject.VBComponents(sIModuleName)
	MsgBox vbclComponent.Name

Here is a pic of my references for this ms word document. I use the same basic refs for all applicastions.

4 My Document References



Project names

Let's talk a bit about VBA Project names.

Open the VBE in a new instance of an application. You'll see that the name of the VBA project is "Project". Well it is in Word anyway, which is what I'm writing this in. In Excel it's "VBAProject". Powerpoint is "VBAProject" as well.

Access is, well, Access. The paradigm of how you work there is different. Though I can't help feeling that the fact the developer teams are separate teams and team coordination has something to do with it. Ahhh well.

For Access, the project name is created from the default filename that Access gives a new database. This is taken from the files in the default save folder and the name will fill the numbered sequence database1, database2 and so on depending on the files already there. If there's already a database2 and no database1 then the file will be called database1 and so will the VBA project.

There is exactly **ONE** VBA project to a Document / WorkBook / DataBase / Presentation.

This is important!

Let's have a look:

- Open Word.
- You should be in Document 1 or the equivalent in your language flavor of Word.
- Go to the VBE.
- You'll see that there is a VBA project called "Project".
- Back to word and create a new Document.
- Back to the VBE.

You'll see **TWO** VBA projects, One for each document and **both** named "Project"

Don't worry... you haven't saved anything yet. These will all disappear when you close Word and don't choose to save.

Try it in Excel or Powerpoint:

- Open the application.
- Go to the VBE.
- You'll see a VBA Project for that document called "VBAProject".
- Back to the application
- Create a new document.
- Back to the VBE

There will be **TWO** VBA projects both named "VBAProject."

So. How do you reference a procedure you want to run that has the same name in say, Excel and Powerpoint but from their own application projects. And yes, I've picked those deliberately, because they have the same default VBA project name, hehehe.

If you haven't noticed, the project names are all appended by the filename in round brackets in the project explorer. Always.

###Outlook?

Together these two items make the name of any VBA project in any application unique.

However! It's possible to, and I **strongly** recommend that you do, change the name of all of your VBA projects to "something meaningful" as soon as you can. **Right after you've created it would be good!**

It's the same process once you're in the VBE in any application.

- Open the VBE
- Select the project in the project explorer
- Press F4
- Change the name

In fact, the same is applicable to module names. **Create 'em, Change 'em.** Trust me. You'll be happy you did it. Instead of Module1 Module2 etc, mTest, mGoodCode, mLoveClarice, mVBAForTheVBE and so on.

You'll notice I prefix module names with a lower-case m. I'd just like to mention here, and you will see me mention it again, that just simply using a naming convention **GREATLY** enhances the VBE and using VBA, and is **VERY** highly recommended by people much better and more experienced than me. I outline the one I use personally in the [appendices](#). I mention this a lot.

Because you can only ever have one copy of outlook open locally on your computer, there can be only one outlook project. This is always called Project1.

Update: As mentioned above, Both MZ-Tools 8 and Rubberduck have implemented a procedure/project explorer. This makes navigating to a particular module and procedure a lot easier.

Some module anatomy

4 Module

1		Option Explicit	First declaration line	.CountOfDeclarationLines = 4
2				.CountOfLines = 20
3		Declaration 1		
4		Declaration 2	Last declaration line	
5	1		First line of subName1	.ProcStartLine("subName1", vbext_pk_Proc) = 5
6	2			.ProcCountLines("subName1", vbext_pk_Proc) = 7
7	3			
8	4			
9	5	Sub subName1		.ProcBodyLine("subName1", vbext_pk_Proc) = 9
10	6			
11	7	End Sub	Last line of subName1	
12	1		First Line of subName2	.ProcStartLine("subName2", vbext_pk_Proc) = 12
13	2			.ProcCountLines("subName2", vbext_pk_Proc) = 9
14	3			
15	4	Sub subName2		.ProcBodyLine("subName2", vbext_pk_Proc) = 15
16	5			
17	6			
18	7			
19	8	End Sub		
20	9		Last line of subName2	

The above shows that if the procedure has blank or comment lines above the declaration, the line count for the procedure, starts at the first of those lines after the last declaration line or the first line after the End of the previous procedure. Being the last procedure in a module causes a few problems as we will see later.

Code to SORT DIMS

I'm not that daft. I know that a big reason you're here is code!

Sooooo! Here's some code.

A bubble sort is used here. A bubble sort is one of the slowest sorts there is. However, it turns out from studies, that the slow sort algorithms are actually pretty good if our data is already reasonably well-ordered, and that it doesn't really matter at all which sort we use if the data set is small anyway. I'm **NOT** going into the relative merits of different sorts. A very brief description of the bubble sort process is in the appendices.

We're choosing this procedure and sort method for a number of reasons.

- It's relatively small
- It's relatively self-contained
- The number of Dims in your proc is not going to be in the hundreds.
- You can **see** it working. Cool!
- It can actually be quite useful
- We like it and it's simple.
- Remember those doors I mentioned in the introduction? This will lead to other procedures **some** of which are...
 - Sorting a selection of Dims instead of all of them
 - Formatting Dims to one a line
 - Moving Dims to the top of a procedure
 - Creating a Class from a Sub
 - Adding code to do debug print of a selected variable
 - Using a UserForm in the VBE while staying in the VBE
 - Clearing the immediate window
 - Adding a toolbar with buttons to run code to the VBE
 - Using the standard VBE menu items in code
 - Setting up a crude procedure "trace"
 - Writing to and reading from the Registry
 - Adding module line numbers to code

So! Lots to look forward to!

This routine works directly on the code. It does not, as is possible, read the all of the lines to sort into an array, sort the array, and replace the originals in one go. Though there is code later that does that for other things later.

This is the original code written quite some years ago. You'll see that an Object is used for the code pane rather than VBIDE.CodePane. Long type local variables also have the prefix ll.

5 subccSortDims

	Option Explicit
	Sub subccSortDims()
	' BATCH.
	'
	' Sort dims in a procedure.
	' Assume dims are together.
	' Assume one to a line.
	' Assume <Definition> Name
	' Bubble sort - This is a slow sort method
	' just so we can see it work :-)
	'
	Dim sIProcName As String
	Dim sLine2 As String
	Dim sLine1 As String
	Dim sIA2() As String
	Dim sIA1() As String
	Dim olPane As Object
	Dim llCompLine2 As Long
	Dim llCompLine1 As Long
	Dim slOLine1 As String
	Dim slOLine2 As String
	Dim slDef1() As String
	Dim ilLenDef1 As Integer
	Dim slDef2() As String
	Dim ilLenDef2 As Integer
	Dim llStartLine As Long
	Dim llSRow As Long
	Dim llSCol As Long
	Dim llLine1 As Long
	Dim llERow As Long
	Dim llECol As Long
	Dim llCountLines As Long
	Dim ilSanityCheck As Integer
	Dim llEndLine As Long

	Set olPane = Application.VBE.ActiveCodePane
	olPane.GetSelection IISRow, IISCol, IISRow, IISCol
	slProcName = olPane.CodeModule.ProcOfLine(IISRow, vbext_pk_Proc)
	llLine1 = olPane.CodeModule.ProcBodyLine(slProcName, vbext_pk_Proc)
	llCountLines = olPane.CodeModule.ProcCountLines(slProcName, vbext_pk_Proc)
	llStartLine = olPane.CodeModule.ProcStartLine(slProcName, vbext_pk_Proc)
	llEndLine = llStartLine + llCountLines - 1
	' Find Dim Line.
	llCompLine1 = llLine1
	Do
	slOLine1 = Trim\$(olPane.CodeModule.Lines(llCompLine1, 1))
	If Left\$(slOLine1, 4) = "Dim " Then
	Exit Do
	Elseif Left\$(slOLine1, 6) = "Const " Then
	Exit Do
	Elseif Left\$(slOLine1, 7) = "Static " Then
	Exit Do
	End If
	llCompLine1 = llCompLine1 + 1
	ilSanityCheck = ilSanityCheck + 1
	If llCompLine1 >= llEndLine Then
	' End of module.
	subMsgBox "@End of module."
	Exit Sub
	End If
	If ilSanityCheck > 200 Then
	' Assume no Dims.
	subMsgBox "@No Dims."
	Exit Sub

	End If
	Loop
	llLine1 = llCompLine1
	Do
	' Move to next dim line
	sLOLine1 = Trim\$(olPane.CodeModule.Lines(llCompLine1, 1))
	sLLine1 = UCase\$(sLOLine1)
	sLDef1 = Split(sLLine1, " ")
	ilLenDef1 = Len(sLDef1(0))
	Select Case sLDef1(0)
	Case "DIM", "PUBLIC", "CONST", "STATIC"
	olPane.SetSelection llCompLine1, 1, llCompLine1, 1
	llCompLine2 = llCompLine1 + 1
	If llCompLine2 > llStartLine + llCountLines - 1 Then
	Exit Do
	End If
	sLOLine2 = Trim\$(olPane.CodeModule.Lines(llCompLine2, 1))
	sLLine2 = UCase\$(sLOLine2)
	sLDef2 = Split(sLLine2, " ")
	If UBound(sLDef2) < 0 Then
	Exit Do
	End If
	ilLenDef2 = Len(sLDef2(0))
	Select Case sLDef2(0)
	Case "DIM", "CONST", "STATIC"
	Case Else
	Exit Do
	End Select
	' Strip to variable names
	sLA1 = Split(Mid\$(sLLine1, ilLenDef1 + 2))

```

slA2 = Split(Mid$(slLine2, ilLenDef2 + 2))

' Compare
If slA2(0) < slA1(0) Then

' Swap
olPane.CodeModule.ReplaceLine llCompLine1, slOLine2
olPane.CodeModule.ReplaceLine llCompLine2, slOLine1

' Reset the indexes to 1 and 2 to start again
llCompLine1 = llLine1

Else

' Increment
llCompLine1 = llCompLine1 + 1
llCompLine2 = llCompLine2 + 1

End If

Case Else
llCompLine1 = llCompLine1 + 1
End Select

If llCompLine1 > llStartLine + llCountLines - 1 Then
Exit Do
End If

Loop

Set olPane = Nothing
'
*****
End Sub

```

To try running it:

- Create a new module in the VBE of your application of choice, Excel, Word, etc
There's a reason for putting it in its own module. I'll get to that in a bit and do a walkthrough as well. Even better would be a different project if you know how to set references to it. I'll talk about that along with creating an add-in later.
- Copy the code and paste it into the new module.
- Be careful not to include two Option Explicit statements because you have set new modules to include Option Explicit, haven't you?
- **SAVE THE WORKBOOK!**
- Put your cursor inside the code you have just created.
- Open the Macros menu.
 - Tools
 - Macros...
- Run the procedure you've just created: subccSortdIMS.
 - You may want to try stepping through instead of running it. You can use the step icon or F8 to do that.

Okay, Here's the kicker.

It won't work.

Gotchya!

You'll need to Kill the application and re start it.

Aren't you glad though, that you saved the workbook and don't have to rebuild it again!

There's a Sub being called that you haven't got. subMsgBox.

Aside: Try using **Debug/Compile** to pick up stuff like this. it's very very fast, and you don't need to actually run any code. Be aware though, that it compiles the whole project and not just the module or procedure you are in. We'll do more with compiling later, a lot more.

Anyroad, get around this by either:

- Commenting the call out or deleting it. There are two of them.
- Changing subMsgBox to MsgBox.
- Adding a simple procedure called subMsgBox (Er, recommended, because later we'll add a pretty complex subMsgBox anyway.)

Ala:

6 subMsgBox

	Sub subMsgBox(_
	spMsg As String _
)
	MsgBox spMsg
	' *****
	End Sub

After correcting for subMsgBox try again.

It still won't work.

HAH!

Remember Remember!

Tenet 1!

Don't do it to yourself!

Okay. I've cheated a bit just to show ya what happens, coz ya need to know this stuff and recognize it!

Now!:

- Add a **new** module
- Add a sub... any sub
- Copy just the Dims from the sortccdims proc to it
- Leave the cursor in the sub
- Open the macros menu
- Run sortccdims

Neat huh!

So that there is no confusion in thje next bit, delete that new module.

I've done this in Excel, Word, Access and Powerpoint so far, using **exactly** the same code. Remember that we're adjusting Code in the VBE and not doing anything in the host application.

If it doesn't work for you, **TELL ME!**

subccSortDims Walkthrough

Okay. I said I'd do a walkthrough.

Here is the subccSortDims code again with line numbers and breakdown. Note the line numbers. All the functions we use are going to use the **MODULE** line numbers because that's what the VBE references. Also note that the last line of the module is a blank one. You can't get rid of this and it is counted as being with the last procedure. Sometimes there's more than one. Try it.

- Add a new module
- Add a sub
- Press CtrlEnd

Your cursor will be one line below the End Sub.

ALSO note... they start at **ONE** not **ZERO**!

7 subccSortDims walkthrough

Module Line	Procedure Line	
1		Option Explicit
2	1	
3	2	
4	3	
5	4	
See Some module anatomy above.		
6	5	Sub subccSortDims()
Some comments about the proc... BATCH = Not interactive, Assumptions and so on. See Appendix C My Personal Naming Convention and procedure layout		
7	6	' BATCH.
8	7	'
9	8	' Sort dims in a procedure.
10	9	' Assume dims are together.
11	10	' Assume one to a line.
12	11	' Assume <Definition> Name

13	12	' Bubble sort - This is a slow sort method
14	13	' just so we can see it work :-)
15	14	'
16	15	
17	16	Dim slProcName As String
18	17	Dim slLine2 As String
19	18	Dim slLine1 As String
20	19	Dim slA2() As String
21	20	Dim slA1() As String

When Lisa first wrote this, she used "Object". We're more specific now and use VBIDE.CodePane.

22	21	Dim olPane As Object
----	----	----------------------

We now use lng for Local Long instead of ll.

23	22	Dim llCompLine2 As Long
24	23	Dim llCompLine1 As Long
25	24	Dim slOLine1 As String
26	25	Dim slOLine2 As String
27	26	Dim slDef1() As String

We use Long almost exclusively now rather than integer.

28	27	Dim ilLenDef1 As Integer
29	28	Dim slDef2() As String
30	29	Dim ilLenDef2 As Integer
31	30	Dim llStartLine As Long
32	31	Dim llSRow As Long
33	32	Dim llSCol As Long
34	33	Dim llLine1 As Long
35	34	Dim llERow As Long
36	35	Dim llECol As Long
37	36	Dim llCountLines As Long
38	37	Dim ilSanityCheck As Integer
39	38	Dim llEndLine As Long
40	39	

Set up a codepane object for where the cursor is in the sub. For this example, where you placed it. Setting a variable makes the code more readable and a mite shorter instead of using Application.VBE.ActiveCodePane all the time.

41	40	Set olPane = Application.VBE.ActiveCodePane
----	----	---

This will get the selection. If nothing is selected it will return the cursor position in the code. Ie: Where you put it. More on this later but for the moment we just need any module line number of where the cursor is to

get the procedure name. This could be the start or end line being returned from the GetSelection call. That is, unless your selection spans more than one procedure! You're usually safe if you use the selection start line number.

Returned are... IISRow = Module Start Row/line number, IISCol = StartCol, IERow = Module End Row/line number, IIECol = End Col. Remember that at this point, in this example, we are in THIS sub somewhere. Wherever you've placed the cursor.

42	41	oIPane.GetSelection IISRow, IISCol, IERow, IIECol
43	42	
Get the procedure name of the returned start line from GetSelection. vbext_pk_Proc, the Procedure kind, will be returned as well. This is the only function that you can use to get this.		
44	43	slProcName = oIPane.CodeModule.ProcOfLine(IISRow, vbext_pk_Proc)
Get the line on which the Declaration/Definition is for that proc name... Sub/Function/Property. In this case 6. vbext_pk_Proc returned from the previous call is needed.		
45	44	llLine1 = oIPane.CodeModule.ProcBodyLine(slProcName, vbext_pk_Proc)
Get the count of lines in this procedure. In this case 149.		
46	45	llCountLines = oIPane.CodeModule.ProcCountLines(slProcName, vbext_pk_Proc)
Get the start line of the procedure. In this case...2.		
47	46	llStartLine = oIPane.CodeModule.ProcStartLine(slProcName, vbext_pk_Proc)
Calculate the end line of the procedure. In this case 150.		
48	47	llEndLine = llStartLine + llCountLines - 1
49	48	
Loop around the procedure code grabbing each MODULE line starting at each sub definition line. Test each code line for Dim Const or Static. This gives us the START line number of the Dims for that sub. Check that we've not gone past the end of the procedure.		
50	49	' Find Dim Line.
51	50	llCompLine1 = llLine1
52	51	
53	52	Do
Grab the line.		
54	53	slOLine1 = Trim\$(oIPane.CodeModule.Lines(llCompLine1, 1))
55	54	
Test for Dim, Const, Static.		
56	55	If Left\$(slOLine1, 4) = "Dim " Then
57	56	Exit Do

58	57	Elseif Left\$(sIOLine1, 6) = "Const " Then
59	58	Exit Do
60	59	Elseif Left\$(sIOLine1, 7) = "Static " Then
61	60	Exit Do
62	61	End If
63	62	lICompLine1 = lICompLine1 + 1
64	63	ilSanityCheck = ilSanityCheck + 1
65	64	
66	65	If lICompLine1 >= lIEndLine Then
67	66	
68	67	' End of module.
69	68	subMsgBox "@End of module."
70	69	Exit Sub
71	70	
72	71	End If
73	72	
Just to be sure! We can be pretty certain we should hit a Dim by line 200. This came back to bite my bottom when trying to sort a proc with no Dims!		
74	73	If ilSanityCheck > 200 Then
75	74	
76	75	' Assume no Dims.
77	76	subMsgBox "@No Dims."
78	77	Exit Sub
79	78	
80	79	End If
81	80	
82	81	Loop
83	82	lILine1 = lICompLine1
84	83	
We have the start line of the Dims and are in fact "on" it. Start the sort. I've mentioned why I chose this sort of sort. Watching it work you can see it's not just, abracadabra now you see it, magic.		
85	84	Do
86	85	
87	86	' Move to next line
Get that line.		
88	87	sIOLine1 = Trim\$(oIPane.CodeModule.Lines(lICompLine1, 1))
89	88	sILine1 = UCase\$(sIOLine1)
90	89	

Split our new line up delimited by space. We use split here because we want the variable name. Above, using if..elseif.. endif, is very slightly faster.

91	90	slDef1 = Split(slLine1, " ")
92	91	ilLenDef1 = Len(slDef1(0))
93	92	

Test our new line to see if that's a Dim line as well.

94	93	Select Case slDef1(0)
95	94	Case "DIM", "PUBLIC", "CONST", "STATIC"
96	95	
97	96	olPane.SetSelection llCompLine1, 1, llCompLine1, 1
98	97	llCompLine2 = llCompLine1 + 1
99	98	If llCompLine2 > llStartLine + llCountLines - 1 Then
100	99	Exit Do
101	100	End If

Get the next line.

102	101	slOLine2 = Trim\$(olPane.CodeModule.Lines(llCompLine2, 1))
103	102	slLine2 = UCase\$(slOLine2)
104	103	
105	104	slDef2 = Split(slLine2, " ")
106	105	If UBound(slDef2) < 0 Then
107	106	Exit Do
108	107	End If
109	108	ilLenDef2 = Len(slDef2(0))
110	109	Select Case slDef2(0)
111	110	Case "DIM", "CONST", "STATIC"
112	111	Case Else
113	112	Exit Do
114	113	End Select
115	114	
116	115	' Strip to variable names
117	116	slA1 = Split(Mid\$(slLine1, ilLenDef1 + 2))
118	117	slA2 = Split(Mid\$(slLine2, ilLenDef2 + 2))
119	118	

Compare the two variable names from adjacent lines to see which is the greater. Remember... We've stripped the "Dim " and so on and are just comparing variable names.

120	119	' Compare
121	120	If slA2(0) < slA1(0) Then
122	121	

Swap the lines by replacing them both with the other.

123	122	' Swap
124	123	oIPane.CodeModule.ReplaceLine llCompLine1, slOLine2
125	124	oIPane.CodeModule.ReplaceLine llCompLine2, slOLine1
126	125	
127	126	' Reset the indexes to 1 and 2 to start again
128	127	llCompLine1 = llLine1
129	128	
130	129	Else
131	130	
132	131	' Increment
Increment both line counts.		
133	132	llCompLine1 = llCompLine1 + 1
134	133	llCompLine2 = llCompLine2 + 1
135	134	
136	135	End If
137	136	
138	137	Case Else
139	138	llCompLine1 = llCompLine1 + 1
140	139	End Select
141	140	
142	141	If llCompLine1 > llStartLine + llCountLines - 1 Then
143	142	Exit Do
144	143	End If
145	144	
146	145	Loop
147	146	
148	147	Set oIPane = Nothing
149	148	' *****
150	149	End Sub
151	150	

Functions used from the Extensibility library in subccSortDims

We will come across more of these functions as we go on. There aren't actually all that many. Anyway, I've expanded the calls here because they are meant as examples. You will also see I've used my "more up to date for me and my naming convention", variable prefix of lng for long instead of a single l. In the subccsortdims code, we used:

1. Application.VBE.ActiveCodePane.**GetSelection** _
 <ModStartLine>, <StartCol>, <ModEndLine>, <EndCol>

We called this to find the line number the cursor is on which is <ModStartLine>. We need this to use for the calls, SetSelection, ReplaceLine, and Lines.

2. slProcName = _
 Application.VBE.ActiveCodePane.CodeModule.**ProcOfLine**(_
 <ModLine>, vbext_pk_Proc _
)

For ProcBodyLine, ProcCountLines, and ProcStartLine, we need the procedure name. ProcOfLine will return it. For those functions, we also need to know the pk (procedure kind) or type. ProcOfLine returns that as well. In fact, it's the **only** procedure that will give us the proc type. Once we have it, we can plug it into the other calls. This makes where you put this call important. It has to be before the calls that need vbext_pk_Proc.

3. lngLine1 = _
 Application.VBE.ActiveCodePane.CodeModule.**ProcBodyLine**(_
 <ProcName>, vbext_pk_Proc _
)

This gives us the declaration line number for the procedure, the line the Sub/Function is on. We use vbext_pk_Proc here from .ProcOfLine.

4. lngCountLines = _
 Application.VBE.ActiveCodePane.CodeModule.**ProcCountLines**(_
 <ProcName>, vbext_pk_Proc _
)

We need the count of lines in the procedure so that we don't go too far. Again we use vbext_pk_Proc here from .ProcOfLine.

5. lngStartLine = _
 Application.VBE.ActiveCodePane.CodeModule.**ProcStartLine**(_
 <ProcName>, vbext_pk_Proc _
)

Gives us the real start line number of the procedure. And again we use vbext_pk_Proc here from .ProcOfLine.

6. sLines = _
Application.VBE.ActiveCodePane.CodeModule.Lines(_
<StartLine>, <NumberOfLines> _
)

Gets all the lines from startline for a number of lines. If there is more than one line, they are separated by vbCrLf.

7. Application.VBE.ActiveCodePane.SetSelection _
<ModStartLine>, <StartCol>, <ModEndLine>, <EndCol>

Sets the selection. This is the opposite of GetSelection. And it moves the cursor to the set selection as well. So, this is a way of going to a place in your code.

8. Application.VBE.ActiveCodePane.CodeModule.ReplaceLine _
<ModLine>, <String>

Replaces a single line with a String. We can use the line number/s from GetSelection here.

These are some of the most used calls you will use. We'll get to some others in due course. You can see that they are all "connected" with each other through the Procedure Name, the Procedure Type, and the Module line number.

A better understanding of what's going on is if you think of doing all of this manually. A sort of pseudocode if you will.

8 Bubble sort psuedocode

	Loop down the procedure till we get to a Dim
	Loop down line by line
	Get the next line
	If that's also a Dim
	If that's a dim as well
	Compare the variable names
	If the top line variable is less than the next line variable
	Swap the lines
	End if
	Endif
	end if
	End loop

9 Bubble sort Manual procedure

	Go to the top of the procedure
	Move the cursor down the procedure till we get to a Dim
	Start a loop
	Look at the next line
	If that's also a Dim
	Compare the variable names
	If the top line variable is less than the next line variable
	Swap the lines
	End if
	Move to the bottom line
	end if
	End the loop

Why Sort the dims

Big question this.

We could get a bit philosophical here, but the simple answer is that it makes variable definitions and groups of definitions easier to find when reading the code. Another is that it can highlight duplicates. And it's a bit prettier.

So! How do **YOU**, normally do Dims?

Scenario 1 - NOOOOOO DIMS

You may have gone minimalist and left out Option Explicit and have no dims at all.

Not much to say here except if you can live with yourself doing this go for it. Having said that I actually know of a poster to some forums who does this. He's pretty good too, but people have had problems with his code because of no Dims and no Option Explicit. There is code in **CodeCode** to check for variables in a procedure and if there isn't a Dim for it insert one. There is also code to go the other way and delete any Dims that aren't used in a procedure. They are both considerably large pieces of VBA code. The code is on thinkz1.com. It's free. Problems? [My email](#) is in the appendices.

Scenario 2 – Nicely grouped DIMS

You may have all of the Dims set out nicely in groups with separator lines and comments at the end.

```
' Procedure details -----
Dim s1ProcName As String      ' Procedure Name
Dim llStartLine As Long      ' Procedure Star of Code
Dim llSRow As Long           ' Start Module line of cursor
Dim llSCol As Long           ' Start Column for cursor/selection
Dim llERow As Long           ' End Module line for cursor
Dim llECol As Long           ' End Column for cursor/selection
Dim llCountLines As Long     ' Count of procedure lines
Dim llEndLine As Long        ' End Module line of procedure

' Work Variables for Sort -----
Dim llLine1 As Long          ' Module line number of line 1 being
                              looked at
Dim s1Line2 As String        ' Line 2
```

```

Dim slLine1 As String      ' Line 1
Dim slA2() As String      ' Array of line 2
Dim slA1() As String      ' Array of line 1
Dim llCompLine2 As Long   ' Modue line number 1 being compared
Dim llCompLine1 As Long   ' Module line number 2 being
compared
Dim slOLine1 As String    ' Original Line 1
Dim slOLine2 As String    ' Original Line 2
Dim slDef1() As String    ' Split of Line 1 on space
Dim ilLenDef1 As Integer  ' Length of 1st element of slDef1
Dim slDef2() As String    ' Split of Line 2 on space
Dim ilLenDef2 As Integer  ' Length of 1st element of slDef2

' Other -----
Dim olPane As Object      ' Active Code Pane

```

That would be difficult to sort indeed! Or would it?

The actual line comparison in subccSortDims picks out the variable names from the Dim/Const/Static and compares just those. When it comes to replacing a line in the bubble sort it replaces the **whole** line. This means that those comments at the end will still be there and with the correct variable!

Scenario 3 – All DIMS on a single line

You may have done the same as a lot of people do in **VBS**, and defined everything as variants in a single line.

```

Dim slProcName, slLine2, slLine1, slA2(), slA1(), olPane, llCompLine2, llCompLine1, slOLine1, slOLine2,
slDef1(), ilLenDef1, slDef2(), ilLenDef2, llStartLine, llRow, llCol, llLine1, llRow, llCol, llCountLines,
ilSanityCheck, llEndLine

```

You should see though, that the names of the variables here give at least a clue to what type of variable is expected even if they are all variants. Clever eh!

Scenario 4 – DIMS defined as they are used

You may have defined the Dim as you use them in the middle of the code possibly as a **reset**. Resetting in this way doesn't actually work because VBA defines all of the Dims up front anyway. Try running the code below... You'll get the number 3 printed in the IM even though the code does the Dim three times.

	Sub CountInsideLoop()	
	' https://excelmacromastery.com/vba-dim/	
	Dim i As Long	
	For i = 1 To 3	
	Dim count As Long	
	count = count + 1	
	Next i	
	' count value will be 3	
	Debug.Print count	
	End Sub	

After all that!

VBS seems to have changed the backdrop because it only uses variants. In VBS, writing Dim a,b,c,d seems to make more sense. However, from zillions of programmers writing gazillions of lines of VB/VBA code for eons, the preferred, best practice, and okay yes, **accepted** method, even though you may as an individual balk at that, is one definition per line defined and typed properly and all at the top of the code. The **big big big** reason for this is maintenance. You **REALLY** want to be able to read the code in a couple of years time. It may be **YOU** who has to change it! I have coined and like the expression:

Remember! You WILL forget!

Here's another very famous quote:

Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live. Code for readability.

John F. Woods in 1991

And an interesting link in the links appendix about [NASA Defensive programming](#).

Sort a selection of Dims

Let's suppose we have scenario **2** where we've laboriously commented each and every variable at the end of the line neatly, and set them up in sections and got everything to look pretty and gorgeous. I propose, it would still be useful to have each **SECTION** sorted. Depending on the variable names, it would put all the numbers together and all of the strings together and make finding variables within a section fairly easy. That's the theory anyway and where we're going with the next bit of code.

UPDATE: MZ-Tools 8 has a feature to sort selected lines.

We'll be using **GetSelection** a bit more here.

If you are into reading code at all, and I **strongly** suggest that you at least give it a go, you will realize the code below is **almost**, identical to subccSortDims.

This can be **literally seen** by copying the code to separate text files and using the wonderful and free [WinMerge](#) program.

10 subccSortSelectedDims

	Sub subccSortSelectedDims()
	' SELECTION.
	'
	' Sort SELECTED dims in a procedure.
	' Assume dims are together.
	' Assume one to a line.
	' Assume <Definition> Name
	' Bubble sort - This is a slow sort method
	' but used here to be the same as
	' SortDims. Using an array makes it
	' quicker though.
	'
	Dim sIVar2 As String
	Dim sIVar1 As String
	Dim sIProcName As String
	Dim sILine2 As String
	Dim sILine1 As String
	Dim sIA2() As String

Dim sIA1() As String
Dim olPane As Object
Dim llCompLine2 As Long
Dim llCompLine1 As Long
Dim intl As Integer
Dim slOLine1 As String
Dim slOLine2 As String
Dim slDef1() As String
Dim ilLenDef1 As Integer
Dim slDef2() As String
Dim ilLenDef2 As Integer
Dim llStartLine As Long
Dim llSRow As Long
Dim llSCol As Long
Dim llLine1 As Long
Dim llERow As Long
Dim llELine As Long
Dim llECol As Long
Dim llCountLines As Long
Dim ilSanityCheck As Integer
Dim slSelectionLines() As String
Dim slSelection As String
Dim vbcplCodePane As VBIDE.CodePane
Dim vbcmlCodeModule As VBIDE.CodeModule
Set olPane = Application.VBE.ActiveCodePane
olPane.GetSelection llSRow, llSCol, llERow, llECol
slProcName = olPane.CodeModule.ProcOfLine(llSRow, vbext_pk_Proc)
llLine1 = llSRow
llCountLines = llERow - llSRow
llStartLine = llSRow
llCompLine1 = llSRow
Do
' Move to next dim line
slOLine1 = Trim\$(olPane.CodeModule.Lines(llCompLine1, 1))
slLine1 = UCase\$(slOLine1)

	sDef1 = Split(sLine1, " ")
	ilLenDef1 = Len(sDef1(0))
	Select Case sDef1(0)
	Case "DIM", "PUBLIC", "CONST", "STATIC"
	oIPane.SetSelection llCompLine1, 1, llCompLine1, 1
	llCompLine2 = llCompLine1 + 1
	If llCompLine2 > llStartLine + llCountLines - 1 Then
	Exit Do
	End If
	sLOLine2 = Trim\$(oIPane.CodeModule.Lines(llCompLine2, 1))
	sLine2 = UCase\$(sLOLine2)
	sDef2 = Split(sLine2, " ")
	If UBound(sDef2) < 0 Then
	Exit Do
	End If
	ilLenDef2 = Len(sDef2(0))
	Select Case sDef2(0)
	Case "DIM", "CONST", "STATIC"
	Case Else
	Exit Do
	End Select
	' Strip to variable names
	sA1 = Split(Mid\$(sLine1, ilLenDef1 + 2))
	sA2 = Split(Mid\$(sLine2, ilLenDef2 + 2))
	' Compare
	If sA2(0) < sA1(0) Then
	' Swap
	oIPane.CodeModule.ReplaceLine llCompLine1, sLOLine2
	oIPane.CodeModule.ReplaceLine llCompLine2, sLOLine1
	' Reset the indexes to 1 and 2 to start again
	llCompLine1 = llLine1

	Else
	' Increment
	llCompLine1 = llCompLine1 + 1
	llCompLine2 = llCompLine2 + 1
	End If
	Case Else
	llCompLine1 = llCompLine1 + 1
	End Select
	If llCompLine1 > llStartLine + llCountLines - 1 Then
	Exit Do
	End If
	Loop
	Set olPane = Nothing
	' *****
	End Sub

We're **NOT** going to walk through this, just point out the differences. Below is a screen shot of the two pieces of code in winmerge. I know it looks funny but I've elongated the picture so you can, I hope, at least sort of read the code. Pun intended hehehe.

On the left is the original subccSortDims.

On the right is subccSort**Selected**Dims.

It's fairly easy to see I think, that there is a whole do..loop of code dedicated to finding the first Dim line on the left in subccSortDims that's totally missing on the right in subccSortSelectedDims.

The left shows the code getting the module start line and count of lines numbers using the VBE calls ProcStartLine, ProcCountLines.

Now the clever bit. On the right, those calls are missing. Not there!

This is because we have the start and end line numbers returned in the GetSelection call mentioned above!

11 GetSelection parameters

	oIPane.GetSelection IISRow, IISCol, IIERow, IIECol
	IISRow=Module start line number of selection
	IISCol= start column number of selection
	IIERow=Module end line number of selection
	IIECol=End column number of selection

5 WinMerge 1

```
C:\Users\think\Desktop\test\new 2.txt
oIPane.GetSelection IISRow, IISCol, IIERow, IIECol

s1ProcName = oIPane.CodeModule.ProcOfLine(IISRow, vbext_pk_Proc)
l1Line1 = oIPane.CodeModule.ProcBodyLine(s1ProcName, vbext_pk_Proc)
l1CountLines = oIPane.CodeModule.ProcCountLines(s1ProcName, vbext_pk_Proc)
l1StartLine = oIPane.CodeModule.ProcStartLine(s1ProcName, vbext_pk_Proc)
l1EndLine = l1StartLine + l1CountLines - 1

' Find Dim Line.
l1CompLine1 = l1Line1

Do
    s1OLine1 = Trim$(oIPane.CodeModule.Lines(l1CompLine1, 1))

    If Left$(s1OLine1, 4) = "Dim " Then
        Exit Do
    ElseIf Left$(s1OLine1, 6) = "Const " Then
        Exit Do
    ElseIf Left$(s1OLine1, 7) = "Static " Then
        Exit Do
    End If
    l1CompLine1 = l1CompLine1 + 1
    i1SanityCheck = i1SanityCheck + 1

    If l1CompLine1 >= l1EndLine Then
        ' End of module.
        subMsgBox "@End of module."
        Exit Sub
    End If

    If i1SanityCheck > 200 Then
        ' Assume no Dims.
        subMsgBox "@No Dims."
        Exit Sub
    End If
Loop
l1Line1 = l1CompLine1

Do
    ' Move to next dim line

C:\Users\think\Desktop\test\new 1.txt
oIPane.GetSelection IISRow, IISCol, IIERow, IIECol

s1ProcName = oIPane.CodeModule.ProcOfLine(IISRow, vbext_pk_Proc)
l1Line1 = IISRow
l1CountLines = IIERow - IISRow
l1StartLine = IISRow
l1CompLine1 = IISRow

Do
    ' Move to next dim line
```

Now we just calculate the count of lines and away we go!

12 Setting lines to sort

	IISRow
	IICountLines = IISRow - IISRow
	IISStartLine = IISRow
	IICompLine1 = IISRow

Here's a shot of both of the subs a bit further down.

6 WinMerge 2

```
Do
    * Move to next dim line
    s10Line1 = Trim$(olPane.CodeModule.Lines(11CompLine1, 1))
    s1Line1 = UCCase$(s10Line1)
    s1Def1 = Split(s1Line1, " ")
    i1LenDef1 = Len(s1Def1(0))
    Select Case s1Def1(0)
    Case "DIM", "PUBLIC", "CONST", "STATIC"
        olPane.SetSelection 11CompLine1, 1, 11CompLine1, 1
        i1CompLine2 = 11CompLine1 + 1
        If i1CompLine2 > i1StartLine + i1CountLines - 1 Then
            Exit Do
        End If
        s10Line2 = Trim$(olPane.CodeModule.Lines(11CompLine2, 1))
        s1Line2 = UCCase$(s10Line2)
        s1Def2 = Split(s1Line2, " ")
        If UBound(s1Def2) < 0 Then
            Exit Do
        End If
        i1LenDef2 = Len(s1Def2(0))
        Select Case s1Def2(0)
        Case "DIM", "CONST", "STATIC"
            Case Else
                Exit Do
            Exit Do
        End Select
        * Strip to variable names
        s1A1 = Split(Mid$(s1Line1, i1LenDef1 + 2))
        s1A2 = Split(Mid$(s1Line2, i1LenDef2 + 2))
        * Compare
        If s1A2(0) < s1A1(0) Then
            * Swap
            olPane.CodeModule.ReplaceLine 11CompLine1, s10Line2
            olPane.CodeModule.ReplaceLine 11CompLine2, s10Line1
            * Reset the indexes to 1 and 2 to start again
            11CompLine1 = 11Line1
        Else
            * Increment
            11CompLine1 = 11CompLine1 + 1
            11CompLine2 = 11CompLine2 + 1
        End If
    Case Else
        11CompLine1 = 11CompLine1 + 1
    End Select
    If 11CompLine1 > i1StartLine + i1CountLines - 1 Then
        Exit Do
    End If
Loop
Set olPane = Nothing
' *****
End Sub

11Line1 = 11CompLine1
Do
    * Move to next dim line
    s10Line1 = Trim$(olPane.CodeModule.Lines(11CompLine1, 1))
    s1Line1 = UCCase$(s10Line1)
    s1Def1 = Split(s1Line1, " ")
    i1LenDef1 = Len(s1Def1(0))
    Select Case s1Def1(0)
    Case "DIM", "PUBLIC", "CONST", "STATIC"
        olPane.SetSelection 11CompLine1, 1, 11CompLine1, 1
        i1CompLine2 = 11CompLine1 + 1
        If i1CompLine2 > i1StartLine + i1CountLines - 1 Then
            Exit Do
        End If
        s10Line2 = Trim$(olPane.CodeModule.Lines(11CompLine2, 1))
        s1Line2 = UCCase$(s10Line2)
        s1Def2 = Split(s1Line2, " ")
        If UBound(s1Def2) < 0 Then
            Exit Do
        End If
        i1LenDef2 = Len(s1Def2(0))
        Select Case s1Def2(0)
        Case "DIM", "CONST", "STATIC"
            Case Else
                Exit Do
            Exit Do
        End Select
        * Strip to variable names
        s1A1 = Split(Mid$(s1Line1, i1LenDef1 + 2))
        s1A2 = Split(Mid$(s1Line2, i1LenDef2 + 2))
        * Compare
        If s1A2(0) < s1A1(0) Then
            * Swap
            olPane.CodeModule.ReplaceLine 11CompLine1, s10Line2
            olPane.CodeModule.ReplaceLine 11CompLine2, s10Line1
            * Reset the indexes to 1 and 2 to start again
            11CompLine1 = 11Line1
        Else
            * Increment
            11CompLine1 = 11CompLine1 + 1
            11CompLine2 = 11CompLine2 + 1
        End If
    Case Else
        11CompLine1 = 11CompLine1 + 1
    End Select
    If 11CompLine1 > i1StartLine + i1CountLines - 1 Then
        Exit Do
    End If
Loop
Set olPane = Nothing
' *****
End Sub
```

But I can't read that! I hear you scream! It's too small! Not the point. The point is that both left and right windows are white, meaning that the rest of the code, the actual sort code, is **IDENTICAL!**

Splitting up Dims from a single line

Let's move on. We've looked at scenario 2. Let's look at scenario 3. And guess which scenario we'll look at next. I know I've skipped scenario 1. There's a reason for that, it's complicated. But we will get to that, or at least [discuss it](#).

We may or may not have a problem with:

13 All Dims on a single line

Dim sIProcName, sLine2, sLine1, sA2(), sA1(), olPane, lICompLine2, lICompLine1, sIOLine1, sIOLine2, sIDef1(), ilLenDef1, sIDef2(), ilLenDef2, lIStartLine, lISRow, lISCol, lILine1, lIERow, lIECol, lICountLines, ilSanityCheck, lIEndLine
--

We're **NOT** going to even entertain sorting the items on a line and replacing it. But, reading the line in, zapping the Dim, removing spaces, splitting it on ",", sorting the trimmed array and doing a Join should work if you really want to.

If we want to sort the Dims properly though they really should all be on different lines by themselves.

14 One Dim per line

Dim sIProcName As String
Dim sLine2 As String
Dim sLine1 As String
Dim sA2() As String
Dim sA1() As String
Dim olPane As Object
Dim lICompLine2 As Long
Dim lICompLine1 As Long
Dim sIOLine1 As String
Dim sIOLine2 As String
Dim sIDef1() As String
Dim ilLenDef1 As Integer
Dim sIDef2() As String
Dim ilLenDef2 As Integer
Dim lIStartLine As Long
Dim lISRow As Long
Dim lISCol As Long
Dim lILine1 As Long

	Dim lERow As Long
	Dim lECol As Long
	Dim lCountLines As Long
	Dim ilSanityCheck As Integer
	Dim lEndLine As Long

The problem of course, is assigning a **TYPE** to a variable. Given the original line, all of the variables would be assigned as variant.

Everybody eschews "variant", and with good reason. The best/worst is that you don't know, without testing, what type of variable the variable is! How many times have I heard "***k! It's an object!" being expressed, er, loudly, and in er, surprise! Hmmmm.

Anyroad, remember that naming thingy? Oh yes, umm, convention. Lower case s = string, lower case i = integer, lowercase l = long, lower case lng = long, and I've not mentioned it before but lower case o = object.

We can make use of that!

The only other alternative is to let everything default to type variant.

Strangely enough, a lot of the "accepted" and normally used naming conventions on forums, begin similarly. s For string, o for object and so on 😊. There are some [links in the appendices](#), but I would actually advocate making up your **own** Naming Convention, for the simple reason that it's easier to stick to, rather than remember someone else's! Use someone else's by all means. **TOTALLY** free to use mine! Hehehe. But I'll bet you will, and would expect you to, adjust it to suit yourself. All I can say to that is, **GOOD ON YER!**

Let's go with mine for now though...s=string, i=integer, l=long, lng=long, o=object.

And oh, **PLEASE PLEASE PLEASE, NEVER use single letters for variables!** Good for Q&D and demos and examples and too often doesn't get changed in the final code. What the, er, hell, does i=j+1 mean? We can make a damn good guess, after reading the code, quite often someone else's, but in too many cases, You. Will. Never. Ever. Ever. Really. **KNOW**. Actually, in the spirit of backwards compatibility, the + sign, is still a concatenation in VBA, so it could be adding strings together! I'll Bet a lot of you didn't realise that! Some habits are good. My advice FWIW... Develop a habit!

Generally, given some sort of naming scheme, we need to:

- Pick up the line.
- Split it up
- Look at the start letter/s of each variable
- Assign a type reflective of the start letters if relevant
- Write a single Dim line for that variable

- Loopy doopy

Most of this is just string manipulation. But we're gonna do it anyway. Here is the code to do that. Because it's a function we need a sub to test it. That's at the top. Remember [my naming setup?](#)

15 fncGuessVarType

Sub subtestfncGuessVarType()
MsgBox fncGuessVarType("IIModuleEndLine")
' *****
End Sub
Function fncGuessVarType(_
spName As String _
) _
As String
' This is a function so that different "naming conventions" can
' be used.
'
' It has nothing to do with the VBE and is simple text/string manipulation.
'
' It's here so that you can code and build on your own convention.
'
' The convention used here is <prefix><scope><Name>
' Where:
' <prefix> is lower case and represents the variable type.
' <scope> is lower case and a single letter.
' <Name> is the name of the actual variable name and begins with a capitol letter.
'
' ### Notes:
' This is very specific to this book and my "real" proc encompasses many
' more types that are read from an INI file. This ia a VERY limited
' function as it looks for prefixes hard coded here.
'
' This assumes that spName is in the above format.
'
Dim sItemvalue As String
Dim sItemName As String
Dim lngIM As Long

Dim slLookForStub As String
Dim slType As String
Dim slPrefixStub As String
Dim slPreAndSuffixes() As String
Dim slPrefix As String
Dim ilN As Integer
Dim ilChr As Integer
Dim slChr As String
Dim slName As String
slName = spName
slPrefix = vbNullString
slPrefixStub = vbNullString
slType = vbNullString
' You can get and fill this array from anywhere.
' I'm using an INI file in my "real" stuff but it could
' equally be a text file, database, or the registry.
ReDim slPreAndSuffixes(1 To 5, 1 To 2)
slPreAndSuffixes(1, 1) = "Long"
slPreAndSuffixes(1, 2) = "l"
slPreAndSuffixes(2, 1) = "Integer"
slPreAndSuffixes(2, 2) = "i"
slPreAndSuffixes(3, 1) = "Long"
slPreAndSuffixes(3, 2) = "Ing"
slPreAndSuffixes(4, 1) = "Object"
slPreAndSuffixes(4, 2) = "o"
slPreAndSuffixes(5, 1) = "String"
slPreAndSuffixes(5, 2) = "s"
' Find the first capitol letter.
' A=65 Z=90 [\]^_` a=97 z=122
For ilN = 1 To Len(slName)

	sChr = Mid\$(sName, iLN, 1)
	iChr = Asc(sChr)
	If iChr < 91 _
	And iChr > 64 Then
	Exit For
	End If
	If iChr < 123 _
	And iChr > 90 Then
	sPrefix = sPrefix & sChr
	End If
	Next iLN
	If sPrefix <> vbNullString Then
	' Chop the scope letter off.
	sPrefixStub = Left\$(sPrefix, Len(sPrefix) - 1)
	sLookForStub = sPrefixStub
	For lngIM = 1 To UBound(sPreAndSuffixes, 1)
	sItemName = sPreAndSuffixes(lngIM, 1)
	sItemvalue = sPreAndSuffixes(lngIM, 2)
	If InStr(sItemvalue, sLookForStub) > 0 Then
	' Found it. Get out.
	sType = sItemName
	Exit For
	End If
	Next lngIM
	End If

	' Have we found something but it's not valid?
	Select Case sIType
	Case ""
	' Catch all.
	sIType = "Variant"
	End Select
	fncGuessVarTypeG = sIType
	' *****
	End Function

Couple of things here.

You'll see that the types are hard coded into an array. You can load that array from anywhere. Typically, this will be a text file, an INI file or the registry. As an example, at the moment I load these from an INI file, and FYI here is my current INI file entry for that.

16 INI File entries for my pre/suffixes

	[PreAndSuffixes]
	DS_PreAndSuffixes="Prefixes/suffixes used to denote variable types."
	D_Exceptions="These prefixes are used by the 'system' for various constants and are exempted from insertion."
	Exceptions=vb,vbext,xl,wd,cc,mso,tvw, fm, _fm, tli
	Long=lng,l,ln
	single=si,sng
	Integer=i,int,n
	String=s,str
	Boolean=bln,b
	Object=o,obj
	Double=d,dbl,db
	Variant=v
	Node=nd
	Range=r,rng,range
	WorkBook=wb
	WorkSheet=ws

	Table=tbl
	UserForm=frm
	DataBase=db
	Document=doc
	Recordset=rs
	Dictionary=dic
	Procedure=sub,fnc,subtest,A_subcc,subcc
	VBIDE.VBProject=vbp
	VBIDE.CodePane=vbcp
	VBIDE.CodeModule=vbcm
	VBIDE.vbComponent=vbc
	CapitalizeFirstLetter=Yes ;Yes/No
	[ScopeLetters]
	Local=l
	Module=m
	Global=g
	Public=g
	Project=g
	Parameter=p

What's in a name

Corny title yeah, but relevant, because that's what the last procedure was all about.

And yes, I know We're repeating things, and you know what? You'll see me do it again, and again, and maybe again!

And again 😊, you can refer to [my personal naming convention](#) in the appendices. You may already have noticed by looking at the list of Dims in subccSortDims that my string variables begin with a lowercase s, integers begin with a lowercase i, and long integers, long, begin with a lowercase l. In fact, I wrote some of the code here a while ago and now always use lower case lng for Long. The very sharp eyed, will note that there's a lowercase l after that and before the first capital letter of the actual variable name. In my setup this is to denote where the variable comes from. A sort of scope. A lower-case l next to the variable is l = local. I use p for a parameter, m for a private module level variable and g for project level global or public variables.

Incidentally:

17 Declaration as Global

Global str as string

... Is a valid definition in the declarations. The variable will be Public.

Stick with what you choose. After a while it becomes automatic.

Having said that, naming conventions can sometimes get a bit cumbersome. For example, you may want the prefix set up so you know that this is a long and a handle and is a variable local to the procedure. As I say in the appendix though, it can get a bit silly.

For example,

lngtpmID.

The prefix denotes a Long in a Type at Module level. It's longer than the variable name!

Sooooo, just be careful and have a think.

I can't emphasise enough, that ANY sort of naming convention, gives a MASSIVE advantage over not using one.

Recap number one!

So, what have we done/learned:

- Need a reference to "Microsoft Visual Basic for Applications Extensibility 5.3"
- Created code to update code
- Used a number of functions contained in the MS VBA Extensibility Library
- DON'T do it to ourselves
- Can use the Compile option in the debug menu
- F8 is tricky sometimes
- Everything uses **MODULE** line numbers
- There is a sort of order to using the VBE functions to feed the next one
- A sub does not necessarily start at the declaration line but the code does.
- The last line/s in a module are empty and are attributed to the last procedure.
- Line numbers of modules start at **ONE** not **ZERO**.
- **Save a lot**
- vbext_pk_Proc is required in functions and is returned by .ProcOfLine.
- It's useful to use a naming convention of some sort.

Yes, I know We're repeating some things, and you know what? You'll see me do it again, and again, and maybe again!

Where Are We

We'll get to use `fncGuessVarType` in a bit. But First! Let's write another sub that **does** have something to do with the VBE.

When writing code to alter code, you'll need to know where you are in the project and module you are currently in. Let's write a sub to tell you that. We'll call it `WhereArweWe`. You'll use this or its alter ego, more on that later, a **lot!**

The following code returns some key items:

- Project Object
- Component Object
- Codepane object
- Project name
- Module name
- Procedure name.
- Module line where the cursor is.
- Module Start Line of Procedure code.
- Number of Lines of code.
- Module End Line of Procedure code.

This will expand and change as you do more in the VBE and need more information about a procedure. There is one calculated value there, Module End Line of Procedure. This is in the procedure we'll write rather than calculated in the calling code because you don't want to have to calculate it multiple times and simply because, it's simpler.

We're going to implement `WhereAreWe` in TWO ways. As a **Sub** and as a **Class**.

Here's the sub code. Again, there is a sub to test the sub at the top.

You'll see that I've put a "Stop" at the end of the test sub. This is so you can examine and verify that `subWhereAreWe` is returning what it should by using the immediate window. As stated in the comments in the proc, the codepane is where most of the work is done. Setting an object to it is a good idea to reduce typing. However, it may be that you never use some of the objects. But they're there if you ever want to.

Aside: I use Stop quite a bit. It's handy because if you set a breakpoint and the process loops or something and you have to kill it, the stop is still there. There's no need to reset the breakpoint. Using Find and Replace it's also pretty simple to remove them. It's also the only way to put a break into the auto run procedure. You can put a breakpoint there and run it separately, but that doesn't reproduce the act of opening a document.

18 subWhereAreWe

Sub subtestsubWhereAreWe()
Dim vbplProject As VBIDE.VBProject
Dim vbclComponent As VBIDE.VBComponent
Dim vbcmlCodeModule As VBIDE.CodeModule
Dim vbcplCodePane As VBIDE.CodePane
Dim slProjectNameName As String
Dim slModuleName As String
Dim slProcedureName As String
Dim lngCurrentModuleLine As Long
Dim lngModuleProcStartLine As Long
Dim lngProcCountLines As Long
Dim lngModuleProcEndLine As Long
subWhereAreWe _
vbplProject, _
vbclComponent, _
vbcmlCodeModule, _
vbcplCodePane, _
slProjectNameName, _
slModuleName, _
slProcedureName, _
lngCurrentModuleLine, _
lngModuleProcStartLine, _
lngProcCountLines, _
lngModuleProcEndLine
Stop
' *****
End Sub
Sub subWhereAreWe(_
vbppProject As VBIDE.VBProject, _
vbcpComponent As VBIDE.VBComponent, _
vbcmpCodeModule As VBIDE.CodeModule, _
vbcppCodePane As VBIDE.CodePane, _
spProjectNameName As String, _
spModuleName As String, _

	spProcedureName As String, _
	IngpCurrentModuleLine As Long, _
	IngpModuleProcStartLine As Long, _
	IngpProcCountLines As Long, _
	IngpModuleProcEndLine As Long _
)
	' Returns:
	' Project Object
	' Component Object
	' Code Module Oject
	' Codepane object
	' Project name
	' Module name
	' Procedure name.
	' Module line where the cursor is.
	' Module Start Line of Procedure code.
	' Number of Lines of code.
	' Module End Line of Procedure code.
	'
	' The CodePane is where all of the real work is done.
	'
	Dim InglSLine As Long
	Dim InglSCol As Long
	Dim InglELine As Long
	Dim InglECol As Long
	Set vbppProject = Application.VBE.ActiveVBProject
	spProjectNameName = vbppProject.Name
	Set vbcppCodePane = Application.VBE.ActiveCodePane
	Set vbcmpCodeModule = vbcppCodePane.CodeModule
	spModuleName = vbcmpCodeModule.Name
	spModuleName = vbcppCodePane.CodeModule.Parent.Name
	Set vbcpComponent = vbppProject.VBComponents(spModuleName)
	vbcppCodePane.GetSelection IngpCurrentModuleLine, InglSCol, InglELine, InglECol

	spProcedureName = vbcppCodePane.CodeModule.ProcOfLine(IngpCurrentModuleLine, vbext_pk_Proc)
	IngpModuleProcStartLine = vbcmpCodeModule.ProcBodyLine(spProcedureName, vbext_pk_Proc)
	IngpProcCountLines = vbcppCodePane.CodeModule.ProcCountLines(spProcedureName, vbext_pk_Proc)
	IngpCurrentModuleLine = IngpCurrentModuleLine
	IngpModuleProcEndLine = IngpModuleProcStartLine + IngpProcCountLines - 1
	' *****
	End Sub

It may look like we've used a few more calls. But we haven't. What we've done here is set objects to the various parts of a VBA project. Previously we used the whole expanded call. Here we've set objects and used the methods and properties of that object.

19 Setting variables to VBE objects

Set an object to the Active project	
	Set vbppProject = Application.VBE.ActiveVBProject
Set an object to the active code pane.	
	Set vbcppCodePane = Application.VBE.ActiveCodePane
Set an object to the current code module .	
	Set vbcmpCodeModule = vbcppCodePane.CodeModule
Set an object to the current component.	
	Set vbcpComponent = vbppProject.VBComponents(spModuleName)

Aaaaand you'll see that we update the parameters directly by allowing the default **ByRef** instead of specifying **ByVal**. Important point!

You will have seen by now that VBE code doesn't seem to like procedures very much. The calls we've used so far, all use **MODULE** line numbers. So, if we want to find if a procedure exists or look at the code in a procedure how do we do it? More on that later.

Procedures to Classes

Oh Sorry. We were talking about subWhereAreWe. Let's get back to that. We've written our **sub** so let's build a **class**.

A little about classes.

Many people never use classes, find no need to, and stick to procedures and so on. No worries! It is however, another string to your bow and even if you never use them, you should have at least a rough understanding of how they work. With this in mind **BUT**, remembering that this is supposed to be about the **VBE**, we're going to turn subWhereAreWe into a class in the simplest way possible.

Class Advantages.

Classes are simple to use. Declare it and it runs through the initialization routine and sets thing up. You can then set or get values for that class.

Class Disadvantages.

They're not that simple to set up and they need setting up properly. Just calling a Public subroutine or function is often much simpler.

For VBA at least, classes are by definition **single** entities. They are embedded in the code. If you write a text manipulation procedure that you need and put it into a class, say to strip all non-printable characters, and then decide to make it more publicly available, the number of instances of the same code is likely to multiply making it more difficult to maintain. Some other languages, though not all, bring the class code in from libraries so they always get the latest version. I have to say though, that it's the same for non class code. In all too many cases, propagating a change in code **everywhere** is a problem. IMHO this is one of the biggest problems in VBA.

Simple class layout

The simplest layout for a class in VBA is...

20 Simple Class layout

	Private Module level variables in Declarations These can be... Private variablename [as...] Or Dim variablename [as...] Both are "Private"
	Private Sub Class_Initialize() Code....

	Module level variables = Local Variables
	End sub
	Get / Let subs that assign the module level variables to the class values and allow the values to be set and retrieved.

21 Simple cNameExample Class code

All this does is print out "Mike&Lisa" in the immediate window. Note that the two parts should be put into a new class module and an ordinary code module. We've talked about changing the module name already. A small "c" at the front of the name in my scheme means er, class. cWhereAreWe is the class containing code for WhereAreWe. Unlike the name of a standard module, the name of the class is **very** important because that's how you reference it from a standard module.

In a new Class Module. Call it cNameExample.

	Option Explicit
	'Module level Private variables.
	Dim smName As String
	Private Sub Class_Initialize()
	' Local Variables
	Dim slName As String
	' Work with the locals
	slName = "Mike&Lisa"
	' Set the module level variables to the locals
	smName = slName
	'

	End Sub
	'Get / Let subs that assign the module level variables to the class.
	Public Property Get Name() As String
	Name = smName
	'

	End Property

22 Using the cNameExample example class

In a standard code module.

	Option Explicit
	Sub subtestcNameExample
	Dim cNameExample as cNameExample
	Set cNameExample = New cNameExample
	Debug.Print cNameExample.Name
	' *****
	End Sub

The more experienced will notice that this uses late binding. This will be the case throughout this document. I don't enter into a discussion of early v late binding.

Note that we use the **name** of the class module in the standard module. I've highlighted that.

Building a class from a Sub

Right then! Let's get back to subWhereAreWe and building a class from that.

We're going to change the code of the sub. You may call that cheating but there's a good reason for it. And anyway, it's our book and I'm going to do it.

In the original sub, I minimize the amount of local variables by using the parameters directly in the code. Being ByRef by default, the parameters were updated and we could harvest the results. I'm going to alter that so local variables are used, and then the parameters are set up at the bottom.

Here's the new code. It's essentially the same but doing it this way will make life easier as you'll see.

Again, the changes are:

- Defined a local variable for each item we want
- Set those local variables to the items we want instead of setting the parameters
- Set the parameters to the local variables all in one go at the bottom of the sub

23 subWhereAreWe recoded to convert to a Class

	Sub subWhereAreWe(_
	vbppProject As VBIDE.VBProject, _
	vbcpcComponent As VBIDE.VBComponent, _
	vbcmpCodeModule As VBIDE.CodeModule, _
	vbcppCodePane As VBIDE.CodePane, _
	spProjectNameName As String, _
	spModuleName As String, _
	spProcedureName As String, _
	IngpCurrentModuleLine As Long, _
	IngpModuleProcStartLine As Long, _
	IngpProcCountLines As Long, _
	IngpModuleProcEndLine As Long _
)
	' Returns:
	' Project Object
	' Component Object
	' Code Module Object
	' Codepane object
	' Project name
	' Module name

'	Procedure name.
'	Module line where the cursor is.
'	Module Start Line of Procedure code.
'	Number of Lines of code.
'	Module End Line of Procedure code.
'	
'	The CodePane is where all of the real work is done.
'	
	Dim lnglSLine As Long
	Dim lnglSCol As Long
	Dim lnglELine As Long
	Dim lnglECol As Long
	Dim vbplProject As VBIDE.VBProject
	Dim vbclComponent As VBIDE.VBComponent
	Dim vbcmlCodeModule As VBIDE.CodeModule
	Dim vbcplCodePane As VBIDE.CodePane
	Dim slProjectNameName As String
	Dim slModuleName As String
	Dim slProcedureName As String
	Dim lnglCurrentModuleLine As Long
	Dim lnglModuleProcStartLine As Long
	Dim lnglProcCountLines As Long
	Dim lnglModuleProcEndLine As Long
	Set vbplProject = Application.VBE.ActiveVBProject
	slProjectNameName = vbplProject.Name
	Set vbcplCodePane = Application.VBE.ActiveCodePane
	Set vbcmlCodeModule = vbcplCodePane.CodeModule
	slModuleName = vbcmlCodeModule.Name
	slModuleName = vbcplCodePane.CodeModule.Parent.Name
	Set vbclComponent = vbplProject.VBComponents(spModuleName)
	vbcplCodePane.GetSelection lnglCurrentModuleLine, lnglSCol, lnglELine, lnglECol

slProcedureName	=	vbcplCodePane.CodeModule.ProcOfLine(InglCurrentModuleLine, vbext_pk_Proc)
InglModuleProcStartLine	=	vbcmlCodeModule.ProcBodyLine(slProcedureName, vbext_pk_Proc)
InglProcCountLines	=	vbcplCodePane.CodeModule.ProcCountLines(slProcedureName, vbext_pk_Proc)
InglCurrentModuleLine	=	InglCurrentModuleLine
InglModuleProcEndLine	=	InglModuleProcStartLine + InglProcCountLines - 1
vbppProject	=	vbplProject
vbcComponent	=	vbclComponent
vbcmpCodeModule	=	vbcmlCodeModule
vbcppCodePane	=	vbcplCodePane
spProjectNameName	=	sIProjectNameName
spModuleName	=	sIModuleName
spProcedureName	=	sIProcedureName
IngpCurrentModuleLine	=	InglCurrentModuleLine
IngpModuleProcStartLine	=	InglModuleProcStartLine
IngpProcCountLines	=	InglProcCountLines
IngpModuleProcEndLine	=	InglModuleProcEndLine
' *****		
End Sub		

We can test this using the same subtestsubWhereAreWe that we used before. It should report exactly the same. It's just that the parameters are all set up together at the end. To belabor a point, notice that the "scope" part of the variable names, the letter immediately before the first capital of the name, are l for local and p for parameter and m for private module level variables.

So! With the naming scheme as it stands, s=String, lng/l=Long, i=Integer, o=Object and so on, we can see that all those Dims as l=locals or p=parameters or m=Module level variables get a leeeeeetle easier to manipulate.

If the biz code is all **LOCAL** variables though rather than directly using the sub **PARAMETERS**, then that code will be **DIRECTLY** copyable to a **CLASS**. Maintaining the code in the sub and class then becomes almost trivial. Just copy the code from one to the other and adjust any Dims and module level variable names!

Even though using a naming doodad simplifies altering code **significantly**, it's still a pain in the bottom to change all those pees to els an els to ems. And then insert all of those little property routines as well!

HAH! Why not do it with code!

The code below makes use of code we've already written and just altered. subWhereAreWe! It will look at code from a sub and add code to make it a class.

Long piece of code warning!

I'm going to do a small walkthrough because there's a couple of points worth ummm, pointing out.

24 subInsertGetProperties

	Sub subInsertGetProperties()
	' Insert properties at the end of the
	' module for all Dims in a procedure.
	'
	' This reflects my own personal preferences
	' but it is possible to alter the code
	' to anything you want!
	'
	' Insert a Dim at the top of the module
	' to reflect the built procedure variable.
	'
	' Insert a line in the procedure to
	' set the prperty to the variable in the
	' property.
	'
	' ### Assumes variables defined as:
	' <type><single letter "scope" l/p/m/g><Name beginning with a capitol>
	'
	' ### Assumes Dims are all together and one to a line.
There are a lot of Dims here. Do not be afraid of inserting as many Dims as you need. Some of this is down to modern computers having more memory. Whatever.	
	Dim sLLDimsArray() As String
	Dim sIMDimsArray() As String
	Dim sINamesArray() As String
	Dim sITypesArray() As String
	Dim sISetArray() As String
	Dim sIVar As String

	Dim vbcmICodeModule As VBIDE.CodeModule
	Dim vbplProject As VBIDE.VBProject
	Dim vbclComponent As VBIDE.VBComponent
	Dim vbcpICodePane As VBIDE.CodePane
	Dim slProjectNameName As String
	Dim slModuleName As String
	Dim slProcedureName As String
	Dim lngICurrentModuleLine As Long
	Dim lngIModuleProcStartLine As Long
	Dim lngIProcCountLines As Long
	Dim lngIModuleProcEndLine As Long
	Dim lngICurrentLine As Long
	Dim slLine As String
	Dim lngISanityCheck As Long
	Dim lngINumOfDims As Long
	Dim slDimLine As String
	Dim slLineArray() As String
	Dim slType As String
	Dim lngILenVar As Long
	Dim lngIChrPos As Long
	Dim slChr As String
	Dim lngIChrCode As Long
	Dim slScope As String
	Dim slName As String
	Dim slMName As String
	Dim lngICountOfModuleLines As Long
	Dim lngIAsPos As Long
	Dim slInsertPropertyCode As String
	Dim slInsertMDimMCode As String
	Dim slInsertMToLCode As String
	Dim slTypePrefix As String
	Dim slLName As String
	Dim lngIModuleProcBodyLine As Long
	Dim slSet As String
YAY! Call to the sub we've just built!	
	subWhereAreWe _
	vbplProject, _
	vbclComponent, _

	vbcmlCodeModule, _
	vbcplCodePane, _
	sIProjectNameName, _
	sIModuleName, _
	sIProcedureName, _
	InglCurrentModuleLine, _
	InglModuleProcStartLine, _
	InglModuleProcBodyLine, _
	InglProcCountLines, _
	InglModuleProcEndLine
	' Find first Dim.
	' Loop through Dims.
	' Find the Capitol and create a variable with m as scope and
	' one with no prefix at all.
	' Go to bottom of Module.
	' Insert Property code.
	' Go to top of Module.
	' Insert module level Dims.
	'
	Hmmm... Been here before. Maybe move this to WhereAreWe.
	' Find Dim Line.
	InglCurrentLine = InglModuleProcStartLine
	Do
	sLine = Trim\$(vbcmlCodeModule.Lines(InglCurrentLine, 1))
	If Left\$(sLine, 4) = "Dim " Then
	Exit Do
	Elseif Left\$(sLine, 6) = "Const " Then
	Exit Do
	Elseif Left\$(sLine, 7) = "Static " Then
	Exit Do
	End If
	InglCurrentLine = InglCurrentLine + 1
	InglSanityCheck = InglSanityCheck + 1
	If InglCurrentLine >= InglModuleProcEndLine Then

	' End of module.
	subMsgBox "@End of module." & vbNewLine & "@No Dims."
	Exit Sub
	End If
	If lngSanityCheck > 200 Then
	' Assume no Dims.
	subMsgBox "@No Dims."
	Exit Sub
	End If
	Loop
	Loop through all the proc code lines. If it's not a Dim Get out.
	lngNumOfDims = 0
	Do
	' Move to next dim line
	slDimLine = Trim\$(vbcmlCodeModule.Lines(lngCurrentLine, 1))
	slLineArray = Split(slDimLine, " ")
	' Get out if at the end of Dims
	If Len(slDimLine) = 0 Then
	Exit Do
	End If
	slType = ""
	Select Case slLineArray(0)
	Got a Dim/Const/Static.
	LOTS of possible line formats here to cope with. Not the least is code with continuation characters!
	This is the simple hard coded form for a few items' version.
	This is crying out for building a function!
	Case "Dim", "Const", "Static"

	' Variable will be in the 2nd element.
	sIVar = sLineArray(1)
	InglLenVar = Len(sIVar)
	' Find the first capitol letter.
	' A=65 Z=90 [\]^_` a=97 z=122
	InglChrPos = 1
	Do
	sIChr = Mid\$(sIVar, InglChrPos, 1)
	InglChrCode = Asc(sIChr)
	If InglChrCode < 91 _
	And InglChrCode > 64 Then
	Exit Do
	End If
	InglChrPos = InglChrPos + 1
	If InglChrPos > InglLenVar Then
	The variable is probably all lower case.
	InglChrPos = 0
	Exit Do
	End If
	Loop
	InglAsPos = InStr(sIDimLine, "As")
	sIType = Mid\$(sIDimLine, InglAsPos)
	If InglChrPos < 2 Then
	Another function opportunity!
	' Probably no naming convention so add/impose one.
	sIVar = StrConv(sIVar, vbProperCase)
	sIVar = "I" & sIVar
	Select Case Mid\$(sIType, 4)
	Case "String"
	sIVar = "s" & sIVar

	InglChrPos = 3
	Case "Long"
	sIVar = "Ing" & sIVar
	InglChrPos = 5
	Case "Boolean"
	sIVar = "bIn" & sIVar
	InglChrPos = 5
	Case "Object"
	sIVar = "o" & sIVar
	InglChrPos = 3
	Case "VBIDE.CodeModule"
	sIVar = "vbcM" & sIVar
	InglChrPos = 6
	Case "VBIDE.VBProject"
	sIVar = "vbp" & sIVar
	InglChrPos = 5
	Case "VBIDE.VBComponent"
	sIVar = "vbc" & sIVar
	InglChrPos = 5
	Case "VBIDE.CodePane"
	sIVar = "vbcp" & sIVar
	InglChrPos = 6
	Case Else
	' Variant.
	sIVar = "v" & sIVar
	InglChrPos = 3
	End Select
	Else
	sIScope = Mid\$(sIVar, InglChrPos - 1, 1)
	End If
	' Need to set sISet.
	Select Case Mid\$(sIType, 4)
	Case "String"
	InglChrPos = 3
	sISet = ""

	Case "Long"
	InglChrPos = 5
	sISet = ""
	Case "Boolean"
	InglChrPos = 5
	sISet = ""
	Case "Object"
	InglChrPos = 3
	sISet = "Set "
	Case "VBIDE.CodeModule"
	InglChrPos = 6
	sISet = "Set "
	Case "VBIDE.VBProject"
	InglChrPos = 5
	sISet = "Set "
	Case "VBIDE.VBComponent"
	InglChrPos = 5
	sISet = "Set "
	Case "VBIDE.CodePane"
	InglChrPos = 6
	sISet = "Set "
	Case "Variant"
	InglChrPos = 3
	sISet = ""
	End Select
	sITypePrefix = Mid\$(sIVar, 1, InglChrPos - 2)
	sIName = Mid\$(sIVar, InglChrPos)
	sIName = Replace(sIName, "()", "")
	sIMName = sITypePrefix & "m" & sIName
	sILName = sIVar
	"Redim Preserve" preserves the contents of the array while adding a new element. However, in VBA, it only works to change the size of the LAST element of an array. We avoid that by using different but coordinated SINGLE ELEMENT arrays of the different items. Also, it's not very good form or practice to loop ReDim Preserve in code. Better is to count the occurrences and ReDim to the total. Even though this does two "passes" it's usually faster then Redim Preserving all the time which has quite an overhead. We do this here because we aren't expecting all that many Dims to process.
	ReDim Preserve sIMDimsArray(InglNumOfDims)

	ReDim Preserve sNamesArray(InglNumOfDims)
	ReDim Preserve sTypesArray(InglNumOfDims)
	ReDim Preserve sLDimsArray(InglNumOfDims)
	ReDim Preserve sSetArray(InglNumOfDims)
We need to be clear here about the items we need to build the code we want.	
	slMDimsArray(InglNumOfDims) = slMName
	sNamesArray(InglNumOfDims) = slName
	sTypesArray(InglNumOfDims) = slType
	sLDimsArray(InglNumOfDims) = slLName
	sSetArray(InglNumOfDims) = slSet
	InglNumOfDims = InglNumOfDims + 1
	Case Else
	' Get out if at the end of Dims
	Exit Do
	End Select
	InglCurrentLine = InglCurrentLine + 1
	Loop
	' -----
	' Create and insert Code.
	For InglNumOfDims = LBound(slMDimsArray) To UBound(slMDimsArray)
We build single strings of code. Each VBE line is "separated" by a vbCrLf. Microsoft says that vbCrLf is slightly faster though. If you breakpoint the code here and look at the lines in the immediate window you should see that the code is built correctly. This means that we only insert ONE line at the insertion line number, but in reality, it is multiple lines. All the code lines are built in this single loop here. It is possible and some may prefer, to go through three times and build the code for the separate insertion points one at a time, or even build code insert it, build code insert it, and so on. I've included examples of the code we want to build in the comments. REMEMBER REMEMBER, even though you set a breakpoint, VBA may refuse to let you do it here!	
	' Public Property Get Name() As String
	' Name = smName

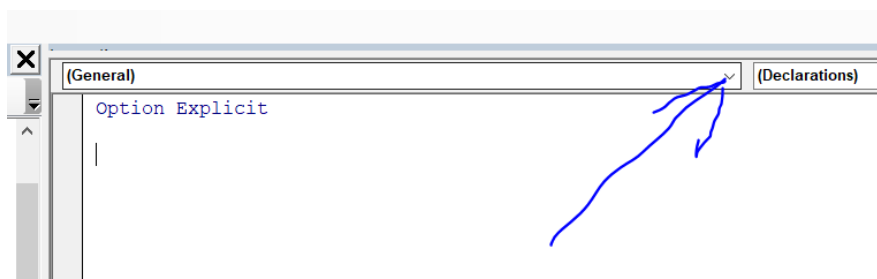
	' ' *****
	' End Property
	slInsertPropertyCode = slInsertPropertyCode _
	& "Public Property Get "
	slInsertPropertyCode = slInsertPropertyCode _
	& slNamesArray(InglNumOfDims) & "(" _
	& " " _
	& slTypesArray(InglNumOfDims)
	slInsertPropertyCode = slInsertPropertyCode & vbCrLf
	slInsertPropertyCode = slInsertPropertyCode _
	& slSetArray(InglNumOfDims) _
	& slNamesArray(InglNumOfDims) _
	& " = " _
	& slMDimsArray(InglNumOfDims)
	slInsertPropertyCode = slInsertPropertyCode & vbCrLf
	slInsertPropertyCode = slInsertPropertyCode & " " & String(69, "**")
	slInsertPropertyCode = slInsertPropertyCode & vbCrLf
	slInsertPropertyCode = slInsertPropertyCode _
	& "End Property"
	slInsertPropertyCode = slInsertPropertyCode & vbCrLf
	' -----
	' <prefix>m<Name> = <prefix>l<Name>
	slInsertMToLCode = slInsertMToLCode _
	& slSetArray(InglNumOfDims) _
	& slMDimsArray(InglNumOfDims) _
	& " = " _
	& slLDimsArray(InglNumOfDims)
	slInsertMToLCode = slInsertMToLCode & vbCrLf
	' -----
	' <prefix>m<Var root> = Old Variable
	slInsertMDimMCode = slInsertMDimMCode _
	& "Dim " _
	& slMDimsArray(InglNumOfDims) & " " _
	& slTypesArray(InglNumOfDims)
	slInsertMDimMCode = slInsertMDimMCode & vbCrLf

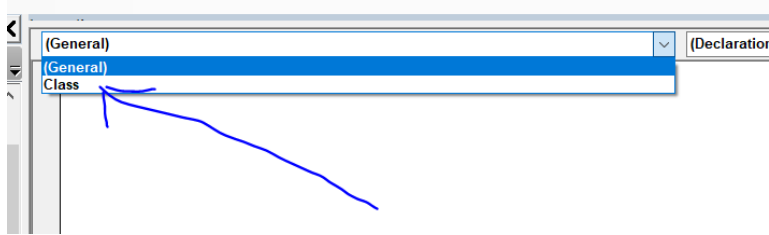
	Next lngINumOfDims
Finished building the code and now we have to put it somewhere.	
Note that we go BACKWARDS up the procedure module code. This is VERY important so that the module line numbers stay the same!	
	vbcmlCodeModule.InsertLines _
	Line:=vbcmlCodeModule.CountOfLines + 1, _
	String:=sInsertPropertyCode
	vbcmlCodeModule.InsertLines _
	Line:=lngIModuleProcEndLine - 1, _
	String:=sInsertMToLCode
	vbcmlCodeModule.InsertLines _
	Line:=vbcmlCodeModule.CountOfDeclarationLines + 1, _
	String:=sInsertMDimMCode
	! *****
	End Sub

Let's use it!

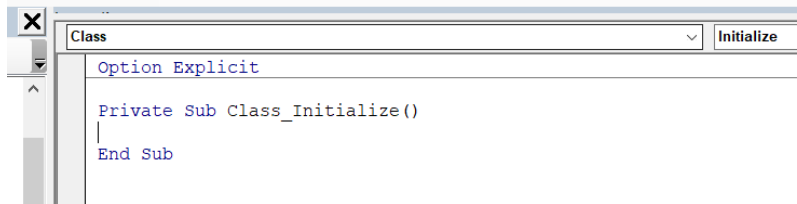
This process is exactly the same as our cNameExample.

- Add a new **CLASS** module.
This is **NOT** the same as a normal module so vbext_pk_Proc is different and needs to be collected from the ProcOfLine call to be used further on.
- Rename it to cWhereAreWe.
- Add a sub Private Sub Class_Initialize()
 - Click the pulldown in the General box
 - Select Class





You'll get...



- Copy the **CODE from subWhereAreWe** to that new sub, Class_Initialize. Just the code from the next line after Sub subWhereAreWe to the line above where the parameters are set because we're going to set the module level variables instead. Remember that lines with the continuation character between them are ONE line.
- Open the Macros menu.
- Run the sub you've just built... subInsertGetProperties
- Have a gander at what you've just created. It's a class and should run just fine.

Here's code to test it.

25 subtestcWhereAreWe

	Sub subtestcWhereAreWe()
	Dim clWhereAreWe As cWhereAreWe
	Set clWhereAreWe = New cWhereAreWe
	Debug.Print clWhereAreWe.ModuleName
	Debug.Print clWhereAreWe.CurrentModuleLine

	'

	End Sub

Now we have a class and a sub that do the same thing. Let's look at it a bit more closely. Let's look at the sub. It only reports so it's okay to run against itself.

You should have a module that just has the test code and the sub in it. This is purely so that it's easy to track and trace. You'll be running code just from one module.

Tell you what, let's add some lines above a sub definition.

Cut and paste subtestsubWhereAreWe and subWhereAreWe into a new module. Add some lines between the procedures.

26 subtestsubWhereAreWe

	Sub subtestsubWhereAreWe()
	Code
	Stop
	'

	End Sub
	Sub subWhereAreWe(_
	Code
	'

	End Sub

Put the cursor in subWhereAreWe and run subtestsubWhereAreWe from the Macros menu.

Now check the module line numbers.

Some of them are wrong.

This is because we're using the wrong start of procedure line number. The procedure actually starts on the line after the end of the last procedure. The line number we are using is the one where the procedure is defined. Let's correct that. Here's the new code complete with line numbers for the module.

27 subWhereAreWe corrected for line numbers

1	Option Explicit
2	
3	Sub subtestsubWhereAreWe()
4	
5	Dim vbplProject As VBIDE.VBProject
6	Dim vbclComponent As VBIDE.VBComponent
7	Dim vbcmlCodeModule As VBIDE.CodeModule
8	Dim vbcplCodePane As VBIDE.CodePane
9	Dim slProjectNameName As String
10	Dim slModuleName As String
11	Dim slProcedureName As String
12	Dim lngCurrentModuleLine As Long
13	Dim lngModuleProcStartLine As Long
Added the line below.	
14	Dim lngModuleProcBodyLine As Long
15	Dim lngProcCountLines As Long
16	Dim lngModuleProcEndLine As Long
17	
18	subWhereAreWe _
19	vbplProject, _
20	vbclComponent, _
21	vbcmlCodeModule, _
22	vbcplCodePane, _
23	slProjectNameName, _
24	slModuleName, _
25	slProcedureName, _
26	lngCurrentModuleLine, _
27	lngModuleProcStartLine, _
Added the line below.	
28	lngModuleProcBodyLine, _
29	lngProcCountLines, _

30	InglModuleProcEndLine
31	
32	Stop
33	' *****
34	End Sub
35	
36	
37	
38	
39	
40	Sub subWhereAreWe(_
41	vbppProject As VBIDE.VBProject, _
42	vbcComponent As VBIDE.VBComponent, _
43	vbcmpCodeModule As VBIDE.CodeModule, _
44	vbcppCodePane As VBIDE.CodePane, _
45	spProjectNameName As String, _
46	spModuleName As String, _
47	spProcedureName As String, _
48	IngpCurrentModuleLine As Long, _
49	IngpModuleProcStartLine As Long, _
50	IngpModuleProcBodyLine As Long, _
51	IngpProcCountLines As Long, _
52	IngpModuleProcEndLine As Long _
53)
54	' Returns:
55	' Project Object
56	' Component Object
57	' Code Module Oject
58	' Codepane object
59	' Project name
60	' Module name
61	' Procedure name.
62	' Module line where the cursor is.
63	' Module Start Line of Procedure code.
64	' Number of Lines of code.
65	' Module End Line of Procedure code.
66	'
67	' The CodePane is where all of the real work is done.
68	'

69	
70	Dim lnglSLine As Long
71	Dim lnglSCol As Long
72	Dim lnglELine As Long
73	Dim lnglECol As Long
74	Dim vbplProject As VBIDE.VBProject
75	Dim vbclComponent As VBIDE.VBComponent
76	Dim vbcmlCodeModule As VBIDE.CodeModule
77	Dim vbcplCodePane As VBIDE.CodePane
78	Dim slProjectNameName As String
79	Dim slModuleName As String
80	Dim slProcedureName As String
81	Dim lnglCurrentModuleLine As Long
82	Dim lnglModuleProcStartLine As Long
Added the line below.	
83	Dim lnglModuleProcBodyLine As Long
84	Dim lnglProcCountLines As Long
85	Dim lnglModuleProcEndLine As Long
86	
87	Set vbplProject = Application.VBE.ActiveVBProject
88	slProjectNameName = vbplProject.Name
89	
90	Set vbcplCodePane = Application.VBE.ActiveCodePane
91	
92	Set vbcmlCodeModule = vbcplCodePane.CodeModule
93	slModuleName = vbcmlCodeModule.Name
94	slModuleName = vbcplCodePane.CodeModule.Parent.Name
95	
96	Set vbclComponent = vbplProject.VBComponents(slModuleName)
97	
98	vbcplCodePane.GetSelection lnglCurrentModuleLine, lnglSCol, lnglELine, lnglECol
99	
100	slProcedureName = vbcmlCodeModule.ProcOfLine(lnglCurrentModuleLine, vbext_pk_Proc)
Changed this line.	
101	lnglModuleProcStartLine = vbcmlCodeModule.ProcStartLine(slProcedureName, vbext_pk_Proc)
Added this line.	

102	InglModuleProcBodyLine = vbcmCodeModule.ProcBodyLine(slProcedureName, vbext_pk_Proc)
103	InglProcCountLines = vbcplCodePane.CodeModule.ProcCountLines(slProcedureName, vbext_pk_Proc)
104	
105	InglCurrentModuleLine = InglCurrentModuleLine
106	
107	InglModuleProcEndLine = InglModuleProcStartLine + InglProcCountLines - 1
108	
109	Set vbppProject = vbplProject
110	Set vbcpComponent = vbclComponent
111	Set vbcmpCodeModule = vbcmlCodeModule
112	Set vbcppCodePane = vbcplCodePane
113	spProjectNameName = slProjectNameName
114	spModuleName = slModuleName
115	spProcedureName = slProcedureName
116	IngpCurrentModuleLine = InglCurrentModuleLine
117	IngpModuleProcStartLine = InglModuleProcStartLine
Added the line below.	
118	IngpModuleProcBodyLine = InglModuleProcBodyLine
119	IngpProcCountLines = InglProcCountLines
120	IngpModuleProcEndLine = InglModuleProcEndLine
121	
122	' *****'
123	End Sub
124	

We've altered the sub and can delete and rebuild the class as well or just change the same lines.

Now the line numbers are reported correctly!

Aside: I write the parameters of my procedures like this:

28 My Procedure parameters layout

	Sub/Function subMike/fncMike(_
	Parameter1 as String, _
	Parameter2 as String, _
	Parameter3 as Long _
) _
	As String

Very simply, for me, it makes counting, adding, deleting, and just reading parameters easier. Did I mention how lazy I am? And while I'm here, depending on the **NAME** of the procedure, comments at the top of a sub/function after the heading line, are a great bonus! They may not say much. It may be just the name of the sub separated out. But, we can then print/list subs and functions in a report! We'll do that later too.

Now then... where were we? Oh yes. Splitting up a Dim line to separate Dims.

Code to Split Dims up

We talked about this some time ago. As we said, this is all about assigning a type to a variable and guess what, a naming convention can help.

Also, some time ago we wrote a function for trying to guess a variable type, `fncGuessVarType`.

We've moved on quite a bit since then, and it's left to the reader to possibly include some of the code from `subInsertGetProperties` in `fncGuessVarType`. It's even possible though we don't do it here, to look for

```
variable = <number>
```

in the code or

```
variable = "
```

and assign `String` or `Long` respectively.

The code in the excel add-in looks at the name and checks for "Num", "Number", "Count", "Array", "Name" and so on in the variable name itself for example, and tries to guess the variable type. A variable beginning with `sl` and with "Array" at the end is assigned

```
Dim slNameArray() As String
```

"i" for example is renamed to `lngI`, and "j" to `lngJ`. I hate single letter variables!

Anyways. We're going to use `fncGuessVarType` and `cWhereAreWe` here.

A side effect of this particular procedure is that all the Dims are moved together to the top of the code, well, to the first line we find a Dim on anyway. We've had to do that a few times now, find the first Dim line in a procedure. We can put that into `cWhereAreWe`! While we're at it, we've been working with the actual code up till now, deleting adding and replacing lines. Working with the code in an array would be a bit trickier but much much faster. We could add that to `cWhereAreWe` as well.

We'll get to that.

Back to splitting up. Remember way back when we sorted Dims, we wrote another procedure to sort a selection of Dims? It's possible to use the same technique to split up a selection of Dims but I leave that to the reader to figure out.

Here's the code to split up all of the Dims from a single line.

29 subccSplitAllDims

	Sub subccSplitAllDims()
	' NO REPORT.
	' NO END MESSAGE.
	'
	' This will go through the whole code of a procedure and
	' split any Dims seperated by commas
	' to seperate lines. They are then all placed
	' where the first Dim was found.
	'
	'
	Dim lngNumLines As Long
	Dim sInsertLines As String
	Dim lngFirstDimLine As Long
	Dim lngStopLines As Long
	Dim llModuleEndLine As Long
	Dim llModuleLine As Long
	Dim lngIM As Long
	Dim lngIN As Long
	Dim lngNewUBound As Long
	Dim lngOldUBound As Long
	Dim lngReplaceUBound As Long
	Dim lngEndLine As Long
	Dim lngStartLine As Long
	Dim lngStartContinuationLines As Long
	Dim lngContinuationLinesCount As Long
	Dim sLine As String
	Dim sNewDimsArray() As String
	Dim sReplaceDims() As String
	Dim sReplaceLines As String
	Dim sComment As String
	Dim vbcmCodeModule As VBIDE.CodeModule
	Dim lngLinesToDelete() As Long
	Dim clWhereAreWe As cWhereAreWe

Dim lngModuleStartLine As Long
Dim lngModuleEndLine As Long
Dim lngModuleLine As Long
Dim lngNumberOfReplaceDims As Long
Dim lngErrNumber As Long
Set clWhereAreWe = New cWhereAreWe
lngModuleLine = clWhereAreWe.ModuleProcBodyLine
lngModuleEndLine = clWhereAreWe.ModuleProcEndLine
Set vbcmlCodeModule = clWhereAreWe.CodeModule
' -----
' Go through ALL of the code.
lngFirstDimLine = 0
Do
Do
lngModuleLine = lngModuleLine + 1
If lngModuleLine >= lngModuleEndLine Then
Exit Do
End If
' Experience tells me it's very possible to have
' multiple lines here connected with the continuation chr.
slLine = ""
lngStartContinuationLines = 0
lngContinuationLinesCount = 1
Do
slLine = slLine & Trim\$(vbcmlCodeModule.Lines(lngModuleLine, 1))
If lngStartContinuationLines = 0 Then
lngStartContinuationLines = lngModuleLine
End If
If Right\$(slLine, 1) <> "_" Then
Exit Do
End If
lngContinuationLinesCount = lngContinuationLinesCount + 1
lngModuleLine = lngModuleLine + 1

	Loop
	sLine = Replace(sLine, "_", "")
	' Skip comments.
	If Left\$(Trim\$(sLine), 1) = "'" Then
	Exit Do
	End If
	' Skip space lines.
	If Len(Trim\$(sLine)) = 0 Then
	Exit Do
	End If
	' We do NOT get rid of spaces as we want to know if there's
	' an As clause.
	' Dim?
	If Left\$(sLine, 4) = "Dim " _
	Or _
	Left\$(sLine, 6) = "Const " _
	Or _
	Left\$(sLine, 7) = "Static " _
	Then
	If lngFirstDimLine = 0 Then
	lngFirstDimLine = lngStartContinuationLines
	End If
	sNewDimsArray = fncSplitDimLine(sLine)
	lngNewUBound = UBound(sNewDimsArray)
	' We may not have initialised this array yet.
	On Error Resume Next
	lngOldUBound = UBound(sReplaceDims)
	lngErrNumber = Err.Number
	On Error GoTo 0
	If lngErrNumber <> 0 Then

	InglReplaceUBound = -1
	End If
	InglReplaceUBound = InglOldUBound + InglNewUBound + 1
	ReDim Preserve sIReplaceDims(InglReplaceUBound)
	For InglM = 0 To InglNewUBound
	sIReplaceDims(InglOldUBound + InglM + 1) _
	= sINewDimsArray(InglM)
	Next InglM
	Else
	Exit Do
	End If
	' Delete the old line.
	vbcmlCodeModule.DeleteLines _
	StartLine:=InglModuleLine - InglContinuationLinesCount + 1, _
	count:=InglContinuationLinesCount
	' Decrement the line counters because of deletion.
	InglModuleLine = InglModuleLine - InglContinuationLinesCount
	InglModuleEndLine = InglModuleEndLine - InglContinuationLinesCount
	Exit Do
	Loop
	If InglModuleLine >= InglModuleEndLine Then
	Exit Do
	End If
	Loop
	' Insert the New set of Dims where we found the first one.
	sIReplaceLines = Join(sIReplaceDims, vbCrLf)
	vbcmlCodeModule.InsertLines _
	Line:=InglFirstDimLine, _
	String:=sIReplaceLines

	Set vbcmCodeModule = Nothing
	' ***** *****
	End Sub

You'll notice we used our **CLASS** to get information about where the cursor is rather than the **SUB**.

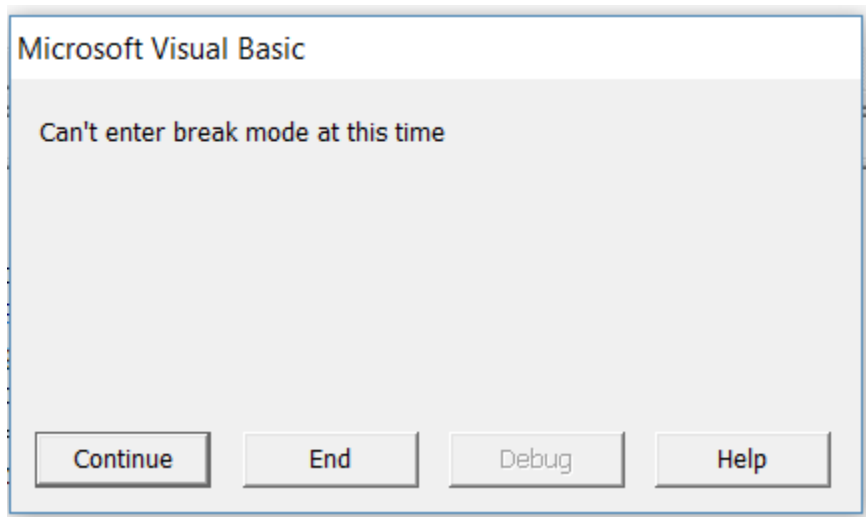
This will be the case for the rest of the procedures here. No more **subWhereAreWe!**

One of the reasons for this is that we will be adding to the WhereAreWe process, and we don't want to keep altering **TWO** sets of code. If you want to maintain the sub, I suggest you wait till the end before doing so and make all the changes in one go by copying the body of the code back and altering any parameters. Structured the way we have laid out, that should be all it should take.

Debug Print variables

While working on that last piece of code I needed to know the value of a few variables. I actually added watches for them but that's not always possible because altering the code while looking at it in the VBE sometimes gives the message...

7 Can't enter break mode



In that case we will want to Debug.Print the variable.

Setting up Debug.Print is simple. You may want to print the variable value before and after a line. What if you want to print a number of variables or indication of the processing path? Simple... you add some text to the Debug.Print to show where you are. After doing all this you may want to delete them all afterwards. Actually, I'm sure you'll want to delete them.

There are lots of simple steps to do. We can pick up a variable by selecting it and using GetSelection. We can insert Lines and we can delete lines.

We've done all of that.

Great!

The hardest part here is working out what to print and how to set that up! The below code just prints a line number, the selected variable, and a comment if you enter one, but we can have it print anything we want really.

30 subInsertSelectionDebug

	Sub subccInsertSelectionDebug()
	' SELECTION.
	' NO REPORT.
	' NO END MESSAGE.
	'
	' Insert the selection as a debug line.
	' This will mostly be used to debug.print a
	' variable.
	'
	Dim clWhereAreWe As cWhereAreWe
	Dim lnglModuleEndLine As Long
	Dim lnglModuleLine As Long
	Dim slCurrentLine As String
	Dim slDQ As String
	Dim slPrintText As String
	Dim slSelection As String
	Dim vbcmlCodeModule As VBIDE.CodeModule
	Dim lnglSLine As Long
	Dim lnglSCol As Long
	Dim lnglELine As Long
	Dim lnglECol As Long
	Dim slPrintBefore As String
	Dim slPrintAfter As String
	Set clWhereAreWe = New cWhereAreWe
	lnglModuleLine = clWhereAreWe.ModuleProcBodyLine
	lnglModuleEndLine = clWhereAreWe.ModuleProcEndLine
	slSelection = clWhereAreWe.Selection
	lnglSLine = clWhereAreWe.CurrentModuleLine
	lnglSCol = clWhereAreWe.SCol
	lnglELine = clWhereAreWe.ELine
	lnglECol = clWhereAreWe.ECol
	slCurrentLine = clWhereAreWe.CurrentLine

	If lnglSLine <> lnglELine Then
	subMsgBox "@Sorry. I don't know what to do" _
	& vbCrLf & " because there is more than one line selected." _
	& vbCrLf & "Please select a variable to Debug.Print."
	Exit Sub
	End If
	If Len(slSelection) = 0 Then
	subMsgBox "@Sorry. You haven't selected a variable." _
	& vbCrLf & "Please select a variable to Debug.Print."
	Exit Sub
	End If
	Set vbcmIcodeModule = clWhereAreWe.CodeModule
	slDQ = Chr(34)
	slPrintText = InputBox("@Please enter Text to print", "Text for Debug.Print")
	slPrintBefore = "Debug.Print " _
	& slDQ & slPrintText & slDQ _
	& " & " _
	& slDQ & " Before L# " & CStr(lnglSLine) _
	& " " & slSelection & " " _
	& ">" & slDQ & " & " _
	& slSelection & " & " _
	& slDQ & "<" & slDQ _
	& "' Debug."
	slPrintAfter = "Debug.Print " _
	& slDQ & slPrintText & slDQ _
	& " & " _
	& slDQ & " After L# " & CStr(lnglSLine) _
	& " " & slSelection & " " _
	& ">" & slDQ & " & " _
	& slSelection & " & " _

& sIDQ & "<" & sIDQ_
& "' Debug."
' Print After?
vbcmlCodeModule.InsertLines lnglSLine + 1, slPrintAfter
' Print Before?
vbcmlCodeModule.InsertLines lnglSLine, slPrintBefore
'

End Sub

Here is a very simple procedure before and after using the above code and selecting "ss"

31 Sub before subInsertSelectionDebug

Sub subtestsubccInsertSelectionDebug()
Dim ss As String
Dim slLine As String
ss = "gg"
'

End Sub

32 Sub after insertSelectionDebug

Sub subtestsubccInsertSelectionDebug()
Dim ss As String
Dim slLine As String
Debug.Print "" & " Before L# 19 ss >" & ss & "<" ' Debug.
ss = "gg"
Debug.Print "" & " After L# 19 ss >" & ss & "<" ' Debug.

	' *****
	End Sub

A caveat for using the code is that it doesn't deal with expressions or continuation lines.

Here is our user form used for a number of options for a similar procedure. I hold the options in an INI file as defaults. The code behind this form does deal with expressions and continuation lines.

8 Insert Debug Options

Insert Debug Options

Text

- Print the Line Numer
- Print the Procedure Namer
- Print the Module Name
- Print the Project Name
- Print the selection as Text
- Insert Line Number
- Create BookMark
- Print the Whole Current Line
- Print Before the current Line
- Print After the Current Line
- Print a Stop after

CANCEL OK

Changing anything here changes the item in the INI file so the options are persistent.

De debug

What do we do when our debugging is finished? We delete the Debug.Print lines of course!

Here's code to do just that.

33 subccDeleteDebugPrint

	Sub subccDeletDebugPrint()
	' Delete Debug.Print Lines.
	'
	Dim clWhereAreWe As cWhereAreWe
	Dim lngModuleEndLine As Long
	Dim lngModuleBodyLine As Long
	Dim slCurrentLine As String
	Dim slDQ As String
	Dim slPrintText As String
	Dim slSelection As String
	Dim vbcmCodeModule As VBIDE.CodeModule
	Dim lngSLine As Long
	Dim lngSCol As Long
	Dim lngELine As Long
	Dim lngECol As Long
	Dim slPrintBefore As String
	Dim slPrintAfter As String
	Dim lngCurrentLine As Long
	Dim slLookFor As String
	Dim lngLenLookFor As Long
	Dim slLine As String
	Set clWhereAreWe = New cWhereAreWe
	lngModuleBodyLine = clWhereAreWe.ModuleProcBodyLine
	lngModuleEndLine = clWhereAreWe.ModuleProcEndLine
	slSelection = clWhereAreWe.Selection
	lngSLine = clWhereAreWe.CurrentModuleLine
	lngSCol = clWhereAreWe.SCol
	lngELine = clWhereAreWe.ELine

	InglECol = clWhereAreWe.ECol
	slCurrentLine = clWhereAreWe.CurrentLine
	Set vbcmIcodeModule = clWhereAreWe.CodeModule
	slLookFor = "Debug.Print"
	InglLenLookFor = Len(slLookFor)
	' Go Back UP the code to preserve line numbers.
	InglCurrentLine = InglModuleEndLine
	Do
	slLine = Trim\$(vbcmIcodeModule.Lines(InglCurrentLine, 1))
	If Len(slLine) >= InglLenLookFor Then
	If InStr(slLine, slLookFor) > 0 Then
	vbcmIcodeModule.DeleteLines _
	StartLine:=InglCurrentLine, _
	count:=1
	End If
	End If
	InglCurrentLine = InglCurrentLine - 1
	If InglCurrentLine <= InglModuleBodyLine Then
	Exit Do
	End If
	Loop
	' -----
	End Sub

About that user form I mentioned for options. It is invoked by entering a question mark in an InputBox instead of text for a comment. It will appear and stay in the VBE and **NOT** in the application thanks to some wonderful code from Chip Pearson.

This code is to keep the userform in the VBE and not appear in the application. It's Chips code, available on his site. Put this in the **FORM** code of the form you are creating. When you load the form, it will stay in the VBE.

34 Keep UserForm in the VBE

Option Explicit
Private Const C_USERFORM_CLASSNAME = "ThunderDFrame"
Private Declare Function SetParent Lib "user32" (_
ByVal hWndChild As Long, _
ByVal hWndNewParent As Long) As Long
Private Declare Function FindWindow Lib "user32" Alias "FindWindowA" (_
ByVal lpClassName As String, _
ByVal lpWindowName As String) As Long
Private Sub UserForm_Initialize()
subMakeParent
' *****
End Sub
Private Sub subMakeParent()
' Chip Pearson.
Dim Res As Long
Dim UserFormHwnd As Long
Dim VBEHwnd As Long
' Chip.
'
' Get the Hwnd of the UserForm
'
UserFormHwnd = FindWindow(C_USERFORM_CLASSNAME, Me.Caption)

```

If UserFormHwnd > 0 Then
' .....
' ' Get the ROOTOWNER Hwnd
' .....
' ' VBEHwnd = GetAncestor(UserFormHwnd, GA_ROOTOWNER)
' Next line is mine.
VBEHwnd = Application.VBE.MainWindow.hwnd
If VBEHwnd > 0 Then
' .....
' Call SetParent to make the form
' a child of the application.
' .....
Res = SetParent(UserFormHwnd, VBEHwnd)
If Res = 0 Then
' .....
' An error occurred.
' .....
MsgBox "The call to SetParent failed."
End If
End If
End If
' *****
End Sub
Private Sub UserForm_Terminate()
Application.VBE.MainWindow.Visible = True
Application.VBE.MainWindow.SetFocus
DoEvents
' *****
End Sub

```

VBE programmers do it Immediately!

Getting back to printing to the immediate window. Trust me. You will want to run your procedure multiple times. The window gets messy and sometime very confusing! Clearing the window manually means going to the immediate window, doing a CtrlA to select everything and then pressing the delete key. It's sometimes convenient to be able to clear the immediate window programatically. There is an easy way and a difficult way.

The following code emulates the manual method.

35 Clear immediate window with SendKeys

	Sub subClearIW()
	Application.VBE.Windows.Item("Immediate").SetFocus
	Application.SendKeys "^a"
	Application.SendKeys "{Del}"
	' ***** *****
	End Sub

However, most people, including me, say that you can't depend on SendKeys.

You are probably aware that if you print a lot to the immediate window then lines will scroll off after about 300. This can be important if you are printing a large array for example, or tracing through some code repeatedly, or just tracing through a lot of code.

In that case maybe you should think about creating a procedure that writes to a log file. We actually do exactly that later when coding a [Compile Report](#).

But! It gets worse. SendKeys is only available in Excel! I've tried Word Powerpoint and Access so far... No go!

So, we need to find another way.

There is some very simple code that works in 99% of cases. Okay, I just guessed the 99%, but I'm pretty sure you won't come across the other "1%" and I'm just doing CYA really.

To get back. If you do a lot of printing to the IW, then you will find your earlier results scrolling off to neverland. As I mentioned, it will only print about 300 lines.

We can use that!

The following code will do it's best to scroll everything in the IW outa sight!

36 Clear immediate window. Debug.Print 1

	Sub subClearIW()
	Debug.Print String(65535, vbCr)
	' *****
	End Sub

There are variations of this...

37 Clear immediate window. Debug.Print 2

	Sub subClearIW()
	Dim lngIN As Long
	For lngIN = 0 To 100
	Debug.Print ""
	Next lngIN
	' *****
	End Sub

But I think you get the idea.

There is more complex code out on the net, some better than others, to clear the immediate window. Basically, all of them do the same as sendkeys, but via API calls. For completeness I present the more complex code to [clear the IW](#) with API calls in the appendices.

Button up

So far, when running a piece of code, we've done this from the Macros menu. You may have noticed this is the first time I've used the word "Macro". IMHO, it's a terrible word. The code you write is in procedures. The procedures may be Subs, Functions, or Properties and I often refer here to "subs" or "functions" or "procs". I call a procedure a procedure!

Flame off!

Anyway, moving on. In Word and so on, in the VBE, it's possible to assign a procedure to a button, or put it in a menu option on a toolbar instead of using the "macros" (Boooohissss) menu.

That would be nice! Yea! I hear ya. But be careful. You may end up with a monster!

However, having said that...

The latter is accomplished in the VBE by referencing the Application.VBE.CommandBars collection. Remember there is no ribbon here!

Before going any further, I would like to point out that this subject is dealt with **VERY** extensively on the internet, most notably by someone I've mentioned a few times already, Chip Pearson. I've distilled the following code from a number of sources all credited in the code. **BUT!** If you follow the links in the code and look at it carefully, you'll find that a **LOT** of the code originates with Mr Pearson. Note that there is no skimping here. You really do need all of the code.

We're going to build a toolbar with code. It's just a toolbar and nothing to be scared of, but there is some pretty tricky stuff here. When I first tried this, I was surprised that it didn't work. Not properly anyways. It would run once and that was it. Until I remembered. Remember Remember, tenet 1! The trick was to put the code in its own project. It worked well then but got bogged down if anything went wrong. So, heads up, this is not 100% stable in all situations and you should definitely put the code in a separate project! The reason probably lies with the event handler resetting and losing its' way. We're going to do it anyway.

You first need a new Project. Remember remember! One project per document so you have to create a new document/spreadsheet/presentation/whatever to get a new Project. Back to the VBE and there it is. Yay!

You'll need a class module and a standard module. The code here is duplicated and mashed from Chip Pearson at <http://www.cpearson.com/excel/VbeMenus.aspx> and from xld on the vbaexpress forum at <http://www.vbaexpress.com/forum/showthread.php?11748-add-an-item-to-a-vbe-toolbar>.

Rename the class module to cBarEvents. The name itself isn't important, but it's all linked together in the code so if you want a different name, you'll have to alter it all over the place. The same goes for other bits of code. My advice is to get it working first before you alter anything.

Here's the code for the class.

38 Button code for the CLASS module

	Option Explicit
	Public WithEvents oCBControlEvents As CommandBarEvents
	Private Sub oCBControlEvents_Click(_
	ByVal cbCommandBarControl As Object, _
	Handled As Boolean, _
	CancelDefault As Boolean)
	'http://www.vbaexpress.com/forum/showthread.php?11748-add-an-item-to-a-vbe- toolbar
	'On Error Resume Next
	'Run the routine given by the commandbar control's OnAction property
	Application.Run cbCommandBarControl.OnAction
	Handled = True
	CancelDefault = True
	End Sub

Now in a standard module, and you can call that anything you like because it's the code that's important.

39 Button code in STANDARD module

	Option Explicit
	Dim mcolBarEvents As New Collection 'collection to store menu item click event handlers
	Sub subBrandNewBarAndButton()
	Dim CBE As cBarEvents
	Dim myBar As CommandBar
	Dim myControl
	On Error Resume Next

```

Application.VBE.CommandBars("Debug Extra").Delete
On Error GoTo 0
' -----
' ToolBar.

Set myBar = Application.VBE.CommandBars.Add("Debug Extra", , False, True)
myBar.Visible = True
' -----
' Button.

Set myControl = myBar.Controls.Add(msoControlButton, , , 1)
With myControl
    .Caption = "Debug Variable"
    .FaceId = 29
    .OnAction = "subccInsertSelectionDebug"
End With

'Create a new instance of our button event-handling class
Set CBE = New cBarEvents

'Tell the class to hook into the events for this button
Set CBE.oCBCControlEvents = Application.VBE.Events.CommandBarEvents(myControl)

'And add the event handler to our collection of handlers
mcolBarEvents.Add CBE
' -----
' Button.

Set myControl = myBar.Controls.Add(msoControlButton, , , 1)
With myControl
    .Caption = "Clear Debug"
    .FaceId = 29
    .OnAction = "subccDeletDebugPrint"
End With

'Create a new instance of our button event-handling class

```

	Set CBE = New cBarEvents
	'Tell the class to hook into the events for this button
	Set CBE.oCBControlEvents = Application.VBE.Events.CommandBarEvents(myControl)
	'And add the event handler to our collection of handlers
	mcolBarEvents.Add CBE
	' _____
	' Button.
	Set myControl = myBar.Controls.Add(msoControlButton, , , 1)
	With myControl
	.Caption = "Mark"
	.FaceId = 29
	.OnAction = "subBookMarkAndBreakpoint"
	End With
	'Create a new instance of our button event-handling class
	Set CBE = New cBarEvents
	'Tell the class to hook into the events for this button
	Set CBE.oCBControlEvents = Application.VBE.Events.CommandBarEvents(myControl)
	'And add the event handler to our collection of handlers
	mcolBarEvents.Add CBE
	' _____
	' Button.
	Set myControl = myBar.Controls.Add(msoControlButton, , , 1)
	With myControl
	.Caption = "Clear IW"
	.FaceId = 29
	.OnAction = "subClearIW"
	End With
	'Create a new instance of our button event-handling class
	Set CBE = New cBarEvents

	'Tell the class to hook into the events for this button
	Set CBE.oCBControlEvents = Application.VBE.Events.CommandBarEvents(myControl)
	'And add the event handler to our collection of handlers
	mcolBarEvents.Add CBE
	'

	End Sub

You'll see that the OnAction Properties for each button has a sub mentioned. These are subs we've worked on previously. Well, subccInsertSelectionDebug, subccDeletDebugPrint, subClearIW are anyway. There's a new one. When debugging it's very likely that you will want to jump around code and set breakpoints. Unfortunately, you can't jump from one breakpoint to the next. Or can you?

The bookmark is a little used feature in VBA. I've asked quite a few people and none of them say they use it and a lot of them have said "Bookmark?". Setting bookmarks allows you to jump around your code. You have to do it in sequence but hey! I encourage you to use it. I use it a lot!

You can set a bookmark and a breakpoint on the same line. That means you can jump through your breakpoints.

Clever huh!

Incidentally, this also gives us a method for saving where a set of breakpoints are set. We can jump to the next bookmark, do WhereAreWe and we have project, module, procedure and module line.

Here's a piece of code to do that. We can't actually "set" a breakpoint as such, we have to toggle it. Same for bookmarks. This code uses the built-in menus for VBA and executes commands from them.

40 subBookMarkAndBreakpoint

	Sub subBookMarkAndBreakpoint()
	On Error Resume Next
	Application.VBE.CommandBars("Menu Bar") _
	.Controls("Edit").Controls("Bookmarks").Controls("Toggle BookMark").Execute
	Application.VBE.CommandBars("Menu Bar") _
	.Controls("Debug").Controls("Toggle Breakpoint").Execute

	On Error GoTo 0
	' ***** *****
	End Sub

This is a VERY big deal!

Using this method, it's possible to run **ANY** of the built-in commands or added menu commands programmatically. If you use Smart Indenter for example, you'll see that it adds an item to the menus. This allows us to use Smart Indenter programmatically for example. Lots of third-party add-ins add menu items. This means you can make a button on a toolbar for your favourite doodad so you don't need to go through the menus.

So! You'll need the code for the subs that are pointed to by the buttons, in the same project. Aaaannnd, don't forget any procedures they call and procedures they call and so on. You can put them anywhere in a normal code module or each in their own module or together in a single module or the same module as the code used to set up the buttons. Possibly a bit of copy and pasting to do then! I personally would encourage you to use lotsa modules. Easier then to go to a particular set of subs, just click in the explorer. On the other hand, putting lots of subs into a single module means you have a drop-down listing all of the subs/functions in alphabetical order. Your choice. It always means proliferation of duplicate code. No real solution for there but we'll discuss that later.

Oh, depending on how you dealt with subMsgBox way back, you may need that as well.

Running subBrandNewBarAndButton will set up a floating toolbar with four buttons. It doesn't look very nice though. For one thing all the icons are the same!

Okay! In the VBE set a breakpoint on the following line.

```
' -----
' Toolbar.

Set myBar = Application.VBE.CommandBars.Add("Extra Debug", , False, True)

myBar.Visible = True

' -----
' Button.

Set myControl = myBar.Controls.Add(msoControlButton, , , 1)
```

Run the sub.

```

' -----
' ToolBar.

Set myBar = Application.VBE.CommandBars.Add("Extra Debug", , False)

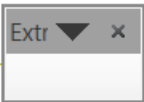
myBar.Visible = True

' -----
' Button.

Set myControl = myBar.Controls.Add(msoControlButton, , , 1)

With myControl

```

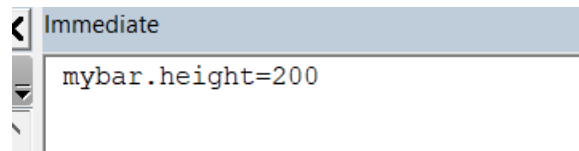


When it stops at the breakpoint, open the Locals window and look at the properties for myBar. You won't be able to look at them any other way.

myBar		CommandBar/CommandBa
AdaptiveMenu	False	Boolean
Application	<Application-defined or object-d	Object
BuiltIn	False	Boolean
Context	""	String
Controls		CommandBarControls/Com
Creator	1398031698	Long
Enabled	True	Boolean
Height	62	Long
Id	1	Long
Index	33	Long
Instanceld	494757816	Long
InstanceldPtr	494757816	Variant/<Unsupported variar
Left	795	Long
Name	"Extra Debug"	String
NameLocal	"Extra Debug"	String
Parent		Object/VBE
Position	msoBarFloating	MsoBarPosition
Protection	msoBarNoProtection	MsoBarProtection
RowIndex	-1	Long
Top	572	Long
Type	msoBarTypeNormal	MsoBarType
Visible	True	Boolean
Width	90	Long
myControl		Variant/Object/CommandB:

You'll see Height and Width.

In the immediate window set the height to a large number and of course press return.



Forget it. Nothing will change.

Release the breakpoint and set another for after the first button has been instantiated.

```
-
' -----
' Button.
Set myControl = myBar.Controls.Add(msoControlButton, , , 1)

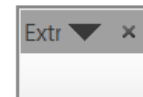
With myControl
    .Caption = "Debug Variable"
    .FaceId = 29
    .OnAction = "subccInsertSelectionDebug"
    .Width = 500
End With

'Create a new instance of our button event-handling class
Set CBE = New cBarEvents

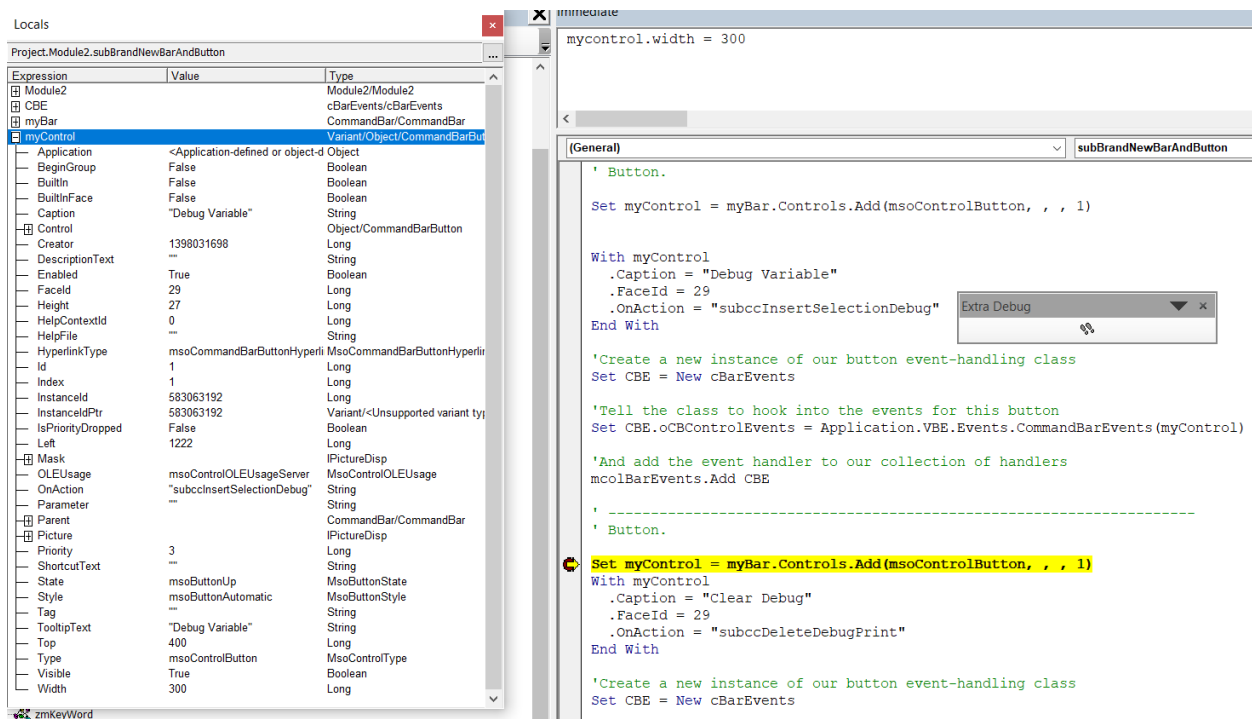
'Tell the class to hook into the events for this button
Set CBE.oCBCControlEvents = Application.VBE.Events.CommandBarEvents(myControl)

'And add the event handler to our collection of handlers
mcolBarEvents.Add CBE

' -----
' Button.
Set myControl = myBar.Controls.Add(msoControlButton, , , 1)
With myControl
    .Caption = "Clear Debug"
```



After stopping at that breakpoint, Set the new button control width in the immediate window. I know the resolution is small here but there's a lot going on and I wanted to capture all of it. There's the button being wider, the breakpoint, the property of the button, and the command in the immediate window.



The command bar will format itself to the size of the buttons horizontally.

So, the button properties control at least some of the toolbar properties.

There's a lot of code and lists on the internet about FaceIds. Put whatever number you like for each button to make it pretty.

A nice place to have your new toolbar is just to the left of the codepane so you don't have to move the mouse too far.

And remember remember, use the toolbar/buttons on **OTHER PROJECTS**.

Here's the final toolbar as set up by the code above.



Messages

We just mentioned subMsgBox. Two things here. An ordinary MsgBox will show in the **application**. If you're in Excel it will show on a spreadsheet. If you're in Word it will show on a document and so on. You don't necessarily want that. In fact, it would be quite useful to have msgs appear in the VBE sometimes. While we're at it, the other thing about MsgBox is you can't copy the text!

We can fix that.

We do this by using a UserForm. We've mentioned those before.

We have code to put into a userform to make it stay in the VBE. Now we need to display a message. We can add some MsgBox functionality to it as well.

A userform is for all intents a class module. We can set up properties in a userform to set and return information.

Let's start with the userform. Insert one. Change its' name to frmMsgBox. Add the code given way above for the initialize and terminate events and subMakeParent. See, initialize and terminate, just like grown up OOP! Hehehe.

Run it just to make sure. You should get a blank form that stays in the VBE. In fact, I guarantee you will.

Now we need code to run it... oh... and code to get let set properties of the class, er, userform.

While we're in the userform here's code to set up the properties. I'm leaving out MakeParent and Terminate.

41 Userform/class properties for subMsgBox

	Option Explicit
	Private Const C_USERFORM_CLASSNAME = "ThunderDFrame"
	Private Declare Function SetParent Lib "user32" (_ ByVal hWndChild As Long, _ ByVal hWndNewParent As Long) As Long
	Private Declare Function FindWindow Lib "user32" Alias "FindWindowA" (_ ByVal lpClassName As String, _ ByVal lpWindowName As String) As Long

Private IngmAnswer As Long
Private Sub cmdCancel_Click()
IngmAnswer = vbNo
Me.Hide
'

End Sub
Private Sub cmdNo_Click()
IngmAnswer = vbNo
Me.Hide
'

End Sub
Private Sub cmdOK_Click()
IngmAnswer = vbOK
Me.Hide
'

End Sub
Private Sub cmdOK_KeyPress(ByVal KeyAscii As MSForms.ReturnInteger)
If KeyAscii = 27 Then
Me.Hide
End If
'

End Sub
Private Sub cmdYes_Click()
IngmAnswer = vbYes
Me.Hide
'

End Sub
Private Sub UserForm_Activate()
Me.Left = 10
Me.Top = 300

```

'
*****
End Sub
Private Sub UserForm_Initialize()

subMakeParent

'Application.VBE.MainWindow.Visible = True
'Application.VBE.MainWindow.SetFocus
'
'
*****
End Sub
Public Property Get Answer() As Long

Answer = lngmAnswer
'
*****
End Property
Public Property Let Answer(ByVal lngpAnswer As Long)

'
*****
End Property
Property Let Title(ByVal spTitle As String)

Me.Caption = spTitle
'
*****
End Property
Public Property Let SetButtons(ByVal lngpSetButtons As Long)

Select Case lngpSetButtons
Case vbOK
Me.cmdCancel.Visible = False
Me.cmdYes.Visible = False

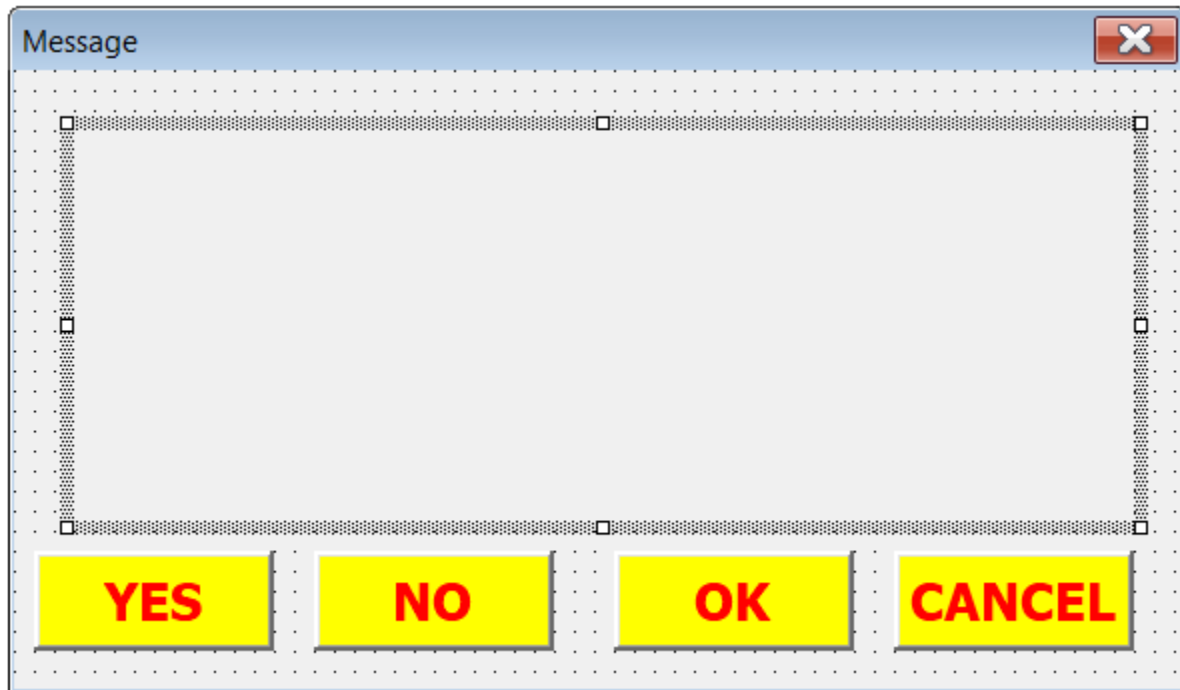
```


Me.cmdNo.Visible = False
Case vbYesNo
Me.cmdCancel.Visible = False
Me.cmdOK.Visible = False
Case vbCritical
Me.cmdCancel.Visible = False
Me.cmdYes.Visible = False
Me.cmdNo.Visible = False
End Select
' *****
End Property
Private Sub UserForm_QueryClose(Cancel As Integer, CloseMode As Integer)
If CloseMode = vbFormControlMenu Then
Me.Hide
Cancel = True
DoEvents
End If
' *****
End Sub

Shades of making a procedure/Sub a class!

Here's a pickie of my form in the designer.

9 My MsgBox UserForm



The big rectangle in the middle is a text box with the same background colour as a normal msgbox, The yellow doodads are command buttons.

I've just done the form code which has the code for the forms and buttons. But! The important bit is how to use it!

Here's the code to do that in a standard code module. I called mine mMsgBox. Yeah, I know, predictable. Also, and lots of people know this and skim over it, there are usually two ways of presenting things. As a Sub or as a Function. Many people are fond of "functional" programming where everything returns a status. I'm presenting the function. The sub is just outputting a message and stopping when the OK button is pressed. The function here returns an "answer".

There's some important stuff here. So, I'm going to mini walkthrough.

42 Using the MsgBox userform

	Option Explicit
	Private Sub subtestfrmMsgBox()
	Debug.Print fncMsgBox("test", vbYesNo)
	' *****
	End Sub
	Public Function fncMsgBox(_ spMessage As String, _ Optional lngpSetButtons As Long = 1, _ Optional spTitle As String = "" _)
	' This is to run a form to display a message.
	'
	' The form is made a subform of the VBE so that
	' the message stays in the VBE rather than
	' displaying in the application.
	'
	Dim vbplProject As VBIDE.VBProject
	Dim vbclComponent As VBIDE.VBComponent
	Note the blnI prefix... bln for Boolean, I for local.
	Dim blnIFound As Boolean
	Dim lngIAnswer As Long
	Dim lngISetButtons As Long
	blnIFound = False

The For..Next loop here is something you'll see a lot of while working in the VBE for looping through Projects.	
	For Each vbplProject In Application.VBE.VBProjects
	If vbplProject.Protection = vbext_pp_locked Then
	Else
	For Each vbclComponent In vbplProject.VBComponents
	Select Case vbclComponent.Type
	Case vbext_ct_MSForm
We're looking for a component with the name of our form. If we don't find it then we run a normal MsgBox. Otherwise we Show the form.	
	If vbclComponent.Name = "frmMsgBox" Then
	InglAnswer = fncRealMsgBox(spMessage, lngpSetButtons, spTitle)
	blnFound = True
	End If
	End Select
	Next vbclComponent
	End If
	If blnFound = True Then
	Exit For
	End If
	Next vbplProject
	If blnFound = False Then
	' Use the standard msgbox.
We haven't found a module anywhere in this project with the name of our form for frmMsgBox so do an ordinary MsgBox.	
	InglAnswer = MsgBox(spMessage, lngpSetButtons, spTitle)
	End If
	fncMsgBox = InglAnswer
	'

	End Function
We run this if we find a module with the name of our form.	
	Private Function fncRealMsgBox(_
	spMessage As String, _
	lngpSetButtons As Long, _

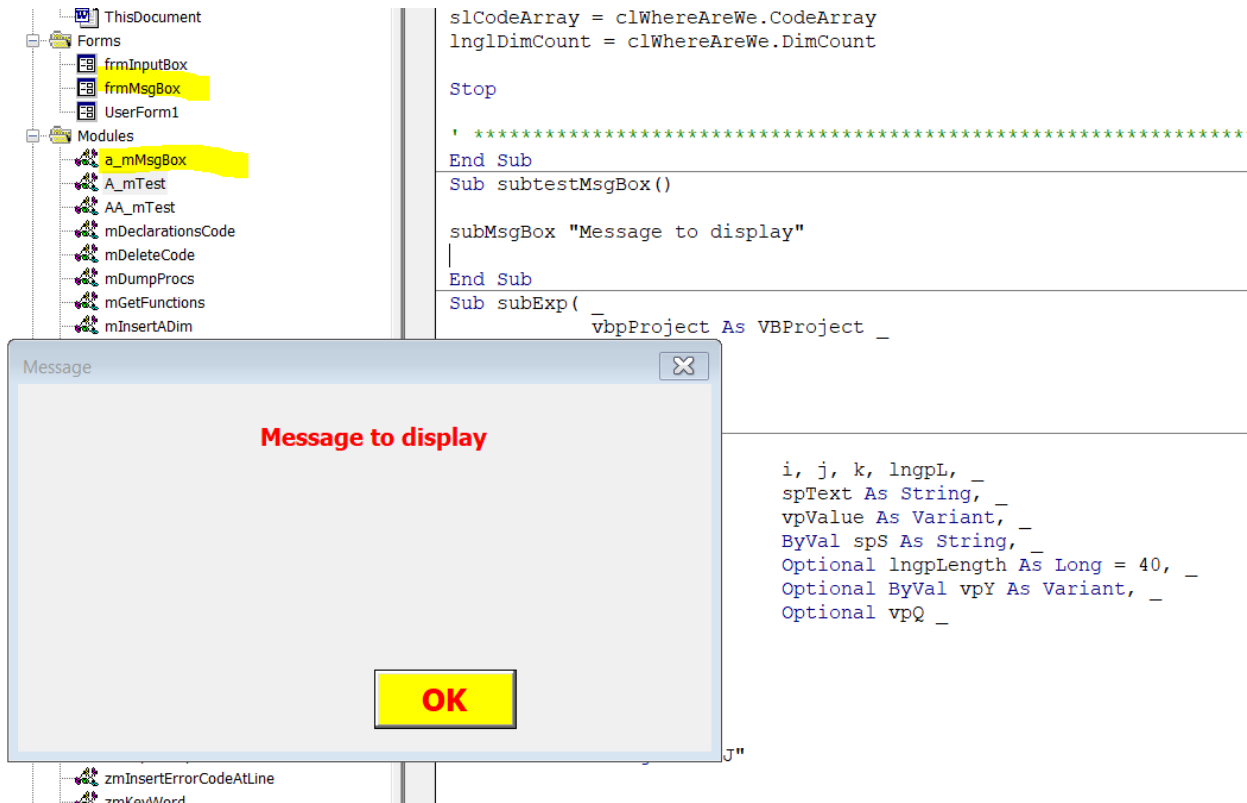
	spTitle As String _
)
	Dim lngAnswer As Long
	Dim lngSetButtons As Long
	Dim frmForm As frmMsgBox
These are the important bits. It's a class. We've set up properties in the class to accept data and set its properties. The data is read by the class/form and the code therein executed.	
	Set frmForm = New frmMsgBox
	frmForm.txtMsg.Text = spMessage
	frmForm.SetButtons = lngSetButtons
	frmForm.Title = spTitle
Ok, we have data in the class properties, now let the dog see the rabbit.	
	frmForm.Show
This is the VERY important bit. Do NOT unload the form until we get our answer back from the class/form.	
	lngAnswer = frmForm.Answer
NOW we can unload the form!	
	Unload frmForm
Return the answer from the form.	
	fncRealMsgBox = lngAnswer
	'

	End Function

Now we have a MsgBox that you can copy text from **AND** is more colourful, **AND** stays in the VBE!

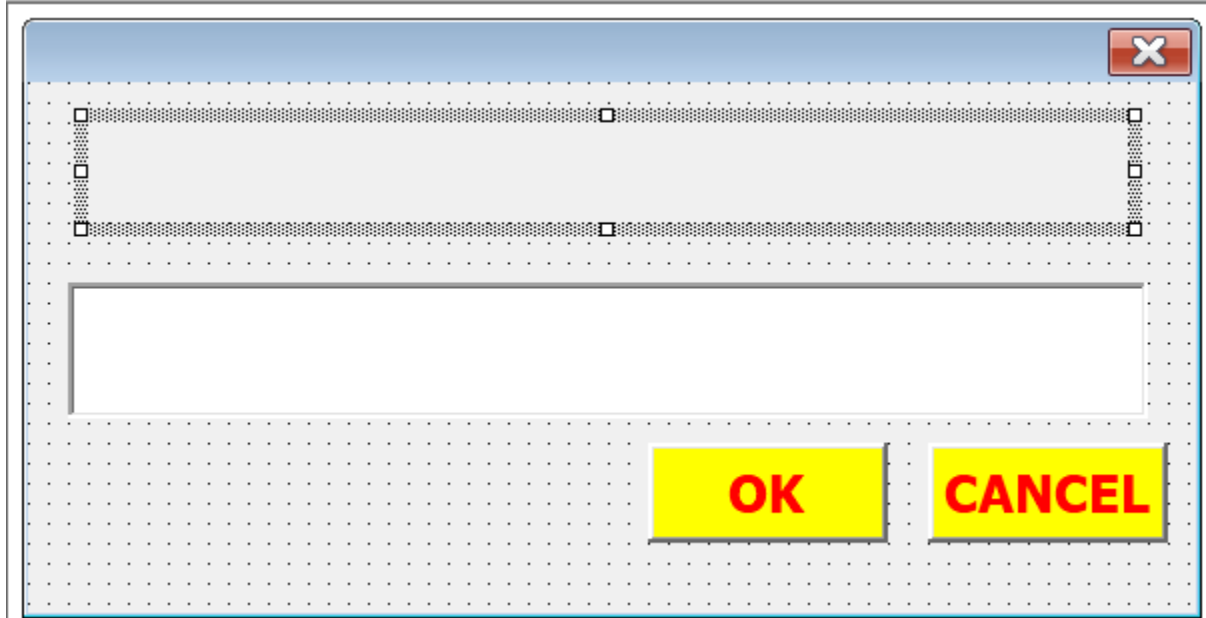
Here's a pic of it in action. You can see the VBE behind it. I've highlighted the two modules concerned in the left explorer pane. Note that when I was working on the module mMsgBox, I prefixed it with A_ so that it appeared at the top of the list. It's not such a big deal but it definately makes it easier to find. Did I mention how lazy I was?

Figure 10 subMsgBox in use.



You can do the same with a userform to make an InputBox. Here's a gander of mine in the designer.

11 My InputBox UserForm



Note the colours again! I'm trying to be friendly and jolly and consistent!

Some of the "important" things about this is that if you for example display an error number and its description, **YOU CAN COPY THE MESSAGE!** Hehehe. You can also show the "message" form where you want. Can't think of anything else at the moment.

Debugging and Tracing

While we are on the subject of debugging, which we weren't really but what the hell, a nice thing to have sometimes is a trace report of the procedures that have been passed through maybe even with times spent inside them.

We know we can add any code we like to procedures now, so we can insert a debug.print at the top and bottom of procedures and also a time at the top and bottom and time spent.

This would look something like:

Put this code in a new module and run subS1.

43 Tracing example 1

	Option Explicit
	Sub subS1()
	Debug.Print "Start subS1 " & Timer()
	subS2
	subS3
	Debug.Print "End subS1 " & Timer()
	'

	End Sub
	Sub subS2()
	Debug.Print "Start subS2 " & Timer()
	subS4
	Debug.Print "End subS2 " & Timer()
	'

	End Sub
	Sub subS3()
	Dim s1Str As String
	Debug.Print "Start subS3 " & Timer()

	sIstr = fncF1()
	subS4
	Debug.Print "End subS3 " & Timer()
	'

	End Sub
	Sub subS4()
	Debug.Print "Start subS4 " & Timer()
	Debug.Print "End subS4 " & Timer()
	'

	End Sub
	Function fncF1()
	Debug.Print "Start fncF1 " & Timer()
	Debug.Print "End fncF1 " & Timer()
	'

	End Function

With no changes, the code will execute so quickly the Timer() value would be practically the same for all of the debug.prints as you can see from the following copied from the immediate window.

	Start subS1 56021.58
	Start subS2 56021.59
	Start subS4 56021.59
	End subS4 56021.59
	End subS2 56021.59
	Start subS3 56021.59
	Start fncF1 56021.59
	End fncF1 56021.59
	Start subS4 56021.59
	End subS4 56021.59
	End subS3 56021.6
	End subS1 56021.6

Let's add a sub to slow things down a bit. A call to this sub goes into each of the other subs and the function.

44 subSpendTime

Sub subSpendTime()
Dim lngIN As Long
For lngIN = 1 To 10000
DoEvents
Next lngIN
'

End Sub

You don't need debug.prints in the slowdown sub.

After running subS1 a few times, you'll find that the immediate window gets a bit hard to follow.

Hang on though... we wrote a procedure to clear that didn't we?! Let's put that at the top of our code!

45 Tracing Example 2

Sub subS1()
Debug.Print "Start subS1 " & Timer()
subSpendTime
subClearIW
subS2
subS3
Debug.Print "End subS1 " & Timer()
'

End Sub

That's better!

Hmmmm. If we have a lot of subs though it's going to be a real pain to write all of those debug.print and subSpendTime lines in at the top and bottom of each one. But we know we can add code with code so why don't we do that!

Great!

But how do we page through each procedure in a module? Interesting.

We know how to go through the modules from the stuff we wrote for frmMsgBox when we looked through all the project modules for the name frmMsgBox. We know we can use cWhereAreWe to get the number of lines of code in procedures. But where do we start?

In the frmMsgBox code we looked at the Component.Name to see if we had our form in the project so we know we can access the component and therefore the code module.

I think I've said before that VBA VBE code is **MODULE** oriented so you can't just go from one procedure to the next even if you know the names. You can however get the number of declaration lines in a module and the total count of module lines using:

```
Application.VBE.ActiveCodePane.CodeModule.CountOfLines
```

And:

```
Application.VBE.ActiveCodePane.CodeModule.CountOfDeclarationLines
```

So, we can get the last line. With **ProcOfLine** we can get the name of the procedure and vbext_pk_Proc . Then use those in **ProcCountLines** to get how many lines there are in it. This will give us the end line number of the above procedure! Even after we've added code!

Let's use this to page through all the procs in a module. That's why I asked you to put the last set of code in a new module. We're only going to use the active code pane. This is in fact the code for a **MODULE**.

The following will list all of the procedures in a module that is showing in the code pane. We could actually go forward or down the module, but since this is part of a procedure that will add code the module and if we plan to insert code in a module then it's better if we go **UP** the module so as to preserve line numbers.

Most of this is taken **DIRECTLY** from cWhereAreWe.

Run it by going to the new module and using the macros (shudder) menu.

46 subListProcsToImediateWindow

	Sub subListProcsToImediateWindow()
	Dim lngCountOfDeclarationLines As Long
	Dim lngOriginalModuleLineCount As Long

Dim lngCurrentModuleLine As Long
Dim vbplProject As VBIDE.VBProject
Dim vbclComponent As VBIDE.VBComponent
Dim vbcmCodeModule As VBIDE.CodeModule
Dim vbcplCodePane As VBIDE.CodePane
Dim slProjectNameName As String
Dim slModuleName As String
Dim slProcedureName As String
Dim lngProcCountLines As Long
Set vbplProject = Application.VBE.ActiveVBProject
slProjectNameName = vbplProject.Name
Set vbcplCodePane = Application.VBE.ActiveCodePane
Set vbcmCodeModule = vbcplCodePane.CodeModule
slModuleName = vbcmCodeModule.Name
slModuleName = vbcplCodePane.CodeModule.Parent.Name
Set vbclComponent = vbplProject.VBComponents(slModuleName)
lngCountOfDeclarationLines = vbcmCodeModule.CountOfDeclarationLines
lngOriginalModuleLineCount = vbcmCodeModule.CountOfLines
lngCurrentModuleLine = lngOriginalModuleLineCount
Do
If lngCurrentModuleLine <= lngCountOfDeclarationLines Then
Exit Sub
End If
slProcedureName = vbcmCodeModule.ProcOfLine(lngCurrentModuleLine, vbext_pk_Proc)
Debug.Print slProcedureName
lngProcCountLines = vbcmCodeModule.ProcCountLines(slProcedureName, vbext_pk_Proc)
lngCurrentModuleLine = lngCurrentModuleLine - lngProcCountLines

	Loop
	' *****'
	End Sub

Here's the result:

	subSpendTime	11	44
	fncF1	6	38
	subS4	6	32
	subS3	12	20
	subS2	8	12
	subS1	11	1

Now then, we want to Insert Lines of code. That last procedure is a bit tricky because the last line of the module **is NOT End Sub**. We have to check the line and go backwards looking for **End**. In fact, it doesn't hurt and we may as well do this for all of the procedures. What about the top of the procedure though, especially using my parameter set up? Ideally, we want to debug.print just before any of the other code in the procedure. If these lines are going to be temporary then we can insert directly after ProcBodyLine.

Before we insert anything we should really delete anything that's there already.

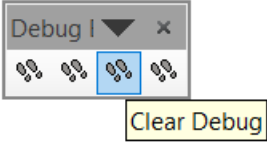
Here's a picture of using the buttons we set up earlier, from a different project of course, to do just that.

```
Option Explicit

Sub subS1 ()

subS2
subS3

' *****'
End Sub
Sub subS2 ()
```



The screenshot shows a code editor window with a 'Debug' menu open. The menu has a dropdown arrow and a close button 'x'. Below the menu are four icons representing different debug actions. The third icon, which appears to be a pair of scissors, is highlighted with a blue selection box. Below the icons is a 'Clear Debug' button. The code in the background shows the end of a 'Sub subS1 ()' procedure and the start of a 'Sub subS2 ()' procedure.

And here's the code to insert trace lines:

47 subAddTraceLinesToAModule

Sub subAddTraceLinesToAModule()
Dim lngCountOfDeclarationLines As Long
Dim lngOriginalModuleLineCount As Long
Dim lngCurrentModuleLine As Long
Dim vbplProject As VBIDE.VBProject
Dim vbclComponent As VBIDE.VBComponent
Dim vbcmlCodeModule As VBIDE.CodeModule
Dim vbcplCodePane As VBIDE.CodePane
Dim slProjectNameName As String
Dim slModuleName As String
Dim slProcedureName As String
Dim lngProcCountLines As Long
Dim lngInsertBottom As Long
Dim lngInsertTop As Long
Dim slInsertTop As String
Dim slInsertBottom As String
Dim slDQ As String
Dim slLine As String
Dim lngEndDefinitionLine As Long
slDQ = Chr(34)
Set vbplProject = Application.VBE.ActiveVBProject
slProjectNameName = vbplProject.Name
Set vbcplCodePane = Application.VBE.ActiveCodePane
Set vbcmlCodeModule = vbcplCodePane.CodeModule
slModuleName = vbcmlCodeModule.Name
slModuleName = vbcplCodePane.CodeModule.Parent.Name
Set vbclComponent = vbplProject.VBComponents(slModuleName)

	InglCountOfDeclarationLines = vbcmlCodeModule.CountOfDeclarationLines
	InglOriginalModuleLineCount = vbcmlCodeModule.CountOfLines
	InglCurrentModuleLine = InglOriginalModuleLineCount
	Do
	If InglCurrentModuleLine <= InglCountOfDeclarationLines Then
	Exit Sub
	End If
	slProcedureName = vbcmlCodeModule.ProcOfLine(InglCurrentModuleLine, vbext_pk_Proc)
	InglInsertBottom = InglCurrentModuleLine
	slInsertBottom = "Debug.Print " & sIDQ & "End Of " & slProcedureName & " " & sIDQ & " & Timer() "
	slInsertTop = "Debug.Print " & sIDQ & "Start Of " & slProcedureName & " " & sIDQ & " & Timer() "
	InglProcCountLines = vbcmlCodeModule.ProcCountLines(slProcedureName, vbext_pk_Proc)
	InglCurrentModuleLine = InglCurrentModuleLine - InglProcCountLines
	Do
	slLine = vbcmlCodeModule.Lines(InglInsertBottom, 1)
	If Left\$(slLine, 4) = "End " Then
	Exit Do
	End If
	InglInsertBottom = InglInsertBottom - 1
	Loop
	vbcmlCodeModule.InsertLines InglInsertBottom, slInsertBottom
	InglInsertTop = vbcmlCodeModule.ProcBodyLine(slProcedureName, vbext_pk_Proc)
	Do
	slLine = vbcmlCodeModule.Lines(InglInsertTop, 1)
	If Right\$(slLine, 1) <> "_" Then
	InglInsertTop = InglInsertTop + 1
	Exit Do

End If
lngInsertTop = lngInsertTop + 1
Loop
vbcmlCodeModule.InsertLines lngInsertTop, slInsertTop
Loop
' *****
End Sub

And the code after running the above:

Option Explicit
Sub subS1()
Debug.Print "Start Of subS1 " & Timer()
subS2
subS3
' *****
Debug.Print "End Of subS1 " & Timer()
End Sub
Sub subS2()
Debug.Print "Start Of subS2 " & Timer()
subS4
' *****
Debug.Print "End Of subS2 " & Timer()
End Sub
Sub subS3()
Debug.Print "Start Of subS3 " & Timer()
Dim slStr As String

	sIstr = fncF1()
	subS4
	' *****
	Debug.Print "End Of subS3 " & Timer()
	End Sub
	Sub subS4()
	Debug.Print "Start Of subS4 " & Timer()
	' *****
	Debug.Print "End Of subS4 " & Timer()
	End Sub
	Function fncF1()
	Debug.Print "Start Of fncF1 " & Timer()
	' *****
	Debug.Print "End Of fncF1 " & Timer()
	End Function
	Sub subSpendTime()
	Debug.Print "Start Of subSpendTime " & Timer()
	Dim lngIN As Long
	For lngIN = 1 To 10000
	DoEvents
	Next lngIN
	' *****
	Debug.Print "End Of subSpendTime " & Timer()
	End Sub

Right. We've got all of this trace code in now that prints to the immediate window. We could just as easily have executed a procedure that logs to a file. As it stands only the procedure name is output but we could also have the module name and line number if we wanted. We could even implement a Stack, and indeed a lot of third party programs do exactly that.

Timer() is a bit crude. We could have a function there that prints out the time spent in the procedure rather than have to work it out. There's a high resolution timer class in the appendices that could be used as well but that is probably a bit of overkill for normal processes.

I think you get the idea though.

We can Add, Delete and Replace code lines in procedures in the VBE but it's essential we know where we are.

Once we're done with all of this debug.printing what are we going to do. Delete it of course! We can use either our debug extra command bar or we can "wrap" the procedure subccDeleteDebugPrint in code similar to subListProcsToImmediateWindow. Within that, instead of printing the procedure name run subccDeleteDebugPrint.

But, you say, that only works if the cursor is actually in the procedure.

You're right.

We know a module line number so we can use SetSelection to actually go to a place in the code.

```
Application.VBE.ActiveCodePane.SetSelection lngCurrentModuleLine, 1, lngCurrentModuleLine, 1
```

There are a number of things of interest here. To delete all of the Debug.Print lines in a module we could:

1.	Loop through the module procedures, go to that procedure and run subccDeletDebugPrint
2.	Loop through all of the module lines, going to them and when we get to a different Procedure, run subccDeletDebugPrint
3.	Loop through all the module lines and run subccDeletDebugPrint for every line
4.	Loop through all of the module lines and delete any that have Debug.Print in them

And of course we could extend this for a whole project or all of the open projects.

A lot of the following is taken directly from code we've already written.

Here's the code for number one:

48 subDeletDebugPrintForModule1

	Sub subDeletDebugPrintForModule1()
	Dim lnglCountOfDeclarationLines As Long
	Dim lnglOriginalModuleLineCount As Long
	Dim lnglCurrentModuleLine As Long
	Dim vbplProject As VBIDE.VBProject
	Dim vbclComponent As VBIDE.VBComponent
	Dim vbcmlCodeModule As VBIDE.CodeModule
	Dim vbcplCodePane As VBIDE.CodePane
	Dim slProjectNameName As String
	Dim slModuleName As String
	Dim slProcedureName As String
	Dim lnglProcCountLines As Long
	Set vbplProject = Application.VBE.ActiveVBProject
	slProjectNameName = vbplProject.Name
	Set vbcplCodePane = Application.VBE.ActiveCodePane
	Set vbcmlCodeModule = vbcplCodePane.CodeModule
	slModuleName = vbcmlCodeModule.Name
	slModuleName = vbcplCodePane.CodeModule.Parent.Name
	Set vbclComponent = vbplProject.VBComponents(slModuleName)
	lnglCountOfDeclarationLines = vbcmlCodeModule.CountOfDeclarationLines
	lnglOriginalModuleLineCount = vbcmlCodeModule.CountOfLines
	lnglCurrentModuleLine = lnglOriginalModuleLineCount
	Do
	Get out?
	If lnglCurrentModuleLine <= lnglCountOfDeclarationLines Then
	Exit Sub
	End If
	slProcedureName = vbcmlCodeModule.ProcOfLine(lnglCurrentModuleLine, vbext_pk_Proc)

Go to the linenumber in the module	
	Application.VBE.ActiveCodePane.SetSelection lngCurrentModuleLine, 1, lngCurrentModuleLine, 1
	lngProcCountLines = vbcmCodeModule.ProcCountLines(sIProcedureName, vbext_pk_Proc)
Set the linenumber to the top of the next procedure. Don't forget we're going UP the module to preserve line numbers!	
	lngCurrentModuleLine = lngCurrentModuleLine - lngProcCountLines
Here's where we make use of the procedure we've already written.	
	subccDeletDebugPrint
	Loop
	' *****
	End Sub

Here's the code for number two:

Guess what the following code is for:

Aaaaannnd, here's the code for number four:

I think you can see that it's possible to also insert highly customized error code too. This could include getting a telephone number from a source and displaying it for support if anything was up. That could be changed depending who was "on duty" as it were. That means a support person could be contacted whenever! Done that, Been there. It worked well.

In fact, we can insert anything we want to!

The Registry

There are some items we've dealt with that are hard coded. These were the prefixes and scope letters used in the naming convention. I keep these in an INI file and have written procedures to retrieve them. I'm **NOT** putting that code here though so you'll have to roll your own. You can even download ready made classes to do that stuff. Lots of stuff out there. I use an INI file because it's easy to edit and very portable as a text file. In fact, MZ-Tools uses an INI file as well as using the registry. Well it does for the free version 3 anyway. Having said that, INI files are a bit crude, and registry access because it's in memory is probably faster. MZ-Tools 8 has embraced XML files.

It sort of highlights though, the need to set and retrieve relatively static data sometimes.

Did you know that VBA has its own special key in the registry and its own set of four built in calls to set values there and get values from there?

It's:

49 Registry key for VBA

Computer\HKEY_CURRENT_USER\Software\VB and VBA Program Settings\
--

If you open the registry and look for this key you may not actually find it. It's created at the first SaveSettings call.

Using the four "native" VBA internal calls makes a lot of working with the registry pretty easy. The calls are:

50 VBA Registry calls

For....
AppName = "Mike&Lisa"
Section = "Registgry"
Key = "Setting A"
Setting = "Page 132"
SaveSetting AppName,Section,Key,Setting SaveSetting "Mike&Lisa","Registgry","Setting A","Page 132"
Writes to/Creates
HKEY_CURRENT_USER\Software\VB and VBA Program Settings\Mike&Lisa\Registry\Setting A\

	Value = 132
	Variable = GetSetting (AppName,Section,Key[,Default])
	Gets the value or key from
	HKEY_CURRENT_USER\Software\VB and VBA Program Settings\Mike&Lisa\Registry\Setting A\ A\
	Variable = 132
	DeleteSettings AppName[,Section[,Key]]
	Deletes a key or section from
	HKEY_CURRENT_USER\Software\VB and VBA Program Settings\Mike&Lisa\Registry\Setting A\ A\
	Variant = GetAllSettings (AppName,Section)
	Gets a list of key settings and their values from
	HKEY_CURRENT_USER\Software\VB and VBA Program Settings\Mike&Lisa\Registry\Setting A\ A\
	Variant = 132

Depending on what's in Mike&Lisa\Registry, Variant may return an array. Hence, it's a er, variant.

There's a non problem here In my not so very humble opinion. If you're going to save stuff in the registry then these four calls are simple and mostly all you need to use. Everything is below a fixed key you can find and check with Regedit and so on and so on.

However, it's **not** free access to the registry. That is possible with API calls though. There's a **LOT** out there on the world wide web about that. But why would you want to do that? If you want to put your data for your application or program in an obscure registry place that no one can find, and I can see some people would want to do that, use the API calls. You may also want to retrieve a value that belongs to some application. Otherwise these four are really all you need for your VBA application.

The documentation says that it's possible, though I don't advise it, to store a massive 1Mb of data in a registry value. I don't advise it because if you need to store that much data then you probably need to rethink why you want to store that much.

This is interesting. Open the immediate window and type or copy/paste:

51 SaveSetting statement

```
SaveSetting "Your Name","Your Name Section","Your Name Key","Your Address"
```

I think you can work out what to put where where.

Now open up regedit and search for "Your Name". Got it? Okay. Hit F3 to search again.

You'll find **two** entries.

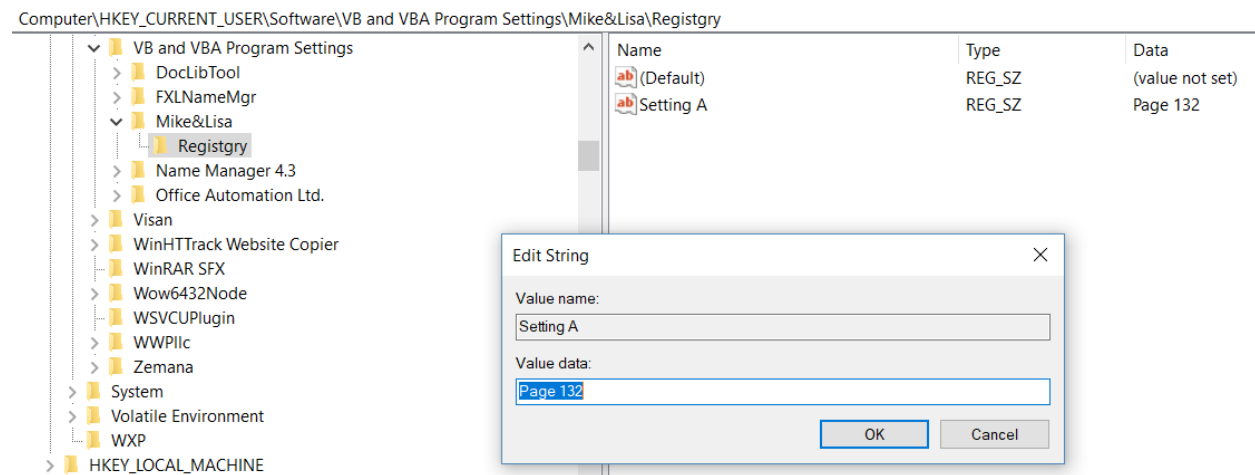
Delete the first one.

Search again.

You won't find the other entry. The second entry was "cloned". If you want the whys and wherefores of the registry google is your friend.

Anyroad, here's a pic of the registry after I've run the SaveSetting line above from the immediate window.

12 Registry after SaveSetting



Line Numbers

A lot of people depend on line numbers to give an exact place for where code goes wrong. We can insert line numbers a lot of ways. One of the most popular, is to use an add-in for VBA. One such is [MZ-Tools](#). The paid for version is version 8, and that version will allow you to set a start line number and an increment.

Version 3 however, the "free" one, will put line numbers into a procedure or a complete module starting at 10 and incrementing by 10 for each procedure. Hmm, I remember that from GWBASIC a while back. Not gonna say when, give my age away! Hehehe. If there is an error in the code then the line number can be printed in an error message with the ERL function, **but**, only if the line is numbered.

MZ-Tools adding line numbers to procedures is fine. It will also add error code that prints out the module and procedure and you can change the error code to add ERL and anything else. Highly customizable.

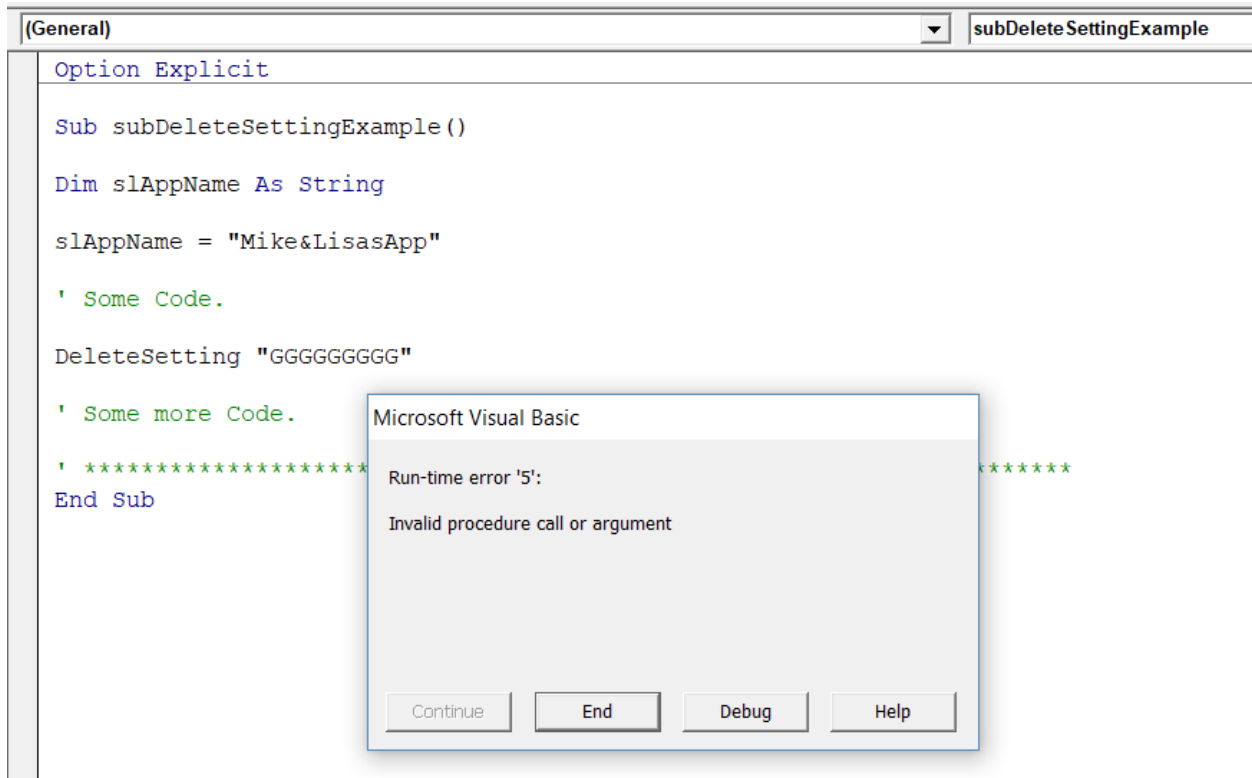
We can add line numbers programmatically as well. Leaving aside MZ-Tools for the nonce, if we want to add line numbers programmatically, we don't have to start at 10 and increment by 10. We've said a few times now that VBA works at the codemodule level with module line numbers. We can add line numbers of the module to a procedure. Reporting the ERL then gives the exact line which is going wrong of the **MODULE**.

It's not very widely known, but you don't actually need to put line numbers on all lines of a procedure. Just putting a line number on a line that maybe will go wrong will work. This means we can number lines in different modules with unique line numbers if we want. For example, things aren't likely to go wrong with ...

```
sName = ""
```

Now! Using the registry functions we talked about earlier, If a key isn't there then DeleteSettings will return an error number 5.

13 Error on DeleteSetting



But, you don't see any line number being reported.

If we use MZ-Tools to add error code and line numbers we get something like:

52 subDeleteSettings example

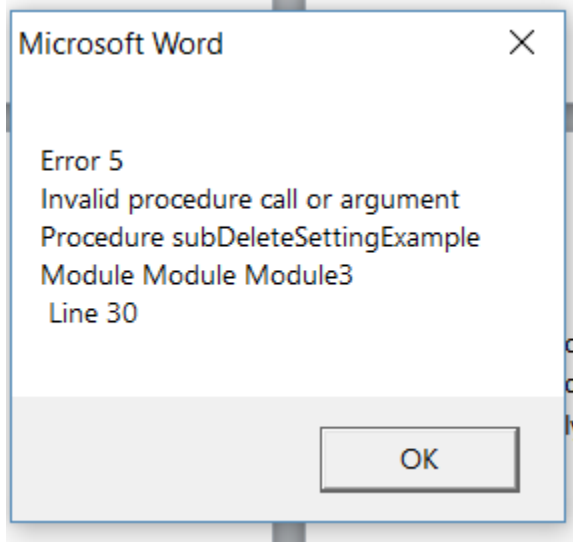
	Option Explicit
	Sub subDeleteSettingExample()
	Dim sAppName As String
10	On Error GoTo subDeleteSettingExample_Error
20	sAppName = "Mike&LisasApp"

	' Some Code.
	30 DeleteSetting slAppName
	' Some more Code.
	' *****
	40 On Error GoTo 0
	50 Exit Sub
	subDeleteSettingExample_Error:
	60 MsgBox "Error " _
	& Err.Number _
	& vbCrLf _
	& Err.Description _
	& vbCrLf _
	& "Procedure subDeleteSettingExample" _
	& vbCrLf _
	& "Module Module Module3" _
	& vbCrLf _
	& " Line " & Erl
	End Sub

You'll see that ERL is on the last line of the MsgBox continuations. This isn't in the standard code inserted by MZ-Tools. In fact, I've altered the whole of the MsgBox line in my copy of MZ-Tools as above.

The error comes out as...

14 Error message from handler code



Line 30 is:

```
30 DeleteSetting slAppName
```

Say we have a large module with lots of procs in it, which actually is my experience of many VBA users. It can take a while to navigate to that particular line.

Aside: MZ-Tools will add line numbers to the complete module if the **MODULE** is highlighted/selected in the project explorer.

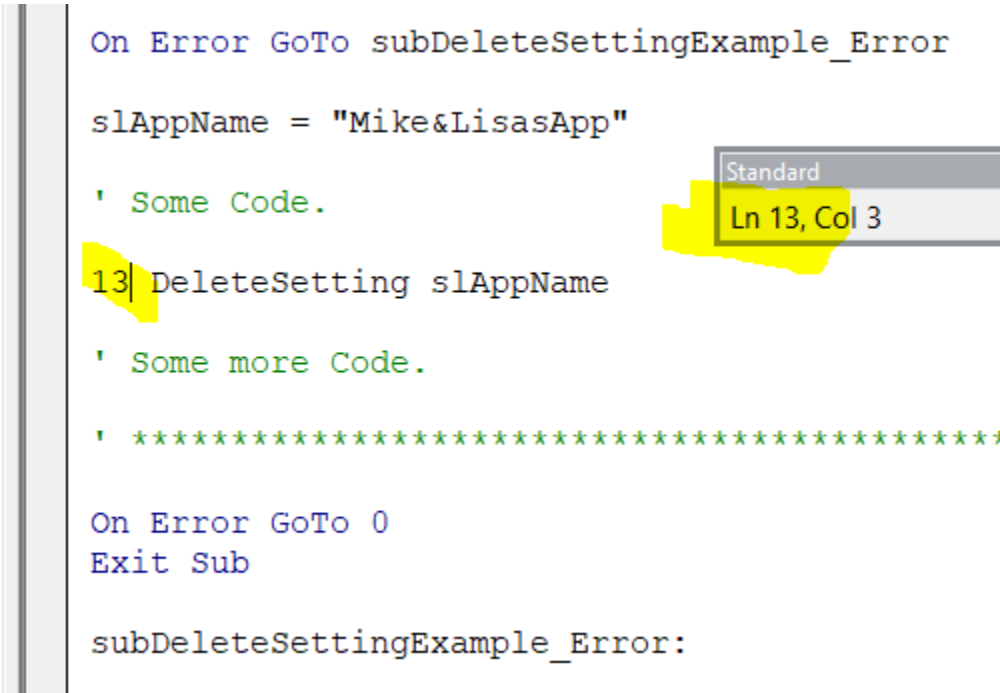
What if we did this:

15 Module line number on code line

```
On Error GoTo subDeleteSettingExample_Error
slAppName = "Mike&LisasApp"
' Some Code.
13 DeleteSetting slAppName
' Some more Code.
' *****

On Error GoTo 0
Exit Sub

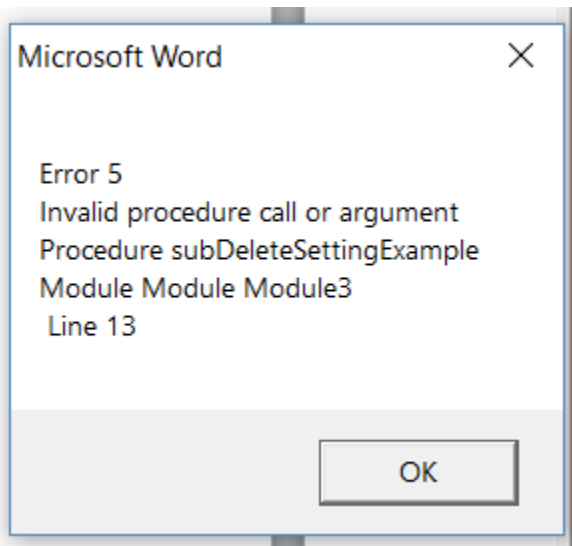
subDeleteSettingExample_Error:
```



I've highlighted the important bits. I've removed the line numbers and just put the **MODULE line number** on the line that may error.

Now the error message looks like...

16 Error message with module line number



And we can do even more! We know the project is the active one, we know the module, it's even in the message, we know the line number. In subDeletDebugPrintForModule we used...

```
Application.VBE.ActiveCodePane.SetSelection  
InglCurrentModuleLine, 1,  
InglCurrentModuleLine, 1
```

... To **go to** a procedure, so we could check things out there.

We can use that!

53 subGoToLine

```
Sub subGoToLine( _  
    spModuleName As String, _  
    lngLineNumber As Long, _  
    Optional vbppProject As VBIDE.VBProject _  
)  
  
Dim vbplProject As VBIDE.VBProject  
Dim vbcmlCodeModule As VBIDE.CodeModule
```

	Dim vbcplCodePane As VBIDE.CodePane
	If vbppProject Is Nothing Then
	Set vbplProject = Application.VBE.ActiveVBProject
	Else
	Set vbplProject = vbppProject
	End If
	Set vbcmlCodeModule = vbplProject.VBComponents(spModuleName).CodeModule
	vbcmlCodeModule.CodePane.Show
	Set vbcplCodePane = Application.VBE.ActiveCodePane
	vbcplCodePane.SetSelection lngplLineNumber, 1, lngplLineNumber, 1
	'

	End Sub

Those who know VBA a little, will realize that the Project is last in the list of parameters because it is optional.

We can put an inputbox or even an fnclnputBox there if we like for the line number, or we can feed the data straight into the sub from the error routine. Lots of options!

Cool!

Let's get back to module line numbers.

It's just not possible to automate picking out lines that may error or not.e to that. Adding module line numbers to a complete procedure though is a different beast.

From as far back as subccSortDims, we know we can pick up each line in a procedure. We know that some lines definately don't need a line number:

- Comment lines above the definition line
- The Definition line
- Blank Lines
- Comment lines in the code
- Dim etc lines
- Definition lines
- The last line

This goes with what we said about not all lines in a procedure needing line numbers.

We can test for all of those.

Some we don't need to. Definition lines do not have a line number. Dim will be okay with a line number but comments and blank lines with line numbers seem to confuse VBA. So, we'll zip past them!

For this I've added a few Do..Loops to cWhereAreWe along with the appropriate local and module variables and Get Procedures to report...

1	First Dim Line
2	Last Dim Line Plus 1
3	Line After the definition Plus 1
4	Whether there is code between first and last Dim lines

	InglFirstDimLine = 0
	InglCLine = InglModuleProcBodyLine
	Do
	sLine = Trim\$(vbcmlCodeModule.Lines(InglCLine, 1))
	If Left\$(sLine, 4) = "Dim " Then
	InglFirstDimLine = InglCLine
	Exit Do
	Elseif Left\$(sLine, 6) = "Const " Then
	InglFirstDimLine = InglCLine
	Exit Do
	Elseif Left\$(sLine, 7) = "Static " Then
	InglFirstDimLine = InglCLine
	Exit Do
	End If
	InglCLine = InglCLine + 1
	If InglCLine >= InglModuleProcEndLine Then

	Exit Do
	End If
	Loop
	InglLastDimLinePlus1 = 0
	InglCLine = InglModuleProcEndLine
	Do
	sLine = Trim\$(vbcmlCodeModule.Lines(InglCLine, 1))
	If Left\$(sLine, 4) = "Dim " Then
	InglLastDimLinePlus1 = InglCLine + 1
	Exit Do
	Elseif Left\$(sLine, 6) = "Const " Then
	InglLastDimLinePlus1 = InglCLine + 1
	Exit Do
	Elseif Left\$(sLine, 7) = "Static " Then
	InglLastDimLinePlus1 = InglCLine + 1
	Exit Do
	End If
	InglCLine = InglCLine - 1
	If InglCLine <= InglModuleProcBodyLine Then
	Exit Do
	End If
	Loop
	InglLineAfterDefinitionPlus1 = 0
	InglCLine = InglModuleProcBodyLine
	Do
	sLine = Trim\$(vbcmlCodeModule.Lines(InglCLine, 1))
	If Right\$(sLine, 1) <> "_" Then

	If lnglCLine = lnglModuleProcBodyLine Then
	lnglLineAfterDefinitionPlus1 = lnglCLine + 1
	Else
	lnglLineAfterDefinitionPlus1 = lnglCLine
	End If
	Exit Do
	End If
	lnglCLine = lnglCLine + 1
	Loop
	bInlNonConsecutiveDims = False
	lnglCLine = lnglLastDimLinePlus1 - 1
	Do
	sLine = Trim\$(vbcmlCodeModule.Lines(lnglCLine, 1))
	If Left\$(sLine, 4) = "Dim " Then
	Elseif Left\$(sLine, 6) = "Const " Then
	Elseif Left\$(sLine, 7) = "Static " Then
	Elseif Left\$(sLine, 1) = "" Then
	Elseif Len(sLine) = 0 Then
	Else
	bInlNonConsecutiveDims = True
	Exit Do
	End If
	lnglCLine = lnglCLine - 1
	If lnglCLine <= lnglLineAfterDefinitionPlus1 - 1 Then
	Exit Do
	End If
	Loop

Now we can start at the end of the Dims if there isn't any code in between them till the end of the procedure. This means we don't have to test for Dim/Const/Static. And don't forget we have code to move all of the Dims together at the top!

Here's the code to insert MODULE line numbers.

54 subccInsertModuleLineNumbersInProcedure

	Sub subccInsertModuleLineNumbersInProcedure()
	' Insert line numbers of the MODULE into the
	' current procedure.
	'
	Dim lngEndLine As Long
	Dim lngStartLine As Long
	Dim lngELine As Long
	Dim lngSLine As Long
	Dim lngECol As Long
	Dim lngSCol As Long
	Dim sSelection As String
	Dim lngCurrentModuleLine As Long
	Dim cWhereAreWe As cWhereAreWe
	Dim lngModuleEndLine As Long
	Dim sLine As String
	Dim sLineArray() As String
	Dim sLOLine As String
	Dim vbcmlCodeModule As VBIDE.CodeModule
	Dim vbcplCodePane As VBIDE.CodePane
	Dim lngCLine As Long
	Dim sTrimLine As String
	Set cWhereAreWe = New cWhereAreWe
	lngModuleEndLine = cWhereAreWe.ModuleProcEndLine
	Set vbcmlCodeModule = cWhereAreWe.CodeModule

Set vbcplCodePane = clWhereAreWe.CodePane
vbcplCodePane.GetSelection lnglCurrentModuleLine, lnglSCol, lnglELine, lnglECol
' Is there a selection?
slLine = clWhereAreWe.CurrentLine
slSelection = clWhereAreWe.Selection
lnglSLine = clWhereAreWe.CurrentModuleLine
lnglSCol = clWhereAreWe.SCol
lnglELine = clWhereAreWe.ELine
lnglECol = clWhereAreWe.ECol
If Len(slSelection) > 0 Then
lnglStartLine = lnglSLine
lnglEndLine = lnglELine
Else
lnglStartLine = clWhereAreWe.LastDimLinePlus1
lnglEndLine = lnglModuleEndLine
End If
' Start replacing lines.
lnglCLine = lnglStartLine
Do
' End of sub?
If lnglCLine >= lnglEndLine Then
Exit Do
End If
slOLine = vbcmlCodeModule.Lines(lnglCLine, 1)
slTrimLine = Trim\$(slOLine)
If Len(Trim\$(slTrimLine)) = 0 Then
Elseif Left\$(slTrimLine, 1) = "" Then

	Else
	' Does this line already have a number?
	sILineArray = Split(sITrimLine, " ")
	If Not IsNumeric(sILineArray(0)) Then
	' Replace line.
	vbcmlCodeModule.ReplaceLine lngIcLine, CStr(lngIcLine) & " " & sIOLine
	End If
	End If
	lngIcLine = lngIcLine + 1
	Loop
	'

	End Sub

And guess what we do when we don't want them.

55 subccDeleteLineNumbersInProcedure

	Sub subccDeleteLineNumbersInProcedure()
	' Delete line numbers
	'
	Dim lngIEndLine As Long
	Dim lngIStartLine As Long
	Dim lngIcLine As Long
	Dim lngISLine As Long
	Dim lngIECol As Long
	Dim lngISCol As Long
	Dim sISelection As String

Dim lngCurrentModuleLine As Long
Dim clWhereAreWe As cWhereAreWe
Dim lngModuleEndLine As Long
Dim slLine As String
Dim slLineArray() As String
Dim slOLine As String
Dim vbcmlCodeModule As VBIDE.CodeModule
Dim vbcplCodePane As VBIDE.CodePane
Dim lngCLine As Long
Dim slTrimLine As String
Dim lngLastDimLinePlus1 As Long
Set clWhereAreWe = New cWhereAreWe
lngModuleEndLine = clWhereAreWe.ModuleProcEndLine
Set vbcmlCodeModule = clWhereAreWe.CodeModule
Set vbcplCodePane = clWhereAreWe.CodePane
vbcplCodePane.GetSelection lngCurrentModuleLine, lngSCol, lngELine, lngECol
' Is there a selection?
slLine = clWhereAreWe.CurrentLine
slSelection = clWhereAreWe.Selection
lngSLine = clWhereAreWe.CurrentModuleLine
lngSCol = clWhereAreWe.SCol
lngELine = clWhereAreWe.ELine
lngECol = clWhereAreWe.ECol
lngLastDimLinePlus1 = clWhereAreWe.LastDimLinePlus1
If lngLastDimLinePlus1 = 0 Then
lngLastDimLinePlus1 = clWhereAreWe.LineAfterDefinitionPlus1
End If
If Len(slSelection) > 0 Then
lngStartLine = lngSLine
lngEndLine = lngELine

Else
InglStartLine = lngLastDimLinePlus1
InglEndLine = lngModuleEndLine
End If
' Start replacing lines.
InglCLine = lngStartLine
Do
' End of sub?
If lngCLine >= lngEndLine Then
Exit Do
End If
sIOLine = vbcmCodeModule.Lines(InglCLine, 1)
sITrimLine = Trim\$(sIOLine)
If Len(Trim\$(sITrimLine)) = 0 Then
Elseif Left\$(sITrimLine, 1) = "" Then
Else
' Does this line already have a number?
sLLineArray = Split(sITrimLine, " ")
If IsNumeric(sLLineArray(0)) Then
sLLine = Mid\$(sIOLine, Len(sLLineArray(0)) + 2)
' Replace line.
vbcmCodeModule.ReplaceLine lngCLine, sLLine
End If
End If

	InglCLine = InglCLine + 1
	Loop
	' ***** *****
	End Sub

Recap number two

Let's have a look at the code we've built or built so far.

1	subccSortDims
2	subMsgBox
3	subccSortSelectedDims
4	fncGuessVarType
5	subWhereAreWe
6	cNameExample
7	subInsertGetProperties
8	cWhereAreWe
9	subccSplitAllDims
10	subccInsertSelectionDebug
11	subccDeleteDebugPrint
12	subClearIW
13	cBarEvents
14	subBrandNewBarAndButton
15	subBookMarkAndBreakpoint
16	frmMsgBox
17	subListProcsToImmediateWindow
18	subAddTraceLinesToAModule
19	subDeleteDebugPrintForModule
20	subGoToLine
21	subccInsertModuleLineNumbersInProcedure
22	subccDeleteLineNumbersInProcedure

That's quite a lot!

Not finished yet though! Hehehehe.

It's fairly clear from the names in the above list which are Subs, which are Functions, which are classes, and we can see there's a UserForm in there as well.

Hang on though... what's this cc business?

I'm glad you asked that!

If you write a lot of procedures and run them from the Macros (waves arms to ward the evil off) menu, then you may find yourself scrolling down that list to get to the one you want. A lot. It would be nice if the one you are testing or using often was at the top. That list is in alphabetic order. But you can't realistically pick the procedure name to put items at the top.Or can you?

Well actually you can. Sort of. You can change the name of the procedure you're working on to a temporary name that puts it at the top of the list. Alternatively, you can add **another** procedure that **calls** your procedure but has a name that puts it it the top of the list. This ensures that the procedure you're working on will always have the same name. Another tip is to position the procedure you are working on at the bottom or top of its module. Getting to it is pretty simple then. Ctrl Home, Ctrl Down Arrow for example. Then again, **USE THE BOOKMARKS LUKE!**

Add a new module. For each procedure with "cc" in the name, insert a procedure in the new module to run it and prefix the procedure name so they are listed **in an order you like/want** alphabetically.

For example, here's a portion of my module called `aamTopOfMacros`. Yes. That name ensures it appears at the top of the explorer list.

56 `aamTopOfMacros`

	Sub A_subccArrangeMacros()
	frmArrangeMacroMenu.Show
	' *****
	End Sub
	Sub A_subccCodeCodeForm()
	frmCodeCode.Show
	' *****
	End Sub
	Sub A_subccShutDown()
	subccShutDown
	' *****
	End Sub
	Sub AA_subccBackMeUp()
	subccBackMeUp
	' *****
	End Sub
	Sub AB_subccDeleteDebugCode()
	subccDeleteDebugCode
	' *****
	End Sub

```

Sub AC_subccGoToBottomOfDims()
subccGoToBottomOfDims
! *****
End Sub
Sub AD_subccInsertDims()
subccInsertDims
! *****
End Sub
Sub AE_subccCleanProcedure()
subccCleanProcedure
! *****
End Sub
Sub AF_subccXRefProcedure()
subccXRefProcedure
! *****
End Sub
Sub AG_subccInsertSelectionDebug()
subccInsertSelectionDebug
! *****
End Sub
Sub AH_subccInsertErrorCodeAtLine()
subccInsertErrorCodeAtLine
! *****
End Sub
Sub AI_subccSortDims()
subccSortDims
! *****
End Sub
Sub AJ_subccSmartIndent()
Application.VBE.CommandBars("Code Window").Controls("&Smart Indent").Controls("Indent
&Procedure").Execute
! *****
End Sub
Sub AK_subccDeleteInBetweenLines()
subccDeleteInBetweenLines
! *****
End Sub

```

	Sub AL_subccInsertComments()
	frmInsertComments.Show
	' *****
	End Sub
	Sub AM_subccRemoveEndOfLineComment()
	subccRemoveEndOfLineComment
	' *****
	End Sub

The names will then appear in the (wispers) macros menu with the ones you want at the top, at the top.

None of this is rocket science. But be aware the the procedures called from the procedures will also appear later in the macros (AAAARRRGHHH) list. So, you are doubling up. So what.

And I use a userform to arrange items.

You can see from the pictures below that I load all of the procedure names with cc in them to the left list box. From there I can add them to or remove them from the list on the right. This sort of methodology with two listboxes is well documented and there are plenty of examples on the Web. The reason I don't present my own code here is, as I've said before, there would be more code listings than text!

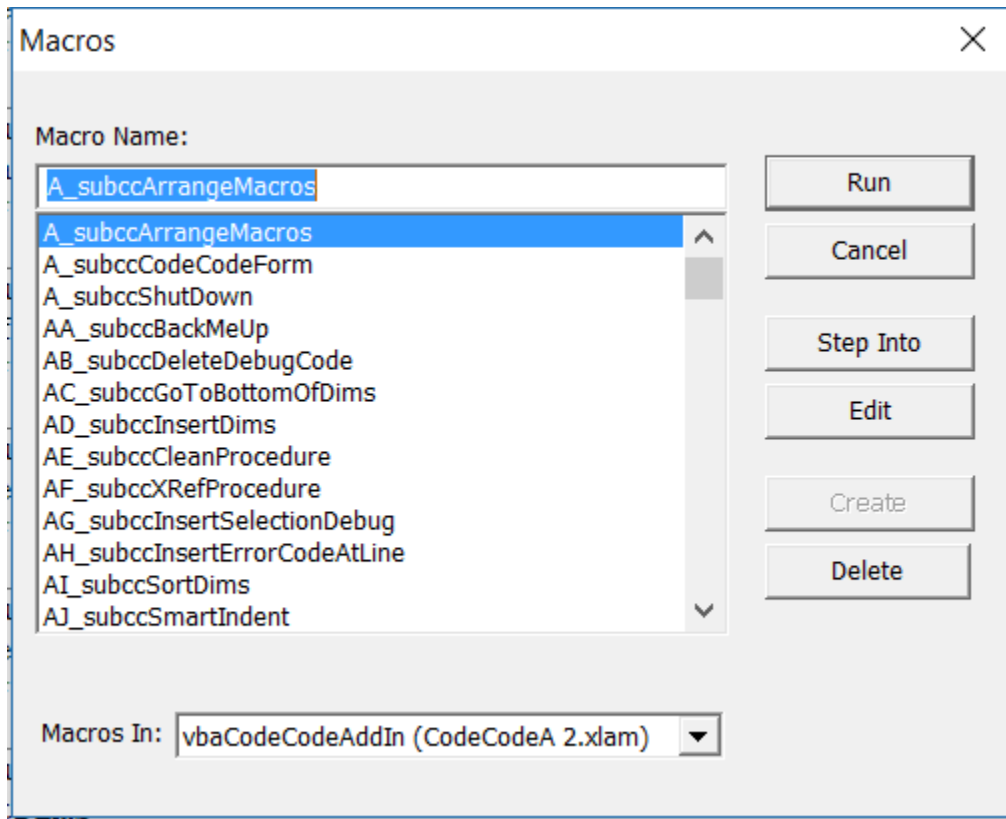
The UP and DOWN cmdButtons will move the items in the list er, UP or ummmm DOWN.

The preview button will then strip all of the prefixes from the LIST and add prefixes in alphabetic order starting at A_ and carrying on from there. When I'm happy, the OK button will change the sub headers in the actual MODULE to reflect the list. This puts items I want at the top of the (hisssss) macros menu. Easy peasy.

If you want the form/form code or anything I've referred to just email me. It's free. The full monty. Welllll, you could make a donation. Hehehe.

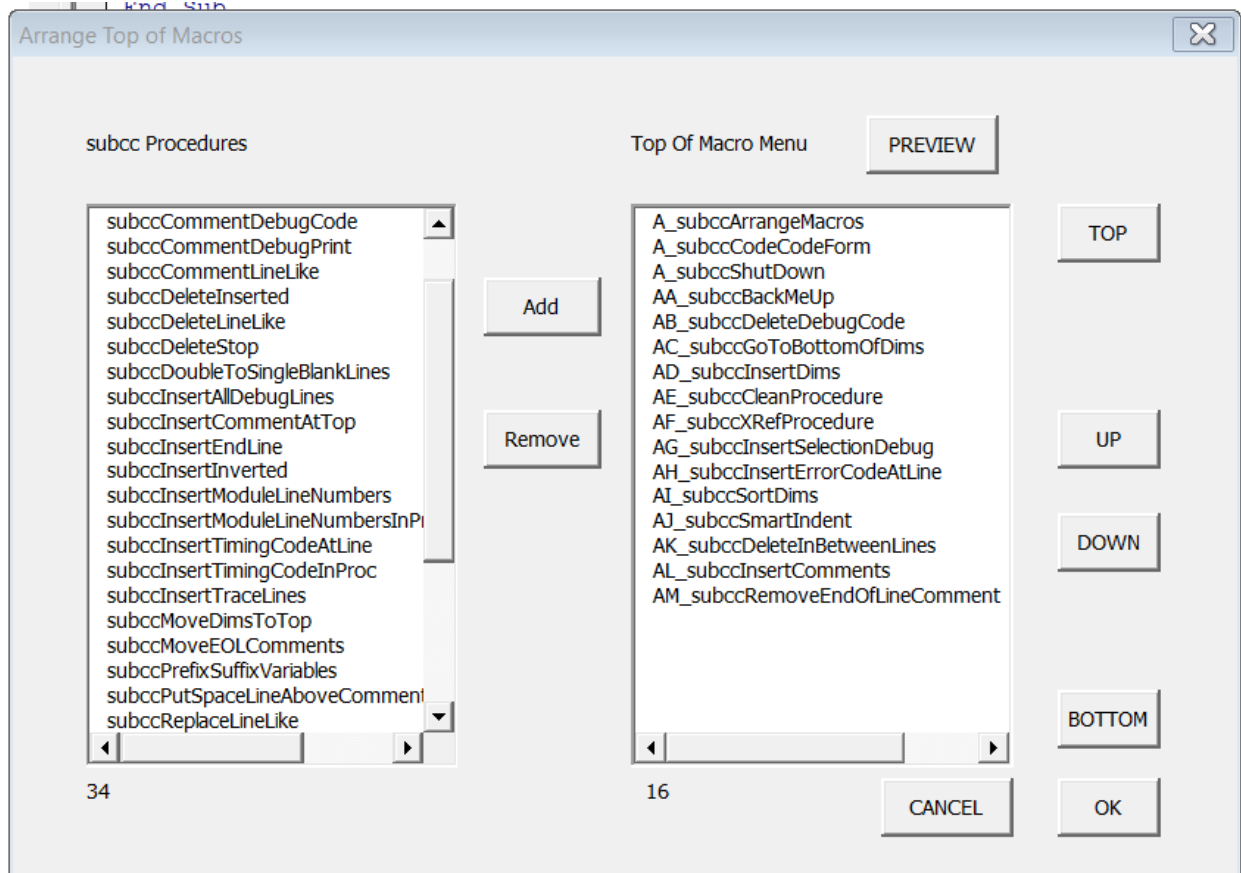
Here's th picture of my mumblemumble menu again.

17 My mumble mumble menu



And a pic of my loaded userform.

18 My Arrange macros menu UserForm



Time out!

One of the reasons some code isn't here, is that some procedures and modules are very big. Lotsa lines. Another is that I expect that people reading this to be familiar with the VBE and able to create forms and so on. Yet another reason harks back to something I said in the disclaimer and why I only posted update code for cWhereAreWe rather than the whole initialization sub. I don't want this to just be filled with code listings.

Although this is mostly about code for the VBE, a **GREAT** deal is editing and massaging lines of code. That's mostly string functions: Left\$, Right\$, Mid\$, Split, and so on. If you need help doing that, I'm quite happy to oblige. Really. I'm on several well-known web sites/forums, just get in touch, and you can always email. But that's not doing things in the VBE. It's editing strings.

Yet another reason is I'm trying to tell a story. I do this but I need to do this and oh, it would be good if this happened as well and while we're at it why don't we...

I'm sure you know what I'm getting at.

Aside: I use the "old" form of string functions like Left\$ Right\$ and Mid\$, adding a dollar sign at the end. They're still valid and what's more return strings! Left Right and Mid return variants of type string. Just one of my little quirks. I think VBS is deranged in using variants for **everything**! You can do a lot with VBS though so don't dismiss it! Maaaaybe more on that later. Probably not though. Incidentally, old **variable suffixes** are still valid as well.

57 Ancient variable suffixes

%	Integer	Dim L%
&	Long	Dim M&
@	Decimal	Const W@ = 37.5
!	Single	Dim Q!
#	Double	Dim X#
\$	String	Dim V\$ = "Secret"

There are a number of conversion functions from one variable type to another in VBA. The one that I've found that I use the most so far though, is Cstr, to convert a number to a string. Purely because I use string arrays a lot. VBA actually does a lot of conversions in the background that not many people realise. It's fairly generally accepted, though AFAIK not proven unequivocally for instance, that **internally**, VBA doesn't use integer, and there is an internal conversion to long, while remembering the original definition, so it can check for integer limits. That's why I now use long in all of my code. Google is your friend.

Try this. Add a module and insert:

58 VBA Converting example

	Option Explicit
	Sub subtestVariables()
	Dim sIS As String
	Dim lngIL As Long
	Dim sISArray(3) As String
	Dim lngILArray(1) As Long
	sIS = "test"
	lngIL = 20
	Debug.Print sIS & " " & lngIL
	sIS = 14
	Debug.Print sIS + lngIL
	sISArray(0) = sIS & " " & lngIL
	sISArray(1) = lngIL
	sISArray(2) = lngIL + sIS
	Debug.Print VarType(sISArray(0))
	Debug.Print VarType(sISArray(1))
	Debug.Print VarType(sISArray(2))
	Debug.Print sIS + "12"
	Debug.Print lngIL + "12"
	lngILArray(0) = sIS
	Debug.Print VarType(lngILArray(0))
	sIS = "12"
	lngILArray(1) = sIS

	Debug.Print VarType(InglArray(0))
	Debug.Print VarType(InglArray(1))
	' *****
	End Sub

The above will run without any compile or runtime errors even though there are mixed variable types without any explicit conversions.

A lot of people would convert everything to a string for example for a debug.print. But aha! VBA is being clever!

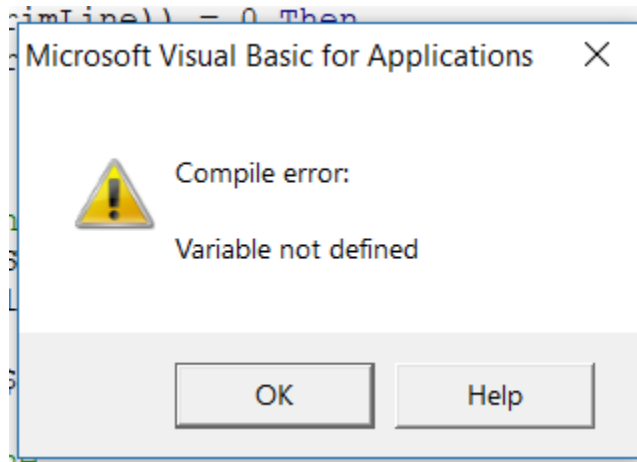
It's up to you whether you want VBA to do that or to use explicit conversion statements. I'm in two minds. Most of the strings you are going to display will be for debugging purposes and so will, I hope, be deleted later. Possibly using a routine we've shown here. Hehehe. So it could be a Q&D. OTOH you may just want to get into a habit and be consistent. Your choice. But! I urge you not to get obsessive about it!

Dims Bottom

Well I thought it was funny.

When you are building a procedure and then compiling or running, you **WILL** get the dreaded:

19 Variable not defined



This is a compile error and so not trappable at runtime. One way of coping with this is to manually, or womanually as the case might be, insert the variable into the Dim list at the top of the proc. You do have all of your Dims at the top of the procedure, don't you? If not use subccSplitAllDims to move them there!

So you dutifully hit the OK button, press CtrlC, scroll up to the bottom of the Dims, Paste as a Dim with maybe some editing, and possibly scroll all the way back down to where you were. Tiresome if you have a large procedure with lotsa new variables.

We can do a little bit better than that. Never mind google... Bookmarks are your friends!

We can get the selection. We can test it. We can go to a line. We can insert lines. Cool! In fact, we already "return" what we need from cWhereAreWe!

Let's Go!

59 subccInsertSingleDim

	Option Explicit
	Sub subccInsertSingleDim()
	' NOT BATCH.

' NO REPORT.
' NO END MESSAGE.
'
' Copy the current selection.
' Try and guess the variable type.
' Go to Bottom of Dims.
' Insert a new Dim.
'
Dim vbcplCodePane As VBIDE.CodePane
Dim clWhereAreWe As cWhereAreWe
Dim lngLastDimLinePlus1 As Long
Dim lngCurrentModuleLineNum As Long
Dim slSelection As String
Dim slSelectionArray() As String
Dim slVar As String
Dim vbcmlCodeModule As VBIDE.CodeModule
Dim slVarType As String
Dim slLine As String
Set clWhereAreWe = New cWhereAreWe
Set vbcplCodePane = clWhereAreWe.CodePane
Set vbcmlCodeModule = vbcplCodePane.CodeModule
slSelection = clWhereAreWe.Selection
slSelectionArray = Split(slSelection, " ")
slSelectionArray = Split(slSelectionArray(0), "(")
slVar = slSelectionArray(0)
slVarType = fncGuessVarType(slVar)
slLine = "Dim " & slVar & " As " & slVarType
lngLastDimLinePlus1 = clWhereAreWe.LastDimLinePlus1
lngCurrentModuleLineNum = clWhereAreWe.CurrentModuleLineNum
vbcplCodePane.SetSelection lngLastDimLinePlus1, 1, lngLastDimLinePlus1, 1

	DoEvents
	vbcmlCodeModule.InsertLines lngLastDimLinePlus1, sLine
	' *****
	End Sub

So we get the variable not defined message and press OK and Reset. Our problem variable is still highlighted. We run our procedure and voila! We have a new Dim and we can run or compile again. Note that if we have a naming convention then fncGuessVarType will set the correct er, var type. You could even put this on a button in a toolbar!

Building A Compile Report and Multithreading

Er... Normally, the above is a two staged process with you sitting behind the wheel as it were. The compile stops, we press return = OK, press reset, correct things and rerun our procedure.

We can go a bit further by being a bit crafty wafty.

We know we can run an item from the VBE menus. Oh look! There's Compile under the debug menu.

The below code will report the module name, the line number in the module, the whole line, and the selection on the line that the compiler is objecting to. Then it will comment out the line. When the code is finished the OnTime statement will take effect and the process will start again. Because we've commented out a compile error, the **NEXT** compile error will be highlighted.

We do need though, to be careful that we don't change the code of the procedure we are running or any code that we depend upon within that procedure. In this case cWhereAreWe. Remember remember!

However! Without multithreading we can't press the OK button programmatically and we can't get the compile error message.

We can easily start a new instance of an application with Shell. The problem comes with starting up any procedure in it. If we do that from within the application we started the new instance from, using Application.Run, then the calling application will wait till the the called procedure has ended. This is called **synchronous** processing. We don't want that because the process will stop and wait forever. We want **asynchronous** processing, where the two programs are running **independently**. Effectively, multithreading.

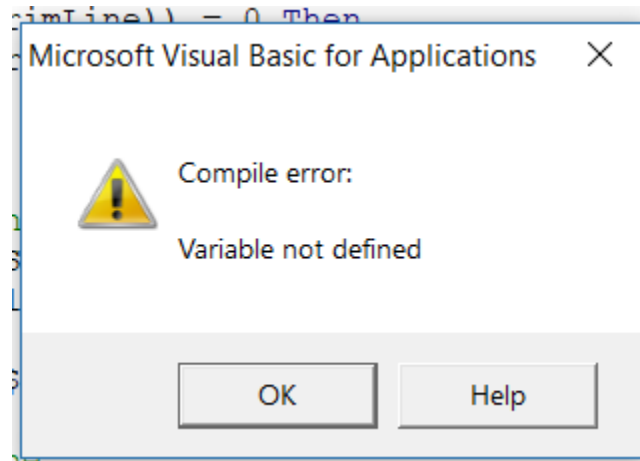
VBA is **inherently** a single thread environment but there are a few ways, that I'm aware of, to "do" multithreading.

1. Use the API call CreateThread.
Every attempt I've made to try and implement this has failed and the application has crashed. Maybe someone else can get it working and let me know how they did it please.
2. Start another instance of a possibly **DIFFERENT** application and run a procedure in that instance to start another instance of our application and run code in it. The usual application used is VBS. It's easy and quick, though the code building code can get a bit complex.
3. Use a completely different programming language that does multithreading like C# C++.
4. Shell a new instance of the current application and run code to add code to the new instance and then run that code.
5. Shell a new instance of the current application to open a file with code in it already, and run code from the new files Open event. Open workbook/doc/presentation/database procedure and so on.

Now while we can be pretty sure VBS exists in our environment, and is useable on most computers, we can pre write code in a file and run it. An alternative is to create VBS code "on the fly" save it and run it. This later is in fact what a lot of people do in VBA. Check the appendices.

We know that everything runs in its own space. We can call this a window. We know there are a number of windows open. We can see them. At least one is a dialog with an OK button. Here it is again.

Figure 20 Variable not defined



If we had information about open windows, we could maybe manipulate or at least get information from that window.

The **ONLY** way to get a **COMPLETE** list of open windows is using API calls. VBS doesn't do API calls. We could wrap the calls up in a COM object and use that in VBS but that would involve using a different language that does that sort of thing like C++ or C# or VB from VS for example.

We have **NO** idea what programs are on the computer so can't write anything that isn't in an application we would expect. We could expect Powershell. The Command window is another. Powershell and the command window command "Tasklist" **do not report ALL windows**. Same I think goes for VBS. There is one application though that we totally know exists. The one we're in.

We're going to use the last method. Run pre-written code built in the same application we're in and run it from the open event. For this I'm using Word... no good reason to do so... I just made a choice. There is no application specific code so it should run if imported to other applications as well. Using pre-written code is being lazy. But it means we don't have to build code in our code to create code with all of the single and double quotation marks and so on. But don't forget that is possible. In that instance we would run a single item and wouldn't depend on a separate file existing

Eventually you may want to do that, or perhaps export a module from the parent and import it to the new child instance, close it and run it again. The code could be built "inline" or actually come from anywhere. Maybe an exported .BAS file. Hmmmmm, didn't we talk about saving things in the registry a little while ago?

And talking about the registry. When I started this, I wanted to use the clipboard to pass messages and information back and forth between the calling application and the shelled child application. Everywhere has a clipboard right?

There's a bug. For some reason when retrieving the clipboard contents you get "??". This is documented and there is a bunch of API code to get around it, originally, I think from MSDN. Quite a bit in fact, and for that reason, I dumped it and decided to use the ummmm, registry. I'm not 100% happy with that for some reason. I don't know why, just makes me twitchy. But I went with it and it worked. However, you do it, there must be a means of passing info between the two open instances of the application. Very specifically, the error message text from the compile error window and a done/finished message.

There are some interesting things going on here between the two proggies as you'll see.

Here's the relevant **Main procedure** code. It's not all that much so I'm going to show the whole module. There are other modules in the project some of which I've deliberately left in to generate a known number of errors of a known type. Testin' y'know.

This is **RAW** code with all the pimples so be prepared!

You'll note that there's a **LOT** of debugging code printing to a file called Log.TXT and that all of these lines have " ' debug." at the end of the line. The child debug code instance prints to the **SAME** file. In this way you can trace through what is happening and when as events occur in either/both programs. This needs the subLog procedure in both instances and addressing the same Log.TXT file. All of these can be deleted. More on that later.

Anyway here's the module. Remember this is the **MAIN** procedure code instantiating the **child**. I'm not skimping here. I've **highlited/coloured** in some parts/lines.

We are looking for compile errors. When you use this code, you must **FIRST** make sure there are **NO COMPILE ERRORS** in this module and in the cWhereAreWe class! **Remember Remember!** Don't do it to yourself.

Walkies!

	Option Explicit
	Private Declare Sub Sleep _
	Lib "kernel32" _

	(ByVal dwMilliseconds As Long)
	Public Enum ASSOCSTR
	ASSOCSTR_COMMAND = 1
	ASSOCSTR_EXECUTABLE
	ASSOCSTR_FRIENDLYDOCNAME
	ASSOCSTR_FRIENDLYAPPNAME
	ASSOCSTR_NOOPEN
	ASSOCSTR_SHELLNEWVALUE
	ASSOCSTR_DDECOMMAND
	ASSOCSTR_DDEIFEXEC
	ASSOCSTR_DDEAPPLICATION
	ASSOCSTR_DDETOPIC
	ASSOCSTR_INFOTIP
	ASSOCSTR_QUICKTIP
	ASSOCSTR_TILEINFO
	ASSOCSTR_CONTENTTYPE
	ASSOCSTR_DEFAULTICON
	ASSOCSTR_SHELLEXTENSION
	ASSOCSTR_MAX
	End Enum
	Public Enum ASSOFCF
	ASSOFCF_INIT_NOREMAPCLSID = &H1
	ASSOFCF_INIT_BYEXENAME = &H2
	ASSOFCF_OPEN_BYEXENAME = &H2
	ASSOFCF_INIT_DEFAULTTOSTAR = &H4
	ASSOFCF_INIT_DEFAULTTOFOLDER = &H8
	ASSOFCF_NOUSERSETTINGS = &H10
	ASSOFCF_NOTRUNCATE = &H20
	ASSOFCF_VERIFY = &H40
	ASSOFCF_REMAPRUNDLL = &H80
	ASSOFCF_NOFIXUPS = &H100
	ASSOFCF_IGNOREBASECLASS = &H200
	End Enum

	Private Const GW_CHILD = 5
	Private Const GW_HWNDNEXT = 2
	Private Declare Function GetDesktopWindow Lib "User32" () As Long
	Private Declare Function GetWindow Lib "User32" (ByVal hwnd As Long, ByVal _ wCmd As Long) As Long
	Private Declare Function GetWindowText Lib "User32" Alias "GetWindowTextA" _ (ByVal hwnd As Long, ByVal lpString As String, ByVal cch As Long) As Long
	Private Type GUID
	IData1 As Long
	IData2 As Integer
	IData3 As Integer
	aData4(0 To 7) As Byte
	End Type
	Public Const MAX_PATH = 260
	Private Declare Function FindWindowEx Lib "User32" _ Alias "FindWindowExA" _ (ByVal hWnd1 As Long, _ ByVal hWnd2 As Long, _ ByVal lpsz1 As String, _ ByVal lpsz2 As String) As Long
	Private Declare Sub AccessibleObjectFromWindow Lib "OLEACC.DLL" _ (ByVal hwnd As Long, _ ByVal dwId As Long, _ riid As GUID, _ ppvObject As Any)
	Private Const OBJID_NATIVEOM = &HFFFFFFF0
	Private Const INFINITE = &HFFFFFFF
	Private Declare Function WaitForSingleObject Lib "kernel32" (ByVal hHandle As Long, ByVal dwMilliseconds As Long) As Long

	Private Declare Function CreateThread Lib "kernel32" (ByVal LpThreadAttributes As Long, _
	ByVal DwStackSize As Long, _
	ByVal LpStartAddress As Long, _
	ByVal LpParameter As Long, _
	ByVal dwCreationFlags As Long, _
	ByRef LpThreadId As Long) As Long
	Private Declare Function CloseHandle Lib "kernel32" (ByVal HANDLE As Long) As Long
	Public Declare Function AssocQueryString Lib "shlwapi.dll" _
	Alias "AssocQueryStringA" (ByVal flags As ASSOCF, _
	ByVal STR As ASSOCSTR, _
	ByVal pszAssoc As String, _
	ByVal pszExtra As String, _
	ByVal pszOut As String, _
	ByRef pcchOut As Long) As Long
	Public smFile As String
	Sub subStartCompile()
	SaveSetting "Mike&Lisa", "Compile", "Status", ""
This is the MAIN bit that runs subCompile	
	subCompile
	'

	End Sub
	Sub subCompile()
	Dim clWhereAreWe As cWhereAreWe
	Dim lngCurrentModuleLineNum As Long
	Dim lngErrNumber As Long
	Dim lngLastDimLinePlus1 As Long
	Dim lngIN As Long
	Dim lngSanityCheck As Long
	Dim slCommentString As String

	Dim slCompileError As String
	Dim slErrDescription As String
	Dim slLine As String
	Dim slReplaceLine As String
	Dim slSelection As String
	Dim slSelectionArray() As String
	Dim slVar As String
	Dim slVarType As String
	Dim vbcmlCodeModule As VBIDE.CodeModule
	Dim vbcplCodePane As VBIDE.CodePane
	Dim slPrintLine As String
	Dim slTextFile As String
	Dim lngAnswer As Long
	Dim slStatus As String
	Dim slMessage As String
This is deliberately using Chr() because when we comment out the "'@@ " we don't want to comment this out as well!	
	slCommentString = Chr(39) & Chr(64) & Chr(64) & Chr(32)
	slTextFile = "Compile.txt"
	' subLog "Compile get message" ' debug.
Has the child sent a Done message?	
	' Do this BEFORE starting a new copy.
	slStatus = fncGetStatus()
	' subLog "Compile message >" & slStatus & "<" ' debug.
	If slStatus = "Done" Then
	slStatus = "Done From Main from Child."
	MsgBox slStatus
	Exit Sub
	End If

Start the other "process/thread" before we compile. The other process will loop for a bit looking for the compile error window. If it doesn't find it it will say so in the status and stop.	
	' Here's where we need info and have to press the OK button.
	' subLog "Compile Starting Child" ' debug.
	subStartChildApplication
	' subLog "Compile Child started" ' debug.
There are TWO tests here. One for the runtime error and one to see if the Compile option is greyed out.	
	' Right. It looks as though If there is a succesfull complie then
	' the .Execute fails with an -2147467259 error.
	' subLog "Compile compile" ' debug.
<p>This is where we do the compile! Note we look for the faceid rather than... Application.VBE.CommandBars("Menu Bar") _ .Controls("Debug").Controls("Compile Project").Execute This is because different applications have different text for "Project". Excel and Powerpoint for example say "Compile VBAProject" and to avoid different language peca-dilloes. This is in line with the Project naming schemes.</p>	
	On Error Resume Next
	Application.VBE.CommandBars.FindControl(ID:=578).Execute
	InglErrNumber = Err.Number
	On Error GoTo 0
If there is a compile error... Variable not defined or something, then we are now sitting there waiting for the OK button to be pressed. If the compile completed because of no errors here's where we check. There could be two possibilities. Either a runtime error -2147467259 or the Compile option is greyed out/disabled.	
	Select Case InglErrNumber
	Case 0
	Case -2147467259
	' subLog "Compile Done -2147467259." ' debug.

	sIStatus = "Done -2147467259."
	subSendStatus sIStatus
	MsgBox sIStatus
	Exit Sub
	Case Else
Leaving a Stop in is VERY BAD programming practice. I've done this to catch any error numbers as a just in case thing and also this is a learning situation. Replace this with whatever you want as error reporting and then exit sub if you like.	
	Stop
	End Select
	' Catch up.
	DoEvents
	' Just Checking.
	If Application.VBE.CommandBars("Menu Bar") _
	.Controls("Debug").Controls("Compile Project").Enabled = False Then
	sIStatus = "Done Enabled = False."
	' subLog "Compile Done Enabled = False." ' debug.
	subSendStatus sIStatus
	MsgBox sIStatus
	Exit Sub
	End If
	' subLog "Compile After Compile and button pressed" ' debug.
Meanwhile... the CHILD instance has been waiting for the dialog window to show up. When it does, the CHILD will pick up the compile error and press OK. This code will carry on from here. Normally when pressing OK the reason for the error is "reported" in the dialog window which disappears and the problem is highlighted on the line. This is no different. We have a line with an error highlighted	

in the codepane. We use cWhereAreWe to pick up the information we need. BUT! We don't have the compile error message. That's gone away with the dialog window.

' This won't happen until we've pressed OK.
Set clWhereAreWe = New cWhereAreWe
Set vbcmIcodeModule = clWhereAreWe.CodeModule
InglCurrentModuleLineNum = clWhereAreWe.CurrentModuleLineNum
Sleep 1000
' Get the compile message text.
' subLog "Compile Get Status/compile error" ' debug.

Having said we don't have the compile error message, the CHILD instance will have picked it up and sent it to the registry just before it pressed OK. Go and get it.

slCompileError = fncGetStatus()
' subLog "Compile Got status/compile error >" & slCompileError & "<" ' debug.
slPrintLine = Now() & "/" & InglCurrentModuleLineNum _
& "/" _
& clWhereAreWe.ModuleName _
& "/" _
& clWhereAreWe.CurrentLineText _
& "/" _
& clWhereAreWe.Selection _
& "/" _
& slCompileError _
& "<"

We don't really need both but what the hell.

Debug.Print slPrintLine
Open slTextFile For Append As #1
Print #1, slPrintLine
Close #1
' subLog "compile printline >" & slPrintLine & "<" ' debug.

<p>Important bit here. Comment out the line the error is on! Without this we won't get to the *next* compile error if there is one.</p>	
	' Replace "bad" line'.
	' Don't do it to ourself!
	Select Case clWhereAreWe.ModuleName
	Case "mCompile", "cWhereAreWe"
	MsgBox "There is a compile error in this module or cWhereAreWe." Stop Case Else
<p>Something to think about here. We have the compile message. If it's "Duplicate declaration in current scope"... We may choose to just delete the line. If it's "Variable not defined"... well we've built a procedure to fix that haven't we!</p>	
	slReplaceLine = slCommentString & clWhereAreWe.CurrentLineText
	' subLog "compile replace line /" & lng!CurrentModuleLineNum & "/" & slReplacelLine ' debug.
<p>MAJOR PROBLEM: If the error is on an If line, say, maybe testing the value of an undefined variable, the If line will be commented out, and the End If and any Elself lines will then come up as a compile error as well. This is something to be aware of when looking at the report or immediate window. Same for Select and Do While and Loop Until and With.</p>	
	vbcmlCodeModule.ReplaceLine lng!CurrentModuleLineNum, slReplaceLine
<p>This is just icing. Bookmark the line so we can get to it quickly.</p>	
	Application.VBE.CommandBars("Menu Bar") _ .Controls("Edit").Controls("Bookmarks").Controls("Toggle BookMark").Execute
	' Start me again in a bit.
	' subLog "compile Ontime " & Now() ' debug.

The comment in the code above says it all.

	Application.OnTime Now() + TimeValue("00:00:4"), "subCompile"
	End Select
	' subLog "compile End" ' debug.
	' ***** End Sub
	Sub subStartChildApplication()
	Dim oIFS As FileSystemObject
	Dim sLEXEName As String
	Dim sIFileName As String
	Dim sIExt As String
	Dim oIObj As Object
	Dim sIPath As String
	Dim oIChildApplication As Object
	Dim sIDocName As String
	Dim sIApplicationName As String
	Dim sIChildFileName As String
	Dim lngErrNumber As Long
	Dim sIProjectCaption As String
	Set oIFS = CreateObject("Scripting.FileSystemObject")
	sIFileName = Application.VBE.ActiveVBProject.FileName
	sIExt = "." & oIFS.GetExtensionName(sIFileName)
	sIPath = oIFS.GetParentFolderName(sIFileName)

This is an interim **HARD CODED** solution. Eventually you'll want to export a module from here and import it to the new instance and then run the procedure in that instance. Maybe you'll do that and then close the file and reopen it so the open event triggers.

At the moment the "other" file is of the same application as this and in the same folder. This might end up as Insert a module and copy and paste rather than import. Dunno.

In all cases, to maintain **asynchronicity**, the call to subGetButtonInfo **CANNOT** be started from here. It **MUST** be started either from a VBS file a batch file or whatever or, as it is here, put into the open event of the **CHILD**.

	sApplicationName = Application.Name
	Select Case sApplicationName
	Case "Microsoft Word"
	sDocName = "doc2b.docm"
	sProjectCaption = "Project"
	Case "Microsoft Excel"
	sProjectCaption = "VBAProject"
	Case "Microsoft Powerpoint"
	Case "Microsoft Outlook"
	Case "Microsoft Access"
	Case "Microsoft Project"
	Case Else
	End Select
	sChildFileName = sPath & "\" & sDocName
Check if the file is open already.	
	On Error Resume Next
	Open sChildFileName For Binary Access Read Write Lock Read Write As #1
	lngErrNumber = Err.Number
	Close #1
	On Error GoTo 0
	Select Case lngErrNumber
	Case 0
	Debug.Print "Opening " & sChildFileName
If the file isn't open then shell a new instance of it using this applications EXE.	
	Shell fncGetAssociatedEXE(sExt) & " " & sDocName, vbNormalFocus
	Debug.Print sChildFileName; " Open"
	Case 70
	' File is open.
	Case Else
	End Select

```

DoEvents
'
*****
End Sub
Public Function fncGetAssociatedEXE(spExtension As String) As String
Dim lngIReturn As Long
Dim slResult As String
Dim lngIPCCHOut As Long
slResult = String$(MAX_PATH, 0)
lngIPCCHOut = Len(slResult)
lngIReturn = AssocQueryString( _
    0, _
    ASSOCSTR_EXECUTABLE, _
    spExtension, _
    "open", _
    slResult, _
    lngIPCCHOut)
If lngIReturn = 0 Then
' AssocQueryString succeeded.
slResult = fncTrimNull(slResult)
fncGetAssociatedEXE = slResult
End If
'
*****
End Function
Private Function fncTrimNull(spToTrim As String) As String
Dim lngChrPos As Long

```

	InglChrPos = InStr(spToTrim, Chr(0))
	If InglChrPos <> 0 Then
	fncTrimNull = Trim\$(Left\$(spToTrim, InglChrPos - 1))
	Else
	fncTrimNull = Trim\$(spToTrim)
	End If
	'

	End Function
	Function fncGetStatus()
	Dim slStatus As String
	slStatus = GetSetting("Mike&Lisa", "Compile", "Status") ' [,Default])
	DoEvents
	The Done message may contain extra information. We only want the Done bit.
	If Left\$(slStatus, Len("Done")) = "Done" Then
	slStatus = "Done"
	End If
	fncGetStatus = slStatus
	'

	End Function
	Sub subSendStatus(_
	spStatus As String _
)
	Dim slStatus As String
	SaveSetting "Mike&Lisa", "Compile", "Status", spStatus
	'

	End Sub

	Sub subLog(_
	spLogMessage As String _
)
	Open "Log.txt" For Append As #1
	Print #1, spLogMessage
	Close #1
	'

	End Sub

Here's one we prepared earlier. Hehehe. Old joke.
Here's the prewritten code module with the procedure executed by the open event in the **CHILD** process. As a total aside... The declarations are formatted with ^^^^

	Option Explicit
	Private Declare Sub Sleep _
	Lib "kernel32" _
	(ByVal dwMilliseconds As Long)
	Private Declare Function FindWindow _
	Lib "User32" _
	Alias "FindWindowA" _
	(_
	ByVal lpClassName As String, _
	ByVal lpWindowName As String _
) _
	As Long
	Private Declare Function GetWindow _
	Lib "User32" _
	(_
	ByVal hwnd As Long, _
	ByVal wCmd As Long _
) _

	As Long
	Private Declare Function SetForegroundWindow _
	Lib "user32.dll" _
	(_
	ByVal hwnd As Long _
) _
	As Long
	Private Declare Function SetFocusAPI _
	Lib "User32" _
	Alias "SetFocus" _
	(_
	ByVal hwnd As Long _
) _
	As Long
	Private Declare Function SendMessage _
	Lib "user32.dll" _
	Alias "SendMessageW" _
	(_
	ByVal hwnd As Long, _
	ByVal uMsg As Long, _
	ByVal wParam As Long, _
	ByRef lParam As Any _
) _
	As Long
	Private Declare Function GetWindowText _
	Lib "User32" _
	Alias "GetWindowTextA" _
	(_
	ByVal hwnd As Long, _
	ByVal lpString As String, _
	ByVal cch As Long _
) _
	As Long
	Private Declare Function GlobalAlloc _
	Lib "kernel32" _
	(_
	ByVal wFlags As Long, _

	ByVal dwBytes As Long _
) _
	As Long
	Private Declare Function GlobalLock _
	Lib "kernel32" _
	(_
	ByVal hMem As Long _
) _
	As Long
	Private Declare Function lstrcpy _
	Lib "kernel32" _
	(_
	ByVal lpString1 As Any, _
	ByVal lpString2 As Any _
) _
	As Long
	Private Declare Function GlobalUnlock _
	Lib "kernel32" _
	(_
	ByVal hMem As Long _
) _
	As Long
	Private Declare Function OpenClipboard _
	Lib "User32" _
	(_
	ByVal hwnd As Long _
) _
	As Long
	Private Declare Function EmptyClipboard _
	Lib "User32" _
	() _
	As Long
	Private Declare Function SetClipboardData _
	Lib "User32" _
	(_
	ByVal wFormat As Long, _
	ByVal hMem As Long _

) _
	As Long
	Private Declare Function CloseClipboard _
	Lib "User32" _
	() _
	As Long
	Private Const GW_HWNDFIRST = 0
	Private Const GW_HWNDLAST = 1
	Private Const GW_HWNDNEXT = 2
	Private Const GW_HWNDPREV = 3
	Private Const GW_OWNER = 4
	Private Const GW_CHILD = 5
	Private Const GW_MAX = 5
	Private Const BM_CLICK = &HF5
	Private Const GHND = &H42
	Private Const CF_TEXT = 1
	Private Const MAXSIZE = 4096
	Public Sub subGetButtonInfo()
	Dim InghIParentHwnd As Long
	Dim sIReturnBuffer As String * 128
	Dim InghINextHwnd As Long
	Dim InghIChildHwnd As Long
	Dim sIParentCaption As String
	Dim sIGetCaption As String
	Dim lngCompStrLen As Long
	Dim sICompileStr As String
	Dim sLOK As String
	Dim lngLenOK As Long
	Dim InghIOKHwnd As Long
	Dim sICompileMessage As String
	Dim lngChr As Long
	Dim lngIN As Long
	Dim sIWindowText As String

	Dim lngTextLen As Long
	Dim lngLoopCount As Long
	Dim slStatus As String
	' We know the beginning of the Compile error message.
	slCompileStr = "Compile error:"
	lngCompStrLen = Len(slCompileStr)
	' We know OK button text.
	slOK = "OK"
	lngLenOK = Len(slOK)
	' We know the window caption.
	slParentCaption = "Microsoft Visual Basic for Applications"
	lngLoopCount = 0
	' subLog "2b starting loop" ' debug.
Loop looking for the Compile Error window. This will be a CHILD of the main VBE window which is why we have to use API calls to find it.	
	Do
	' Use the Registry to communicate a stop.
	DoEvents
	' subLog "2b get status" ' debug.
Has the Main instance that shelled me, finished compiling?	
	slStatus = fncGetStatus()
	' subLog "2b status >" & slStatus & "<" ' debug.
	If slStatus = "Done" Then
	' subLog "2b Exit sub" ' debug.

	Exit Sub
	End If
	' subLog "2b Find window" ' debug.
Look for the Compile Error window.	
	InghlParentHwnd = FindWindow(vbNullString, sIParentCaption)
	If InghlParentHwnd <> 0 Then
	' subLog "2b FOUND window" ' debug.
	sIGetCaption = ""
Found it! The buttons are all windows as well so get the four CHILD windows and the captions and the window handle of the OK button.	
	' Get the child windows.
	' There are four of them.
	' It doesn't seem worth it to create a loop for just four.
	' subLog "2b Button 1" ' debug.
First window is a CHILD of the Compile error window.	
	InghlChildHwnd = GetWindow(InghlParentHwnd, GW_CHILD)
	InglTextLen = GetWindowText(InghlChildHwnd, sIReturnBuffer, 128)
	sIGetCaption = Trim(sIReturnBuffer)
	If Left\$(sIGetCaption, InglLenOK) = sLOK Then
	InghlOKHwnd = InghlChildHwnd
	Elseif Left\$(sIGetCaption, InglCompStrLen) = sICompileStr Then
	subSendStatus sIGetCaption
	End If
	' subLog "2b Button 1 caption >" & sIGetCaption & "<" ' debug.
	' subLog "2b Button 2" ' debug.
Now we get the NEXT window in the Z Order.	
	InghlNextHwnd = GetWindow(InghlChildHwnd, GW_HWNDNEXT)

	InglTextLen = GetWindowText(InglNextHwnd, slReturnBuffer, 128)
	slGetCaption = Trim(slReturnBuffer)
	If Left\$(slGetCaption, InglLenOK) = sLOK Then
	InglOKHwnd = InglNextHwnd
	Elseif Left\$(slGetCaption, InglCompStrLen) = slCompileStr Then
	subSendStatus slGetCaption
	End If
	' subLog "2b Button 2 caption >" & slGetCaption & "<" ' debug.
	' subLog "2b Button 3" ' debug.
And the NEXT.	
	InglNextHwnd = GetWindow(InglNextHwnd, GW_HWNDNEXT)
	InglTextLen = GetWindowText(InglNextHwnd, slReturnBuffer, 128)
	slGetCaption = Trim(slReturnBuffer)
	If Left\$(slGetCaption, InglLenOK) = sLOK Then
	InglOKHwnd = InglNextHwnd
	Elseif Left\$(slGetCaption, InglCompStrLen) = slCompileStr Then
	subSendStatus slGetCaption
	End If
	' subLog "2b Button 3 caption >" & slGetCaption & "<" ' debug.
	' subLog "2b Button 4" ' debug.
And the NEXT.	
	InglNextHwnd = GetWindow(InglNextHwnd, GW_HWNDNEXT)
	InglTextLen = GetWindowText(InglNextHwnd, slReturnBuffer, 128)
	slGetCaption = Trim(slReturnBuffer)
	If Left\$(slGetCaption, InglLenOK) = sLOK Then
	InglOKHwnd = InglNextHwnd
	Elseif Left\$(slGetCaption, InglCompStrLen) = slCompileStr Then
	subSendStatus slGetCaption
	End If
	' subLog "2b Button 4 caption >" & slGetCaption & "<" ' debug.

	' Press the OK button.
	' subLog "2b PRESS BUTTON" ' debug.
Okay! We have the compile error message and we have the window handle of the OK button. We've sent the compile error to the registry. Time to press that button with SendMessage!	
	SetForegroundWindow InghIOKHwnd
	Call SetFocusAPI(InghIOKHwnd)
	DoEvents
	Call SendMessage(InghIOKHwnd, BM_CLICK, 0, 0)
	DoEvents
	Sleep 1000
	Else
Haven't found the compile error window yet, hang around for a bit. Then try again. If this needs to be increased you can add to the sleep time or the number of loops.	
	Sleep 2000
	InglLoopCount = InglLoopCount + 1
	' subLog "2b Looping to wait >" & InglLoopCount & "<" ' debug.
Tired of waiting... 2 secs x 20... probably not going to happen. Get out!	
	If InglLoopCount > 20 Then
	subSendStatus "Done END From Child TIMEOUT"
	Exit Do
	End If
	End If
	Loop
	' subLog "2b END" ' debug.

```

'
*****
End Sub
Sub subSendStatus( _
    spStatus As String, _
    Optional spCompileStr As String = "Compile error:" _
)
Dim slStatus As String
' Strip the Compile message because we only want the error.
Do
    If InStr(spStatus, " ") > 0 Then
        spStatus = Replace(spStatus, " ", "")
    Else
        Exit Do
    End If
Loop
spStatus = Replace(spStatus, Chr(10), "")
spStatus = Replace(spStatus, Chr(0), "")
spStatus = Trim$(Replace(spStatus, spCompileStr, ""))
' subLog "2b Sending >" & spStatus & "<" ' debug.
SaveSetting "Mike&Lisa", "Compile", "Status", spStatus
'
*****
End Sub
Function fncGetStatus()
Dim slStatus As String
slStatus = GetSetting("Mike&Lisa", "Compile", "Status") ' [,Default]
' subLog "2b Got status fncGetStatus >" & slStatus & "<" ' debug.

```


	If Left\$(sIStatus, Len("Done")) = "Done" Then
	sIStatus = "Done"
	End If
	fncGetStatus = sIStatus
	'

	End Function
	Sub subLog(_
	spLogMessage As String _
)
	Open "Log.txt" For Append As #1
	Print #1, spLogMessage
	Close #1
	'

	End Sub

Other module code with known errors so we get a meaningful report to check out.

	Option Explicit
	Sub subtestCompile()
	Dim lngEndLine As Long
	Dim lngStartLine As Long
	Dim lngELine As Long
	Dim lngEndLine As Long
	Dim lngStartLine As Long
	Dim lngELine As Long
	Dim lngSLine As Long
	Dim lngECol As Long
	Dim lngSCol As Long

Dim sSelection As String
Dim lngCurrentModuleLine As Long
Dim clWhereAreWe As cWhereAreWe
Dim lngModuleEndLine As Long
Dim sLine As String
Dim sLineArray() As String
Dim sLOLine As String
Dim vbcmlCodeModule As VBIDE.CodeModule
Dim vbcplCodePane As VBIDE.CodePane
Dim lngICLine As Long
Dim sTrimLine As String
Set clWhereAreWe = New cWhereAreWe
lngModuleEndLine = clWhereAreWe.ModuleProcEndLine
Set vbcmlCodeModule = clWhereAreWe.CodeModule
Set vbcplCodePane = clWhereAreWe.CodePane
vbcplCodePane.GetSelection lngCurrentModuleLine, lngISCol, lngILine, lngIECol
' Is there a selection?
sLine = clWhereAreWe.CurrentLine
sSelection = clWhereAreWe.Selection
lngISLine = clWhereAreWe.CurrentModuleLine
lngISCol = clWhereAreWe.SCol
lngILine = clWhereAreWe.ELine
lngIECol = clWhereAreWe.ECol
'

End Sub

Debug Reprise

And there may be another reprise even. Playing it by ear at the mo. Anyways, there are a lot of lines in the previous section about compiling with " ' debug'" at the end. In another piece of code,

subccDeletDebugPrint, we deleted all code with debug.print. That can easily be altered to delete lines with our tag, " ' debug." at the end.

Change....

	<code>sILookFor = "Debug.Print"</code>
	<code>IngLenLookFor = Len(sILookFor)</code>
	<code>' Go Back UP the code to preserve line numbers.</code>
	<code>IngCurrentLine = IngModuleEndLine</code>
	<code>Do</code>
	<code>sLine = Trim\$(vbcmlCodeModule.Lines(IngCurrentLine, 1))</code>
	<code>If Len(sLine) >= IngLenLookFor Then</code>
	<code> If InStr(sLine, sILookFor) > 0 Then</code>
	<code> vbcmlCodeModule.DeleteLines _</code>
	<code> StartLine:=IngCurrentLine, _</code>
	<code> count:=1</code>
	<code> End If</code>
	<code>End If</code>

To....

	<code>sILookFor = ucase\$(" ' Debug")</code>
	<code>IngLenLookFor = Len(sILookFor)</code>
	<code>' Go Back UP the code to preserve line numbers.</code>
	<code>IngCurrentLine = IngModuleEndLine</code>
	<code>Do</code>
	<code>sLine = Trim\$(vbcmlCodeModule.Lines(IngCurrentLine, 1))</code>
	<code>If Len(sLine) >= IngLenLookFor Then</code>

	If InStr(sLine, sLookFor) > 0 Then
	vbcmlCodeModule.DeleteLines _
	StartLine:=InglCurrentLine, _
	count:=1
	End If
	End If

Maybe you feel " ' debug." Is too long to type. It could be anything. As long as it's a unique at the end of the line comment. " ']" is probably the quickest to type.

It's possible even to make sLookFor a parameter! Something to think about to delete lines eh?

And this can be made module or project wide as we've done before... maybe as a parameter also!

In fact, we don't need to delete them even. As long as we have that tag we can comment out the lines instead. Maybe yet another parameter?

What we're sometimes left with after that though is a bunch of code with multiple space lines. We can do something about that!

We've looped through the lines of a procedure a few times now. The general code is...

	Sub subLoopThroughProcLines()
	'
	Dim clWhereAreWe As cWhereAreWe
	Dim lngModuleEndLine As Long
	Dim lngModuleBodyLine As Long
	Dim sCurrentLine As String
	Dim sSelection As String
	Dim vbcmlCodeModule As VBIDE.CodeModule
	Dim lngSLine As Long
	Dim lngSCol As Long
	Dim lngELine As Long
	Dim lngECol As Long
	Dim lngCurrentLine As Long
	Dim sLookFor As String
	Dim lngLenLookFor As Long

Dim sLine As String
Set clWhereAreWe = New cWhereAreWe
InglModuleBodyLine = clWhereAreWe.ModuleProcBodyLine
InglModuleEndLine = clWhereAreWe.ModuleProcEndLine
sSelection = clWhereAreWe.Selection
InglSLine = clWhereAreWe.CurrentModuleLineNum
InglSCol = clWhereAreWe.SCol
InglELine = clWhereAreWe.ELine
InglECol = clWhereAreWe.ECol
sCurrentLine = clWhereAreWe.CurrentLineText
Set vbcmlCodeModule = clWhereAreWe.CodeModule
InglLenLookFor = Len(sLookFor)
' Go Back UP the code to preserve line numbers.
InglCurrentLine = InglModuleEndLine
Do
sLine = Trim\$(vbcmlCodeModule.Lines(InglCurrentLine, 1))
InglCurrentLine = InglCurrentLine - 1
If InglCurrentLine <= InglModuleBodyLine Then
Exit Do
End If
Loop
'

End Sub

To check for double blank lines and reduce them to single ones, we need to look at the next line as well. Here's the code.

Sub subDeleteDoubleBlankLines()
'
Dim clWhereAreWe As cWhereAreWe
Dim lngCurrentLine As Long
Dim lngECol As Long
Dim lngELine As Long
Dim lngEndProcLine As Long
Dim lngLenLookFor As Long
Dim lngModuleProcBodyLine As Long
Dim lngModuleProcEndLine As Long
Dim lngSCol As Long
Dim slNextLine As String
Dim lngSLine As Long
Dim slCurrentLine As String
Dim slLine As String
Dim slLookFor As String
Dim slSelection As String
Dim vbclComponent As VBIDE.VBComponent
Dim vbcmlCodeModule As VBIDE.CodeModule
Dim vbcplCodePane As VBIDE.CodePane
Dim vbplProject As VBIDE.VBProject
Set clWhereAreWe = New cWhereAreWe
' We don't use some of these but they're there if we
' need them for something else.
lngModuleProcBodyLine = clWhereAreWe.ModuleProcBodyLine
lngModuleProcEndLine = clWhereAreWe.ModuleProcEndLine
slSelection = clWhereAreWe.Selection
lngSLine = clWhereAreWe.CurrentModuleLineNum
lngSCol = clWhereAreWe.SCol
lngELine = clWhereAreWe.ELine
lngECol = clWhereAreWe.ECol
lngEndProcLine = clWhereAreWe.EndProcLine
slCurrentLine = clWhereAreWe.CurrentLineText

	Set vbplProject = clWhereAreWe.Project
	Set vbcmlCodeModule = clWhereAreWe.CodeModule
	Set vbcplCodePane = clWhereAreWe.CodePane
	Set vbclComponent = clWhereAreWe.Component
	' Go Back UP the code to preserve line numbers.
	' We could just as easily go down as well
	' and we would have to look for:
	' "End Sub", _
	' "End Function", _
	' "End Property"
	' We can't look for just "End " because of:
	' "End If"
	' "End Select"
	InglCurrentLine = InglEndProcLine
	Do
	slLine = Trim\$(vbcmlCodeModule.Lines(InglCurrentLine, 1))
	If LenB(Trim\$(slLine)) = 0 Then
	slNextLine = vbcmlCodeModule.Lines(InglCurrentLine + 1, 1)
	If LenB(Trim\$(slNextLine)) = 0 Then
	vbcmlCodeModule.DeleteLines _
	StartLine:=InglCurrentLine + 1, _
	count:=1
	End If
	End If
	InglCurrentLine = InglCurrentLine - 1
	If InglCurrentLine <= InglModuleProcBodyLine Then
	Exit Do
	End If
	Loop
	'

	End Sub
--	---------

We need to be careful when going from the end of a sub because it may be the last in the module and we have to deal with pesky blank lines that are included in the subs line count.

We've had to do this at least once before. Repeating code, Hmmmm, maybe we can put it in cWhereAreWe. And that's what I did. The new class item is EndProcLine. It will be the same as ModuleProcEndLine **UNLESS** the procedure is the last in the module.

Continuations

Very often we'll want to look at a line in the VBE and ask things about it. Is it a Declare line? Does it have a comment on the end? Is it a Dim Line? Use your imagination. But we need the whole line.

When working with picking up lines from VBA procedures you'll often come across lines that are "continued" to the next one with the "_" continuation character. To process a line, you need to pick up the whole doodad. A good example of this is how I personally declare Sub and Function definitions.

If we are in the middle of a set of continued lines we need to go back up and then start going down again.

It's a lot of bother just to pick up a line but worth it and in some cases essential.

The process is...

1. Table 1

	Get a line = B
	Get the line above it = A
	Does A have the continuation chr at the end?
	Loop upwards getting lines until no more continuation chrs
	If the line has a continuation chr add it to the FRONT of B
	If B has a continuation chr at the end loop down getting lines till no more continuation chrs
	If the line has a continuation chr add it to the END of B

Here's the code...

2. Table 2

	Function fncGetSingleLine(_
	vbcmpCodeModule As VBIDE.CodeModule, _
	IngpCurrentModuleLineNum As Long)
	' Join lines with continuation character up into
	' a single line.
	' Increment the given line number.
	'
	Dim ilCommentPos As Integer

Dim lngCurrentLine As Long
Dim sJoinChr As String
Dim sGetLine As String
Dim sOldLine As String
Dim sNewLine As String
Dim blnLoopUp As Boolean
Dim blnLoopDown As Boolean
sJoinChr = " _ "
sOldLine = Trim\$(vbcmpCodeModule.Lines(lngpCurrentModuleLineNum, 1))
lngCurrentLine = lngpCurrentModuleLineNum
blnLoopDown = False
blnLoopUp = False
If LenB(sOldLine) > 2 Then
If Right\$(sOldLine, 2) = sJoinChr Then
blnLoopDown = True
End If
Else
fncGetSingleLine = sOldLine
Exit Function
End If
sGetLine = Trim\$(vbcmpCodeModule.Lines(lngCurrentLine - 1, 1))
If LenB(sGetLine) > 2 Then
If Right\$(sGetLine, 2) = sJoinChr Then
sNewLine = sOldLine & " " & sGetLine
blnLoopUp = True
End If
Else
fncGetSingleLine = sOldLine
Exit Function
End If
If blnLoopDown = True Then

Do
InglCurrentLine = InglCurrentLine + 1
slGetLine = Trim\$(vbcmpCodeModule.Lines(InglCurrentLine, 1))
slNewLine = slNewLine & slGetLine
If Right\$(slNewLine, 2) <> slJoinChr Then
Exit Do
End If
Loop
End If
If blnlLoopUp = True Then
' We already have the previous line.
InglCurrentLine = IngpCurrentModuleLineNum - 1
Do
InglCurrentLine = InglCurrentLine - 1
slGetLine = vbcmpCodeModule.Lines(InglCurrentLine, 1)
If Right\$(slGetLine, 2) <> slJoinChr Then
Exit Do
End If
slNewLine = slGetLine & slNewLine
Loop
End If
' Get rid of the continuation chrs.
slNewLine = Replace(slNewLine, slJoinChr, "")
' Set up any &s correctly.
slNewLine = Replace(slNewLine, "&", " & ")
' Get rid of lots of spaces.
slNewLine = fncRemoveDoubleSpaces(slNewLine)

IngpCurrentModuleLineNum = IngpCurrentModuleLineNum + 1
fncGetSingleLine = sNewLine
'

End Function
Function fncRemoveDoubleSpaces(_
spLine As String _
) _
As String
If LenB(spLine) = 0 Then
fncRemoveDoubleSpaces = ""
Exit Function
End If
Do
If InStr(spLine, " ") > 0 Then
spLine = Replace(spLine, " ", " ")
Else
Exit Do
End If
Loop
fncRemoveDoubleSpaces = spLine
'

End Function

fncGetSingleLine, only returns a string. It doesn't do anything with it. I've included a procedure to remove double spaces as well.

I do Declare

While we're on the subject of continuations, don't you just get frustrated by all the different ways people write API Declarations!? Sometimes they are on one single line meaning you need to scroll the window to see the parameters. Other times they're split into a totally arbitrary number of lines with totally arbitrary indentation, if any!

To fix this we'll:

- Grab All of the declaration lines
- Join each one up into single lines
- Separate them out and indent them "properly"
- Delete all of the old declaration lines
- Insert our new set of pretty lines

Wait a minute! Haven't we just written a procedure to join up lines? That's handy!

We'll add someother functions as well.

- `fncGetDeclarations`
Of course!
- `fncGetJoinedArray`
This will take an array of lines with continuation characters and return an array of single lines.

At the moment, our function `fncGetSingleLine` gets it's lines directly from the code. Rather than alter that, we'll build another function based on that that works with an array so that we have both. Good eh?! I'm going to rename `fncGetSingleLine` to `fncGetSingleLineFromCode` and call the new one `fncGetSingleLineFromArray`. Snappy huh!

Anyway... here's the `fncGetDeclarations` function.

60 `fncGetDeclarations`

	Function <code>fncGetDeclarations()</code>
	<code>Dim sIDeclarations() As String</code>
	<code>Dim vbplProject As VBIDE.VBProject</code>
	<code>Dim vbclComponent As VBIDE.VBComponent</code>
	<code>Dim vbcmlCodeModule As VBIDE.CodeModule</code>
	<code>Dim vbcplCodePane As VBIDE.CodePane</code>
	<code>Dim sLines As String</code>

Dim lnglCountOfDeclarationLines As Long
'Set vbplProject = Application.VBE.ActiveVBProject
Set vbcmlCodeModule = Application.VBE.ActiveCodePane.CodeModule
lnglCountOfDeclarationLines = vbcmlCodeModule.CountOfDeclarationLines
slLines = vbcmlCodeModule.Lines(1, lnglCountOfDeclarationLines)
slDeclarations = Split(slLines, vbCrLf)
fncGetDeclarations = slDeclarations()
'

End Function

And the fncGetJoinedArray function that calls fncGetSingleLineFromArray.

61 fncGetJoinedArray

Function fncGetJoinedArray(spLines() As String)
' Return an array of joined lines.
' Skip comments and blank lines.
'
' I built this mostly to return joined
' declaration lines though it will
' work for any set of passed lines.
'
Dim lnglN As Integer
Dim lnglLineIndex As Integer
Dim lnglNumJoinedLines As Integer
Dim slJoinedLine As String
Dim slJoinedLinesArray() As String
Dim lnglUB As Integer
lnglLineIndex = 0
lnglNumJoinedLines = 0
lnglUB = UBound(spLines)
' First COUNT the lines for the array redim.

' Redim preserve Array(UBound(Array)+1) will work inside the
' loop but there seems to be quite an overhead for that.
Do
sIJoinedLine = _
Trim\$(fncJoinLines(spLines(), lngILineIndex))
If LenB(sIJoinedLine) = 0 Then
Elseif Left\$(sIJoinedLine, 1) = "" Then
Else
lngNumJoinedLines = lngNumJoinedLines + 1
End If
If lngILineIndex >= lngIUB Then
Exit Do
End If
Loop
' Redim the EXACT array size.
ReDim sIJoinedLinesArray(lngNumJoinedLines)
' Populate our array.
lngILineIndex = 0
lngNumJoinedLines = 0
Do
sIJoinedLine = _
Trim\$(fncJoinLines(spLines(), lngILineIndex))
If sIJoinedLine = "" Then
Elseif Left\$(sIJoinedLine, 1) = "" Then
Else
sIJoinedLinesArray(lngNumJoinedLines) = sIJoinedLine
lngNumJoinedLines = lngNumJoinedLines + 1
End If
If lngILineIndex >= lngIUB Then
Exit Do
End If
Loop
fncGetJoinedArray = sIJoinedLinesArray()
'

End Function

Whenever possible always count the dimensions of the array you want to end up with and try NOT to use

```
Redim preserve Array(UBound(Array)+1)
```

in a loop.

As stated in the procedure comments, ReDim Preserve will incur an overhead. Think about what needs to be done. Make another temporary array, save all of the information to it, redefine the original array one element bigger, copy the saved information back. It's not quite as simple as just adding more space to the end of the original array because for one thing the array formats have to be preserved... string, long, variant, object, whatever. Not so important in most cases maybe but think about having a couple of thousand lines in a separate module defining "global" variables and defining API calls.

Won't happen you say. HAH! Think about the number of pre defined variables Microsoft uses. All those vb variables and so on. There's a lot. There's more than a lot.

Having said that, If you know, **VERY DEFINATELY** that the number of lines in the declarations is **FIXED** at a small number then go right ahead. That will never **EVER** be updated. **Right?**

Hah!

Just a hint/reminder. Think year 2000. If you don't know about that, google is your friend.

About this code... fncGetSingleLine is a *general* function. Give it a set of lines and a line number and it will look up from that line number and down from that line number to check for continuation characters, and return a single line. We've also tried to take as many "scenarios" of code lines into consideration as possible. Having said that and written the function, we know that we are going to look at all the declaration lines and we know we're going to start from the top. The coding would be much simpler just going down line by line. We've also set things up to leave comments and commented continuations alone and not join them up into a single line. So in a way we've made things difficult for ourselves. Here is a simplified version of fncGetJoinedArray that simply loops down from the top to the bottom of the declarations. No need to call fncGetSingleLine at all! This version doesn't count the lines first either but uses the ReDim Preserve statement. We could use other methods to size an array. A common one is to use a very big array and then count upwards till we hit a non blank item. Then we have the size of our new array and can ReDim and populate it. Hang on... we did that in the main calling procedure subSplitDeclarations. Cool!

fncGetJoinedArray but just loops down and uses ReDim.

62 fncGetJoinedArrayFromDeclarations

	Function fncGetJoinedArrayFromDeclarations(_
	spLines() As String _
)

' Return an array of "joined" lines.
'
' This version just starts at the top and goes down.
' This uses Redim Preserve.
'
Dim lngIn As Long
Dim lngInArrayLineIndex As Long
Dim lngOutArrayLineIndex As Long
Dim sOutArray() As String
Dim lngNumJoinedLines As Long
Dim sJoinedLine As String
Dim sJoinedLinesArray() As String
Dim sCommentTest As String
Dim lngUBInArray As Long
Dim blnComment As Boolean
Dim sTrimmedLine As String
Dim sJoinChr As String
Dim sOriginalLine As String
Dim blnContinuation As Boolean
sJoinChr = " _"
lngOutArrayLineIndex = -1
lngNumJoinedLines = 0
lngUBInArray = UBound(spLines)
For lngInArrayLineIndex = 0 To lngUBInArray
Do
sOriginalLine = spLines(lngInArrayLineIndex)
sTrimmedLine = Trim\$(sOriginalLine)
If Left\$(sTrimmedLine, 1) = "" Then
lngOutArrayLineIndex = lngOutArrayLineIndex + 1
ReDim Preserve sOutArray(lngOutArrayLineIndex)
sOutArray(lngOutArrayLineIndex) = sOriginalLine
blnComment = True
Exit Do
End If

```

If bInComment = True Then
    If LenB(sTrimmedLine) > 2 Then
        If Right$(sTrimmedLine, 2) <> sJoinChr Then
            ' End of continuations in comments.
            ' Write the last one.
            lngOutArrayLineIndex = lngOutArrayLineIndex + 1
            ReDim Preserve sOutArray(lngOutArrayLineIndex)
            sOutArray(lngOutArrayLineIndex) = sOriginalLine
            bInComment = False
            Exit Do
        Else
            ' Comment and continuation.
            ' Write line.
            lngOutArrayLineIndex = lngOutArrayLineIndex + 1
            ReDim Preserve sOutArray(lngOutArrayLineIndex)
            sOutArray(lngOutArrayLineIndex) = sOriginalLine
            bInComment = False
            Exit Do
        End If
    Else
        ' End of continuations in comments.
        ' Write the last one.
        lngOutArrayLineIndex = lngOutArrayLineIndex + 1
        ReDim Preserve sOutArray(lngOutArrayLineIndex)
        sOutArray(lngOutArrayLineIndex) = sOriginalLine
        bInComment = False
        Exit Do
    End If
Else

```

	' Not in Comments.
	If blnContinuation = False Then
	If LenB(sTrimmedLine) > 2 Then
	If Right\$(sTrimmedLine, 2) = sJoinChr Then
	' First line of continuation.
	sJoinedLine = sTrimmedLine
	blnContinuation = True
	Exit Do
	End If
	End If
	End If
	If blnContinuation = True Then
	If LenB(sTrimmedLine) > 2 Then
	If Right\$(sTrimmedLine, 2) <> sJoinChr Then
	' End of continuations outside comments.
	' Add to Line.
	' Write the complete line.
	InglOutArrayLineIndex = InglOutArrayLineIndex + 1
	ReDim Preserve sOutArray(InglOutArrayLineIndex)
	sJoinedLine = sJoinedLine & " " & sTrimmedLine
	sOutArray(InglOutArrayLineIndex) = sJoinedLine
	blnContinuation = False
	Exit Do
	Else
	' In a continuation and this is a continuation as well.
	' Add to Line.
	sJoinedLine = sJoinedLine & " " & sTrimmedLine
	End If
	Else

' End of continuations outside comments.
' Add to Line.
' Write the complete line.
InglOutArrayLineIndex = InglOutArrayLineIndex + 1
ReDim Preserve sOutArray(InglOutArrayLineIndex)
sJoinedLine = sJoinedLine & " " & sTrimmedLine
sOutArray(InglOutArrayLineIndex) = sJoinedLine
bInContinuation = False
Exit Do
End If
Else
' Not a continued line.
' Write it away.
InglOutArrayLineIndex = InglOutArrayLineIndex + 1
ReDim Preserve sOutArray(InglOutArrayLineIndex)
sOutArray(InglOutArrayLineIndex) = sOriginalLine
bInContinuation = False
Exit Do
End If
End If
Exit Do
Loop
Next InglInArrayLineIndex
fncGetJoinedArrayFromDeclarations = sOutArray()
' *****
End Function

Time Ladies and Gentlemen please!

In the last bit we talked about ReDim preserve having an overhead, and that counting items first to size an array and then populating it is usually quicker.

How do we know?

We time it of course!

Timing procedures for any computer program in any computer is tricky. The normal way is to catch the before and end times of running the procedure a **lot** of times and give an average. Even this is prone to being misleading because of all the other stuff going on in a computer. Maybe a background process is checking the internet just at the time you run a timing test. Perhaps the internet tries to connect to you. It could be that an anti virus program decides to do a scan. There's all those pesky svchost processes. If you use google chrome there will be **LOTS** of instances of that chugging away. It's instructive to run a program like the free AutoRuns by Mark Russinovich that was bought by Microsoft. Link in the appendices. The number of processes that run at startup is staggering! You just have to run task manager and count the number of running processes... if you can! With so much going on it's hardly suprising that timing a single procedure in VBA may or may not give a true measure of the time it takes to run. Hence doing it a zillion times, and even then you may not be sure. You need to take into account the time your timing procedure runs as well. It could even be said that timing the same processes on different computers is totally useless. Certainly, running timers on the same computer gives more of a **comparison** measure than an exact running time.

You can try to give yourself an edge though by closing down the internet and zapping all the processes you don't need and stopping any virus/malware programs. Most people don't bother.

However... it's better than having no metrics at all!

The VBA Timer() function returns the number of seconds elapsed since midnight using a single-precision floating point value. This is not a threaded control so may not be triggered at exactly the correct time. Delays can be caused by other applications and or system processes.

Lots of people are of the opinion that the VBA functions, Timer() and Now() are adequate to time their code. Timer() gets the number of milliseconds since midnight with a resolution of approximately 10 milliseconds. Now() has a resolution of approximately one second. Some people use the "tick count" which is the number milliseconds that have elapsed since Windows was started.

There are a number of ways of getting a more precise time measurement. The timeGetTime API call returns the number of milliseconds that have elapsed since Windows was started. The MSDN GetSystemTimePreciseAsFileTime function retrieves the current system date and time with the highest possible level of precision (<1us). The retrieved information is in Coordinated Universal Time (UTC)

format. The MSDN QueryPerformanceCounter function retrieves the current value of the performance counter, which is a high resolution (<1us) time stamp.

MSDN and other sources, seem to agree that the most accurate is to use an API call combination of QueryFrequency and QueryCounter. This gets the current value of the performance counter to <1μs.

I present in the appendices, a [High Definition timer](#) class that does that. It's not by me. I got it from the internet and as far as I know the original is by the people I list in the comments at the top of the class code. Copy this to a class module and rename the module cHiResTimer. Remember the name of the class module is important because that's the name of the class.

So, we have a timer class, cHiResTimer. How do we use it?

The same as any other class! An example was our cWhereAreWe class and our cNameExample class.

3. High Definition timer class Example

	' In an ordinary code module.
	Sub subtestcHiResTimer()
	Dim cHiResTimer as cHiResTimer
	Set cHiResTimer = New cHiResTimer
	cHiResTimer.Start
	' Code.
	cHiResTimer.Stop
	Debug.Print cTimer.Elapsed
	'

	End Sub

I'm pretty confident that the timing class tackles most timing questions fairly well, albeit perhaps a mite too accurately! I've used it with results consistent enough to draw conclusions about performance bottlenecks that have enabled me to alter code to noticeably improve speed.

Just a reminder. We're talking about a class to test and report how long a procedure or some code takes to execute, so we can measure the speed of doing things one way or another way.

Where you put the code is dependent on whether you want to report exact timings with all of the snakes and worms or whether you want to average over a large number of iterations. Sometime you have no choice especially when talking about altering code with code.

Insert timing code

Inserting code is something we've done before. What we haven't done is checked Dims for needed definitions. Specifically, in this case...

Dim cHiResTimer as cHiResTimer

Collecting the Dims though is something we have done. Remember subInsertGetProperties? We collected a set of Dims and inserted them altered for module level in the declarations. We don't need to do as much here. But! Having a procedure to return an array of Dims for a procedure may be useful in its own right! Even better might be to put them into cWhereAreWe so we have them set up when we instantiate that class. We also have an item returned from that class that tells us if the Dims are all together, where they start and where they end.

So, here's the code I added to cWhereAreWe to return the Dims and DimCount.

63 WhereAreWe Update for Dims Array and Dims Count

' Count All of the Dims.
InglDimCount = 0
If InglLastDimLinePlus1 = 0 Then
InglCLine = InglModuleProcBodyLine
Else
InglCLine = InglLastDimLinePlus1 - 1
End If
Do
sLine = Trim\$(vbcmIcodeModule.Lines(InglCLine, 1))
If LenB(sLine) <> 0 Then
sLineArray = Split(sLine, " ")
Select Case sLineArray(0)
Case "Dim", "Const", "Static"
InglDimCount = InglDimCount + 1
End Select
End If
InglCLine = InglCLine - 1

	If lnglCLine <= lnglLineAfterDefinitionPlus1 - 1 Then
	Exit Do
	End If
	Loop
	ReDim sDimsArray(lnglDimCount - 1)
	' Populate the array.
	If lnglLastDimLinePlus1 = 0 Then
	lnglCLine = lnglModuleProcBodyLine
	Else
	lnglCLine = lnglLastDimLinePlus1 - 1
	End If
	lnglDimCount = -1
	Do
	lnglDimCount = lnglDimCount + 1
	sLine = Trim\$(vbcmlCodeModule.Lines(lnglCLine, 1))
	If LenB(sLine) <> 0 Then
	sLineArray = Split(sLine, " ")
	Select Case sLineArray(0)
	Case "Dim", "Const", "Static"
	sDimsArray(lnglDimCount) = sLine
	End Select
	End If
	lnglCLine = lnglCLine - 1
	If lnglCLine <= lnglLineAfterDefinitionPlus1 - 1 Then
	Exit Do
	End If
	Loop

Along with adding the property procedures and the module level variables.

Now, where were we? Oh yes... We need to check for

```
Dim cHiResTimer as cHiResTimer
```

Now we have a list of Dims from cWhereAreWe, it becomes almost trivial!

The following code will check for that Dim line. If it's there we can fairly safely assume that the line to instantiate the class...

```
Set cWhereAreWe = New cWhereAreWe
```

... is there as well. But if we want to be doubly sure, we can always check for that too! We have all of the lines of code in an array from cWhereAreWe! Just checking the Dims though is a teeny bit shorter.

After we are pretty sure those lines are in the procedure a line to start the timer is put above the line we are on and one to stop the timer below the line we are on. Finally, we can insert a debug.print to tell us the elapsed time. So, to insert timing at a line... maybe one that runs another procedure... we put our cursor on it and run our subcInsertTimerAtLine routine.

Here is the code to just check the Dims. Do you see the assumption in the function?

64 fncLookThroughDims 1

```
Sub subtestfncLookThroughDims()  
MsgBox fncLookThroughDims("Dim cHiResTimer As CHiResTimer")  
' *****  
End Sub  
Function fncLookThroughDims( _  
    spLookFor As String _  
    ) _  
    As Boolean  
  
    Dim cWhereAreWe As cWhereAreWe  
    Dim sIDimsArray() As String  
    Dim lngIDimCount As Long  
    Dim lngIN As Long  
    Dim bInIFound As Boolean  
    Dim sLLookFor As String  
    Dim sIDimLine As String  
  
    Set cWhereAreWe = New cWhereAreWe
```

	sIDimsArray = clWhereAreWe.DimsArray
	lngIDimCount = clWhereAreWe.DimCount
	blnFound = False
	sIDimLine = spLookFor
	For lngIN = 0 To lngIDimCount - 1
	sIDimLine = sIDimsArray(lngIN)
	If sIDimLine = sIDimLine Then
	blnFound = True
	Exit For
	End If
	Next lngIN
	fncLookThroughDims = blnFound
	' *****
	End Function

Because you will probably run this from a procedure that has already instantiated cWhereAreWe, you may choose to pass sIDimsArray instead. And here's how that would look.

65 fncLookThroughDims 2

	Sub subtestfncLookThroughDims()
	Dim clWhereAreWe As cWhereAreWe
	Dim sIDimsArray() As String
	Dim lngIDimCount As Long
	Set clWhereAreWe = New cWhereAreWe
	sIDimsArray = clWhereAreWe.DimsArray
	lngIDimCount = clWhereAreWe.DimCount
	MsgBox fncLookThroughDims(_
	"Dim clHiResTimer As CHiResTimer", _
	sIDimsArray _
)
	' *****
	End Sub

	Function fncLookThroughDims(_
	spLookFor As String, _
	spDimsArray() As String _
) _
	As Boolean
	Dim lngIN As Long
	Dim blnFound As Boolean
	Dim slLookFor As String
	Dim slDimLine As String
	blnFound = False
	slLookFor = spLookFor
	For lngIN = 0 To lngIDimCount - 1
	slDimLine = spDimsArray(lngIN)
	If slDimLine = slLookFor Then
	blnFound = True
	Exit For
	End If
	Next lngIN
	fncLookThroughDims = blnFound
	' *****
	End Function

Annnnnnd, here's the code to insert timing around a selection. This is very similar to the code [to insert Debug.Print around a line](#). Note that we append "' Timer code" to our inserted lines at the end so we can delete them when done. We could use another string. "' inserted." maybe... "' test" ... "' This line to be deleted" ... whatever.

The differences to subInsertSelectionDebug are:

- We look through the variables to check if we need to insert one
- We put code around a selection rather than a single line
- We insert an extra line to instantiate the class

Also note:

- We go backward **UP** the procedure when adding lines to avoid altering original line numbers collected by cWhereAreWe.
- We insert code to loop 1000 times.
- There is a msgbox at the end telling us we're "Done.". This isn't nesessary but very useful so that we aren't sitting there waiting after the code has run. I do this in a lot of procedures.

66 subccInsertTimingCodeAroundSelection

You should be used to the subtest code at the top by now.

Sub a__subtestsubccInsertTimingCodeAroundSelection()
subccInsertTimingCodeAroundSelection 1000
' *****
End Sub
Sub subccInsertTimingCodeAroundSelection(_
Optional lngpLoopCounter As Long = 1 _
)
' NOT BATCH.
' NO REPORT.
' NO END MESSAGE.
'
'
' Insert Timing Code Around the current line.
' Puts "" Timer code" at the end of the Inserted lines.
' D/Prints the module and procedure name and line
' number, and the DECIMAL elapsed time in milliseconds.
'
' The CHiResTimer class uses the APIs...
' QueryPerformanceFrequency
' QueryPerformanceCounter
'
' BASIC Code.
' Dim clTimer As CHiResTimer
' Set clTimer = New CHiResTimer
' clTimer.StartTimer
' clTimer.StopTimer
' Debug.Print clTimer.Elapsed
' Set clTimer = Nothing
'

	' ### Be aware that timing code is tricky because
	' of all the different things a computer is doing
	' all the time as well as the temperature and the
	' price of eggs.
	' It's <i>*very*</i> unlikely that two runs will ever
	' give the same/identical elapsed times.
	' It's possible that you may want to perform
	' the timing a number of times and average the
	' result.
	'
	Dim blnIMessageFound As Boolean
	Dim blnITimerClassVar As Boolean
	Dim clWhereAreWe As cWhereAreWe
	Dim ilN As Integer
	Dim lngIECol As Long
	Dim lngIELine As Long
	Dim lngIModuleEndLine As Long
	Dim lngIModuleLine As Long
	Dim lngISCol As Long
	Dim lngISLine As Long
	Dim slClassName As String
	Dim slCurrentLine As String
	Dim slDQ As String
	Dim slEndOfLine As String
	Dim slIndent As String
	Dim slMessageVar As String
	Dim slModuleName As String
	Dim slProcedureName As String
	Dim slSelection As String
	Dim slTimerClassVar As String
	Dim slVarArray() As String
	Dim vbcmlCodeModule As VBIDE.CodeModule
	Dim lngLastDimLinePlus1 As Long
	Dim slSetClass As String
	Dim slStartTimer As String
	Dim slStopTimer As String
	Dim slTimerToNothing As String
	Dim slTimeElapsed As String

Dim lnglDimCount As Long
Dim slStartLoop As String
Dim slEndLoop As String
Dim slLoopCounterVar As String
Dim blnLoopCountVar As Boolean
Set clWhereAreWe = New cWhereAreWe
lnglModuleLine = clWhereAreWe.ModuleProcBodyLine
lnglModuleEndLine = clWhereAreWe.ModuleProcEndLine
slModuleName = clWhereAreWe.ModuleName
slSelection = clWhereAreWe.Selection
lnglSLine = clWhereAreWe.CurrentModuleLineNum
lnglSCol = clWhereAreWe.SCol
lnglELine = clWhereAreWe.ELine + 1
lnglECol = clWhereAreWe.ECol
slCurrentLine = clWhereAreWe.CurrentLineText
slProcedureName = clWhereAreWe.ProcedureName
slIndent = clWhereAreWe.FirstIndent
lnglLastDimLinePlus1 = clWhereAreWe.LastDimLinePlus1
lnglDimCount = clWhereAreWe.DimCount
Set vbcmCodeModule = clWhereAreWe.CodeModule
slDQ = Chr(34)
slEndOfLine = " ' Timer code"
slTimerClassVar = "cTimer"
slClassName = "CHiResTimer"
slLoopCounterVar = "lnglLoopCounter"
slSetClass = "Set " & slTimerClassVar _
& " = New " _
& slClassName _
& slEndOfLine
slStartLoop = "For " & slLoopCounterVar & " = 1 to " & lngpLoopCounter _
& slEndOfLine
slStartTimer = "clTimer.StartTimer" _

& sEndOfLine
slEndLoop = "Next " & slLoopCounterVar _
& sEndOfLine
slStopTimer = "clTimer.StopTimer" _
& sEndOfLine
slTimeElapsed = "Debug.Print "
slTimeElapsed = slTimeElapsed _
& slDQ & slModuleName & slDQ
slTimeElapsed = slTimeElapsed _
& " & " & slDQ & "/" & slDQ
slTimeElapsed = slTimeElapsed _
& " & " _
& slDQ & slProcedureName & slDQ
slTimeElapsed = slTimeElapsed _
& " & " & slDQ & "/" & slDQ
slTimeElapsed = slTimeElapsed _
& " & " _
& slDQ & CStr(IngISLine) & slDQ
slTimeElapsed = slTimeElapsed _
& " & " & slDQ & "/" & slDQ
slTimeElapsed = slTimeElapsed _
& " & " _
& slDQ & "Time elapswed : " & slDQ
slTimeElapsed = slTimeElapsed _
& " & " _
& "CStr(clTimer.Elapsed)"
slTimeElapsed = slTimeElapsed _
& sEndOfLine
slTimerToNothing = "Set clTimer = Nothing" _
& sEndOfLine
' -----
' Put the built lines into the code.

vbcmlCodeModule.InsertLines _
Line:=InglELine, _
String:=slTimerToNothing
vbcmlCodeModule.InsertLines _
Line:=InglELine, _
String:=slTimeElapsed
vbcmlCodeModule.InsertLines _
Line:=InglELine, _
String:=slStopTimer
vbcmlCodeModule.InsertLines _
Line:=InglELine, _
String:=slEndLoop
vbcmlCodeModule.InsertLines _
Line:=InglSLine, _
String:=slStartLoop
vbcmlCodeModule.InsertLines _
Line:=InglSLine, _
String:=slStartTimer
vbcmlCodeModule.InsertLines _
Line:=InglSLine, _
String:=slSetClass
' Timer Code inserted.
' -----
' Check for variables.
blnTimerClassVar = False
blnLoopCountVar = False
' If there are no dims then we have to insert ours.
' Rather than use a function to see if the array is
' allocated, we can just check DimCount.

	If lnglDimCount > 0 Then
	' cWhereAreWe also passes a plain list of local variables.
	' We could equally use fncLookThroughDims.
	slVarArray = clWhereAreWe.LocalVariablesArray
	For iIN = 0 To UBound(slVarArray)
	If slVarArray(iIN) = slTimerClassVar Then
	bInlTimerClassVar = True
	Exit For
	End If
	Next iIN
	For iIN = 0 To UBound(slVarArray)
	If slVarArray(iIN) = slLoopCounterVar Then
	bInlLoopCountVar = True
	Exit For
	End If
	Next iIN
	End If
	If Not bInlTimerClassVar Then
	vbcmlCodeModule.InsertLines _
	Line:=lnglLastDimLinePlus1, _
	String:= _
	"Dim " & slTimerClassVar & " As " & slClassName
	End If
	If Not bInlLoopCountVar Then
	vbcmlCodeModule.InsertLines _
	Line:=lnglLastDimLinePlus1, _
	String:= _
	"Dim " & slLoopCounterVar & " As Long" & slClassName
	End If
	' *****
	End Sub

A Timer example

Hmmmmmm, Lets run some of what we've just done and see what happens.

We were talking way back about Redim Preserve having an overhead, so let's check it out. To be significant. we need to do it a lot though!

Here's the test code without timing code.

67 Test ReDim code for Timing

	Sub subtestRedim()
	' This is to test the timing of redim preserve against counting
	' and doing a single redim.
	'
	' It is hard coded to look at all the modules here and create an array
	' of all the lines.
	'
	' Because it doesn't alter anything it's okay to look at this module as well.
	' Remember, we're not doing anything with the lines, just populating an
	' array of them.
	'
	Dim lngTotalLineCount As Long
	Dim lngModuleLineCount As Long
	Dim lngModuleLineNumber As Long
	Dim sLine As String
	Dim sAllProjectLinesRedimPreserve() As String
	Dim sAllProjectLinesCountFirst() As String
	Dim vbplProject As VBIDE.VBProject
	Dim vbcmCodeModule As VBIDE.CodeModule
	Dim vbclComponent As VBIDE.VBComponent
	Dim clTimer As CHiResTimer
	Set vbplProject = Application.VBE.ActiveVBProject
	' -----
	' Redim Preserve.

```

InglTotalLineCount = 0
For Each vbclComponent In vbplProject.VBComponents
  Set vbcmlCodeModule = vbclComponent.CodeModule
  With vbcmlCodeModule
    InglModuleLineCount = 0
    Do
      InglModuleLineCount = InglModuleLineCount + 1
      InglTotalLineCount = InglTotalLineCount + 1
      If InglModuleLineCount >= .CountOfLines Then
        Exit Do
      End If
    Loop
    sLine = vbcmlCodeModule.Lines(InglModuleLineCount, 1)
    ReDim Preserve sAllProjectLinesRedimPreserve(InglTotalLineCount)
    sAllProjectLinesRedimPreserve(InglTotalLineCount) = sLine
  End With
Next vbclComponent

' -----
' Count First.

InglTotalLineCount = 0
For Each vbclComponent In vbplProject.VBComponents
  Set vbcmlCodeModule = vbclComponent.CodeModule
  With vbcmlCodeModule
    InglModuleLineCount = 0
    Do
      InglModuleLineCount = InglModuleLineCount + 1
      InglTotalLineCount = InglTotalLineCount + 1
      If InglModuleLineCount >= .CountOfLines Then
        Exit Do
      End If
    Loop
  End With

```

	Next vbclComponent
	ReDim sAllProjectLinesCountFirst(InglTotalLineCount)
	InglTotalLineCount = 0
	For Each vbclComponent In vbplProject.VBComponents
	Set vbcmlCodeModule = vbclComponent.CodeModule
	With vbcmlCodeModule
	InglModuleLineCount = 0
	Do
	InglModuleLineCount = InglModuleLineCount + 1
	InglTotalLineCount = InglTotalLineCount + 1
	sLine = vbcmlCodeModule.Lines(InglModuleLineCount, 1)
	If InglModuleLineCount >= .CountOfLines Then
	Exit Do
	End If
	sAllProjectLinesCountFirst(InglTotalLineCount) = sLine
	Loop
	End With
	Next vbclComponent
	Stop
	MsgBox "Done."
	' *****
	End Sub

Run it. The Stop at the end will allow you to look at sAllProjectLinesRedimPreserve and sAllProjectLinesCountFirst in the watch window. The two arrays should be identical.

Now select each different section in turn and run our a__subtestsubccInsertTimingCodeAroundSelection. Isn't putting a__ at the front of procedures and having the macros (Hissssss) button on the toolbar useful! You can delete the stop afterwards as well.

68 subtestRedim after adding Timer code

	Sub subtestRedim()
	' This is to test the timing of redim preserve against counting

' and doing a single redim.
'
' It is hard coded to look at all the modules here and create an array
' of all the lines.
'
' Because it doesn't alter anything it's okay to look at this module as well.
' Remember, we're not doing anything with the lines, just populating an
' array of them.
'
Dim lngTotalLineCount As Long
Dim lngModuleLineCount As Long
Dim lngModuleLineNumber As Long
Dim sLine As String
Dim sAllProjectLinesRedimPreserve() As String
Dim sAllProjectLinesCountFirst() As String
Dim vbplProject As VBIDE.VBProject
Dim vbcmCodeModule As VBIDE.CodeModule
Dim vbclComponent As VBIDE.VBComponent
Dim lngLoopCounter As Long
Dim clTimer As CHiResTimer
Set vbplProject = Application.VBE.ActiveVBProject
' -----
' Redim Preserve.
Set clTimer = New CHiResTimer ' Timer code
clTimer.StartTimer ' Timer code
For lngLoopCounter = 1 To 1000 ' Timer code
ReDim sAllProjectLinesRedimPreserve(0)
lngTotalLineCount = 0
For Each vbclComponent In vbplProject.VBComponents
Set vbcmCodeModule = vbclComponent.CodeModule
With vbcmCodeModule
lngModuleLineCount = 0
Do
lngModuleLineCount = lngModuleLineCount + 1
lngTotalLineCount = lngTotalLineCount + 1

	If lnglModuleLineCount >= .CountOfLines Then
	Exit Do
	End If
	slLine = vbcmlCodeModule.Lines(lnglModuleLineCount, 1)
	ReDim Preserve slAllProjectLinesRedimPreserve(lnglTotalLineCount)
	slAllProjectLinesRedimPreserve(lnglTotalLineCount) = slLine
	Loop
	End With
	Next vbclComponent
	Next lnglLoopCounter ' Timer code
	clTimer.StopTimer ' Timer code
	Debug.Print "zmTimingCode" & "/" & "subtestRedim" & "/" & "280" & "/" & "Time elapsed:" & CStr(clTimer.Elapsed)
	Set clTimer = Nothing ' Timer code
	' -----
	' Count First.
	Set clTimer = New CHiResTimer ' Timer code
	clTimer.StartTimer ' Timer code
	For lnglLoopCounter = 1 To 1000 ' Timer code
	ReDim slAllProjectLinesCountFirst(0)
	lnglTotalLineCount = 0
	For Each vbclComponent In vbplProject.VBComponents
	Set vbcmlCodeModule = vbclComponent.CodeModule
	With vbcmlCodeModule
	lnglModuleLineCount = 0
	Do
	lnglModuleLineCount = lnglModuleLineCount + 1
	lnglTotalLineCount = lnglTotalLineCount + 1
	If lnglModuleLineCount >= .CountOfLines Then
	Exit Do
	End If
	Loop
	End With
	Next vbclComponent

```

ReDim sAllProjectLinesCountFirst(InglTotalLineCount - 1)

InglTotalLineCount = 0
For Each vbclComponent In vbplProject.VBComponents
  Set vbcmlCodeModule = vbclComponent.CodeModule
  With vbcmlCodeModule
    InglModuleLineCount = 0
    Do
      InglModuleLineCount = InglModuleLineCount + 1
      InglTotalLineCount = InglTotalLineCount + 1
      sLine = vbcmlCodeModule.Lines(InglModuleLineCount, 1)
      If InglModuleLineCount >= .CountOfLines Then
        Exit Do
      End If
      sAllProjectLinesCountFirst(InglTotalLineCount) = sLine
    Loop
  End With
Next vbclComponent
Next InglLoopCounter ' Timer code
clTimer.StopTimer ' Timer code
Debug.Print "zmTimingCode" & "/" & "subtestRedim" & "/" & "313" & "/" & "Time elapswed :" & CStr(clTimer.Elapsed)
Set clTimer = Nothing ' Timer code

MsgBox "Done."
' *****
End Sub

```


69 Timing results after three iteration of 1000

	zmTimingCode/subtestRedim/280/Time elapsed :45.7332562325209
	zmTimingCode/subtestRedim/313/Time elapsed :42.7459380434622
	zmTimingCode/subtestRedim/280/Time elapsed :45.7286757545452
	zmTimingCode/subtestRedim/313/Time elapsed :42.1558093286235
	zmTimingCode/subtestRedim/280/Time elapsed :45.5667217621354
	zmTimingCode/subtestRedim/313/Time elapsed :41.9430500884319

The above is a copy paste from the immediate window. You can see that counting first is indeed faster. You can also see that I've made a spelling mistake. Ah well. However, that's for 1000 loops! If you play around with the number of loops, say take it down to 100. Then the difference is quite small and sometimes counting actually takes longer! It's up to you then, and what you are doing inside your code. But at least you now have the tools to get some numbers! Given all the code we've built so far, it should be relatively simple to add timing code for whole procedures throughout the project. This would give a "report" in the immediate window, though it could be to a file as we did when compiling, that you could look at while commuting on the train or whatever, in order to make decisions about which code to trim down, look at to improve, split up, get rid of, or just leave alone.

It's Complicated

While we are looking at performance, we can also measure how complex our procedure is. Sort of.

There are a lot of "metrics" for program code. Some of these are pretty self evident. All of the following can increase the "complexity":

- More code lines
- More variables
- More "branches" If .. For.. and so on
- Deeper nesting

Warning here: All of the next are my thoughts, that's the proviso.

There have been soooo many attempts to produce code metrics that will definitively say, this code is good or this code is bad. IMHO they've all failed. Put them all together though and we can get a picture of the code without even looking at it. I think it still doesn't mean much unless you compare it to other code stats that were produced by the same people in the same organization with the same rules with similar code, and possibly when the wind was from the south, or north, or maybe east.

If you are going to use these metrics you need to build up a subjective knowledge of this stuff over time for **yourself** and in **your** situation for the code **you** write. If your calculations consistently give a CC of 15... which is classed as "Complex Code", but you find that that's okay for you and your organization then that's exactly what it is. Okay for you and your organization. Actually, the next classifications description 20-40, is "Very complex code". Pretty definitive huh!?

I think what I'm getting at is that code metrics work well **COMPARATIVELY** but on their own they can only produce very course results at best and misleading at worst.

That's not to say they aren't useful! If one of the numbers is way way high, there is probably a problem or at the very least something that needs to be looked at.

As I said, there are a lot of different "measures" and there are at least as many tools to measure them!

"Profiling" a piece of code with metrics is great. We can tell how "complex" it is relative to the rest of our code and how quickly or slowly it runs compared to the rest of our code.

And there's the rub. "Relative", "Compared", to the rest of our code". To gain an overview we need to apply this to a number of procedures. The more the better. Possibly every one of them in every module. Wo/Manually doing this is out of the question. At least in my world!

To do this though we would have to go through the whole project and in every proc in all modules insert our timing code.

Just a mo though! We've got code to loop through procedures in a module and to loop through modules in a procedure! And even loop through lines in a procedure1

WOW... Are we good or what!

VBA Key Words

I'm including this as a sort of novelty. However, I use it in my add-in and has saved my bottom more than a few times.

If you want to see if a "word" is defined somewhere in your project you're usually SOL. **Typlib libraries** give you all of the keywords if they are set up. Accessing them is quick and easy and then you have to check that your word is or isn't there. But it ain't simple to do.

There maaaaayyyyyy be an easier way.

Let me ask you, how do you know if what you are typing in is a keyword or something already defined and not needing to be added to the Dims or whatever.

You do just that... type it in, but you do it in lower case.

If it's a keyword, trust me, the case will change. This is consistent across all of VBA as far as I can tell and I've spent a loooooong time looking. There is no keyword that is all lower case. Anywhere.

Hmmmmm.

So, we can type/insert a word into a procedure in all lower case and if the case alters it's a keyword.

For this we need to add a new project. Remember remember **TENET ONE**. What was it again? Oh yeah... don't do it to yourself! We do all this somewhere else. We can add a new project by adding a new workbook/project/presentation/document.

We can add a new module in the new VBA project there and we can add a new procedure in that module. Hold on though. We've done that before I think!

Anyway, we also know how to add code to that procedure!

Here it gets a bit tricky dickie.

If we have a word we want to test to see if it's a keyword we would normally just type it in. In lower case. But! That won't get evaluated until it's in a **valid** expression.

Try it. New module... add a sub... in the sub type "Mike&Lisa"... or anything. Whatever you type will be accepted because VBA thinks it's a new variable with upper and lower case. It won't get evaluated and give an error until you try running the procedure. You do have Option Explicit set don'tchya.

In the same procedure now add "Dim Mike As String". Then underneath that, or anywhere really add a line mike ="stupid". The case of mike will change from mike to Mike.

Don't panic! This is normal and you are probably well aware of it. In fact, it's so normal as to not be mentioned very much!

We can do something with that.

In our new module in our new project in our new procedure try a few keyword you know... Try "dim"... on its own. You'll get an error. **But!** dim will stay all lower case.

VBA will parse lines and if valid will enter it into the VBE. **THEN** it will change any case that needs to be changed.

ALL VBA keywords change case because none of them are all lower case. **None**. That appears to be how Microsoft has designed things. There is noooooo reason after sooooo many years, to even **imagine** they are going to change.

And, if you stick to a naming convention that includes upper and lower case, so will **all** of your procedure calls and variables as well! They will **all** be included in the **VALID** list.

Did I mention previously that I hate single character lower case variables. "i" is a favourite closely followed by "j" then there's a gap till we get to "s" with "n" coming in a poor fourth. It's horrid!!

Getting back, and being brief, we add code to our new procedure in our new module in our new project that contains our word/variable/string all in lower case. Then we get it back and compare it to the original lower case. If it's different... It's a key word. The "trick" is that our word has to be part of a **VALID** line of code

Simple.

We need to differentiate from a user defined one though like a procedure definition. Easy peasy if our variables and procedure names are mixed case. Oh, I think I've mentioned that already.

If we find a keyword, we can add it to a list of key words. Where would we keep that I wonder. Oh, we have code to add to the registry! Maybe that's a good place.

Anyroad up, Here is a procedure created by code, to test for the keyword olPane.

	Public Sub subKWTest()

	Dim olpane
	Olpane
	a = olpane(a)
	olpane a = a
	olpane a = a then
	olpane a
	If a = a olpane
	If olpane Then
	Redim (olpane)
	Select olpane
	For Each a In olpane
	For Each olpane In a
	For Each a olpane a
	For olpane = a To a
	For a = olpane To a
	For a = a To olpane
	For a = a olpane a
	do olpane a
	olpane do
	Object.olpane
	While olpane
	Do Until olpane
	Loop Until olpane
	End Sub

It's clear from this that olPane is NOT a VBA keyword **because it stays lower case**. Some of the lines are in red meaning a syntax error. It doesn't matter. **As long as the line as a whole is valid**, we're good. We're not bothered about compile errors because there's totally no way in an IKEA kitchen we are going to try and run this!

I keep all of these "valid" lines in my INI file at the moment so I don't have to recode to add another line if I need to. Here are the INI entries. I hope you see the correlation with the above generated code.

	[KeyWordCodeTestLines]
	keywordCodeTest1=Dim KW
	keywordCodeTest2=KW
	keywordCodeTest3=a = KW(a)
	keywordCodeTest4=KW a = a

keywordCodeTest5=KW a = a then
keywordCodeTest6=KW a
keywordCodeTest7=If a = a KW
keywordCodeTest8=If KW Then
keywordCodeTest9=Redim (KW)
keywordCodeTest10=Select KW
keywordCodeTest11=For Each a In KW
keywordCodeTest12=For Each KW in a
keywordCodeTest13=For Each a KW a
keywordCodeTest14=For KW = a to a
keywordCodeTest15=For a = KW to a
keywordCodeTest16=For a = a to KW
keywordCodeTest17=For a = a KW a
keywordCodeTest18=do KW a
keywordCodeTest19=KW do
keywordCodeTest20=Object.KW
keywordCodeTest21=While KW
keywordCodeTest22=Do Until KW
keywordCodeTest23=Loop Until KW

I grab the whole section, strip off the front, replace "KW" with the word I want to test **IN LOWER CASE**, and then start inserting lines in the separate project/module/procedure.

After that I read the lines back and have a look at them. If the word is still lower case then it's **NOT** a VBA keyword.

As you can imagine there's an overhead. But **ALL** keywords are identified.

Other people have taken er, other approaches. Notably Chris Greaves.

Code for keywords

HTML Report

This has nothing to do with the VBE apart from creating a HTML report.

Reports of this type, that document values of things, can usually be separated out into a minimum of two parts. A heading and the items that are reported, and sometimes a footer/summary.

These parts can be sliced, diced, sorted, and ported, to procedures to create whatever format report you want in whatever application you want. CSV, Excel, Word, to name a few.

For CodeCode, the report will nearly always be as a table in a HTML file.

If you check out the example report of our [cross-reference procedure report](#) in the appendices, then there is a bit more. First, I print the items from **sub**WhereAreWe. Then I print the headers, then I print the details. All packaged nicely in a bunch of tables.

I present this “as is” as an example. Practical working code is in CodeCode. This is mostly to show how to build a HTML file.

Recap number three

Let's have a look at the code we've built so far... and yes I'm repeating so it looks more impressive.

	Last Recap
1	subccSortDims
2	subMsgBox
3	subccSortSelectedDims
4	fncGuessVarType
5	subWhereAreWe
6	cNameExample
7	subInsertGetProperties
8	cWhereAreWe
9	subccSplitAllDims
10	subccInsertSelectionDebug
11	subccDeleteDebugPrint
12	subClearIW
13	cBarEvents
14	subBrandNewBarAndButton
15	subBookMarkAndBreakpoint
16	frmMsgBox and using a form as a class to return information.
17	subListProcsToImmediateWindow
18	subAddTraceLinesToAModule
19	subDeleteDebugPrintForModule
20	subGoToLine
21	subccInsertModuleLineNumbersInProcedure
22	subccDeleteLineNumbersInProcedure
	Since last Recap.
23	We've talked about arranging procedures by name so the one you're testing is at the top of the (whisper) macros... menu
24	We've gone to the Dims Bottom
25	subccInsertSingleDim
26	subLoopThroughProLines

27	"Multithreaded" to create a compile report
28	Deleted lines tagged with end comments
29	subDeleteDoubleBlankLines
30	Concatonated continued lines to single lines from modules
31	Concatonated continued lines to single lines from an array
32	Formatted Declarations
33	Code to count and return Dims in a procedure
34	Added Timing code and code to add that code to do timing.
35	Added code to count various lines in a procedure for metrics like Cyclic complexity, Halstead metrics and Lines of code.
36	Code to list procedures and the top comments
37	Code to create a HTML report
38	Procedure to check keywords without using TLI.

Not bad!

What's to come?

Welllll, I want to talk about making all this stuff available in an Add-in. Being able to insert comments consistently formatted may be a good idea. My personal methodology for handling errors. Backing up, and adding an array for parameters to cWhereAreWe. I also want to present code to export a whole project either as separate modules or as a single text file. And of course, importing from that. A big thing about exporting to a text file is the ability to **SEE** differences using for example WinMerge. There are loads of proggies out there that will sort procedures but we'll do that too and chat about where to put procedures that are being worked on. A big advantage to an export/import process is that code gets "debloated". [Rob Boveys code cleaner](#) is perfect for doing just this and I mention it in the appendices. But be aware that as soon as the VBE compile process kicks in, it's big again, but maybe not quite so big as before.

Though the code is too large to comfortably present here, I also want to at least look at, what needs to be done to produce a Procedure Cross Reference. This has a bearing on inserting Dims into code where they aren't defined, and to cleaning a procedure of Dims that aren't used. I'm thinking of presenting those three as a separate item in the near future. It's a lot of VBA code.

Some code analyzing programs go much further than I have and do things like show up or delete code that will never be used/gone through, so called dead or redundant code. And they do other stuff like drawing flow charts and so on. I'm not that good. It would take me a loooong time to get as far as that.

However, I'll bet none of them have even all of the procedures we've covered here, hehehe. And all this in totally free VBA as well, for you to personalize tweak improve or whatever! No royalties or anything either!

That actually begs the question. Is any of this useful? If the big boys and girls haven't done it, is it worth it?

My opinion FWIW? Unequivitably, for me, **YES**.

I use this stuff and more all the time. I love having my Dims sorted. I think being able to insert and especially delete, **ALL** debugging code in one go is great! Being able to see if I already have code to do something is a boon and has saved me much repeat coding where in the back, well pretty much to the front actually, of my mind, I'm thinking "I'm sure I've written this before". The compile stuff has shown up multiple problems all in one go that I could solve by a one line change to a class Get procedure for example. I can insert a Dim for a local variable if I get a compile error without typing! The timing code has helped me find bottlenecks in procedures that needed recoding or splitting up. I **really** like having pretty formatted declarations that I can read without scrolling to the right. Being able to change all Integers to Long in a whole project at once has been terrific! We've also got code to go through each procedure in a project and apply a set of procedures to it. That means I can format **ALL** of my procedures in any or all of the open projects, in any way I choose with code I've written.

A **KEY** element of this has been the cWhereAreWe class and the associated subroutine. It's been updated and updated and updated and I think it's still got some legs. With this in mind, a date stamped and complete copy of the latest cWhereAreWe class module is on the web site. It's long. Our, Lisas' and mine, original **sub**WhereAreWe returned an array of about 70 items. Some of these were sort of "doubled up" because I returned the same line number items of the module for the array code so they didn't have to be recalculated, and some calculated values. Did I mention I was lazy? But still.

We have the Dims in there, we have the code in there, we have some metrics in there, pretty soon we'll have parameters in there. We have the project, codemodule, codepane and component in there. We even have **where we are** in there! Woohoo!

Again, and as always, if you think it needs an update and you do that and it works and is useful to you, then let me know why and so on along with your code and if it's etc ect you **WILL** get credit. I've been guilty of not crediting in the past and have been quite rightly booed and shouted at. I go to a lot of trouble to make every attempt to give credit where it's due.

Sharing your code with yourself

When you have your code to do stuff in the VBE you'll probably want it available to all of your VBA coding efforts. Of necessity this has to be on an application by application basis. There's our duplication of code problem again.

You need to create an **Add-In**.

Unfortunately the naming conventions for add-in files, how they work, and where the files are placed is different for each application.

Having said that, you can change at least the latter.

There are two ways you can use a VBA add-in. Note I say a VBA add-in. COM add-ins are beyond the scope of this document so I'm studiously ignoring them.

If you write a piece of VBA code that you want implemented across applications, it ain't easy to share. I think this is at least, and in MS WORDs case very much so, the fault of having different development teams, and that perhaps they didn't talk to each other. Probably the two most mature products in office age wise, are WORD and EXCEL. They work differently in so many respects that the above conclusion is not difficult to come to. I have to say though, that the latter versions have gone some way towards rectifying that.

Add-in files extensions for the big four are...

- Excel
xlam
- Powerpoint
potm
- Word
dotm
- Access
mda

The best way to install an add-in? By hand. In ACCESS you have no other choice. You have to insert a module and copy code to it.

This means that it's pretty well inevitable that there will eventually be a code version problem across applications. Using an add in goes at least part way to solving that. If you only ever use access VBA or Excel VBA or Word VBA, then you're quids in.

Excel

In Excel you can save the code in a .XLAM file.

Word

In Word you can use a .DOTM file... but it's not the same mechanism.

PowerPoint

To create a PowerPoint add-in, begin by creating a macro-enabled PowerPoint presentation (.pptm) file. PowerPoint uses a unique file format for add-ins (.ppam), but that format is read-only. So, you write and edit your code in the .pptm file and then save a copy as a .ppam add-in.

By default, loaded PowerPoint add-in files don't appear in the **Project Explorer** of the Visual Basic editor. To view and open loaded .ppam projects, add a new value to the Windows Registry. In the key HKEY_CURRENT_USER\Software\Microsoft\Office\14.0\PowerPoint\Options, add a **DWORD** value named **DebugAddins**, with a value of **1**. Then, restart PowerPoint. Note that you can view, export, and copy content from the loaded .ppam project. You can also make edits to test them in the loaded add-in, which makes the add-in function similar to a handy scratch pad when you're troubleshooting or want to try something new. However, you cannot save changes directly in the loaded add-in file.

Access

Some Tips hints and gotchas!

<http://www.granite.ab.ca/access/addins.htm>

You may or may not need an add-in because you can also set a reference to the database that contains your code.

Outlook

Add ins for outlook are pretty much silly. This is because you can only ever have one copy of outlook running at any one time on any single computer. Having said that it's not strictly true. You can have a local copy of outlook running and a web copy at the same time. But because the web versions of the office application don't do VBA then it's all hunky dory.

No Comment?

If we have a lot of procedures, then sometimes we tend to write another procedure to do the same thing as a procedure we've already written, just because we've forgotten, or can't be bothered to look for, the original code. Bin there, got multiple T-shirts, read the book and got the DVD.

If, and that's a big **IF**, we've named our functions subs and properties in a reasonably understandable way then we have a reprieve. Even better if we've included comments after the procedure definition. We come back now to a bit of standardization. Most people are free spirits. If you don't want to document, then don't. But here's code that will create a list of procedures with any comments below the definition. It's useful sometimes. Especially if you're working for a client. Hehehe. Don't they love their documentation. Don't get me wrong here, that's not so much of a joke!

This will create an array. The array could be passed to anywhere: a text file or back into word or excel for example. And there's a deliberate attempt to be "general" here. We create an array that can be used by other procedures whether it be one to create a HTML report, which I present later, or imported into a table in excel or word or even an access database.

The process is:

1. Loop through all procedures in all open projects in the VBE
 - We got that covered
2. Collect project name, module name, procedure name, module start line, number of lines
 - We can do that too
3. Create elements in an array for this
 - Awww Arrays are just big buckets
4. Sort it
 - Gonna steal JKPs' quicksort maybe!
5. Loop down through the code collecting comments till we hit a non comment line
 - Hey We've done that before
6. Add the comments to the array
 - See 3

Inserting comments

Code to insert comments with userform.

Runtime Errors

Errors like "Variable not defined" are **compile** errors. Procedures will not even start to run until they are resolved.

Errors that occur at **runtime** may include subscript errors, type errors, and lots of others. Those two though, are I think possibly the most common ones.

There are a number of ways of dealing with runtime errors. Some are:

1. Ignore them
2. Put general error code into procedures and possibly loop back
3. Try to deal with errors as they occur
4. Stop the code where it is
5. Quit with a message
6. Just Quit
7. Use third party software like vbWatchdog

I go with number three. My reasoning is that I don't want the user to get an **error message at all!**

If you have no code at all for errors then you are asking for trouble and phone calls in the night from irate users and others.

If you put **general** error code into a procedure then almost by definition you will report an error to the user and stop. The result is a slightly less irate user who **may** call in the morning.

If there is an error in "normal" code, there is something wrong with something else on that system, and not your code. And it does happen. More than you'd think. I've had errors saying the function Left\$ isn't known/recognized. That's really scary. It's an inbuilt string function! And it's tricky to track down. SO, a user might get a message, or should, that there is an error in module SortDims and to call 0002020002022202. That's not a valid phone number BTW. You drag yourself to a computer at 04:27 and track finally down the errant line. It's:

```
slLine = Left$(slLine,1)
```

WT* is wrong with that!?

Deleting any missing references and the Temp folder, clearing the cache, and possibly rebooting, will in my humble experience get rid of that particular message. Then there's the little-known String object. You can always change string functions to String.Function. Intellisense will help there. There's little you can

do about that stuff except be aware of it and have an idea of how to deal with it. If it's going to happen it will, wether you like it or not.

Back to normal errors.

Most people who code, are fairly well aware of items that could produce an error. Database access, File access, Network access. Subscripts in arrays. Empty or Null strings. If you've tested enough, you're damn **sure** where there may be problems. My own personal preference for dealing with errors is to identify as many of those places as possible and put specific code there to cope with problems.

This takes the form of:

70 My method of dealing with errors

	Dim lngErrNumber As Long
	Dim slErrDescription As String
Then around a possible problem line:	
	On error resume next
	POSSIBLE ERROR LINE
Immediately after, set variables from the Err object.	
	lngErrNumber = Err.Number
	slErrDescription = Err.Description
Immediately after that turn error checking back on!	
	On error goto 0
Now deal with what's happened.	
	Select case lngErrNumber
	Case 0
	Case n1
	Case n2
	Case else
Consider always having an Else clause. It can be empty. But then you know it's empty! Same for If blocks.	
	End select

My methodology allows me to say:

- Hey guys we've got a problem here what do we do?
- Oh... it's a file error /empty string / negative index. Let's tell the user and go back and give her/him another chance.
- Oh I know this one, Nothing found. Tell User and go back.
- Oh crap. Not come across this before, can we rollback? Better say what's happening though and give the the user a choice of aborting or trying again.
- No input. I'm outa here. Think I should tell the user what's happening though.

Get my drift?

I can't remember where it came from but the philosophy is, **Don't leave your user in the dust!** And remember, it may be you! You can't help but bemusing the user though if you have a **general** procedure error routine. It's quite likely that if there is an error in modue doodad and procedure thingy you may know, and are able say, oh yeah, that has to be the foo error. Then say so!!! Better yet, deal with it and move on!

Dealing with errors in this way adds quite a few lines to the code. About 15 per possible error. I'm tempted to say, so what. Instead, I'll say, **it's worth it**. You get specific code and actions for specific or general errors.

Okay. Enough evangelizing.

Here's code to insert general error code around a line. In fact, it's nothing we haven't done before!

Note the end of line tags that are inserted so you can look for them to delete if you want.

Also note that this is, by its nature, generic, and needs updating if you suspect specific errors may occur. Having said that, you may wish to use a parameter or some such to specify a file access error or a database error or some such.

71 subccInsertErrorCodeAtLine

	Sub subccInsertErrorCodeAtLine()
	' NOT BATCH.
	' NO REPORT.
	' NO END MESSAGE.
	'
	' Insert Error Code Around the current line.
	' Puts "" Inserted" at the end of the lines.
	'

	Dim blnErrDescriptionFound As Boolean
	Dim blnErrNumberFound As Boolean
	Dim blnMessageFound As Boolean
	Dim clWhereAreWe As cWhereAreWe
	Dim lnglArrayStartOfCode As Long
	Dim lnglCurrentLine As Long
	Dim lnglIncr As Long
	Dim lnglModuleStartOfCode As Long
	Dim lnglModuleEndProcedureLineNumber As Long
	Dim lnglN As Long
	Dim lnglNumberOfCodeLines As Long
	Dim lnglStartOfDims As Long
	Dim lnglStartLine As Long
	Dim llStartOfSub As Long
	Dim slCodeLine As String
	Dim slDimList() As String
	Dim slDQ As String
	Dim slEndOfLine As String
	Dim slErrDescriptionVar As String
	Dim slErrNumberVar As String
	Dim slIndent As String
	Dim slMessageVar As String
	Dim slModuleName As String
	Dim slProcedureName As String
	Dim slVarArray() As String
	Dim slWhereAreWe() As String
	Dim vbclComponent As VBIDE.VBComponent
	Dim vbcmlCodeModule As VBIDE.CodeModule
	Dim lnglInsertPoint As Long
	' -----
	' Start.
	slDQ = Chr(34)
	slEndOfLine = "' Inserted."
	slErrNumberVar = "lnglErrNumber"

	sErrMsgDescriptionVar = "sErrMsgDescription"
	sMsgVar = "sMsg"
	Set clWhereAreWe = New cWhereAreWe
	InglArrayStartOfCode = clWhereAreWe.LastDimLinePlus1
	If InglArrayStartOfCode = 0 Then
	Exit Sub
	End If
	Set vbcmlCodeModule = clWhereAreWe.CodeModule
	InglStartLine = clWhereAreWe.ModuleProcBodyLine
	InglNumberOfCodeLines = clWhereAreWe.ProcCountLines
	sModuleName = clWhereAreWe.ModuleName
	lStartOfSub = clWhereAreWe.ModuleProcBodyLine
	sProcedureName = clWhereAreWe.ProcedureName
	InglStartOfDims = clWhereAreWe.FirstDimLine
	InglModuleStartOfCode = clWhereAreWe.LastDimLinePlus1
	InglModuleEndProcedureLineNumber = clWhereAreWe.ModuleProcEndLine
	InglCurrentLine = clWhereAreWe.CurrentModuleLineNum
	' Code gets inserted ABOVE the line number specified.
	InglInsertPoint = InglCurrentLine
	If InglModuleStartOfCode = InglModuleEndProcedureLineNumber Then
	' No code or Dims.
	InglInsertPoint = InglModuleEndProcedureLineNumber
	InglIncr = 1
	Else
	InglIncr = 2
	End If
	' Get the indent.
	sCodeLine = vbcmlCodeModule.Lines(InglCurrentLine, 1)

	sIIndent = fncGetIndent(sICodeLine)
	vbcmlCodeModule.InsertLines _
	Line:=InglCurrentLine, _
	String:= _
	sIIndent _
	& "On Error Resume Next" _
	& sIEndOfLine
	InglCurrentLine = InglCurrentLine + InglIncr
	vbcmlCodeModule.InsertLines _
	Line:=InglCurrentLine, _
	String:= _
	vbCrLf _
	& sIIndent _
	& "InglErrNumber = Err.Number" _
	& sIEndOfLine
	InglCurrentLine = InglCurrentLine + 2
	vbcmlCodeModule.InsertLines _
	Line:=InglCurrentLine, _
	String:= _
	sIIndent _
	& "sIErrDescription = Err.Description" _
	& sIEndOfLine
	InglCurrentLine = InglCurrentLine + 1
	vbcmlCodeModule.InsertLines _
	Line:=InglCurrentLine, _
	String:= _
	sIIndent _
	& "On Error GoTo 0" _
	& sIEndOfLine
	InglCurrentLine = InglCurrentLine + 1
	vbcmlCodeModule.InsertLines _
	Line:=InglCurrentLine, _
	String:= _
	sIIndent _

	& "Select Case lngErrNumber" _
	& sEndOfLine
	lngCurrentLine = lngCurrentLine + 1
	vbcmlCodeModule.InsertLines _
	Line:=lngCurrentLine, _
	String:= _
	sIndent _
	& "Case 0" _
	& sEndOfLine
	lngCurrentLine = lngCurrentLine + 1
	vbcmlCodeModule.InsertLines _
	Line:=lngCurrentLine, _
	String:= _
	sIndent _
	& "Case Else" _
	& sEndOfLine
	lngCurrentLine = lngCurrentLine + 1
	vbcmlCodeModule.InsertLines _
	Line:=lngCurrentLine, _
	String:= _
	sIndent _
	& " smessage = " _
	& sIDQ & "Procedure " _
	& sProcedureName & sIDQ _
	& sEndOfLine
	lngCurrentLine = lngCurrentLine + 1
	vbcmlCodeModule.InsertLines _
	Line:=lngCurrentLine, _
	String:= _
	sIndent _
	& " smessage = sMessage & vbCrLf & " _
	& sIDQ & "Module Line " _
	& CStr(lngCurrentLine - 13) & sIDQ _
	& sEndOfLine
	lngCurrentLine = lngCurrentLine + 1

	vbcmlCodeModule.InsertLines _
	Line:=InglCurrentLine, _
	String:= _
	sllndent _
	& " slmessage = slMessage & vbcrLf & " _
	& sIDQ & "Err Number " _
	& sIDQ & " & CStr(InglErrNumber)" _
	& slEndOfLine
	InglCurrentLine = InglCurrentLine + 1
	vbcmlCodeModule.InsertLines _
	Line:=InglCurrentLine, _
	String:= _
	sllndent _
	& " slmessage = slMessage & vbcrLf & " _
	& sIDQ & "Err > " _
	& sIDQ & " & slErrDescription" _
	& slEndOfLine
	InglCurrentLine = InglCurrentLine + 1
	vbcmlCodeModule.InsertLines _
	Line:=InglCurrentLine, _
	String:= _
	sllndent _
	& " submsgbox slMessage" _
	& slEndOfLine
	InglCurrentLine = InglCurrentLine + 1
	vbcmlCodeModule.InsertLines _
	Line:=InglCurrentLine, _
	String:= _
	sllndent _
	& "End Select" _
	& slEndOfLine _
	& vbCrLf
	'-----
	'#####

	' Code inserted... Check for variables.
	'sIVarArray = fncGetVarNamesFromArray(sIDimList())
	'subGetVarNamesFromDimList sIDimList(), sIVarArray()
	blnErrNumberFound = False
	blnErrDescriptionFound = False
	blnMessageFound = False
	If fnclsArrayAllocated(sIVarArray) = True Then
	sIVarArray = fncGetNames(sIDimList())
	For lngIN = 0 To UBound(sIVarArray)
	If sIVarArray(lngIN) = sErrNumberVar Then
	blnErrNumberFound = True
	End If
	If sIVarArray(lngIN) = sErrDescriptionVar Then
	blnErrDescriptionFound = True
	End If
	If sIVarArray(lngIN) = sMessageVar Then
	blnMessageFound = True
	End If
	Next lngIN
	End If
	If lngStartOfDims = 0 Then
	lngStartOfDims = clWhereAreWe.LineAfterDefinitionPlus1 ' CInt(sIWhereAreWe(9, 4)) + 1
	Else
	lngStartOfDims = clWhereAreWe.LastDimLinePlus1 'CInt(sIWhereAreWe(9, 1)) + 1
	End If
	If Not blnErrNumberFound Then
	vbcmlCodeModule.InsertLines _
	Line:=lngStartOfDims, _
	String:= _
	"Dim " & sErrNumberVar & " As Long"
	lngStartOfDims = lngStartOfDims + 1
	End If
	If Not blnErrDescriptionFound Then
	Stop

	vbcmlCodeModule.InsertLines _
	Line:=InglStartOfDims, _
	String:= _
	"Dim " & slErrDescriptionVar & " As String"
	InglStartOfDims = InglStartOfDims + 1
	End If
	If Not blnIErrorMessageFound Then
	Stop
	vbcmlCodeModule.InsertLines _
	Line:=InglStartOfDims, _
	String:= _
	"Dim " & slMessageVar & " As String"
	InglStartOfDims = InglStartOfDims + 1
	End If
	Stop
	' *****
	End Sub

Here is before and after code using our infamous subtestsubccWhereAreWe.

72 Before inserting error code

	Sub subtestCWhereAreWe()
	Dim clWhereAreWe As cWhereAreWe
	Dim slDimsArray() As String
	Dim slCodeArray() As String
	Dim lngDimCount As Long
	Dim vlVar As Variant
	Set clWhereAreWe = New cWhereAreWe
	slDimsArray = clWhereAreWe.DimsArray
	slCodeArray = clWhereAreWe.CodeArray
	InglDimCount = clWhereAreWe.DimCount

Stop
' *****
End Sub

Put the cursor on line "InglDimCount = clWhereAreWe.DimCount" and run subccInsertErrorCodeAtLine.

73 After inserting error code

Sub subtestCWhereAreWe()
Dim clWhereAreWe As cWhereAreWe
Dim slDimsArray() As String
Dim slCodeArray() As String
Dim InglDimCount As Long
Dim vlVar As Variant
Dim lngErrNumber As Long
Dim slErrDescription As String
Dim slMessage As String
Set clWhereAreWe = New cWhereAreWe
slDimsArray = clWhereAreWe.DimsArray
slCodeArray = clWhereAreWe.CodeArray
On Error Resume Next ' Inserted.
InglDimCount = clWhereAreWe.DimCount
InglErrNumber = Err.Number ' Inserted.
slErrDescription = Err.Description ' Inserted.
On Error GoTo 0 ' Inserted.
Select Case lngErrNumber ' Inserted.
Case 0 ' Inserted.
Case Else ' Inserted.
slMessage = "Procedure subtestCWhereAreWe" ' Inserted.
slMessage = slMessage & vbCrLf & "Module Line 55" ' Inserted.
slMessage = slMessage & vbCrLf & "Err Number " & CStr(InglErrNumber) ' Inserted.

	sIErrorMessage = sIErrorMessage & vbCrLf & "Err > " & sIErrDescription ' Inserted.
	subMsgBox sIErrorMessage ' Inserted.
	End Select ' Inserted.
	Stop
	' *****
	End Sub

And what do we do if we don't want the error code? We use the tags at the end of the lines to delete the inserted code!

Backup

You must backup.

This is because, as sure as the sun comes up every day and water is wet, you will need a backup at some time or other. I know of **no one, anywhere**, who hasn't needed to look at past code, or recover/restore, from backups.

It doesn't really matter how you do this. Some options are:

- Just save to a different filename every time you save.
- Use a third-party program.
 - Git
- Build your own system.
- Use a different program than the IDE for editing that has an autosave.
- Use code to do an autosave.

A lot depends on your situation.

If you are on a network of some sort in a multiple user environment, then there will be backups made anyway. If they're not then you really should consider changing your employer. You will need to contact the network admin to help you with restoring. In fact, you may very well be the network admin! Be careful not to overwrite the current code/files!!

If you are a single user, coding mostly for yourself with a small or even a large audience, you may choose to make an "on demand" backup. That is, manually backup before there are significant changes. You could implement an automatic backup, but that will possibly slow you down. This would be in the autoopen of the relative application addin. The code would probably use the OnTime function to start a backup every so often. We used OnTime in the Compile routines. I've "played" with add in auto open routines in add-ins, and I'm not happy with the result so cannot speak to them. Just be careful if you go down this route and test thoroughly! If you succeed I would **very** much like to hear from you.

I present here a procedure set to do an on demand backup.

The process is:

I emphasise that this is **no** substitute for a commercial product! It's very definitely home grown roll your own.

Parameters in WhereAreWe

Code to retrieve parameter names.

Export and Import

Notably there is an add-in to do this by Jan Karel Pieterse. Please **do** check [JKPs' link](#) in the appendix.

There is code here to export each module separately or to a single text file. I'm stealing that idea from Jan Karel in that I'm sorting the code modules in the single text file as well. We could even export a sorted list of procedures singly over all modules but I don't recommend that. The reason being dependencies. Exporting a procedure is fine as long as it doesn't call any other procedures. If it does, you're bugged. This is one of the reasons I advocate lots of modules. By setting up all of the called procedures in the same module as private, the module becomes "self supporting" as it were. This could mean multiple copies of the same procedure being in different modules and that has to be carefully monitored. The same goes for classes. By putting all the code needed for the class into the class we are more than likely going to have multiple copies of the same procedure code all over the place.

This is unavoidable in all but the most strictly controlled environments.

One way to do this is to have a librarian. That person would control, and be responsible for, any published code and make sure any new code is propagated through the rest of any relevant projects.

This is all just for us to use in our own VBE environment though, so it should be okay. Shouldn't it? Well let's assume that anyway.

Having said that, CodeCode, available for free from the web site, has a couple of tree controls that lists, those procedures being called by other procedures and those procedures that call other procedures and what procedures they call. There is a button on that form to create a module from a procedure. This will try and load the selected procedure and all of the called procedures and procedures they call into one module. I've used it and it works pretty well.

Phew!!! Was that a mouthful or what!

If we want to sort procedures, we need to pick up the procedure names, sort them, and then pick up the procedure code in that order, delete the old procedures and insert the sorted procedures in the new order. This isn't as simple as it sounds because we can't just sort all of the procedure names. We have to do it on a module by module basis. It's also a bit scary. I mean, c'mon, are you comfortable with deleting code you know works? Hehehe.

So the first thing we do is save everything. The whole shebang. The file containing the project and all the modules are saved to a folder with a date and time stamp. We don't delete that folder either. It's there until you are satisfied everything works and then you can delete it yourself. If you want.

This also has a lot to do with backups of course.

Code for export and import.

Sort Procedures

Making use of procedure names

Considerations for Inserting, Cross referencing and Cleaning

Amongst all the lists that VBA must use **internally for EVERY procedure**, is a list of variables local to that procedure, a list of module level variables for every procedure in a module, and a list of global variables for that project. It must also be able to resolve global procedures and variables that are visible to it possibly from other projects.

This isn't simple. I take my hat off to the dev team in fact!

Take a Cross reference. We need to:

- Know where we are so we can get the procedure code
- Create a list of **Defined variables** from the procedure
 - Dim
 - Const
 - Static
 - Parameters
- Create a list of **Used variables** from the procedure code
- Filter out key words
- Create a list of **available variables** from the **same module** as the procedure
 - Private Declare procedures
 - Private variables
 - Dim
 - Const
 - Static
 - Private ENums
 - Private Types
 - Global/Public Declare procedures
 - Global/Public variables
 - Global/Public ENums
 - Global/Public Types
- Create a list of **Available variables** in **other modules**
 - Public Procedures
 - Public variables
 - Dim
 - Const
 - Static
 - Public ENums
 - Public Types
 - Friend Procedures

- Create a list of **Available procedures** in the **same module** as the procedure
 - Private procedures
 - Public procedures
 - Friend Properties
 - Friend Procedures
- Create a list of **Available procedures** from **other modules** that do not use Option Private Module
 - Private procedures
 - Public procedures
 - Friend Properties
 - Friend Procedures
- Create a list of **Available items** from other projects that are **referenced** by this project
 - Public Procedures
 - Public variables
 - Public ENums
 - Public Types

Do let me know if you see or know of something that I've missed. With so much, it's more likely than not!

All of this means looping through projects and, looping through the module declarations in those projects, and looping through all procedures in those modules.

We've done that!

Multilinguality

This is a big subject. But we're not going to let that put us off are we? To be multilingual in any application means having a database of some sort of different languages. Typically, this consists of different files that need to be loaded to the application. To create that database, you need to capture all of the literals that are displayed anywhere in the original languages and translate them. There's a bit more involved. There always is isn't there. There's catching compile error messages and runtime error messages. If you're using userforms there's the length of the error message. Saying "This field must be a number" in English is "Dieses Feld muss numerisch sein" in German. Different lengths. Especially when proportional fonts are being used. Then again, how do you know whether you've missed even a single piece of text?

Some of the very eagle eyed out there will have noticed I prefix a lot of text with @. This is my attempt to tell if I've missed something. When I code a literal string, I try to remember to put an "@" in front. I'll load a language "database", and none of the translations have an @ in front. Then I do my tests. If I see an @ then I know I've missed a translation.

Getting the original language strings is a matter of trawling through the application and gathering all of the text strings being used. It's not just a question of going through code though. If any controls are being used their captions or defaults need to be scraped as well. Then there's tooltips.

It **IS** possible to tell the developer that a translated string is too long to fit into a label on a form say or a checkbox caption. But you know what? The best way is to just look at it. I know it means a lot of time consuming eyeballing work, but you know you're worth it. Oops, you know it's worth it!

This latter item, checking length programmatically, involves a label or text box on a userform. Let's just have a look at that. Open your favourite application, get to the VBE and Insert a userform. On the userform insert two labels and two text boxes. This is an example so no need to change any names, though it's a good habit to get into! By default, the text boxes will be named TextBox1 and TextBox2 and the Labels will be Label1 and Label2.

In the userform code for the change events of the text boxes, enter:

74 String length example

	Private Sub TextBox1_Change()
	Me.Label1.Caption = Me.TextBox1.Width
	End Sub

	Private Sub TextBox2_Change()
	Me.Label2.Caption = Me.TextBox2.Width
	End Sub

Now type something into textbox1 and something else, preferable a translation of what you have just typed in in a different language, into textbox2.

I guarantee the numbers in label1 and label2 will be different.

But! We now have a way of measuring lengths of translations and therefore whether the translations will fit into the same control caption as the original. We can flag that so the translation can be looked at and if necessary altered.

First though, we need to gather all possible literals and put them somewhere.

Hmmmm... Registry? Nah, let's put everything into a language file this time like everyone else.

It's a two stage process.

- Get the strings
- Save them

The translate process could possibly include information from the above temporary userform. Let's do that.

Get the strings

We need to loop through all code in a project and collect strings. Let's start with that. We're collecting strings from modules. We don't need to know the procedure name or anything but we do need the module name and line number in the module so we can alter that. All strings are delimited by " " and there's no way to write multiple lines with " _ " in code so we're good for continuation lines.

Code to get strings.

Finally

This is not the end. Neither is it the beginning of the end. It is the start of the beginning.

Well, whatever **that** means, I intend to keep building tools for the VBE that are useful at least to me and letting people like you know about them.

On that note let me know if you need to or want to do anything in the VBE with VBA or even if you have some crazy idea! We can maybe work on it together!

Any road up, I hope you've enjoyed the journey and thank you for putting up with me. I doubt it could have been easy.

I'll be back!

**A
P
P
E
N
D
I
C
E
S**

Appendix Notes

Appendix A Links

Yes. I know. I've probably missed out your favourite site. These are here because I find myself going back to most of them time and time again, and or found them interesting. If you are really upset about me not including a site get in touch and I'll check it out. If I decide to include it you **WILL** get credit.

Chip Pearson

<http://www.cpearson.com/excel/vbe.aspx>

Whatever you do do NOT miss visiting this site. A lot of my code is thanks to Chip and his insights. He explains the object model and the various parts and calls in the VBE. LOTS of downloadable code. DO check the rest of his site, especially about arrays and classes! The caveat is it's for Excel but there's a bunch of other stuff there as well.

Deconstruction of a code module

<https://www.codeproject.com/Articles/640258/Deconstruction-of-a-VBA-Code-Module>

This needs to be worked through. Worth it in the end.

Code for Menus and buttons in the VBE.

Chip Pearson at <http://www.cpearson.com/excel/VbeMenus.aspx>

xld on the vbaexpress forum at <http://www.vbaexpress.com/forum/showthread.php?11748-add-an-item-to-a-vbe-toolbar>.

Jan Karel Pieterse

<https://www.jkp-ads.com/>

This site is in dutch and English. Emphasis is on Excel.

Stephen Bullen

<http://www.oaltd.co.uk/>

Lots of VBA. Again though, Stephens emphasis is on Excel.

Multithreading with vbs

<https://analystcave.com/excel-multithreading-in-vba-using-vbscript/>

<http://www.databison.com/multithreaded-vba-an-approach-to-processing-using-vbscript/>

<https://analystcave.com/excel-vba-multithreading-tool/>

This is about spawning VBS scripts to start instances of an application and running code in that application.

Cyclomatic Complexity

https://www.tutorialspoint.com/software_testing_dictionary/cyclomatic_complexity.htm

<https://www.guru99.com/cyclomatic-complexity.html>

https://en.wikipedia.org/wiki/Halstead_complexity_measures

MSDN Discussion of high-performance time stamps.

<https://msdn.microsoft.com/nl-nl/D66E0FC2-3AF2-489B-B4B5-78648905B77B>

Defensive programming and reliability. Analysis of post mortem NASA software.

<https://coder.today/nasa-coding-standards-defensive-programming-and-reliability-a-postmortem-static-analysis-832d0f146b6f>

Naming Conventions

https://rosettacode.org/wiki/Naming_conventions#Visual_Basic

<https://docs.microsoft.com/en-us/dotnet/visual-basic/programming-guide/program-structure/naming-conventions>

<https://docs.microsoft.com/en-us/dotnet/visual-basic/programming-guide/program-structure/program-structure-and-code-conventions>

https://en.wikibooks.org/wiki/Visual_Basic/Coding_Standards

<http://softwaresaved.github.io/distance-consultancy/conventions/VisualBasicCodingConventions.html>

<http://www.csidata.com/custserv/onlinehelp/vbsdocs/vbs3.htm>

https://en.wikipedia.org/wiki/Hungarian_notation

https://en.wikipedia.org/wiki/Leszynski_naming_convention

<http://www.sourceformat.com/coding-standard-vb.htm>

Version Control

<https://github.com/>

String optimization

<http://www.aivosto.com/articles/stringopt.html>

export/import

Appendix B Software for the VBE

There are a lot of add-ins and utilities on the internet. These are the ones I think are most popular and useful or interesting. Some are free. And of course, Google is your friend!

Pretty Code Print

<https://submain.com/products/prettycode.print.aspx>

Free

COM AddIn

Print code with lines showing indents. VERY useful. This is not being maintained any more.

MZ-Tools

V8 <https://www.mztools.com/>

V3 Free. Google for it. It's not available on the MZ-Tools site.

V8 Commercial ± 65 USD

COM AddIn.

Version 3 is free. Google for it. They're up to version 8 at the mo but you have to pay for that. Well worth downloading and using V3 in any case. You may have a bit of trouble on a 64 bit machine. Works fine on 32 bit machines.

UPDATE: I got very angry about web sites pointing at V3 and downloading the V8 trial. I sent an ugly and nasty message to the author, Carlos Quintero. It was in the heat of the moment and he was so gracious in his reply, which he didn't have to do at all. I totally and publicly apologise.

Smart Indenter

<http://www.oaltd.co.uk/Indenter/Default.htm>

Free

COM AddIn.

Wonderful program! Though not being supported by the author Stephen Bullen any more. As with MZ-Tools 3, you may have trouble installing on 64 bit machines but works well on 32 bit. There is an

annoyance that, as it stands, the indent is four chrs. To alter that needs a registry update to "HKEY_CURRENT_USER\Software\Microsoft\VBA\6.0\Common\TabWidth".

Here is a vba procedure to update it from the setting you have in the VBE courtesy of Hartmut Gruenhagen.

75 Sub Update SmartIndenter Tab Width

'	-----
'	Procedure : Update_SmartIndenter_Tab_Width
'	Author : Hartmut Gruenhagen
'	Date : 31-Jan-14
'	Purpose : In Office 2010/13 SmartIndenter fails to pick up the
'	VBA Editor tab width from the registry.
'	The procedure writes the VBA editor tab width into a
'	that Smart Indenter will then be able to read it.
'	-----
'	
	Public Sub Update_SmartIndenter_Tab_Width()
	Dim myWS As Object
	Dim intEditorTabWidth As Integer
	Dim intIndenterTabWidth As Integer
	On Error Resume Next
	Set myWS = CreateObject("WScript.Shell")
	intEditorTabWidth =
	myWS.RegRead("HKEY_CURRENT_USER\Software\Microsoft\VBA\7.0\Common\TabWidth")
	myWS.RegWrite
	"HKEY_CURRENT_USER\Software\Microsoft\VBA\6.0\Common\TabWidth",
	intEditorTabWidth, "REG_DWORD"
	intIndenterTabWidth =
	myWS.RegRead("HKEY_CURRENT_USER\Software\Microsoft\VBA\6.0\Common\TabWidth")
	If intIndenterTabWidth = intEditorTabWidth Then
	MsgBox "The tab width for SmartIndenter has been updated successfully in the registry."
	& vbCrLf & vbCrLf & _
	"The new tab width is " & intIndenterTabWidth
	Else
	MsgBox "Oops! This didn't work." & vbCrLf & vbCrLf & _
	"Writing the VBA Editor tab width to the registry for SmartIndenter failed."
	End If
	End Sub

UPDATE: This whole add in is part of the RubberDuck suit now. Their code also supports 64 bit which the original sometimes had problems with.

WinMerge

<https://sourceforge.net/projects/winmerge/files/latest/download>

Free

Open source program to compare two files. Doesn't matter what files... this will compare them. I use it a lot. Copy your procedures or modules to text files and compare them. Simple! You *see* where the differences are! We've "automated" this for procedures in the add-in.

Spy++

<http://mdb-blog.blogspot.com/2017/02/download-microsoft-spy-140-2016-06-20.html>

Free

Part of Visual Studio 2017 which is free to download from

<https://visualstudio.microsoft.com/downloads/>

Some other links for spy++:

<https://docs.microsoft.com/en-us/visualstudio/debugger/introducing-spy-increment?view=vs-2017>

<https://stackoverflow.com/questions/43360339/how-do-i-get-spy-with-visual-studio-2017>

A Microsoft program to "spy" on all windows. Does a good job and I couldn't have written the compile routines without it.

RubberDuck

<http://rubberduckvba.com/>

Free

COM AddIn

As mentioned, a "relatively" new kid on the block from 2014. Does lots including the smart indent above for which it has refactored the code from the author. This indenter will work fine with 64 bit machine too, and you can alter the indent spaces easily. The parser is quite impressive. It's GNU as well so you can look at the code and even contribute to it.

CodeCleaner

<http://www.appspro.com/Utilities/CodeCleaner.htm>

Free

COM AddIn

From the inimitable Rob Bovey. In his own words:

"During the process of creating VBA programs a lot of junk code builds up in your files. If you don't clean your files periodically you will begin to experience strange problems caused by this extra baggage. Cleaning a project involves exporting the contents of all its VBAComponents to text files, deleting the components and then importing the components back from the text files."

Notepad++

<https://notepad-plus-plus.org/>

Free

My favourite editor. Not for the VBE I know but I thought I'd mention it. You can switch on coding in VB and that gives a similar colouring scheme. Being able to collapse the blocks of code is nice. If you put a comment at the top of the block you end up with something like pseudo code.

Code Manager

<https://www.rondebruin.nl/win/dennis/codemanager.htm>

Free.

EXCEL ONLY.

Small toolbox by Dennis M Wallentin. Indentation and working with files instead of in the VBE. There is an export to VB Editor button to put code into the VBE. Dennis also has a free Windows API Viewer for MS Excel.

vbWatchdog

<https://www.everythingaccess.com/vbwatchdog.asp>

Commercial ± 135 Euros

Global error handling. "vbWatchdog provides complete global control over error handling". Worth downloading the free trial just to see it work!

VBA Code Compare

<http://www.formulasoft.com/vba-code-compare.html>

Free

Picks up the VBA from the host files so no need to copy/paste to txt files. Does not save any altered code back!

Appendix C Bubble Sort

A good tutorial is at...

https://www.tutorialspoint.com/data_structures_algorithms/bubble_sort_algorithm.htm

In its very simplest form for a list of n items:

76 Bubble sort psuedocode/Algorhythm

	For Counter1 = 1 to n
	For Counter2 = 1 to n-1
	Get item(Counter2)
	Get Item(Counter2 +1)
	If item(Counter2) < Item(Counter2+1)
	Items are already sorted.
	else
	Swap the items
	End If
	Next Counter2
	Next Counter1

This will iterate through the items n times. Some bubble sorts take into account already sorted items and are a bit quicker. As stated though, This sort is fine for what we use it for here.

Appendix D *My Personal Naming Convention and procedure layout*

Variables

It's simple. Relatively.

What do we want to know of a variable?

- Type
- Value

Usually that's it.

But I want more!

I want to know if it's global or local and so on.

Sooooo, my Variable naming convention is:

- <type><scope><Name>

With no underscores or anything. I use "scope" loosely because p=parameter but a parameters scope is local.

- Scope: Single letter. One of g=global, m=module level, l=local, p=local parameter

A few examples are...

- slPrefix
s=String, l=local, Variable name=Prefix
- spPrefix
s=string, p=parameter, Variable name=Prefix
- smPrefix
s=string, m=defined at Module level, Variable name=Prefix
These should be PRIVATE to that module by decaring them with Private or Dim.
- sgPrefix
s=string g=public global, Variable name=Prefix
These are GLOBAL to the project.

It would be nice to know if it's an enum or a type variable as well but that's enough for me... I can always add an en or tp to the prefix if I like. The problem is that it soon becomes unmanageable. For example, we could have a variable named:

IngtpmID

the prefix denoting a Long in a Type at Module level is longer than the Name!

So I don't do that unless it's important to know if the variable is in a type or an enum.

That's variables.

Procedures

I name all of my subs, or try to, beginning with lowercase "sub"

Similarly, my functions begin with lowercase "fnc"

All my classes begin with lowercase "c".

I know immediately if I am referencing sub or function, class.

I do a lot of testing of functions and Sub procedures. I name all of the test sub routines

- subtest<complete sub/function Name>

In fact I have a module with just test procedures in it. The module is called mTerst because I spelt it wrongly when I set the name. I just kept it.

Amongst other things, it makes life easier if I want to list/dump all subroutines. I exclude all of those with "subtest" at the start of the name.

That's it. Nothing fancy but a bit more than "s" or "i".

Procedure Comments

An explanation of some comments we sometimes use at the top of some procedures:

77 Some of my procedure comments

	<ul style="list-style-type: none">• BATCH	Execute until done. Could update multiple procedures. No prompts or notification messages
--	---	---

	• END MESSAGE	Display an End Message. Usually "Done."
	• SELECTION	Works with a selection or a line.
	• REPORT	Produce a HTML report.
	• PROJECT	Go through the whole project.
	• MODULE	Only work on this module.
	• PROCEDURE	Only work on this procedure.
	• ALL PROJECTS	Go through ALL Projects.

This gives, up front, an idea of the scope of the procedure.

It also points to some normal stuff that procedures regularly do. This means you can plan at least a bit of the procedure in advance. A sort of template maybe.

It's very useful too when printing out a list of procedures with the comments under the header line.

Having said all this, I'm pretty lax in keeping this "feature" up to date.

Procedure layout

My procedures are laid out like so...

78 My procedure layout

NO COMMENTS BEFORE THE SUB/FUNCTION LINE.	
	Sub/Function sub/fncName (_
	Parameter1 as Type, _
	Parameter2 as Type, _
	Parameter2 as Type _
)
	' Comments IMMEDIATELY below the definition
	' Comments IMMEDIATELY below the definition
	' Comments IMMEDIATELY below the definition
	'
	SPACE LINE
	Dims
	SPACE LINE
	Code....
	' -----
	' Major section comment.
	SPACE LINE
	Code....
	SPACE LINE
	' ### Special notes like problems or things still to do.
	SPACE LINE
	Code....
	SPACE LINE
	' Comment on the next lines of code IMMEDIATELY AFTER WITHOUT SPACE LINE.
	Code....
	SPACE LINE
	' Comment on the next lines of code IMMEDIATELY AFTER WITHOUT SPACE LINE.
	Code....
	SPACE LINE
	Function = variable
	' *****
	End Sub/Function/Property

Comments about code on separate lines immediately above the code they're commenting. I hate end of line comments. It's personal. Except for lines I want to tag for some future reason like deleting or otherwise.

Note the space lines. In this scheme they sometimes act as markers to stop processing code if we examine the lines in a procedure. This makes it simpler to find the first or last dim line for example. It is also possible to use lines with specific text for this like ' Start of Dims. And ' End of Dims.

Add comments directly below the definition so we get a description of the procedures. That's why I said about two billion years ago, if the name isn't enough, do try to put some sort of comment below the header.

Special notes, like somewhere there is a problem to be looked at later have hashes after the comment character.

You'll see a line of stars at the bottom of the procedure. This is a hangover from using continuous paper when printing on a dot matrix printer. It shows the end of the procedure more clearly. I like it so I've kept it. Quite handy when scrolling down on screen as well.

Please feel free to comment on this or anything in my naming style. You **WILL** get a reply from me and if it leads to me altering this document you **WILL** get credit! You may get a mention in any case!

Module Names

As implied, I use a lot of modules. All my standard modules are prefixed with an m. All my class modules are prefixed with c. Standard modules with code in that is stable and works well I prefix with zm. Modules that are very stable and work very well I prefix with zzm. Some modules I access a lot I prefix with A_. This is because the modules tree in the project explorer is in alphabetical order. The ones prefixed A_ are then at the top of the modules tree and are quicker to find.

So...

- A_mTest
- mInsertPrefixes
- zmInsertDims
- zzmWhereAreWe

Appendix E VBA as an OOP programming language

"To be sure, **VBA** is not a full object-oriented programming language, as it lacks important OOP features such as inheritance and function overloading. However, it does include two very important OOP features: **Classes** and **Interfaces**" Chip Pearson.

An important, informed, and fairly definitive discussion is at:

<https://rubberduckvba.wordpress.com/2015/12/24/oop-in-vba/>

The subject is beaten to death on the internet. I'm not going any further here.

Appendix F Clearing the immediate window

Here is a complete module for code to clear the immediate window with APIs.

This is from <http://www.vbforums.com/showthread.php?672465-RESOLVED-How-to-clear-Immediate-Window-in-IDE>

79 Clear immediate window with API

Option Explicit
Private Declare Function FindWindow Lib "User32" Alias "FindWindowA" _ (ByVal lpClassName As String, ByVal lpWindowName As String) As Long
Private Declare Function FindWindowEx Lib "User32" Alias "FindWindowExA" _ (ByVal hWnd1 As Long, ByVal hWnd2 As Long, ByVal lpsz1 As String, _ ByVal lpsz2 As String) As Long
Private Declare Function PostMessage Lib "User32" Alias "PostMessageA" _ (ByVal hwnd As Long, ByVal wParam As Long, ByVal lParam As Long, _ ByVal lParam As Long) As Long
Private Declare Sub keybd_event Lib "User32" (ByVal bVk As Byte, ByVal bScan As Byte, ByVal dwFlags As Long, ByVal dwExtraInfo As Long)
Private Const WM_ACTIVATE As Long = &H6
Private Const KEYEVENTF_KEYUP = &H2
Private Const VK_CANCEL = &H3
Private Const VK_CONTROL = &H11
Private Sub ClearImmediateWindow() Dim IWinVB As Long, IWinEmmediate As Long IWinVB = FindWindow("wndclass_desked_gsk", vbNullString) 'Last param depends on languages, use your immediate window caption: IWinEmmediate = FindWindowEx(IWinVB, ByVal 0&, "VbaWindow", "Immediate") PostMessage IWinEmmediate, WM_ACTIVATE, 1, 0& keybd_event VK_CANCEL, 0, 0, 0 ' (This is Control-Break) keybd_event VK_CONTROL, 0, 0, 0 'Select All

	keybd_event vbKeyA, 0, 0, 0 'Select All
	keybd_event vbKeyDelete, 0, 0, 0 'Clear
	keybd_event vbKeyA, 0, KEYEVENTF_KEYUP, 0
	keybd_event VK_CONTROL, 0, KEYEVENTF_KEYUP, 0
	keybd_event vbKeyF5, 0, 0, 0 'Continue execution
	keybd_event vbKeyF5, 0, KEYEVENTF_KEYUP, 0
	End Sub

Appendix G **P-Code**

As with VBA and OOP, I'm not going into detail here. In fact, there isn't an awful lot out there describing VBA P-Code. Here are some useful links though.

<http://www.woodmann.com/crackz/Tutorials/Vbpcode.htm>

<http://www.cpap.com.br/orlando/VBACompilerMore.asp>

https://en.wikipedia.org/wiki/P-code_machine

<https://web.archive.org/web/20151222171103/http://www.woodmann.com/crackz/Tutorials/Vbpcode.htm>

<http://archive.li/0c2is>

There is a book. But I'm loath to fork out the dosh so I can't comment to it:

https://www.betterworldbooks.com/product/detail/microsoft-p-code-5511610046?utm_source=affiliate&utm_campaign=Text&utm_medium=CJ_Link&utm_term=3630151&utm_content=Homepage&cjevent=e65cbfa5d5ef11e8806a035f0a180510

ISBN-13 9785511610047

ISBN-10 5511610046

There is a Microsoft link

[MS-OVBA]: Office VBA File Format Structure

[https://msdn.microsoft.com/en-us/library/cc313094\(v=office.12\).aspx](https://msdn.microsoft.com/en-us/library/cc313094(v=office.12).aspx)

Tools to extract VBA Macro source code from MS Office Documents

https://www.decalage.info/vba_tools

Appendix H Acronyms

IMHO	In My Humble Opinion
VBA	Visual Basic for Applications
VBIDE	Visual Basic Inegrated Development Environment
IDE	Inegrated Development Environment
WT*	What The <fill in expletive>
OOP	Object Oriented Programming
IMNSVHO	In My Not So Very Humble Opinion
DRY	Don't Repeat yourself
KISS	Keep it simple stupid
YMMV	Your Milage May Vary
VBE	Visual Basic Editor
SOL	Shit Out of Luck
QAT	Quick Access Toolbar

Appendix I Comments and Contribut as of <>

As time has passed, I've also learnt the value of using INI files to store user preferences. While it is possible to store such data in the database front-end itself, it is lost upon updating. Another option would be to use the registry, but I dislike using the registry except for registration information and the likes. Beyond which, I cannot easily be pushed out to a new computer. By using a simple INI file, you can store any information you choose, it remains intact when updates are performed and can be transferred with great ease to other computers so the user can retain their settings.

Late Binding on the other hand does not make use of pre-defined Libraries and hence it's MAJOR benefit and thus does not suffer from versioning issues. Code written in Office 2016 will typically run just fine in Office 2013, 2010, ..., 97 (always assuming the library is registered on the PC – You can't perform say Excel automation if Excel isn't installed!).

BAckup

IT Department

Windows Scheduled Task

Database Startup

RTF MsgBox

Own msgs on buttons

there is no concept of a procedure beyond the various "Proc" methods. To make it worse, they all need to know what kind of procedure they're being called on to work.

For example, the ProcOfLine property takes two required parameters.

line – A long specifying the line to check

prockind – Specifies the kind of procedure to locate. Because property procedures can have multiple representations in the module, you must specify the kind of procedure you want to locate. All procedures other than property procedures (that is, Sub and Function procedures) use vbext_pk_Proc.

So, we could find the name first procedure in a module like so.

Note that the pprockind argument indicates whether the line belongs to a Sub or Function procedure, a Property Get procedure, a Property Let procedure, or aProperty Set procedure. To determine what type of procedure a line is in, pass a variable of type Long to the ProcOfLine property, then check the value of that variable.

That's right. prockind is an Out parameter. It gets passed by reference. So where all of the other "Proc" properties of a CodeModule need to know what kind of procedure they're working with, ProcOfLine is the one that actually returns it.

The correct way to do this is to declare an uninitialized prockind variable to pass to ProcOfLine before calling any of the other properties that require a Prockind parameter.

Appendix J Latest cWhereAreWe Class Module

This just go to be too big.

I've put it on thinkz1.Com

The last update date is on the web site.

Appendix K Cross Reference example

XREF

07 May 2018 02:23

3 Cross reference example

CROSS REFERENCE PROCEDURE	
Monday 07 May 02:23:06 2018	
Project name	vbaCodeCodeAddIn
Module name	zmSorts
Procedure name	subccSortDims
Current Module Line Number	19
Module Sub/Function Line	5
Number of lines in procedure	154
Module Start line of procedure ProcStartLine	2
Array End Of Dims + 1	41
Array Start Of Dims	14
Module End Of Dims + 1	43

Module Start Of Dims	16
Array Start Of Code	42
Module Start Of Code	44
Array End Of Header +1	4
Selection Start Column	22
Selection End Column	22
Selection End Line	19
Selection	
Array Sub/Function Line	3
Procedure type	0
Current Line Text	Dim sLine2 As String
Current Array Line Number	17
Number of Dims	27
Number of Parameters	0
Module End Line	155
Procedure Scope	Public
Module End Of Header +1	6
Number Of Header Comment Lines	9
Total Comment Lines	19
Code Lines	72
Space Lines	32

Module End Procedure Line Number	155
Array End Procedure Line Number	153
Project File Name	G:\Lisa\Projects\CodeCode_CodeCodeA 2 Testing\CodeCodeA 2.xlam
Module Declaration line count	1
Unique Project name	vbaCodeCodeAddIn(CodeCodeA 2.xlam)
Module None Consecutive Dims Line	0
Module Space line number before Dims	15
Module Space line number after Dims	43
Module End Star line number	154
Module Space Line number after Header Comments	0
Number of Dims	27
Not Defined	1
Not Used	4

#	Name	Type	From	Defn.	# Found	Found at
1	olPane	Object	Dims	23	2	44 153
2	IISRow	Long	Dims	34	2	45 47

3	IISCol	Long	Dims	35	1	45
4	IERow	Long	Dims	37	1	45
5	IIECol	Long	Dims	39	1	45
6	sIProcName	String	Dims	18	4	47 48 49 50
7	IILine1	Long	Dims	36	4	48 54 88 133
8	IICountLines	Long	Dims	40	4	49 51 104 147
9	IIStartLine	Long	Dims	33	4	50 51 104 147
10	IIEndLine	Long	Dims	42	2	51 69
11	IICompLine1	Long	Dims	25	17	54 57 66 66 69 88 93 102 102 103 129 133 138 138 144 144 147
12	sIOLine1	String	Dims	27	7	57 59 61 63 93 94 130
13	iISanityCheck	Integer	Dims	41	3	67 67 77
14	subMsgBox		Not Local		2	72 80
15	sILine1	String	Dims	20	3	94 96 122
16	UCase\$		Not Defined		2	94 108
17	sIDef1	String Array	Dims	29	3	96 97 99
18	iILenDef1	Integer	Dims	30	2	97 122
19	IICompLine2	Long	Dims	24	6	103 104 107 130 139 139
20	sIOLine2	String	Dims	28	3	107 108 129

21	sLine2	String	Dims	19	3	108 110 123
22	sDef2	String Array	Dims	31	4	110 111 114 115
23	iLenDef2	Integer	Dims	32	2	114 123
24	sIA1	String Array	Dims	22	2	122 126
25	sIA2	String Array	Dims	21	2	123 126
26	sVar2	String	Dims	16	Not Used	
27	sVar1	String	Dims	17	Not Used	
28	intll	Integer	Dims	26	Not Used	
29	llELine	Long	Dims	38	Not Used	

Appendix L High Definition Timer Class

80 High definition timer class

	Option Explicit
	'
	' Though I copied this from a different source, I believe this
	' class is from...
	' Excel 2007 VBS Programmer's reference
	' by
	' John Green, Stephen Bullen, Rob Bovey, Michael Alexander
	'
	' Dim cTimer As CHiResTimer
	' Set cTimer = New CHiResTimer
	' cTimer.StartTimer
	' cTimer.StopTimer
	' Debug.Print cTimer.Elapsed
	'
	' http://www.mrexcel.com/forum/showthread.php?t=65448
	' http://www.tech-archive.net/Archive/Excel/microsoft.public.excel.programming/2011-04/msg00240.html
	'
	'How many times per second is the counter updated?
	Private Declare _
	Function QueryFrequency _
	Lib "kernel32" _
	Alias "QueryPerformanceFrequency" _
	(lpFrequency As Currency) _
	As Long
	'What is the counter's value
	Private Declare Function _
	QueryCounter _
	Lib "kernel32" _
	Alias "QueryPerformanceCounter" _
	(lpPerformanceCount As Currency) _

	As Long
	'Variables to store the counter information
	Dim cFrequency As Currency
	Dim cOverhead As Currency
	Dim cStarted As Currency
	Dim cStopped As Currency
	Private Sub Class_Initialize()
	Dim cCount1 As Currency
	Dim cCount2 As Currency
	'Get the counter frequency
	QueryFrequency cFrequency
	'Call the hi-res counter twice, to check how long it takes
	QueryCounter cCount1
	QueryCounter cCount2
	'Store the call overhead
	cOverhead = cCount2 - cCount1
	End Sub
	Public Sub StartTimer()
	'Get the time that we started
	QueryCounter cStarted
	End Sub
	Public Sub StopTimer()
	'Get the time that we stopped
	QueryCounter cStopped
	End Sub
	Public Property Get Elapsed() As Double
	Dim cTimer As Currency

	'Have we stopped or not?
	If cStopped = 0 Then
	QueryCounter cTimer
	Else
	cTimer = cStopped
	End If
	'If we have a frequency, return the duration, in seconds
	If cFrequency > 0 Then
	Elapsed = (cTimer - cStarted - cOverhead) / cFrequency
	End If
	End Property

Appendix M **Word Doc stuff**

This has nothing to do with the VBE.

When I look at some documents I often wonder, how did they do that? Sorting out how to do things in word can sometimes be very tricky and involve lots of internet searches, trying things, and hair pulling. And **time**. I just thought I'd explain a few things about this word document. For me, it wasn't easy.

At the top of this document is a Table of Contents, a Table of Figures, and a Table of Tables. It turns out that these, IMHO, are all different flavours of a Table of contents. This is not something that's obvious or clear, at least to me. Again, I emphasise that this is how **I** see it.

A TOC is easy and separate. Probably because it's the most wanted/used. You mark all of your chapter headings and sub headings with an appropriate style and then just insert the TOC. I use the standard styles mostly so that's not been a problem. Apply a style to the chapter headings and insert a TOC.

For the other two you have to use a Table Of Figures. Yes, I know a table isn't a figure. Go figure!

Most of the code here is created by:

1. Creating the code in the VBE and making sure it works.
2. Copy/Pasting the code to Excel.
3. Copy/Pasting the code from Excel plus an extra column back into Word.

Thus, the code sits in a table.

Warning: If you copy/paste the code from that table back to the VBE, be aware that the quote used for comments has possibly changed. I mentioned this in the Disclaimer but it bears repeating.

Rather than scrolling through a **LOT** of pages multiple times I built a couple of little procs to go to the next figure or table. They're in the Table of Figures and Table of Tables I know, but they weren't always when we were writing/creating this! Using these procs I was able to put captions on each table and figure quicker, or should that be more quickly. Hmmmmm.

About those two procedures. Remember when we were talking about procedure names in [Recap number two](#)? While I was building this doc, I changed the procedure names from subGoToNextTable and subGoToNextFigure to A_subGoToNextTable and A_subGoToNextFigure, that is, prepended them with "A_". When I opened the macros menu they were at the top and I didn't have to scroll down. Neat! I could also have set them up as buttons in a separate floating toolbar as in [Buttons](#). In fact, I

created another group on the the ribbon and put them there. You can do that with any of your ummmmm ugh!!! "macros".

Captions are essential. Even more essential are caption types! That's what separates the figures from the tables!

Here are some results devined from lots of and many hours of experimentation and is meant to show some results of those experiments. This is with Word 16 on Windows 10. Again, nothing to do with the VBE.

All of this is about Insert Caption in the references Tab and Insert Table Of Figures on the references tab.

- Go to word.
- Insert a blank doc.
- Insert a few pictures.
- Insert two tables.
- Go to the top of the doc and add aboiut 10 or so lines. This is just to give you a place to insert the tables.
- For each table Insert a caption using defaults and "Table" as the LABEL
- At the top or a few lines down, Build a table of figures with "Table" as the CAPTION LABEL.
- You should see a table with both of the tables listed.
- Delete the caption for table 1.
- The caption for table 2 will stay the same.
- Updating the table of figure will remove table 1 and table 2 will still be there.
- Update the Caption field for table 2 and it will change to table 1.
- Now update the table of figs and you will get table 1.
- To delete a caption, delete the line and then a backspace to totally remove it and it's formatting.
- Table of figures can have a style applied to it.
- You can alter the style of the caption and it will not alter the style in the table of figures. I've not checked what happens if they are the same style.
- The whole table must be selected to right click for Insert Caption. If you're in a table you can click on Table tools. Be careful because there are TWO Layout tabs. You'll find Select table in the one on the right.
- You can also use References insert caption.
- To remove the automatic numbering AFTER the caption, insert the caption and then delete it IN THE DOC.
- To put numbers in FRONT of the caption... Insert caption New caption .. space.. OK.
- To add to that caption type IN THE DOCUMENT.
- Now Insert table of figures and select the space for caption label.
- You can also use the Don't use caption option.

Appendix N Contact

Mike and Lisa Green

Kriekengarde 30

3992 KJ Houten

The Netherlands

Thinkz1@hotmail.com

Web Site <http://thinkz1.com/>