1. ABOUT THIS BOOK

Microsoft Excel Visual Basic Examples

Release 5, February 19, 2008

Copyright 2003-2008, Add-ins.com LLC

The examples and information in these books are for illustration only, without warranty either expressed or implied, including, but not limited to, the implied warranties of merchantability and/or fitness for a particular purpose.

The easiest way to search for a subject is to click on the Search tab and type in key words in lower case. Then choose a topic to display. To turn search highlighting off:

- Click on Options
- Click the menu item "Search Highlighting Off"

If on, then when you do a search and click on a topic, all matching words are highlighted. If you change the setting, you will need to display a different topic and then
click back on the original topic to see the change.

To copy an example:

- Highlight the code you want to copy
- Right click and select "Copy"
- Or, press CTL-C

Topics can be stored in a favorites list for quick access. First display the topic if it is not already displayed. Then click on the Favorites tab and click the "Add" button.
1. ABOUT THIS BOOK .................................................................................................................1
2. New Examples ......................................................................................................................21
3. GENERAL INTEREST TOPICS ..........................................................................................22
   3.1 Problems Accessing Visual Basic Help ........................................................................22
   3.2 The Menu Editor And Excel 97/2000 ...........................................................................22
   3.3 Determining the Excel Version ..................................................................................22
   3.4 Protecting Your Code From Others ..........................................................................23
   3.5 Excel 2000 VBA vs. Excel 97 VBA ............................................................................23
   3.6 Excel Runtime Version ............................................................................................23
   3.7 Country And Language Versions Of Excel ................................................................24
   3.8 High Security And Enabling Macros .........................................................................25
   3.9 How To Determine Regional Settings or Properties ...................................................25
   3.10 Controlling The Cursor Appearance ........................................................................29
   3.11 Displaying the Developer Tab ................................................................................29
   3.12 Using The Immediate Window ................................................................................30
   3.13 How To Clean Your Code .......................................................................................30
   3.14 Useful Module Level Statements .............................................................................30
   3.15 Recovering Code From A Corrupt File .....................................................................30
   3.16 Naming Your Visual Basic Projects .........................................................................31
   3.17 Docking Windows In The Visual Basic Editor ...........................................................31
   3.18 Books On Learning Windows API ............................................................................32
   3.19 Disabling Macro Virus Check ..................................................................................32
   3.20 Translating 1-2-3 Macros To Excel Macros .............................................................32
   3.21 Converting Lotus 1-2-3 Macros To Visual Basic ......................................................32
   3.22 The Equivalent Of A Lotus 1-2-3 Macro Pause .........................................................32
4. ADD-INS ...............................................................................................................................34
   4.1 Creating Add-Ins .........................................................................................................34
   4.2 Certification of Your Add-ins ....................................................................................34
   4.3 Running Add-In Procedures From Other Workbooks .................................................35
   4.4 How To Create XLL's ................................................................................................35
   4.5 Creating COM Add-ins ...............................................................................................35
   4.6 Using DLL Functions In Excel ....................................................................................36
   4.7 Problems With Add-Ins - ActiveWorkbook Problem ..................................................37
   4.8 Installing Add-Ins Via Visual Basic Code ...................................................................37
   4.9 Using Solver With Visual Basic ................................................................................38
5. MODULES .............................................................................................................................40
   5.1 Naming Your Modules And UserForms ......................................................................40
   5.2 Copying Modules .........................................................................................................40
   5.3 Showing Just A Single Procedure ...............................................................................40
   5.4 Removing Modules Via Visual Basic Code ...................................................................41
   5.5 Delete Modules With Code .........................................................................................41
   5.6 Removing All Modules From A Workbook ..................................................................41
   5.7 Exporting And Importing Modules ............................................................................42
9.6 SELECTING THE LAST CELL........................................................................ 105
9.6.1 The VBA Equivalents Of Ctrl-Shift-Down And Ctrl-Down......................... 105
9.6.2 Determining The Last Cell In A Column.................................................. 106
9.6.3 Finding The Last Entry In A Column ...................................................... 107
9.6.4 Finding The Last Non-Blank Cell In A Column ........................................ 108
9.6.5 Finding The Last Entry In A Row ............................................................ 108
9.6.6 Determining The Last Cell In A Row ...................................................... 109
9.6.7 Finding The Last Cell, Last Row, or Last Column................................. 110
9.6.8 Selecting from the ActiveCell to the Last Used Cell ................................ 110
9.6.9 Determining The Last Cell When Multiple Areas Are Selected .......... 111
9.6.10 Finding the Last Row and Column Numbers ....................................... 112
9.6.11 Fill Down .............................................................................................. 113
9.7 COLOR AND FORMAT EXAMPLES.............................................................. 113
9.7.1 Color Every Other Row Gray And Bold Text ........................................... 113
9.7.2 Coloring Cells Based On Their Value ................................................... 114
9.7.3 Coloring Cells Example ......................................................................... 115
9.7.4 Copying Formats From One Sheet To Another ...................................... 115
9.7.5 Summing Cells Based On Cell Color ..................................................... 116
9.7.6 Outlining A Selection ............................................................................ 116
9.7.7 Getting The Formatted Contents Of A Cell ......................................... 116
9.8 WORKING WITH FORMULAS....................................................................... 117
9.8.1 Writing Formulas That Require Double Quotes ...................................... 117
9.8.2 The Difference Between Formula And FormulaR1C1 .............................. 117
9.8.3 Modifying A Cell's Formula .................................................................. 118
9.8.4 Determining If A Cell Contains A Formula ........................................... 119
9.9 WORKING WITH COMMENTS...................................................................... 119
9.9.1 Checking For Comments ....................................................................... 119
9.9.2 Commenting A Cell With A Macro ....................................................... 119
9.9.3 Working With Comments ...................................................................... 120
9.9.4 How To Create Or Append A Comment On A Cell ................................. 121
9.9.5 Deleting Comments ............................................................................. 122
9.9.6 Auto-Sizing Comments ........................................................................ 122
9.10 CELL EXAMPLES ......................................................................................... 123
9.10.1 Determining What Is In A Cell ............................................................... 123
9.10.2 Determining Information About A Cell ............................................... 124
9.10.3 Reading And Writing Cell Values Without Switching Sheets .................. 126
9.10.4 Determining If A Cell Is Empty And Problems WithIsEmpty ................ 126
9.10.5 Testing To See If A Cell Is Empty ........................................................... 127
9.10.6 Assigning A Value To A Cell ................................................................. 128
9.10.7 Using Visual Basic To Extract Data From Cells .................................... 128
9.10.8 Copying Values Without Using PasteSpecial ....................................... 129
9.10.9 Checking For Division By Zero .............................................................. 130
9.10.10 Filling A Range With A Formula .......................................................... 130
9.10.11 Changing The Value Of Cells In A Range Based On Each Cell's Value... 131
9.10.12 Undoing The Last Manual Entry .......................................................... 132
9.10.13 Determining The Number Of Selected Cells ...................................... 132
9.10.14 How To Determine If A Range Is Empty ................................................. 133
9.10.15 Determining The Number Of Empty Cells In A Range ............................ 133
9.10.16 Cell References And Merge Cells ............................................................ 133
9.10.17 Determining if there are Merged Cells in a Range .............................. 133
9.10.18 Determining The Number Of Cells With Entries .................................. 134
9.10.19 Modifying Cell Values Based On Two Tests ......................................... 134
9.10.20 Replacing Characters in a String ........................................................... 135
9.10.21 VBA Code for ALT-ENTER .................................................................... 136

9.11 SELECTING AND SPECIFYING CELLS ...................................................... 136
9.11.1 Using Column Letters to Reference Cells .................................................. 136
9.11.2 How To Reference The Selected Cells ..................................................... 136
9.11.3 Specifying Cells Relative To Other Cells .................................................. 137
9.11.4 Referring To Cells And Ranges ............................................................... 137
9.11.5 Using The Offset Function To Specify Cells ............................................. 139
9.11.6 Use The Offset Method To Specify Cells Relative To Other Cells ............. 139
9.11.7 Scrolling To A Particular Cell ................................................................. 140
9.11.8 Controlling Cell Selection And The Scroll Area ..................................... 140
9.11.9 Selecting A Range For Sorting Or Other Use .......................................... 141
9.11.10 Making Certain That A Selection Consists Of Only A Single Area .......... 142
9.11.11 Counting And Selecting Cells With Certain Characteristics ................. 142
9.11.12 How To Expand Or Resize A Range: ...................................................... 144
9.11.13 Resizing Or Expanding A Range ............................................................ 145
9.11.14 Selecting Just Blank Cells ..................................................................... 146
9.11.15 Selecting Just Number Cells ................................................................. 146
9.11.16 Setting Number Cells to Zero ............................................................... 150
9.11.17 Selecting The Current Region ............................................................... 151
9.11.18 Using the Used Range Property In Your Code ...................................... 151
9.11.19 Resetting the Used Range .................................................................... 152
9.11.20 Selecting The Used Range On A Sheet ................................................. 152
9.11.21 Restricting A Selection To The Cells In The Sheet's Used Range .......... 153
9.11.22 Using The Intersect Method With Ranges ............................................. 155
9.11.23 Getting The Intersection Of Two Ranges .............................................. 155
9.11.24 Union Method Problem ....................................................................... 159
9.11.25 Limiting Access To Cells ...................................................................... 160
9.11.26 Hiding The Cursor Frame / Preventing Cell Selection ......................... 160
9.11.27 Preventing Cell Drag And Drop ............................................................ 160
9.11.28 Using The Merge Command In Your Code .......................................... 160
9.11.29 VBA and Validation List ....................................................................... 161

9.12 DETERMINING IF A RANGE IS IN ANOTHER RANGE ............................ 161
9.12.1 Determining If A Selection Is Within A Named Range ............................ 161
9.12.2 Determining If A Range Is Within A Specific Range ............................. 162
9.12.3 Determining If A Cell Is Within A Certain Range .................................. 163
9.12.4 Determining When A Cell Is Within A Range ........................................ 163
9.12.5 Determining If One Range Is Within Another ........................................ 165
9.12.6 How To Determine If The ActiveCell Is Within A Named Range .......... 165
9.12.7 A Function That Determines If A Range Is Within Another Range ......... 167
15. PRINTING ............................................................................................................ 329
  15.1 A Fast Way To Set The Page Setup.................................................................. 329
  15.2 How To Speed Up Changing Print Settings...................................................... 329
  15.3 How To Set The Print Area.............................................................................. 329
  15.4 Determining The Print Area .......................................................................... 330
  15.5 Enlarging A Print Area Range.......................................................................... 331
  15.6 Add Or Exclude An Area From Print_Area...................................................... 331
  15.7 Updating The Header Or Footer Before Printing .............................................. 332
  15.8 Restricting Options in PrintPreview ................................................................. 333
  15.9 Memory Problems With Page Setup................................................................. 333
  15.10 How To Fit The Printout To One Page............................................................ 335
  15.11 Controlling Printing ....................................................................................... 335
  15.12 Printing Directly To A Printer ......................................................................... 335
  15.13 How To Have The User Change The Active Printer ........................................ 336
  15.14 How To Determine The Number Of Pages That Will Print............................. 336
  15.15 Getting The Number Of Pages That Will Print .............................................. 338
  15.16 Printing Using Range Names ......................................................................... 338
  15.17 Adding Page Breaks To Your Code ................................................................ 339
  15.18 Determining PageBreaks Locations................................................................. 339
  15.19 Locating Page Breaks .................................................................................... 340
  15.20 How To Find Next Automatic Page Break ..................................................... 342
  15.21 Removing Page Breaks .................................................................................. 342
  15.22 Printing Each Row In A Selection Onto A Separate Page............................... 343
  15.23 Printing From A Dialogsheet ......................................................................... 343
  15.24 How To Printout A Sheet Or An Entire Workbook ........................................ 345
  15.25 Printing All The Files In A Directory .............................................................. 345
  15.26 Printing Embedded Charts ........................................................................... 347
  15.27 Case Of The Disappearing PageBreak Constant ............................................. 348
  15.28 Changing the Paper Type on each Sheet in a Workbook............................... 348
  15.29 File Path In Footer ....................................................................................... 349
  15.30 Hiding the Windows Print Dialog .................................................................. 349

16. DIRECTORIES ..................................................................................................... 351
  16.1 Displaying The Windows 95 Folder Dialog To Select A Directory ................. 351
  16.2 Specifying the Windows Dialog Starting Directory ....................................... 352
  16.3 Specifying A Starting Directory .................................................................... 352
  16.4 Getting A Directory Using The File Open Dialog .......................................... 353
  16.5 How To Have The User Select A Directory ..................................................... 353
  16.6 Setting The Directory For UnMapped Network Drives .................................. 354
  16.7 Getting A List Of Subdirectories ..................................................................... 354
  16.8 Listing Sub Directories In A Directory ............................................................ 356
  16.9 Determining If A Directory Exists .................................................................. 357
  16.10 Listing Files In A Directory And/Or Its Subdirectories .................................. 358
  16.11 Counting The Number Of Files In A Directory ............................................. 360
  16.12 How To Obtain The User's Temp Directory .................................................. 362
  16.13 Getting The Windows Directory ................................................................. 362
  16.14 Getting File Information From A Directory .................................................. 363
19.13 How To Change The Excel Window Caption ................................................. 397
19.14 How To Keep The Workbook Window Maximized .................................... 397
19.15 How To Maximize The Window ................................................................... 397
19.16 Positioning The Excel Window .................................................................... 397
19.17 Setting All Worksheets To The Same Scroll Position .................................. 398
19.18 Sizing A Worksheet To Fit The Screen ...................................................... 399
19.19 Synchronizing Windows On Different Sheets .............................................. 400
19.20 Unhiding Hidden Workbooks ..................................................................... 400
20. FILTERING DATA .............................................................................................. 402
20.1 AutoFilter's Range ........................................................................................ 402
20.2 Determining Filter Settings .......................................................................... 402
20.3 How To Select The Data In A Filtered List ................................................. 403
20.4 How To Turn AutoFilter Off And On ............................................................ 403
20.5 Determining If AutoFilter Is Turned On ...................................................... 404
20.6 Determining The AutoFilter's Settings ....................................................... 404
20.7 Working With Just The Filtered Cells On A Sheet ....................................... 405
21. PIVOT TABLES ................................................................................................ 406
21.1 Expanding Pivot Table Ranges ..................................................................... 406
21.2 Clearing Incorrect Field Names in PivotTable Field Dialog Box ................. 406
21.3 Pivot Table Events ....................................................................................... 407
22. DATE AND TIME ............................................................................................. 408
22.1 Converting The Date To A Day's Name ....................................................... 408
22.2 Converting Now() To Hours, Minutes, Day, Month And Year ................... 408
22.3 Getting A Date Input From A User ............................................................... 408
22.4 How To Find A Specified Time In A Specific Range ................................... 411
22.5 How To Find A Date In A Range .................................................................. 411
22.6 Using A Macro To Insert Current Time ...................................................... 412
22.7 Having Excel Wait For A Few Seconds ...................................................... 413
22.8 Application.Wait ......................................................................................... 413
22.9 Date Comparisons ....................................................................................... 413
22.10 Using Code To Create A Calendar In A Worksheet .................................... 414
22.11 Getting The End Of A Month ...................................................................... 414
22.12 Inserting The Date On Every Worksheet And Footer ............................... 414
22.13 Using Milliseconds when Excel Waits ....................................................... 415
22.14 Writing The Date And Time Out To A Cell ............................................... 415
22.15 Measuring Time Change ........................................................................... 415
22.16 Automatically Entering The Date Into A Edit Box .................................... 416
22.17 Days Left Counter ..................................................................................... 416
22.18 Select Case Using Dates ........................................................................... 416
22.19 Validating Date Entries ............................................................................. 416
23. SHORTCUT KEYS ........................................................................................... 419
23.1 Individual Disable Shortcut Keys ................................................................. 419
23.2 Redefining The Plus And Minus Keys ......................................................... 422
23.3 Disabling Almost All Of The Shortcut Keys ............................................... 422
23.4 Disabling Shortcut Menu Commands ......................................................... 424
23.5 Making shortcut Keys Sheet Specific ......................................................... 424
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>27.2 Replacing The Cell Pop-Up Menu</td>
<td>457</td>
</tr>
<tr>
<td>27.3 Disabling The Right Click Pop-Up Menu In A Workbook</td>
<td>457</td>
</tr>
<tr>
<td>27.4 Disabling The Tool List Pop-Up Menu</td>
<td>458</td>
</tr>
<tr>
<td>27.5 Replacing The Cell Pop-Up Menu With A Custom Menu</td>
<td>459</td>
</tr>
<tr>
<td>27.6 How To Customize The Popup Menus</td>
<td>459</td>
</tr>
<tr>
<td>27.7 Creating and assigning a custom Pop-up Menu</td>
<td>461</td>
</tr>
<tr>
<td>27.8 Disabling The Worksheet Tab And Navigation Pop-Up Menus</td>
<td>462</td>
</tr>
<tr>
<td>28. DEBUGGING AND HANDLING ERRORS</td>
<td>464</td>
</tr>
<tr>
<td>28.1 Debugging Tricks</td>
<td>464</td>
</tr>
<tr>
<td>28.2 Break On Unhandled Errors</td>
<td>465</td>
</tr>
<tr>
<td>28.3 Error Trapping</td>
<td>465</td>
</tr>
<tr>
<td>28.4 Avoiding Excel/VBA Crashes</td>
<td>467</td>
</tr>
<tr>
<td>28.5 Modifying Code And Repeating Steps While Debugging</td>
<td>467</td>
</tr>
<tr>
<td>28.6 Error Handling Different In Excel 97/2000 For Functions</td>
<td>467</td>
</tr>
<tr>
<td>28.7 What To Do If You Get Strange Problems With Perfectly Good Code</td>
<td>468</td>
</tr>
<tr>
<td>28.8 Observing Excel While Debugging In Visual Basic</td>
<td>468</td>
</tr>
<tr>
<td>28.9 Detecting Error Values In Cells</td>
<td>469</td>
</tr>
<tr>
<td>28.10 Out Of Memory Error Solutions</td>
<td>469</td>
</tr>
<tr>
<td>28.11 Excel Crashes When Using A Range</td>
<td>469</td>
</tr>
<tr>
<td>28.12 Stack Overflow / Out Of Memory Problems</td>
<td>470</td>
</tr>
<tr>
<td>28.13 Keeping An Error Handling In Effect After An Error Occurs</td>
<td>471</td>
</tr>
<tr>
<td>28.14 Excel Crashes When A UserForm Is Displayed</td>
<td>472</td>
</tr>
<tr>
<td>28.15 Error Handling And Getting the Error Line</td>
<td>472</td>
</tr>
<tr>
<td>28.16 ErrObject</td>
<td>472</td>
</tr>
<tr>
<td>29. DIALOGSHEETS</td>
<td>473</td>
</tr>
<tr>
<td>29.1 How To Create And Display Dialogsheets</td>
<td>473</td>
</tr>
<tr>
<td>29.2 Selecting A Range Using An Excel 5/7 Dialog Sheet</td>
<td>473</td>
</tr>
<tr>
<td>29.3 Changing The Name Of Your Dialogsheet Objects</td>
<td>475</td>
</tr>
<tr>
<td>29.4 Setting The Tab Order In A DialogSheet</td>
<td>475</td>
</tr>
<tr>
<td>29.5 Displaying Dialogsheets</td>
<td>476</td>
</tr>
<tr>
<td>30. CONTROLLING USER INTERRUPTIONS</td>
<td>479</td>
</tr>
<tr>
<td>30.1 Capturing When Esc Or Ctrl-Break Are Pressed</td>
<td>479</td>
</tr>
<tr>
<td>30.2 Keeping Your Code From Being Stopped By The Esc Or Ctrl-Break Keys</td>
<td>481</td>
</tr>
<tr>
<td>30.3 Determining Which Key Was Pressed</td>
<td>481</td>
</tr>
<tr>
<td>30.4 Traping the Key Pressed Event</td>
<td>482</td>
</tr>
<tr>
<td>31. EVENT HANDLING</td>
<td>483</td>
</tr>
<tr>
<td>31.1 Auto_Open And Workbook_Open Macros</td>
<td>483</td>
</tr>
<tr>
<td>31.2 Preventing An Auto_Open or Workbook_Open Macro From Running</td>
<td>483</td>
</tr>
<tr>
<td>31.3 Having A Dialog Appear When A Workbook Is First Opened</td>
<td>484</td>
</tr>
<tr>
<td>31.4 Running A Macro Whenever A Workbook Is Closed</td>
<td>484</td>
</tr>
<tr>
<td>31.5 Order Of Close Events</td>
<td>484</td>
</tr>
<tr>
<td>31.6 Intercepting The Excel and Workbook Close Events</td>
<td>485</td>
</tr>
<tr>
<td>31.7 Disabling Events From Running</td>
<td>485</td>
</tr>
<tr>
<td>31.8 Running A Macro Every Minute</td>
<td>485</td>
</tr>
<tr>
<td>31.9 OnTime method - how to handle fractions of seconds</td>
<td>486</td>
</tr>
<tr>
<td>31.10 How To Make A Macro Run Every Two Minutes</td>
<td>486</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
</tr>
<tr>
<td>---------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>33.19</td>
<td>Getting Data From Access</td>
</tr>
<tr>
<td>33.20</td>
<td>How To Exchange Data Between Access And Excel</td>
</tr>
<tr>
<td>33.21</td>
<td>SQL Query Strings</td>
</tr>
<tr>
<td>33.22</td>
<td>Excel GetObject To Open Word</td>
</tr>
<tr>
<td>33.23</td>
<td>Using Barcodes in Excel</td>
</tr>
<tr>
<td>34.</td>
<td>NEAT THINGS TO KNOW</td>
</tr>
<tr>
<td>34.1</td>
<td>Using SendKeys In Your Macros</td>
</tr>
<tr>
<td>34.2</td>
<td>Hiding The Active Menu And Using Full Screen</td>
</tr>
<tr>
<td>34.3</td>
<td>Hiding Screen Update Activity - Stop Screen Flashing</td>
</tr>
<tr>
<td>34.4</td>
<td>Stopping Alert Messages / Display Alert Warning</td>
</tr>
<tr>
<td>34.5</td>
<td>Speeding Up Your Procedures And Controlling Calculation</td>
</tr>
<tr>
<td>34.6</td>
<td>Speeding Up Your Procedures - More Suggestions</td>
</tr>
<tr>
<td>34.7</td>
<td>Macros Run Really Slow</td>
</tr>
<tr>
<td>34.8</td>
<td>Determining How Long Your Code Took To Run</td>
</tr>
<tr>
<td>34.9</td>
<td>A Solution To Excel Running Slow</td>
</tr>
<tr>
<td>34.10</td>
<td>How To Hide Excel Itself</td>
</tr>
<tr>
<td>34.11</td>
<td>Opening Without A Blank Workbook And No Splash Screen</td>
</tr>
<tr>
<td>34.12</td>
<td>Closing Excel Via Visual Basic with Application.Quit</td>
</tr>
<tr>
<td>34.13</td>
<td>Playing WAV Files</td>
</tr>
<tr>
<td>34.14</td>
<td>Running Macros That Are Located In A Different Workbook</td>
</tr>
<tr>
<td>34.15</td>
<td>Writing Text To A Shape Or Text Box</td>
</tr>
<tr>
<td>34.16</td>
<td>How To Prevent A Macro From Showing In The Macro List</td>
</tr>
<tr>
<td>34.17</td>
<td>Using SendKeys To Force A Recalculation</td>
</tr>
<tr>
<td>34.18</td>
<td>Bypassing The Warning About Macros</td>
</tr>
<tr>
<td>34.19</td>
<td>Hiding The Cell Pointer</td>
</tr>
<tr>
<td>34.20</td>
<td>Turning The Caps Lock Key Off Or On</td>
</tr>
<tr>
<td>34.21</td>
<td>Modifying The Windows Registry</td>
</tr>
<tr>
<td>34.22</td>
<td>Getting Values from the Registry</td>
</tr>
<tr>
<td>34.23</td>
<td>Removing An Outline</td>
</tr>
<tr>
<td>34.24</td>
<td>Turning Num Lock Off Or On</td>
</tr>
<tr>
<td>34.25</td>
<td>Turning Scroll Lock Off Or On</td>
</tr>
<tr>
<td>34.26</td>
<td>Disabling The Delete Key</td>
</tr>
<tr>
<td>34.27</td>
<td>Writing To The Serial Port</td>
</tr>
<tr>
<td>34.28</td>
<td>COM PORTs</td>
</tr>
<tr>
<td>34.29</td>
<td>Capturing Win 95 Network Login User Name</td>
</tr>
<tr>
<td>34.30</td>
<td>Convert To PDF File Via VBA</td>
</tr>
<tr>
<td>34.31</td>
<td>How To Display HTML Help Files</td>
</tr>
<tr>
<td>34.32</td>
<td>Turn Off Asterisk As Wildcard</td>
</tr>
<tr>
<td>34.33</td>
<td>Preventing VBA Help from Resizing the Editor</td>
</tr>
<tr>
<td>34.34</td>
<td>VBA Screen Capture Routines</td>
</tr>
<tr>
<td>35.</td>
<td>INTERESTING MACRO EXAMPLES</td>
</tr>
<tr>
<td>35.1</td>
<td>An Example Of A Rounding Macro</td>
</tr>
<tr>
<td>35.2</td>
<td>Finding Entries That Are Not In A List</td>
</tr>
<tr>
<td>35.3</td>
<td>Deleting Leading Tick Marks From A Selection</td>
</tr>
<tr>
<td>35.4</td>
<td>How To Convert Formulas To Absolute References</td>
</tr>
<tr>
<td>35.5</td>
<td>A Database Modification Example</td>
</tr>
</tbody>
</table>
2. New Examples

Feb 23, 2008

- Counting Entries In A Filtered Column
- Last Row Number and Last Column Number
3. GENERAL INTEREST TOPICS

3.1 Problems Accessing Visual Basic Help

The default installation of Excel 97 does not install the Visual Basic help files. If you can not access Visual Basic help, then re-run Excel's setup program and do a custom install that installs the help.

If you have problems with accessing the help files once they are installed, such as repeatedly getting the message "Preparing help file…", check out the following Internet article in the Microsoft knowledge base.

http://support.microsoft.com/support/kb/articles/q162/6/56.asp

"Preparing Help File for First Use" Continues to Appear

3.2 The Menu Editor And Excel 97/2000

The Excel 5/7 menu editor allows one to modify the menus in Excel. It is accessed from an Excel 5/7 module via the Tools menu. However, the menu editor is not available in Excel 97/2000. One way to remove such changes is edit the file in Excel 5 or 7, go to a module, and from the tool menu select the menu editor. Then do a reset on each of the menus.

You can also remove Excel 5/7 Tools menu edits and Menu Editor edits by copying the contents of the Excel 5/7 workbook to a new Excel workbook, using the following procedure:

1 Open Excel 5/95 workbook.

2 Group the sheets (right-click a sheet tab and click Select All Sheets).

3 Click on Edit Move or Copy Sheet...

4 In the Move or Copy dialog box, in the workbook dropdown, click on (new book). Click the Create a copy check box. Click OK. You should now have a new Excel workbook containing all the sheets from the Excel 5/95 workbook, except for the modules.

5 In the VB Editor window, in the Project Explorer window, drag any modules from the Excel 5/95 workbook to the new workbook.

6 Save the Excel workbook. It should now be a complete copy of the Excel 5/95 workbook, except for the menu edits.

3.3 Determining the Excel Version

There are slight differences between Excel 97 and Excel 2000. If you need to determine what version of Excel is running, then use Application.Version. It will return either a number or a
text string depending on the version. For example, 8.0e or 9.0. To determine if it is Excel 97 or Excel 2000, use the following function. It returns an 8 if Excel 97, a 9 if Excel 2000, and most likely a 10 for the next release of Excel.

```vbnet
Function ExcelVersion() As Integer
    ExcelVersion = Val(Application.Version)
End Function
```

### 3.4 Protecting Your Code From Others

In Excel access to code modules is now through the Visual Basic Editor (VBE) and projects must be explicitly password-protected in order for their code to be made unavailable. To hide the code, go to the VB editor.

- Select Tools, and click on VBA Project Properties.
- Click on the Protection Tab.
- Check Lock Project for viewing and type in your password.

This applies to XLA projects as much as it applies to XLS projects. Unfortunately, a utility is available that will crack VBA project passwords across Office 97, so, again, this protection only works against casual users.

If you are concerned that you need more heavyweight protection then you should consider wrapping your most important code in an ActiveX DLL (most Excel code can be ported pretty much straight into VB5 or VB6 with a few tweaks to object references) and calling that from a protected XLA.

### 3.5 Excel 2000 VBA vs. Excel 97 VBA

Excel 2000 has only a few new features versus Excel 97/2000. There is enhanced web code. Some VB specific keywords like "Implements", "CallByName" etc. are added. There are some new objects, properties, methods, and events which are not normally used which were added.

Excel 2000 has a reverse `InStr`, a routine to split the elements of a delimited string into elements of an array,. It also has a routine like the worksheet function SUBSTITUTE. There are also some new features relating to macro security. You have the ability to "sign" your VBA code in Excel 2000. This allows users with Excel 2000 to automatically accept your code as safe when they set macro security to High.

There are many cosmetic changes. You can get a complete list of new additions through the help topic: What’s New for Microsoft Excel 2000 Developers In 2000 VBA help

If you don't use any of the new things, your code should run in Excel 97. If you are developing applications for Excel 97 users, you should do the development in Excel 97 to insure compatibility.

### 3.6 Excel Runtime Version
A frequently asked question regarding Excel is about the existence of a run-time version of Excel for users who do not have Excel installed on their machines, but need to run macros or access data in Excel workbooks. The answer is straightforward: There is not a run-time version of Excel. Each user must purchase a copy of Excel.

However, you can put your data in Excel workbooks and distribute those files to your audience even if they don't have Excel. They would have to have the Excel viewer available free from the Microsoft Web site. They would be able to look at your files, as if they were pictures - they would not be able to change the values or enter data and run macros.

### 3.7 Country And Language Versions Of Excel

The `xlCountryCode` and `xlCountrySetting` parameters return the LANGUAGE versions of Excel and Windows, not necessarily the actual country. For example, USA and UK both return `xlCountryCode` of 1, even though they are (very) different countries.

Microsoft Excel is published in over 30 languages. You can determine the language version by using `Application.International(xlCountryCode)`. It returns a number that indicates which language is in use:

<table>
<thead>
<tr>
<th>Language</th>
<th>Code</th>
<th>Country</th>
</tr>
</thead>
<tbody>
<tr>
<td>English</td>
<td>1</td>
<td>The United States of America</td>
</tr>
<tr>
<td>Russian</td>
<td>7</td>
<td>Russian Federation</td>
</tr>
<tr>
<td>Greek</td>
<td>30</td>
<td>Greece</td>
</tr>
<tr>
<td>Dutch</td>
<td>31</td>
<td>The Netherlands</td>
</tr>
<tr>
<td>French</td>
<td>33</td>
<td>France</td>
</tr>
<tr>
<td>Spanish</td>
<td>34</td>
<td>Spain</td>
</tr>
<tr>
<td>Hungarian</td>
<td>36</td>
<td>Hungary</td>
</tr>
<tr>
<td>Italian</td>
<td>39</td>
<td>Italy</td>
</tr>
<tr>
<td>Czech</td>
<td>42</td>
<td>Czech Republic</td>
</tr>
<tr>
<td>Danish</td>
<td>45</td>
<td>Denmark</td>
</tr>
<tr>
<td>Swedish</td>
<td>46</td>
<td>Sweden</td>
</tr>
<tr>
<td>Norwegian</td>
<td>47</td>
<td>Norway</td>
</tr>
<tr>
<td>Polish</td>
<td>48</td>
<td>Poland</td>
</tr>
<tr>
<td>German</td>
<td>49</td>
<td>Germany</td>
</tr>
<tr>
<td>Portuguese Brazil</td>
<td>55</td>
<td>Brazil</td>
</tr>
<tr>
<td>Thai</td>
<td>66</td>
<td>Thailand</td>
</tr>
<tr>
<td>Vietnamese</td>
<td>84</td>
<td>Vietnam</td>
</tr>
<tr>
<td>Simplified Chinese</td>
<td>86</td>
<td>People's Republic of China</td>
</tr>
<tr>
<td>Japanese</td>
<td>81</td>
<td>Japan</td>
</tr>
<tr>
<td>Korean</td>
<td>82</td>
<td>South Korea</td>
</tr>
<tr>
<td>Turkish</td>
<td>90</td>
<td>Turkey</td>
</tr>
<tr>
<td>Indian</td>
<td>91</td>
<td>India</td>
</tr>
<tr>
<td>Urdu</td>
<td>92</td>
<td>Pakistan</td>
</tr>
<tr>
<td>Portuguese</td>
<td>351</td>
<td>Portugal</td>
</tr>
<tr>
<td>Finnish</td>
<td>358</td>
<td>Finland</td>
</tr>
<tr>
<td>Traditional Chinese</td>
<td>886</td>
<td>Taiwan</td>
</tr>
<tr>
<td>Arabic</td>
<td>966</td>
<td>Saudi Arabia</td>
</tr>
<tr>
<td>Hebrew</td>
<td>972</td>
<td>Israel</td>
</tr>
<tr>
<td>Farsi</td>
<td>982</td>
<td>Iran</td>
</tr>
</tbody>
</table>
For example

Sub LanguageUsed()

    Dim sLang As String
    Select Case Application.International(xlCountryCode)
    Case 1: sLang = "U.S"
    Case 34: sLang = "Spanish"
    Case Else: sLang = "other"
    End Select
    MsgBox sLang
End Sub

The following illustrates how to determine if the user is using a US version of Excel.

If Application.International(xlCountryCode)=1 Then
    MsgBox "US"
Else
    MsgBox "Not US"
End If

The main problem with writing code for other languages typically has been currency. Microsoft Excel will sometimes default to the U.S. conventions of currency separators with obviously unsatisfactory results.

3.8 High Security And Enabling Macros

The default setting in Excel 2000 and above is high security. This means that users are not notified that a file contains macros and are not even given the opportunity to enable macros. Macros are automatically disabled.

If you want users to enable macros when they open a file, you should have the file open onto a worksheet with a message saying "If you see this message then you did not enable macros and they macro features of this workbook are disabled." If the user enables macros, then simply hide this worksheet as the first command in a subroutine named "Auto_Open".

3.9 How To Determine Regional Settings or Properties

To determine regional Excel settings, use Application.International(index), where index indicates which setting you want returned. To see a list of the various index values, simply highlight the keyword International in your code and press the F1 key (or look below).

The following are a few examples:

'decimal separator:

Dim sDecimal As String
sDecimal = Application.International(xlDecimalSeparator)

'Day symbol and a statement that displays the day in a message box
```vba
Dim sDay As String
sDay = Application.International(xlDayCode)
MsgBox Format(Now(), sDay & sDay & sDay & sDay)

'currency code:

Dim sCurrency As String
sCurrency = Application.International(xlCurrencyCode)

The following are just a few of the index values for International() queries

**Index Type Meaning**

- **xlCountryCode** Long Country version of Microsoft Excel.
- **xlCountrySetting** Long Current country setting in the Windows Control Panel, or the country number as determined by your Macintosh system software.
- **xlCurrencyCode** String Returns the currency character
- **xlDecimalSeparator** String Decimal separator.
- **xlThousandsSeparator** String Zero or thousands separator.
- **xlListSeparator** String List separator.
- **xlUpperCaseRowLetter** String Uppercase row letter (for R1C1-style references).
- **xlUpperCaseColumnLetter** String Uppercase column letter.
- **xlLowerCaseRowLetter** String Lowercase row letter.
- **xlLowerCaseColumnLetter** String Lowercase column letter.
- **xlLeftBracket** String Character used instead of the left bracket ([) in R1C1-style relative references.
- **xlRightBracket** String Character used instead of the right bracket (]) in R1C1-style references.
- **xlLeftBrace** String Character used instead of the left brace ( {) in array literals.
- **xlRightBrace** String Character used instead of the right brace ( }) in array literals.
**xlColumnSeparator** String Character used to separate columns in array literals.

**xlRowSeparator** String Character used to separate rows in array literals.

**xlAlternateArraySeparator** String Alternate array item separator to use if the current array separator is the same as the decimal separator.

**xlDateSeparator** String Date separator (/ in U.S. version).

**xlTimeSeparator** String Time separator (:) in U.S. version).

**xlYearCode** String Year symbol in number formats (y in U.S. version).

**xlMonthCode** String Month symbol (m in U.S. version).

**xlDayCode** String Day symbol (d in U.S. version).

**xlHourCode** String Hour symbol (h in U.S. version).

**xlMinuteCode** String Minute symbol (m in U.S. version).

**xlSecondCode** String Second symbol (s in U.S. version).

**xlCurrencyCode** String Currency symbol ($ in U.S. version).

**xlGeneralFormatName** String Name of the General number format.

**xlCurrencyDigits** Long Number of decimal digits to use in currency formats.

**xlCurrencyNegative** Long Currency format for negative currency values:

- 0 = ($x) or (x$)
- 1 = -$x or -x$
- 2 = $-x$ or x-
- 3 = $x$- or x$-
Note that the position of the currency symbol is determined by xlCurrencyBefore.

xlNoncurrencyDigits Long Number of decimal digits to use in non-currency formats.

xlMonthNameChars Long Always returns three for backwards compatibility.

xlWeekdayNameChars Long Always returns three for backwards compatibility.

xlDateOrder Long Order of date elements:

0 = month-day-year
1 = day-month-year
2 = year-month-day

xl24HourClock Boolean True if using 24-hour time, False if using 12-hour time.

xlNonEnglishFunctions Boolean True if not displaying functions in English.

xlMetric Boolean True if using the metric system, False if using the English measurement system.

xlCurrencySpaceBefore Boolean True if a space is added before the currency symbol.

xlCurrencyBefore Boolean True if the currency symbol precedes the currency values, False if it follows them.

xlCurrencyMinusSign Boolean True if using a minus sign for negative numbers, False if using parentheses.

xlCurrencyTrailingZeros Boolean True if trailing zeros are displayed for zero currency values.

xlCurrencyLeadingZeros Boolean True if leading zeros are displayed for zero currency values.

xlMonthLeadingZero Boolean True if a leading zero is displayed in months (when months are displayed as numbers).

xlDayLeadingZero Boolean True if a leading zero is displayed in days.
**xl4DigitYears** Boolean True if using four-digit years, False if using two-digit years.

**xlMDY** Boolean True if the date order is month-day-year for dates displayed in the long form, False if the date order is day-month-year.

**xlTimeLeadingZero** Boolean True if a leading zero is displayed in times.

### 3.10 Controlling The Cursor Appearance

The cursor can jiggle back and forth between an hourglass and an arrow as your macros run. It is possible to control the appearance of the cursor via code. The following sets it to a nice sedate hourglass:

```
Application.Cursor = xlWait
```

To set it back, which you must do before your code completes, use the following statement:

```
Application.Cursor = xlDefault
```

If you do not set it back (for example an error occurs that crashes your code), then the cursor will stay an hourglass.

### 3.11 Displaying the Developer Tab

In Excel 2007, you can display the developer tab to access macro functions such as recording a macro by

- Office Button > Excel Optoins > Popular

or

- Press ALT tms

### 3.12 Using The Immediate Window

The Immediate window in the VB editor is a very useful debug tool. You can use it if you have paused your macro (either through an error and clicking Debug, using a break point or using a Stop statement). The Immediate window can be displayed by choosing “Immediate Window” from the View menu or by pressing CTL-G.

In the immediate window, you can get the a value by typing a question mark and a valid statement:

```
?ActiveCell.Value
```
If you have a variable named "myVar", you can get its value by:

?myVar

You can also type in a statement and it is executed when you press enter:

ActiveCell.Value = 123

You can also put statements in your code that write to the Immediate window:

Debug.Print "The value of myVar is " & myVar

### 3.13 How To Clean Your Code

For some reason, Visual Basic modules seem to grow in size beyond the amount of code that you put into a module. It appears that Visual Basic is still retaining copies of old, deleted code. To eliminate this excess code, you can use Rob Bovey's code cleaner that will clean your code. It is available at


Basically, the code cleaners store the code in an ASCII file, delete the module, and then recreate it with the code stored in the ASCII file. Code that has not been cleaned has been know to be unstable and cause crashes.

### 3.14 Useful Module Level Statements

The following statements, placed at the top of a module, can be very useful:

**Option Explicit**

The above statement insures that all variables that you use in the module must be declared with a Dim statement. This insures that you do not misspell a variable's name and accidentally create a new variable.

**Option Compare Text**

The above statement makes any text comparison in the module case insensitive.

**Option Private Module**

The above statement prevents subroutines and functions in a module from being visible to the macro list available from the Tools menu. They can however be referenced by other subroutines and functions in other modules in the workbook.

### 3.15 Recovering Code From A Corrupt File
Andrew Baker has created a free program called a workbook rebuilder available at his site:

http://www.vbusers.com

The direct link to the download is:

http://www.vbusers.com/downloads/download.asp#item2

3.16 Naming Your Visual Basic Projects

In Excel Visual Basic, each workbook is considered a "project". When you click on the Project Explorer button in the visual basic editor, you will see a listing of each project or workbook. The default name of each project is "VBAProject". To change this name to a more descriptive name, do the following:

In the Project Explorer, click on the project line, the one with the name of the workbook.

Click on the properties button to display the properties window.

Change the name to the project to a more descriptive name. Spaces are not allowed. Use underscores instead of spaces.

3.17 Docking Windows In The Visual Basic Editor

Assuming that you inadvertently undocked these windows, you'll first need to turn on the docking property for each of the desired windows. Right click anywhere inside the desired window & click Dockable on the shortcut menu, then drag the window to the side you want it docked to. You can also turn docking on or off using (from the VBE) Tools, Options, Docking Tab.

To dock the Explorer window, move it left until the thick gray line around it turns into a thin dotted one. It should attach to the left side of the screen.

To dock the properties window move it left and down almost until it disappears off the lower left part of your screen. At this point the gray line should change again into a dotted one. Let go and, hopefully, it should return to normal. If it does not on the first try just double click on the properties window blue menu bar and try again... and again... and again...

The key thing is to watch the window outline as you drag. The thick gray outline will leave the window floating in the middle of the screen, on top of any undocked windows. A thin dotted outline will dock the window at an edge or against another docked window.

However, the final docking place is not quite as haphazard as it might seem. If you drag the thin outline left and right across the bottom of the screen (keeping it low enough to avoid the thick outline), you will see its shape go through a number of transitions that indicate how it will dock. It is best to have a maximized code module open while you do this. Otherwise, the gray border is difficult to see against the gray window background.
A slightly taller outline indicates that it will dock against the left or right edge of the screen or left or right edge of another docked window. A slightly flatter short outline indicates it will dock under another docked window or against the bottom of the screen. If you drag the outline really low, it will widen and indicate that it will cover the entire bottom of the screen. These changes are fairly subtle, so you really have to look carefully. In Excel 2000, these changes are more exaggerated and easier to see.

3.18 Books On Learning Windows API

The Application Programmer's Interface (API) is a library of hundreds of windows functions that require exact programming. One mistake can render your system useless. The definitive book on this library is Dan Applem an's, "Visual Basic 5.0 Programmer's Guide to the WIN32 API." It is extensive. Another good book that encapsulates some of these functions in classes with examples on CD is "VBA Developer's Handbook," by Ken Getz and Mike Gilbert. Of course spread across the web at microsoft.com and many of the VB sites you'll find examples of using the API. Another good book which makes use of the API is "Visual Basic Graphics Programming," by Rod Stephens.

3.19 Disabling Macro Virus Check

In case you are wondering, it can't be done. Doing so would make the virus protection. You can however, disable the warning screen that appears if you don't want to be told that workbooks contain macros. This is an option on the warning screen.

3.20 Translating 123 Macros To Excel Macros

There are not any tools (other than consultants <g>) that translate macros from 1-2-3 to Excel. Even if there were, the translation would never be as good as determining exactly what they should do and then write the VBA code from scratch. If you simply try to do a line-by-line translation of a complex 1-2-3 macro, you probably won't be taking full advantage of the features available in Excel and VBA.

3.21 Converting Lotus 1-2-3 Macros To Visual Basic

For information on converting 1-2-3 macros go to the following Microsoft web site:

http://support.microsoft.com/support/kb/articles/q148/2/40.asp

That informs you about an MS Application Note "WE1277:XL7:Visual Basic Equivalents for Lotus Macro Commands". Click on the WE1277.EXE to download this file. When you download an run it, It creates a word documentation that you can then read.

3.22 The Equivalent Of A Lotus 1-2-3 Macro Pause

In Lotus, you could pause your code so that the user could modify a cell before the macro resumed. In Excel, you would use Application.InputBox or InputBox to get user input, and
then have your code modify the cells. Please see the topics on `Application.InputBox` or `InputBox` for examples of how to use these features.
4. ADD-INS

4.1 Creating Add-Ins

There are two ways to create an add-in in Excel. From the Excel interface you can choose "File" -> "Save As" from the menu and change the file type to add-in. This works if there is at least one worksheet in the file.

The other approach (and the one I recommend) is from the visual basic editor interface. First select the workbook object in your project and change its `IsAddIn` property to `True` in the properties window. While this property is `True`, none of the worksheets in the add-in are visible or accessible in Excel. The file can only be accessed in the Visual Basic editor. To access a worksheet of an add-in, change the `IsAddin` property back to `False`.

Please note that neither approach hides your code or prevents a user from changing it unless you lock the project from the Visual Basic editor and assign a password. You can password protect your code by choosing Tools/VBAProject Protection/Protection from the menu. Then check the Lock Project For Viewing checkbox and enter a password. You'll then have to save the project, close it and reopen it for the password to take effect. Also, be aware passwords can be easily broken by commercial password crackers.

4.2 Certification of Your Add-ins

If you want to certify your macros, you do so in a two step procedure: First you create a certificate and then you certify it in Excel 2000 and above. Excel 97 does not recognize certificates.

For information on creating a certificate, go to

http://support.microsoft.com/support/kb/articles/Q217/2/21.ASP

Another other URL that will help you with certificates is:

Using SelfCert to Create a Digital Certificate for VBA Projects

With the article on "Using SelfCert To Create A Digital Certificate for VBA Projects, you may get the error message "Could not create a certificate". If you get this message, use the procedure in the second article above to create your certificates.

The second one is probably the best source. We recommend that in addition to listing your name on the certificate that you list your e-mail address and your phone number.

To sign the macros, open the workbook and go to the VB editor. Then select in the VB editor Tools, Digital Signature. Use the Choose button to specify the certificate you wish to use.
When you open a workbook with a certificate and your macro security is not set to high, you will be given an option to trust the source. Doing so adds the source to your trusted list and you will not be prompted to enable macros from this source – they will automatically be enabled.

Please note that an workbook that has a certificate means very little. Such a file can be infected with a macro virus and the certificate stays in place. Thus you can get a false sense of security. **About the only value of a certificate is to a user who wants to set a medium or high level of security against macros but wants his or her macro containing files to open without being warned that they contain macros.**

### 4.3 Running Add-In Procedures From Other Workbooks

To run an add-in's procedure from another workbook's code, use a statement like the following:

```vba
Application.Run "MyAddin.xla!MySub"
```

The double quotes are required. If the file name contains spaces then single quotes are required:

```vba
Application.Run "'My Add in.xla'!MySub"
```

If it is a function, then use a statement like the following:

```vba
retVal = Application.Run("MyAddin.xla!MyFunction")
```

"retVal" is a variable to capture the result of the function.

### 4.4 How To Create XLL's

An XLL is a standard DLL written in *C or C++*, which uses special functions and data types (principally XLOPER structures) in order to "communicate" with Excel. XLL code is far more difficult to write, but runs far faster.

If you are interested in creating XLL's, you need a good knowledge and practice of C or C++ and the "Microsoft Excel Developer's Kit", published by Microsoft Press.

### 4.5 Creating COM Add-ins

In the Visual Basic Window, there is a new pull down menu called "Add-Ins." Under this, there is "Add-In Manager." This menu is for COM (Component Object Model) Add-Ins that can be created using languages such as VB6. These add-ins are DLLs that are not visible in the VBE. The Developer Edition of Office 2000 also has the necessary power to create COM Add-Ins. Stephen Bullen has converted two of his popular VBE Add-Ins (Indenter and VBETools) into COM Add-Ins, which you can download from his site if you want to see them in action.

A COM add-in is just like a VB ActiveX DLL. You can't use it to directly add custom worksheet functions. What you can do is compile all the logic of your function into the DLL, then create a
normal Excel add-in that references that DLL and exposes the DLL functions as Excel worksheet functions.

To build COM add-ins from the VBE in the Developer Edition of Office 2000, you have to start out with a special project type. Choose File/New Project from the VBE and select Add-in Project as the project type. This will give you a COM add-in designer. The events for the COM add-in are accessed from the class module behind the designer.

For most users, creating regular add-ins via File, SaveAs or by changing the workbook IsAddin property to True is the best and easiest approach. However, COM add-ins can not be cracked by commercial password crackers (at least those I am aware of), and so offer a better level of code protection than regular add-ins.

A major disadvantage of COM add-ins is that you do not have a worksheet in the add-in to act as a storage or a working area for data.

4.6 Using DLL Functions In Excel

You can access a DLL function from VBA provided that the function is exported from the DLL using the standard call calling convention. Note that the default calling convention for a DLL created from C/C++ is cdecl and such a DLL will NOT be accessible from VB or VBA.

The DLL function is made available to VBA using the Declare statement.

By default the name by which a function is exported from your DLL will differ from the name you actually gave it in your source code. If you named your function MyFunction then the name by which it must be called will be _MyFunction@n, where n is an integer representing the size of the return stack (you can see this name by opening the file from Explorer using Quick View (from the right click menu in Explorer) and scrolling down to the export table).

The underscore is not allowed in a VB function name so to call your DLL function from VBA you must use the Declare statement with the Alias clause for example "Declare Function MyFunction Lib "DLLlibname" Alias "myfunction@12" (...) As Integer"

There are several items available from Microsoft Technet which will help you. One particularly useful one is an article entitled "DLLs for Beginners".

You can use functions from DLLs on your Excel worksheet using the CALL function.

If you are producing your own C/C++ DLLs then you can get around the name decoration by using a module definition file in your project and directly defining the exported names in the Export section of your .def file.

Be very careful if you experiment with DLL functions in Excel or Excel VBA as there is no built-in error handling and any errors are likely at the best to crash Excel and at worst bring down your machine. So make very sure that everything is properly saved before you call an untested function.
4.7 Problems With Add-Ins - ActiveWorkbook Problem

A typical error when writing code that will be an add-in is the use of `ActiveWorkbook` instead of `ThisWorkbook`. If you run a macro in a workbook, both methods reference the same workbook, but if you run an add-in, they mean different things. `ThisWorkbook` is the actual book running the macro. `ActiveWorkbook` is the active workbook visible in Excel. If you wish to refer to a sheet in the file containing the macro code, use `ThisWorkbook.Sheets("sheetname")`. If you wish to refer to a sheet in the active workbook, use `ActiveWorkbook.Sheets("sheetname")`.

4.8 Installing Add-Ins Via Visual Basic Code

To install or un-install an add-in using Visual Basic code, use statements like the following:

```
AddIns("Visual Basic Macros Made Easy").Installed = False
AddIns("Visual Basic Macros Made Easy").Installed = True
```

The following illustrates how to install the Solver add-in via Visual Basic statements:

' determine if the add-in exits in the library directory

```
Dim tempStr As String
tempStr = Application.LibraryPath & "\SOLVER\SOLVER.XLA")
If Dir(tempStr) = "" Then
    MsgBox "you do not have solver.xla installed"
    End
End If
```

' Add Solver Add-in if needed. Please note this statement may take up to

'a minute to run

```
If AddIns("Solver Add-In").Installed = False _
    Then AddIns("Solver Add-In").Installed = True
```

If you also need to add a reference to an add-in, then use the following statements which illustrate how to do set a reference to Solver.xls. Solver must be installed as an add-in for this to work.

```
Dim bFound As Boolean
Dim obj

'see if there is a reference to solver already

For Each obj In ThisWorkbook.VBProject.References
    If UCase(obj.Name) = "SOLVER.XLS" Then
        bFound = True
        Exit For
    End If
Next obj
```
'if no reference then set a reference.

If bFound = False Then 
    ThisWorkbook.VBProject.References.AddFromFile 
    Application.LibraryPath & 
    "\SOLVER\SOLVER.XLA"

4.9 Using Solver With Visual Basic

Frontline developed Solver. Apparently Solver defines names For each constraint, then hides the name definitions. After un-hiding them, you will find that the references were to cells like $AC$46, $AD$46, etc.. Apparently, in the named definitions it had the correct cells identified, but in the actual Solver dialog box it will not show cells beyond the "Z" column. Frontline's web site is:

http://www.solver.com

You can also find a number of internet articles by Microsoft at:

http://support.microsoft.com/support/Excel/Content/Solver/SOLVER.asp

They have a number of suggestions on it to help you in using Solver.

Here are the steps to display the hidden names and "restart" Solver fresh:

1. Use "Save Model" in the options dialog to save the model onto the sheet. (This will save the embedded formulas onto the worksheet.)

2. Use this macro that makes all the hidden defined names available:

   Sub unhide_Names()
       Dim na As Name
       For Each na In ActiveWorkbook.Names
           na.Visible = True
       Next
   End Sub

3. Go to Insert-Name-Define, you will see all the Solver names, starting with solver_.

4. Simply delete all of these names.

5. Start the Solver; the dialog will be blank, but you can use the "Load Model" option to restore the model you saved before.

Finally, here's another tip Frontline Systems provided:

Frontline does not recommend using formulas in the right hand sides of constraints. It is allowed, but causes severe performance problems. Instead, if you have:

$A$1 <= 0.5*$D$5*93
just place the right hand side in a cell, such as $B$10, so that $B$10 has the formula:

\[=0.5*DS^5*93,\]

and make the constraint:

$A1 <= B10$
5. MODULES

5.1 Naming Your Modules And UserForms

To change the names of your modules and userforms from "module1", "module2", etc. to more descriptive names, do the following:

- Display the project explorer by clicking on the project explorer button
- Click on the module or userform you wish to rename
- Click on the properties button to display the properties window (or press F4)
- Change the name property to a new name. Spaces are not allowed. Use underscores instead of spaces.

One technique that you may wish to use is to begin all modules with the phrase "Mdl_" and all user forms with the name "Form_". Also, you can display the module's properties window by pressing F4 while in the module.

5.2 Copying Modules

You can copy modules by manually exporting them and then importing them. You can also do this with code. The following code exports Module1 from the workbook containing the code and copies into Book2, then removes the intermediate file:

```vba
Sub CopyModule()
    Dim fName As String
    fName = "C:\TempFile.bas"
    Workbooks("book1").VBProject.VBComponents("Module1") .Export FileName:=fName
    Workbooks("book2").VBProject.VBComponents .Import(FileName:=fName).Name = "NewModule"
    Kill fName
End Sub
```

To use the above code, you must have a reference set to the VBA Extensibility library. In the VBA editor, go to Tools, References, and check "Microsoft Visual Basic For Application Extensibility".

5.3 Showing Just A Single Procedure

At the bottom left of the code panel in a module are two small buttons. Clicking on these buttons will toggle you from viewing just the active procedure (the one where the cursor is located, and all procedures.
5.4 Removing Modules Via Visual Basic Code

To delete a module via code, use statements like the following:

```vba
Sub Delete_A_Module()
    With ActiveWorkbook.VBProject.VBComponents
        .Remove .Item("Module1")
    End With
End Sub
```

Another approach is the following. It creates a reference to the VBA Extensibility Library, so you can use the following code to see if Module1 exists and then remove it:

```vba
Sub Delete_A_Module()
    Set VBEref = Application.VBE.ActiveVBProject.VBComponents
    For Each VBComponent In VBEref
        If VBComponent.Name = "Module1" Then
            VBEref.Remove VBComponent
        End If
    Next
End Sub
```

5.5 Delete Modules With Code

The following deletes regular Visual Basic modules

```vba
With ThisWorkbook.VBProject.VBComponents
    .Remove .Item("Module1")
End With
```

You can not remove the workbook or worksheet modules. To get rid of code in the workbook or worksheet modules, use the **DeleteLines** method:

```vba
With ThisWorkbook.VBProject.VBComponents("ThisWorkbook")
    .CodeModule.DeleteLines 1, .CountOfLines
End With
```

5.6 Removing All Modules From A Workbook

Here's a function that removes all modules and all code from the workbook and worksheet objects. It assumes there are no user forms, though you could easily modify the first If test to include them.

```vba
Function bRemoveAllCode(ByVal szBook As String) As Boolean
    Const lModule As Long = 1
    Const lOther As Long = 100
    Dim objCode As Object
    Dim objComponents As Object
    Dim wkbBook As Workbook

    For Each wkbBook In ThisWorkbook.Workbooks
        wkbBook = ThisWorkbook
        Set objCode = wkbBook.VBProject.VBComponents(lModule)
        Set objComponents = wkbBook.VBProject.VBComponents(lOther)
        Set objComponents.Delete
    Next
    bRemoveAllCode = True
End Function
```
On Error GoTo bRemoveAllCodeError

'set workbook and components based on the workbook name passed to
'this function

Set wkbBook = Workbooks(szBook)
Set objComponents = wkbBook.VBProject.VBComponents

'Remove all modules & code

For Each objCode In objComponents
    If objCode.Type = lModule Then

        'if the type is a module, delete the module

        objComponents.Remove objCode

    ElseIf objCode.Type = lOther Then

        'if the type is a code module, remove the lines

        objCode.CodeModule.DeleteLines 1, _

    End If
Next objCode

'if no error occurs, set function value to true and exit

bRemoveAllCode = True
Exit Function

bRemoveAllCodeError:

'if an error occurs set function value to false and exit

bRemoveAllCode = False
End Function

To use it, place the code in a workbook that you do not want to delete the modules, activate the
workbook containing the modules to be deleted, and then run this procedure:

Sub CleanWorkBook()
    If Not bRemoveAllCode(ActiveWorkbook.Name) Then
        MsgBox "An error occurred!", vbCritical
    Else
        MsgBox "Modules and code Removed!"
    End If
End Sub

5.7 Exporting And Importing Modules
You can manually export the desired module to a text file by selecting the module in the VBE (Visual Basic Editor) and using the File-->Export File command. Once saved out to a file, you can use the following to import it into any workbook...

```
ActiveWorkbook.VBProject.VBComponents.Import _
  "C:\Test\AutoMacros.bas"
```

You can export a module or form using...

```
ActiveWorkbook.VBProject.VBComponents("AutoMacros").Export _
  "C:\Test\AutoMacros.bas"
```

Both lines of code assume the module is coming into/being exported from the active workbook.

### 5.8 Deleting A Macro Via Code

The following code will delete the Auto_Open macro in the module named "Module1". Before using the code, set a reference to "Microsoft Visual Basic For Applications Extensibility" from VBA Tools menu, References.

```vba
Sub RemoveAutoOpen()
    Dim startLine As Long
    Dim nLines As Long
    With ThisWorkbook
        .VBProject.VBComponents("Module1").CodeModule
            startLine = .ProcStartLine(procName:="Auto_Open", _
                prockind:=vbext_pk_Proc)
            nLines = .ProcCountLines(procName:="Auto_Open", _
                prockind:=vbext_pk_Proc)
        .DeleteLines startLine, nLines
    End With
End Sub
```

The following is another similar example:

```vba
With ThisWorkbook.VBProject.VBComponents("Module1").CodeModule
    .DeleteLines .ProcStartLine("MacroName", 0), _
    .ProcCountLines("MacroName", 0)
End With
```

### 5.9 Listing The Subroutines In A Workbook

The following macro by Myrna Larson will search through a workbook and display all the subroutines without any arguments (i.e. those you can run from outside VBA). The code can be amended to list other items (e.g. Functions, Subroutines with arguments as necessary). It displays the names of the macros that are found.

```vba
Sub CheckForSubs()
    Dim li_CurrentLine As Integer
    Dim li_ArguementsStart As Integer
```
Dim ls_Line As String
Dim l_Component As Object

' Look at each VB Component (form/class/module) in turn
For Each l_Component In ActiveWorkbook.VBProject.VBComponents

' Only look at modules. Other types are: 2=Class, 3=Form, 100=Worksheet
If l_Component.Type = 1 Then

' Work through each line of code in turn
For li_CurrentLine = 1 To _
    ls_Line = l_Component.CodeModule._
        .Lines(li_CurrentLine, 1)

' Remove spaces from the start in case of indentation
    ls_Line = Trim$(ls_Line)

' See if this line is what we want.
    If Left$(ls_Line, 3) = "Sub" Then
        li_ArguementsStart = InStr(ls_Line, ")")
        If li_ArguementsStart > 0 Then
            MsgBox Mid$(ls_Line, 4, _
                li_ArguementsStart - 4)
        End If
    End If
Next li_CurrentLine
End If
Next l_Component
End Sub

5.10 Using A Class Module To Capture Events In Excel

A class module can be used to capture events anywhere in Excel. The following illustrates how
to capture several different events. First, create a class module in the Visual Basic editor. Name
it captureEvents and put the following code in it:

Public WithEvents excelApp As Application

Private Sub excelApp_NewWorkbook(ByVal Wb As Workbook)
    MsgBox "New workbook trapped"
End Sub

Private Sub excelApp_SheetBeforeRightClick(ByVal Sh As Object, ByVal Target As Range, Cancel As Boolean)
    MsgBox "right click trapped"
End Sub
Private Sub excelApp_SheetChange( _
    ByVal Sh As Object, ByVal Target As Range)
    MsgBox "Sheet change trapped"
End Sub

You can see what additional events can be trapped by clicking in one of the above routines and then clicking on the right (procedure) drop down.

The next step is to activate the class module. Do this by putting the following code in a regular module:

Dim excelAppClass As New captureEvents
Sub ActivateTrapping()
    Set excelAppClass.excelApp = Application
End Sub

When you run ActivateTrapping and insert a new workbook, change sheets, or right click, then the three routines in the class module are activated. You can then have these procedures call other procedures (located in regular modules). You should minimize code that you put in class modules.

5.11 Declaring A New WithEvent Class

A WithEvents class is the same as any other class module, but it has a special variable declaration inside it. To create one, add a class module to your project. At the top of the class module add the following variable declaration:

Public WithEvents MyBook As Workbook

As soon as you do this you'll notice that there are now two objects listed in the upper left dropdown of the class module, class and MyBook. If you select MyBook you will have all of the Workbook object's events available to you in the upper right dropdown.

Before you can actually use this class module, you must declare a variable for it, initiate it, then connect the MyBook variable to a workbook. The variable you use for the class module must have module-level or global-scope. For this example we'll assume global. Place the following variable declaration at the top of a normal module:

Public clsEventHandler As Class1

Next you need to initiate the class and set the MyBook variable. In your Auto_Open code or entry point procedure add the following lines:

Set clsEventHandler = New Class1
Set clsEventHandler.MyBook = ActiveWorkbook
That's it. Not you have a **WithEvents** class in which you can trap all the events for the workbook that was active when the above code was run. You can set the MyBook variable to any other workbook as well. The class will trap events for any valid Workbook object.

Note: trapping events in this manner does not stop them from firing in the specified workbook. Therefore you need to make sure you don't have any conflicting code between the two.

Also you need to initiate a new copy of the class **For Each** workbook you're trapping the events for. For a multiple-workbook solution, you may be better of using a public Collection to store the instances of the events class:

```vba
Public colEventHandler As New Collection

Dim clsNewEventHandler As Class1

Set clsNewEventHandler = New Class1
Set clsNewEventHandler.MyBook = ThisWorkbook

colEventHandler.Add clsNewEventHandler
```
6. VARIABLES AND THEIR USE

6.1 Declaring Variables

Declaring variables as specific types is the best way to insure that you use a variable as intended. For example, you do not want to accidentally assign a text string to a variable that should only hold integer values.

Each variable must be explicited declared or else it is of type Variant which means that it can hold any value. It also means that you don't have any check to insure the variable is being used correctly. These examples illustrate the most frequently used declarations. The key word Dim is used to declare a variable in a procedure.

```vba
Dim N As Integer, iMonth As Integer
Dim someText As String
Dim rowNum As Long
Dim bResponse As Boolean
Dim rNum As Single
Dim bigNum As Double
Dim oSheet As Worksheet
Dim oBook As Workbook
Dim tempR As Range, cell As Range
```

To insure that you declare all variables and that you do not accidentally create new variables by misspelling a name, put the following at the top of each of your modules:

```vba
Option Explicit
```

The following are some of the most popular types one can declare:

Type Description / Examples

**Integer** Whole numbers such as 1, 2, 3. The range can be from -32,768 to 32,767

**Long** Whole numbers from -2,147,483,648 to 2,147,483,647. If you are referring to rows numbers, then Long is the preferred declaration since the maximum number of rows exceeds 65,000

**Single** Real numbers. For example, 1.75, 356.7894, or -32.0057. The range can be from -3.4x1038 to 3.4x1038
**Double** Real numbers. The range can be from -1.8x10^{308} to 1.8x10^{308}. Also, more decimal places are carried on small numbers.

**Currency** Real numbers with exactly 4 decimals points. For example, 12.1234. The range is from -922 trillion to 922 trillion

**Boolean** True or False

**String** Text of any length

**Date** #7/4/1994# or #12:00 AM#

You can further the type of an object variable by specifying the exact type of object instead of using Object as the type. You can use the name of any kind of object as the type. The following list just a few:

Type Description / Examples

**Range** Cell or a range of cells

**Worksheet** A worksheet

**Chart** A chart sheet

**Workbook** A workbook

**Object** Cell, a range of cells, a sheet, a workbook
If the type of a variable is not declared, then it is assigned the default type **Variant**. A variable of type **Variant** can be used for any of the above types. Also, variables of type **Variant** are the only variables that can accept an error value such as #DIV/0!, #N/A, #NAME?, #NULL!, #NUM!, #VALUE, and #REF!.

### 6.2 Variable Names To Avoid

When naming your variable names, you should avoid names that Visual Basic uses or that refer to words likely to have meaning to Excel. For example, avoid names such as **Row**, **Column**, **Next**, **Worksheet**, etc. If you are uncertain about a name, prefix it with a letter indicating the type of variable. For example,

```vba
Dim iRow As Integer  
Dim bResponse As Boolean  
Dim oSheet As Object
```

Another trick is to type the name in all lower case and see if Visual Basic capitalizes it. If it does, then it has meaning to Visual Basic and should not be used. Lastly, you can select the word and press F1. If Visual Basic finds a help topic on the word, then you should not use that word as a variable.

### 6.3 Environment Variable Values

Run this code to get all the environment variables available

```vba
Sub ListEnvironVars()
  Dim i As Integer

  With Cells(1, 1)
    .Value = "Environment Variables"
    .Font.Size = 14
    .Font.Bold = True
  End With

  Cells(3, 1).Value = "Number"
  Cells(3, 2).Value = "Name and value"

  For i = 1 To 30
    Cells(i + 3, 2) = Environ(i)
    Cells(i + 3, 1) = i
  Next i
End Sub
```

### 6.4 About Local Variables, Module Variables, And Global Variables

Local variables are variables declared within a procedure or subroutine. statement:
Sub MySub()
    Dim I As Integer

    'code here

End Sub

The variable "I" is a local variable. VBA initializes local variables to empty strings and zeros when an SUB is executed.

Module variables are variables declared at the top of a module with a Dim statement.

'top of module
'module variable declaration:

Dim fileName As String

Global variables are variables declared at the top of a module with a Public or Global

'top of module
'global variable declaration:

Public fileName As String

Global variables and module level variables do not normally get re-initialized, but retain their values between macro executions. However they can be reset. For example, if an End statement is executed (a line with just the word "End" on it), such variables are re-initialized. And, when one edits a sub-routine, such variables may (but not always) be re-initialized.

6.5 Global Or Public Variables In A UserForm's Code Module

If you put global or public variables in a userform's code module, these variables will be re-initialized (lose any values) when the user form is unloaded. Therefore, you should put your global or public variables only in regular modules.

6.6 Actions That Reset Variables

Variables that are declared at the top of a module will retain their values from one macro run to another, unless certain events occur. The following are some of those events:

♦ Execution of an End statement (not an End Sub, but just the word End)

♦ Placing a worksheet into design mode. For example by adding controls to a worksheet manually.

♦ Sometimes, but not always debugging will reset the variables.
♦ Selecting halt in the Visual Basic debug screen or an error crash

♦ Creation of a new module with the Option Explicit statement added automatically

Local variables - those defined within in a Sub or Function procedure - are cleared at the end of the procedure's execution.

6.7 Setting Variables To Refer To Cell Ranges

To set a variable to refer to a range, you need to use the Set command

Sub VariableSetToRange()

' this sets the variable myRange to the current worksheet's selection

Dim myRange As Range
Set myRange = Selection
End Sub

Sub VariableSetToCell()

' this sets the variable curCell to the active cell

Dim curCell As Range
Set curCell = ActiveCell
End Sub

Sub VariableSetToLargeRang

' this sets the variable cellsOfInterest to the cells from the active cell to
' the last cell in the cell's column with an entry. It assumes that there are
' entries in the column below the active cell.

Dim R As Integer, cellsOfInterest As Range
With ActiveSheet.UsedRange
R = .Cells(.Cells.Count).Row + 1
End With
Set cellsOfInterest = Range(ActiveCell, Cells(R, _
    ActiveCell.Column).End(xlUp))
cellsOfInterest.Select
End Sub

6.8 Sharing Variable Values Between Workbooks

The following illustrates an un-documented approach to sharing variables between workbooks.
If you use it you have no guarantee that Microsoft will continue to support it in future versions of Excel.

First create an "environmental" variable like this:
Sub SetVariable()
    Application.ExecuteExcel4Macro ("SET.NAME(""CapitalOfUS"".""Wash DC""))"
End Sub

and read it in any other open workbook like this:

Sub ReadVariable()
    MsgBox Application.ExecuteExcel4Macro("CapitalOfUS")
End Sub

This works even if the first workbook is later closed (since the variable belongs to Excel not the first workbook).

6.9 Resetting Or Clearing An Object Variable

if you have set an object variable to refer to an object, it is possible to reset it. An object variable is one which is set to an object by using the Set command. Examples of an object are a workbook, a worksheet, a range on a sheet, part of chart, etc. For example, the following sets several object variables

Set tempR = Range("A1:B4")
Set sourceSheet = Sheets("Sheet1")

To reset an object variable, just set it to Nothing. For example:

Set tempR = Nothing
Set sourceSheet = Nothing

6.10 Disabling Toolbar Right Click

To disable the pop-up menu that appears if you right click on a toolbar, use:

CommandBars("Toolbar List").Enabled = False

This works in Excel 97 SR2 or higher.

6.11 Testing To See If An Object Variable Is Set

Examples of an object are a workbook, a worksheet, a range on a sheet, part of chart, etc. For example, the following sets several object variables

Set tempR = Range("A1:B4")
Set sourceSheet = Sheets("Sheet1")

To test to see if an object variable is set, use the Is test and the value Nothing:
If tempR Is Nothing Then
    MsgBox "tempR is not set"
End If

6.12 Setting An Object Variable To A Column:

The following are three ways to set a range object variable to refer to a column:

Dim myRange As Range

Set myRange = Columns("B:B")

Set myRange = Columns(2)

Set myRange = [b:b]

6.13 Storing Values In Workbook Names

If you want, you can store values in workbook names instead of in a variable or in a cell. The advantage of this approach over use of cells is that you do not have to maintain a sheet to retain values. The advantage over variables is that variables lose their values when Excel is closed or an End statement is executed.

The following statement illustrates how to assign a workbook name a value:

ActiveSheet.Names.Add "setting1", "ABC"
ActiveSheet.Names.Add "setting2", 5

To return the value assigned to the name, use the following statement:

setting1 = Mid(ActiveSheet.Names("setting1").Value, 2)

If the value being stored is a number then you will need to convert the name’s value to a number using the Val function:

setting2 = Val(ActiveSheet.Names("setting2").Value)
7. ARRAYS

7.1 Determining The Size Of An Array

It is always a good idea to determine the size of an array because Visual Basic can start an array with an index of 0. For example if you had an array by the name of myPets() and its members may be myPets(0) to myPets(2) or myPets(1) to myPets(3). One way to force Visual Basic to use the 1…3 approach is to put the statement

```
Option Base 1
```

at the top of your module.

To determine the size of an array, use **LBound** and **UBound**:  

```vbnet
Dim firstIndex As Integer, lastIndex As Integer
firstIndex = LBound(myArray)
lastIndex = UBound(myArray)
```

If the array has two dimensions and you want the limits of the second dimension, then use

```vbnet
firstIndex = LBound(myArray, 2)
lastIndex = UBound(myArray, 2)
```

7.2 Passing An Array To A Subroutine

Let's assume that you have a subroutine like the following one that requires an array as an argument:

```vbnet
Sub MyProc(AnArray() As Integer)

'code that uses and changes AnArray()

End Sub
```

you call it from code like this:

```vbnet
MyProc SomeArray()
```

Note the empty parentheses. That’s what tells Excel you mean the entire array rather than an element of it.

7.3 Clearing Arrays and Object Variables

To clear or reset an array, use the Erase method:
Erase MyArray()

To clear or reset an object variable, set it to Nothing:

Set oVariable = Nothing

7.4 How To Get The Unique Entries In A Selection

The following code will return an array named "theList" that contains the unique values in a selection. The selection in this example must be only a single column wide.

Sub UniqueEntries()

'uArray will contain the unique values in the selection

Dim rng1 As Range
Dim theList As Variant
Dim uArray
Dim J As Long, I As Long

'make sure the selection is only a single column wide

If Selection.Columns.Count > 1 Then
    MsgBox "the selection can only be a single column wide"
    Exit Sub
End If

'set the range to a variable and to just cells in the used range

Set rng1 = Intersect(Selection, ActiveSheet.UsedRange)

'transpose the range so that it can be used by QuickSort and assign
'values to a variant variable which can act as an array

theList = Application.Transpose(rng1)

'sort the list using the routine below this one

QuickSort theList, 1, UBound(theList) - LBound(theList) + 1

'remove non-unique entries and assign to a new variant array

J = 0
For I = LBound(theList) To UBound(theList)
    If J = 0 Then

'create the array and put in the first entry

    J = 1
    ReDim uArray(1 To 1)
uArray(I) = theList(I)
Else
    If theList(I) <> theList(I - 1) Then

        'if the next item in the theList is unique, add it to theList array

            J = J + 1
            ReDim Preserve uArray(1 To J)
            uArray(J) = theList(I)
        End If
    End If
Next

'display the list

For I = 1 To UBound(uArray)
    MsgBox uArray(I)
Next uArray(I)

End Sub

Sub QuickSort(SortArray, L, R)
    Dim I, J, X, Y
    I = L
    J = R
    X = SortArray((L + R) / 2)

    While (I <= J)
        While (SortArray(I) < X And I < R)
            I = I + 1
        Wend
        While (X < SortArray(J) And J > L)
            J = J - 1
        Wend
        If (I <= J) Then
            Y = SortArray(I)
            SortArray(I) = SortArray(J)
            SortArray(J) = Y
            I = I + 1
            J = J - 1
        End If
    Wend
    If (L < J) Then Call QuickSort(SortArray, L, J)
    If (I < R) Then Call QuickSort(SortArray, I, R)
End Sub

7.5 Getting A List Of The Unique Items In A List

The code below in the subroutine named CreateUniqueList returns a modified array that contains a sorted list of the items in a range. To use, the user supplies the range containing the list and an array. The CreateUniqueList updates the array with the sorted list. The range passed to the routine can contain duplicate values.
The following two examples show how to use the CreateUniqueList routine by using the sorted list to populate a listbox.

' how to use the CreateUniqueList routine.

Sub Excel97Example()

'declare an array but do not specify its size
Dim X() As String

turn off screen updating
Application.ScreenUpdating = False

call the routine and pass a range and an array
CreateUniqueList Worksheets(1).Range("a2:a10"), X
With UserForm1.ListBox1

'populate the list box as X now has values
.List = X

'turn on screen updating
Application.ScreenUpdating = True

'loop until an item is selected in the list
While .ListIndex = -1
    UserForm1.Show

'if no item selected, tell the user to select an item
If .ListIndex = -1 Then _
    MsgBox "You must select an item in the list."
Wend

'display the item selected
MsgBox .List(.ListIndex) & " was selected"
End With

'unload the userform
Unload UserForm1
End Sub

'this is the routine that returns a sorted list
Sub CreateUniqueList(srceRange As Range, X() As String)
Dim newBook As Workbook
Dim destCell As Range
Dim destSheet As Worksheet
Dim originalSetting As Long
Dim tempR As Range
Dim I As Long

'make certain the range passed to this routine is one column wide
If srceRange.Columns.Count > 1 Then
    MsgBox "Only a single wide range is allowed."
    End
End If

'make certain only one area in the range
If srceRange.Areas.Count > 1 Then
    MsgBox "Only a single area can be passed to this routine"
    End
End If

'add new workbook with two sheets
originalSetting = Application.SheetsInNewWorkbook
Application.SheetsInNewWorkbook = 2
Set newBook = Workbooks.Add
Application.SheetsInNewWorkbook = originalSetting

'set object variables to be used when creating a pivot table
Set destSheet = newBook.Sheets(1)
Set destCell = destSheet.Cells(1)

'copy the source data and paste onto sheet 2 of the new workbook
srceRange.Copy
Worksheets(2).Cells(2, 1).PasteSpecial _
    Paste:=xlValues, Operation:=xlNone, _
    SkipBlanks:=False, Transpose:=False

'add a label above the pasted data
Worksheets(2).Cells(1, 1).Value = "Name"

'create a pivot table on the first sheet
destSheet.PivotTableWizard SourceType:=xlDatabase, _
    SourceData:=Worksheets(2).UsedRange, _
    TableDestination:=destCell, TableName:="PivotTable1", _
    RowGrand:=False, ColumnGrand:=False

destSheet.PivotTables("PivotTable1").AddFields _
RowFields := "Name"
destSheet.PivotTables("PivotTable1") _
.PivotFields("Name").Orientation = xlDataField

'sort the data

destCell.Offset(2, 0).Sort Order1:=xlAscending, _
Type:=xlSortLabels, Order Custom:=1, _
Orientation:=xlTopToBottom

'set a range variable to the sorted list

Set tempR = destCell.PivotTable.DataBodyRange.Offset(0, -1)

'declare the array size equal to the cells in the above range

ReDim X(1 To tempR.Cells.Count)

'assign values to the array elements

For I = 1 To tempR.Cells.Count
X(I) = tempR.Cells(I).Value
Next

'close the workbook

newBook.Close False
End Sub

7.6 Storing Range Values In An Array

You can use a variant variable to save range values. That variant then becomes a two-
dimensional array. Example with the used range (but any range object will do):

'declare at the top of the module so that the array can be used by other
'routines in the module

Dim varArray

Sub SaveUsedRange()

'this assigns all cell values in the active sheet's used range to an array.
'UsedRange can be replaced with any range reference to store a different set
'of values

varArray = ActiveSheet.UsedRange
MsgBox UBound(varArray, 1) & " Rows"
MsgBox UBound(varArray, 2) & " Columns"
End Sub
You can restore the array on another place with code like this, which writes it to sheet2 of the active workbook, starting at cell A1.

```vba
Sub RestoreRange()
    Sheets(2).Range("A1").Resize(UBound(varArray, 1), _
                          UBound(varArray, 2)) = varArray
End Sub
```

Please note that you must use a variant variable (not a variant array) to pick up a multi-cell range from the spreadsheet.

The following are some additional examples of storing range values in a variant variable:

you can also pick up a one dimensional array by using the Application.Transpose function. If your values are going down a column, you can use

```vba
varArray = Application.Transpose(Range("A1:A10"))
```

varArray will contain a single dimension variant array with elements 1 to 10

If your values are all in one row you can pick them up with

```vba
varArray = _
    Application.Transpose(Application.Transpose(Application.Transpose(Range("A1:J1"))))
```

Again, varArray will contain a single dimension array with dimensions 1 to 10

Similarly, you can assign a range to a variant to create an array of values:

```vba
Dim myArray As Variant
myArray = Range("A1:A10").Value
```

### 7.7 Setting Array Size Dynamically

The following illustrates how to set an array's size dynamically:

```vba
Dim numElements As Integer
Dim myArray() As Integer
numElements = 2 + 3
ReDim myArray(1 To numElements)
```

If you need later to expand the array and want to preserve any values that you have assigned to the array, use Preserve in the ReDim statement:

```vba
Dim L As Integer, U As Integer
L = LBound(myArray)
U = UBound(myArray)
ReDim Preserve myArray(L To U + 1)
```
If you do not use Preserve, all values assigned to an array are erased.

### 7.8 Copying Array Values To A Range Of Cells

When transferring data from an array to a range of cells, you have to be aware that Excel treats a single dimensional array as a set of columns, not rows, unless it is transposed. If you work with a two dimensional array, the first dimension represents rows, and the second represents columns:

If the array and the destination range are different sizes, then Excel will either partially fill the range if it too small, and if it is too large, it will fill the excess cells with #N/A.

```vba
Sub ArrayToRows()
    Dim intData(1 To 10)
    Dim I As Long

    'populate the array
    For I = 1 To 10
        intData(I) = I
    Next I

    'write results out to a row range
    Range("A1:A10").Value = Application.Transpose(intData)
End Sub

Sub ArrayToColumns()
    Dim intData(1 To 10)
    Dim I As Long

    'populate the array
    For I = 1 To 10
        intData(I) = I
    Next I

    'write results out to a column range
    Range("A1:J1").Value = intData
End Sub
```

You can also use a two dimensional area:

```vba
Sub ArrayToRange()
    Dim intData(1 To 10, 1 To 2)
    Dim I As Long

    'populate the array. The first dimension is the row, and second the column
    For I = 1 To 10
        intData(I, 1) = I
        intData(I, 2) = I
    Next I

    'write results out to a range
    Range("A1:J20").Value = intData
End Sub
```
For I = 1 To 10
intData(I, 1) = I
intData(I, 2) = I * 100
Next I

'write results out to a range
Range("A1:B10").Value = intData
End Sub

7.9 How To Transpose A Range Of Values

Application.Transpose(any array) will transpose arrays. The following illustrates its use:

Array2 = Application.Transpose(Array1)

The following is another example:

Sub ReverseRange()
    Dim myArray, cell As Range, I As Long
    If Selection.Areas.Count > 1 Then

        'Only works on contiguous range

        Exit Sub
    End If
    If Selection.Columns.Count = 1 Then

        'assign the range to an array and transpose at the same time
        'if one column wide

        myArray = Application.Transpose(Selection)
    Else

        If Selection.Rows.Count = 1 Then

            'transpose twice if more than one row

            myArray = Application.Transpose/Application.Transpose(Selection)
        Else

            'only works on single column or single row

            Exit Sub
        End If
    End If

    'write values back to the cells

    I = UBound(myArray)
    For Each cell In Selection
cell.Value = myArray(I)
    I = I - 1
Next
End Sub

7.10 Editing Cells The Fast Way

The following code removes leading letters from a group of cells that contain entries like tt123456 or b456789. If you have to work on each cell in a range, it's very fast to transfer the range to an array, do what you have to do and transfer the array back to the range. This example illustrates this approach.

Sub RemoveAlpha()
    Dim cArray As Variant
    Dim i, j, dummy

    'write range's value to a variant variable which acts like an array
    cArray = Range("A1:A100")

    'loop through the values in the array
    For i = 1 To UBound(cArray, 1)

        'check each character in the value/string
        For j = 1 To Len(cArray(i, 1))

            'extract a single character
            dummy = Mid(cArray(i, 1), j, 1)

            'this Like test returns true if the character is a number
            If dummy Like "#" Then

                'remove the text character
                cArray(i, 1) = Right(cArray(i, 1), _
                                  Len(cArray(i, 1)) - j + 1)

            End If
        Next j
    Next i

    'write array values back to the original cells
7.11 Sorting An Array

The following illustrates how to sort an arrays:

```vba
Sub MyProcedure()
    Dim MyArray As Variant, i As Integer

    'create an array to use to demonstrate the sorting
    MyArray = Array("Oranges", "Pears", "Apples")

    'call the QuickSort procedure
    QuickSort MyArray

    'display the results. Note that MyArray has been changed by QuickSort
    For i = LBound(MyArray) To UBound(MyArray)
        MsgBox MyArray(i)
    Next i
End Sub
```

The above could also have been written this way,

```vba
Sub MyProcedure()
    Dim MyArray() As String, i As Integer

    'create an array to use to demonstrate the sorting
    ReDim MyArray(1 To 3)
    MyArray(1) = "Oranges"
    MyArray(2) = "Pears"
    MyArray(3) = "Apples"

    'call the QuickSort procedure
    QuickSort MyArray

    'display the results. Note that MyArray has been changed by QuickSort
    For i = LBound(MyArray) To UBound(MyArray)
        MsgBox MyArray(i)
    Next i
End Sub
```

The following is the routine that does the sorting. The only required argument is the first, the array. The next two arguments are optional and allows you to specify the starting and stopping
sort positions. Please note that if you do not have Option Base 1 declared in your module or have declared the array size, the first array element is 0, not 1.

Sub QuickSort(VA_array, Optional V_Low1, Optional V_high1)
  On Error Resume Next

'Dimension variables

Dim V_Low2, V_high2, V_loop As Integer
Dim V_val1, V_val2 As Variant

'If first time, get the size of the array to sort

If IsMissing(V_Low1) Then
  V_Low1 = LBound(VA_array, 1)
End If

If IsMissing(V_high1) Then
  V_high1 = UBound(VA_array, 1)
End If

'Set new extremes to old extremes

V_Low2 = V_Low1
V_high2 = V_high1

'Get value of array item in middle of new extremes

V_val1 = VA_array((V_Low1 + V_high1) / 2)

'Loop for all the items in the array between the extremes

While (V_Low2 <= V_high2)

'Find the first item that is greater than the mid-point item

  While (VA_array(V_Low2) < V_val1 And V_Low2 < V_high1)
    V_Low2 = V_Low2 + 1
  Wend

'Find the last item that is less than the mid-point item

  While (VA_array(V_high2) > V_val1 And V_high2 > V_Low1)
    V_high2 = V_high2 - 1
  Wend

'If the new 'greater' item comes before the new 'less' item, swap them

  If (V_Low2 <= V_high2) Then
    V_val2 = VA_array(V_Low2)
    VA_array(V_Low2) = VA_array(V_high2)
    VA_array(V_high2) = V_val2
  End If

Wend
'Advance the pointers to the next item

V_Low2 = V_Low2 + 1
V_high2 = V_high2 - 1
End If
Wend

'Iterate to sort the lower half of the extremes

If (V_high2 > V_Low1) Then
    Call QuickSort(VA_array, V_Low1, V_high2)
End If

'Iterate to sort the upper half of the extremes

If (V_Low2 < V_high1) Then
    Call QuickSort(VA_array, V_Low2, V_high1)
End Sub

7.12 Writing Arrays To A Worksheet

The following article in the Microsoft knowledge base has examples of putting array values in a worksheet without looping and doesn't use the restrictive FormulaArray method

http://support.microsoft.com/support/kb/articles/q149/6/89.asp

XL: Visual Basic Macro Examples for Working with Arrays

7.13 Assign Range Values Directly To An Array

In Visual Basic, it is possible to assign the values in a range directly to an array variable. The trick is that variable must be a Variant variable. The following illustrates how to do this:

Sub WriteToAnArray()
    Dim myArray As Variant, n As Integer, m As Integer

    'assign range values to the variant variable
    myArray = WorkSheets(1).Range("a1:c5")

    'determine the array size display in a message
    n = UBound(myArray, 1)
    m = UBound(myArray, 2)
    MsgBox n & " " & m

    'display several values
MsgBox myArray(1, 1)
MsgBox myArray(5, 3)
End Sub

You can also assign a cell range the values in an Variant array, if the cell range is the same size as the Variant array:

WorkSheets(1).Range("a1:c5").Value = myArray

The easiest way to pickup and place an array as a block on a worksheet is to use a variant variable as an array when assigned to a range. Note the following unique features. If the variant is assigned to several cells on a single row it contains a two dimensional array with dimensions

horizontalArray( 1 to 1, 1 to Number_of_Cells)

If the range is a single column of values, the variant contains an array with the following dimensions

verticalArray(1 to Number_of_Cells, 1 to 1)

You can also transpose a range by using a variant variable as an array and Application.Transpose:

Dim rangeValues As Variant
rangeValues = Range("A1:A5").Value
rangeValues = Application.Transpose(rangeValues)
Range("a10:e10").Value = rangeValues

7.14 Looping through an Array of Workbooks.htm

If you have a set of workbooks you want to loop through, using an array is a good approach. The following illustrates how to assign values to the array and loop through it:

Sub test()
Dim x()
Dim y As Integer
'assign workbook names to the array
x() = Array("One.xls", "Two.xls", "Three.xls")
'the following illustrates looping through the array
For y = LBound(x) To UBound(x)
    MsgBox "workbook " & y + 1 & " name " & (x(y))
Next y
End Sub

The variable X should not be declared as a string array. Doing so will create an error. As it is undeclared, it is a Variant type variable, and thus one does not need to declare the array size.
To refer to one of the workbooks, you could use `Workbooks(x(y))` in a statement.
8. LOOPING

8.1 Using Case Instead Of If Tests

Instead of using a series of IF tests, you can use a Case statement instead. For example:

```vba
Select Case ActiveCell.Value
    Case "A", "B", "C"
        ' do this
    Case "D", "F"
        'do something else
    Case Else
        'third option
End Select
```

or, if the values are always in sequence, with no gaps you can write

```vba
Case "A" To "C"
```

Multiple statements to be executed can be placed between the Case tests. However, a better approach is to run subroutines instead. This keeps the code compact, and you may be able to use the subroutines multiple times in your code.

8.2 Using A Select Statement To Take Action

The following illustrates how to use a Select statement to take action based on the first value in a string. In this case, the string is set based on the value of the cell being checked, which can contain entries like "ABC", "23 XX", and so forth. The user only wants action to take place if the first character in the string is a number. In this example, the actions are simulated by message boxes. In actual practice, they could be Visual Basic statements or calls to other routines.

```vba
Sub TakeAction()
    Dim cell As Range
    Dim nwTestVar As Variant
    For Each cell In Selection
        nwTestVar = Left(cell.Value, 1)
        If IsNumeric(nwTestVar) Then
            Select Case nwTestVar
                Case 0
                    MsgBox "Starts with 0"
                Case 1
                    MsgBox "Starts with 1"
            End Select
        End If
    Next cell
End Sub
```
Case 2, 3
    MsgBox "Starts with 2 or 3"
Case 4
    MsgBox "Starts with 4"
Case Is > 4
    MsgBox "Starts with > 4"
End Select
End If
Next
End Sub

8.3 Determining What Type A Value Is

The function **TypeName** returns Error, String, Double, etc. when supplied with a value:

```vba
Sub WhatType()
    MsgBox TypeName(ActiveCell.Value)
End Sub
```

You can supply **TypeName** with variables also:

```vba
Dim R As Variant
R = Application.InputBox("Enter something")
MsgBox TypeName(R)
```

which returns "Boolean" if cancel is selected, and "String" if any entry is supplied.

8.4 Using Select As A Multiple Or Statement

The **Select** statement is basically a way to write complex **Or** statements. The following illustrates this:

```vba
Sub test()
    Dim X As Integer
    Do
        X = Val(InputBox(_
            "Enter a number. 0, blank, or text exits.")
        Select Case X
        Case 0
            Exit Sub
        Case 1, 3, 7, 92
            MsgBox "first select"
        Case 5, 10, 60
            MsgBox "second select"
        Case 93 To 193
            MsgBox "third select"
        Case Else
            MsgBox "case else selected"
        End Select
    Loop
End Sub
```
8.5 How To Return To Your Starting Location

Although the best and fastest Visual Basic code is code that does not do any Selects or Activates, you may find a need to so such operations. Also, creating new workbooks with change the active workbook and use of the a statement like ActiveCell.End(xlDown) will frequently change the scroll region.

To save these settings, put the following variables at the top of a module:

```vba
Dim actWB As Workbook
Dim actSh As Object
Dim actSelect As Range
Dim sRow As Long
Dim sCol As Long
```

Before running your code, run the following macro

```vba
Sub storeSettings()
    Set actWB = ActiveWorkbook
    Set actSh = ActiveSheet
    Set actSelect = Selection
    sRow = ActiveWindow.ScrollRow
    sCol = ActiveWindow.ScrollColumn
End Sub
```

When done running all your code run the following code. Please note that if you have deleted workbooks, sheets, or inserted or moved cells you will need to modify this code. Also, if you try to activate a workbook that has been closed, you most likely will cause Excel to crash. The same can happen with sheets and ranges. So, be careful.

```vba
Sub ApplySettings()
    actWB.Activate
    actSh.Select
    actSelect.Select
    ActiveWindow.ScrollRow = sRow
    ActiveWindow.ScrollColumn = sCol
End Sub
```

8.6 Processing All The Entries In A Column

The following is a simple example that processes all the cells in a column from the selected cell downward. When it hits a blank cell, it stops

```vba
While Not IsEmpty(ActiveCell)
    'macro code that acts on the active cell
    'change the active cell to the next cell
```
ActiveCell.Offset(1, 0).Select
Wend

The above code is inefficient as it requires the active cell to change each time through the While...Wend loop. The following avoids this issue:

Dim cell As Range

' set the range variable cell to the ActiveCell
Set cell = ActiveCell

While Not IsEmpty(cell)

' macro code that acts on the cell
' change cell so that it refers to the next cell
Set cell = cell.Offset(1, 0)

Wend

You can also use a Do...Loop to achieve the same results:

Dim cell As Range

' set the range variable cell to the ActiveCell
Set cell = ActiveCell
Do

' macro code that acts on the cell
' change cell so that it refers to the next cell
Set cell = cell.Offset(1, 0)

' test to see if all the next cell is empty, exit the loop if it is
If IsEmpty(cell) Then Exit Do

Loop

If you want to process all the cells in a column that have entries with some cells not having entries and you want to start at a certain row, then use code like the following.

Dim cell As Range
Dim lastCell As Range
Dim theCells As Range

' find the last cell with an entry in the column containing the ActiveCell
Set lastCell = Cells(16000, ActiveCell.Column).End(xlUp)

'define a variable that refers to all cells from the second cell in the column to the last cell with an entry

Set theCells = Range(Cells(2, ActiveCell.Column), lastCell)
For Each cell In theCells
  If Not IsEmpty(cell) Then
    'macro code that acts on the cell
  End If
Next

8.7 Some Simple Loop Examples

A looping routine is one that repeats itself over and over until a test tells Visual Basic to exit the loop. The following are the typical constructions:

Do
  'code
  If <test to exit the loop> Then Exit Do
Loop

Do <test to start and repeat loop>
  'code
Loop <Test to repeat loop>

The following are illustrations of the above constructions. These run at least once

Do
  'your code
  I = I + 1
  If isEmpty(Cells(I, 1)) Then Exit Do
Loop
Do
    i = i + 1
Loop Until IsEmpty(Cells(i, 1))

Do
    i = i + 1
Loop While Not IsEmpty(Cells(i, 1))

The following Do..Loops are tested at the top and may not execute at all

Do
    i = i + 1
    If IsEmpty(Cells(I, 1)) Then Exit Do

'your code

Loop

    i = 1
    Do Until IsEmpty(Cells(i, 1))
        i = i + 1
    Loop

Loop

    i = 1
    Do While Not IsEmpty(Cells(i, 1))
        i = i + 1
    Loop
9. CELL AND RANGES

9.1 Excel 2007 versus Prior Versions.htm

One of the key differences in Excel 2007 is the number of rows and columns in a workbook. In Excel 97-2003, there are

- 65,536 rows
- 256 columns

In Excel 2007 there are

1048576 rows
16384 columns

If you are writing code that may be used in Excel 2007 and earlier versions, or if you are going to open XLS files in Excel 2007, then you should use statements like the following to determine the number of rows versus hard coding the number:

```
numberOfRows = ActiveSheet.Rows.Count
```

and

```
numberOfColumns = ActiveSheet.Columns.Count
```

In many of the examples you will see the Excel 97-2003 values of 65,536 rows and 256 columns hardcoded, as most users have not upgraded, and such references are easier to understand and use. If you are using Excel 2007, you should change to the Count approach.

9.2 EDITING, COPYING, AND PASTING

9.2.1 Copying And Pasting

When copying a range from one sheet and pasting to another sheet, you do not need to activate or select the destination sheet. Also, you should always specify the destination range. Otherwise Excel assumes that you wish to paste to the current selection on the destination sheet.

```vba
Sub DoCopyExample1 ()
    Dim srcRng As Range
    Dim destRng As Range
```

75
'set range variables to refer to the range to be copied and the
'destination range. Note that the destination range is a single cell

    Set srcRng = _
    Workbooks("book1.xls").Sheets("sheet1").Range("A1:A10")
    Set destRng = _
    Workbooks("book1.xls").Sheets("sheet2").Range("A1")

'A PRO APPROACH 1

'copy the range

    srcRng.Copy

'paste, specifying the destination sheet and range

    Workbooks("book1.xls").Sheets("sheet2").Paste destRng

'A PRO APPROACH 2

'specify the destination sheet by referring to the parent of the range, and
'do a copy paste on one row

    destRng.Parent.Paste destRng

'A PRO APPROACH 3 - THE SIMPLEST

'copy the source range and specify the destination range

    srcRng.Copy destRng
End Sub

The following illustrates how to copy the same range on one sheet to another sheet:

Sub DoCopyExample3()
    Dim szRange As String
    szRange = "C3:K3"
    Worksheets("Data").Range(szRange).Copy _
    'Destination:=Worksheets("Timesheet").Range(szRange)
End Sub

9.2.2 Writing Large Numbers To Cells

If you have to write large numbers to cells, like123456478944561, it is displayed as
1234456478944560. The work around is to write the number as a text string, using a single
quote:

    Workbooks("WebOrder.xls").Sheets(orderNumber) _
    .Cells(26, 5).Value = "'" & cardNumber
9.2.3 A Technique To Avoid

You should never directly assign one cell's value to that of another cell if the destination cell is formatted as currency or as a date. Nor should you assign a variable value's to that of such a cell. The following illustrates the problem:

Sub test()
    Range("A1:B1").NumberFormat = "$#.#0.00"
    Range("A1").Value = 1.23456789
    Range("B1").Value = Range("A1").Value
    'The wrong value is assigned to the cell:
    MsgBox Range("B1").Value
    Dim x As Double
    x = Range("A1").Value
    'and the variable has the wrong value also:
    MsgBox x
End Sub

The value of B1 should be 1.23456789. However, it instead ends up as 1.23! Variable X ends up as 1.23456.

One solution is to do a Copy and PasteSpecial. Values:

Sub Test2()
    Range("A1:B1").NumberFormat = "$#.#000"
    Range("A1").Value = 1.23456789
    Range("A1").Copy
    Range("B1").PasteSpecial Paste:=xlPasteValues, _
        Operation:=xlNone, _
        SkipBlanks:=False, Transpose:=False
End Sub

If you want to get the correct value assigned to a variable, then you must first clear the cell's format, assign the value to the variable, and then reset the format:

Sub Test4()
    Range("A1").NumberFormat = "$#.#00000"
    Range("A1").Value = 1.23456789
    Dim x As Double
    Dim cellFormat As String
    cellFormat = Range("a1").NumberFormat
    Range("a1").NumberFormat = ""
    Range("a1").ClearFormats
    x = Range("a1")
    Range("a1").NumberFormat = cellFormat
    MsgBox x
End Sub

If you create a sheet in your workbook named "Temp", you can use the following approach:
Sub test5()
    Range("A1").NumberFormat = "/#,###0.000"
    Range("A1").Value = 1.23456789
    Dim X As Double
    'use this function whenever you need to return a cell's value
    X = ReturnVal(Range("a1"))
    MsgBox X
End Sub

Function ReturnVal(anyCell As Range)
    anyCell.Copy
    'a sheet named Temp must be in the workbook containing this function
    'cell A1 in this workbook should not be formatted
    With ThisWorkbook.Sheets("temp").Range("a1")
        .PasteSpecial Paste:=xlPasteValues, _
        Operation:=xlNone, _
        SkipBlanks:=False, Transpose:=False
        ReturnVal = .Value
    End With
End Function

Microsoft is aware of the above problem and it is discussed in article: http://support.microsoft.com/support/kb/articles/Q213/7/19.ASP The article says that this is a problem with currency or date formatted cells. Excel 2000 and above they have added a Value2 property that avoids the above problem. Using the Value function in Excel 2000 and above continues the problem Unfortunately, a macro that uses this property will not work in Excel 97.

If you are going to work with currency or date formatted cells, then you need to use one of the solutions above.

9.2.4 Writing Text To The Clipboard

One way to write text to the clipboard is to first write it to a cell and then copy the cell. Another way is to use code like the following:

    Dim ClipData As DataObject
    Dim ClipString As String

    'declare clipData as a data object
    Set ClipData = New DataObject

    'put something into the variable clipString
    ClipString = "Some text"

    'this copies the above string to the data object
    ClipData.SetText ClipString, 1
'this copies it to the clipboard

ClipData.PutInClipboard

The following code places the value of a VBA variable in The clipboard

Sub PutOnClipboard(Obj As Variant)
    Dim MyDataObj As New DataObject
    MyDataObj.SetText Format(Obj)
    MyDataObj.PutInClipboard
End Sub

The following code places the contents of the clipboard Into a VBA variable:

Function GetFromClipboard() As Variant
    Dim MyDataObj As New DataObject
    MyDataObj.GetFromClipboard
    GetFromClipboard = MyDataObj.GetText()
End Function

9.2.5 Clearing The Clipboard After A Copy Command

The following statement will clear the clipboard after a copy command:

    Application.CutCopyMode= False

It is important to do this before a macro doing copying completes. If the clipboard is not cleared and the user presses the enter key, the contents of the clipboard will be copied to the spreadsheet, and undoubtedly upset the user.

9.2.6 An Example Of How To Copy One Range To Another Range

The following illustrates how to copy a range on one worksheet to a range on another worksheet: - without changing sheets.

Sub CopyIt()
    Dim FromRange As Range
    Dim ToRange As Range

    'set range variable to the range to be copied
    Set FromRange = Worksheets("FromSheet").Range("A1:F15")

    'set an object variable equal to the first cell of the destination range
    Set ToRange = Worksheets("ToSheet").Range("c1")

    'single line approach on doing a copy paste
FromRange.Copy ToRange

'two line approach on doing a copy paste

FromRange.Copy

Worksheets("ToSheet").Paste Worksheets("ToSheet").Range("c1")
End Sub

9.3 ROW EXAMPLES

9.3.1 Determining The Currently Selected Cell's Row

The following returns the row number of the active cell and stores in a variable for later use

R = ActiveCell.Row

9.3.2 Testing Whether A Row Is Selected

The following tests to see if a row is highlighted:

If Selection.Address = Selection.EntireRow.Address Then

'do something if highlighted

End If

9.3.3 How To Select All The Rows In A Database

Assume that your data starts in cell A1 of your worksheet and is bordered by blank rows, the following statement will select all the cells in the database:

Range("A1").CurrentRegion.Select

If you want to assign it to a range variable, then use a statement like the following:

Dim dataRange As Range
Set dataRange = Range("A1").CurrentRegion

If the data is not on the active worksheet, then qualify the Range reference with the sheet and if necessary the workbook:

Dim dataRange As Range
Set dataRange = _

Workbooks("Book1.Xls").Sheets("Sheet1") _

.Range("A1").CurrentRegion

Or:
9.3.4 Selecting Rows Based On Cell Entries

By using AutoFilter, you can select just the rows that contain a certain entry. In this case, all rows with the value of 780 in column 1.

```vba
With Range("A1").CurrentRegion
    .AutoFilter 1, ",=780"
    .Offset(1).Resize(.Rows.Count-1) _
        .SpecialCells(xlVisible).Select
    .AutoFilter
End With
```

The following macro will highlight all cells containing a string in all open workbooks.

```vba
Sub FindAll()
    Dim strWhat As String
    Dim wb As Workbook
    Dim ws As Worksheet
    Dim r As Range
    Dim rFirst As Range
    Dim bFirstStep As Boolean

    strWhat = InputBox("Enter a string to find:")

    For Each wb In Application.Workbooks
        For Each ws In wb.Sheets
            Set rFirst = ws.Cells.Find(What:=strWhat, _
                                       After:= ws.Cells(1, 1), _
                                       LookIn:=xlValues, LookAt:= _
                                       xlPart, SearchOrder:=xlByRows, _
                                       SearchDirection:=xlNext, MatchCase:=False)
            If Not rFirst Is Nothing Then
                HighlightIt rFirst
                Set r = ws.Cells.FindNext(After:=rFirst)
                While (Not rFirst.Address = r.Address)
                    HighlightIt r
                    Set r = ws.Cells.FindNext(After:=r)
                Wend
            End If
        Next
    Next
    End Sub

Sub HighlightIt(r As Range)
```

81
9.3.5 Select Odd-Numbered Rows

The following selects every odd row in the worksheet's used range:

```vba
Sub OddRows()
    Dim rngRows As Range
    Dim i As Long
    Set rngRows = Rows(1)
    For i = 3 To Cells.SpecialCells(xlLastCell).Row
        If i Mod 2 = 1 Then
            Set rngRows = Union(rngRows, Rows(i))
        End If
    Next i
    Set rngRows = Intersect(rngRows, ActiveSheet.UsedRange)
    rngRows.Select
End Sub
```

9.3.6 How To Determine If A Selection Has Non-Contiguous Rows

The Range object has an Areas collection that allows you to access multiple non-contiguous ranges in a Selection. Therefore, if Selection.Areas.Count = 1 then you know the selected range is contiguous. If Selection.Areas.Count > 1 then you can access the different parts of the selection with

```vba
Dim rArea As Range
For Each rArea In Selection.Areas
    'Do your stuff here
Next
```

Note that each rArea object iterated above is a Range object that can contain any number of cells/rows/columns, so you may need a second iteration inside the first one to do what you want if you want to work with the cells in each area.

9.3.7 Determining If A Row Or Column Is Empty

The following illustrates how to determine if a row is empty, in this case row 10.

```vba
If Application.CountA(Rows(10)) = 0 Then
    MsgBox "The row is empty"
End If
```

The following determine if column C is empty:
If Application.CountA(Columns(3)) = 0 Then
    MsgBox "The column is empty"
End If

If you need to determine if a the row containing the active cell is empty, then use the following approach:

If Application.CountA(ActiveCell.EntireRow) = 0 Then
    MsgBox "The row is empty"
End If

For a column check, you would use **EntireColumn** instead of **EntireRow**.

### 9.3.8 Duplicating The Last Row In A Set Of Data

The following code will duplicate the last row in a set of data. It assumes that there is always an entry in column A.

```vba
Dim lastRow As Long
lastRow = Cells(Rows.Count, "A").End(xlUp).Row
Rows(lastRow).Copy Cells(lastRow + 1, "A")
```

If you want to clear the entries in certain cells in this new last row, then you can use statements like the following:

```
Cells(lastRow + 1, "C").ClearContents
```

### 9.3.9 Inserting Multiple Rows

The following example looks for the word "test" (any case) in the range A1 to A50, and if found inserts two rows below the cell containing the word

```vba
Sub Insert_Row()
    Dim cell As Range

    'rotate through all the cells in the range
    For Each cell In Range("A1:A50")

    'check value and see if it equal to the word test
    If LCase(cell.Value) = "test" Then
        'if equal, insert two rows below the cell
        cell.Offset(1, 0).EntireRow.Resize(2).Insert
    End If
    Next
End Sub
```
9.3.10 Insert Rows And Sum Formula When Cells Change

The following example checks a column of entries, ands and every time the entry in the column changes the macro inserts two rows and sums the numbers in another column. This macro also prompts the use to specify the first cell to be checked and for any cell in the column to be summed.

Sub Insert_Rows_And_Sum()
    Dim cell As Range, sumCell As Range, comparisonValue
    Dim topSumRow As Integer, sumColumn As Integer

    'get ranges
    On Error Resume Next
    Set cell = Application.InputBox(_
        prompt:="Select the first cell in the ID column", _
        Type:=8)
    If cell Is Nothing Then Exit Sub
    Set sumCell = Application.InputBox(_
        prompt:="Select any cell in the column to be summed", _
        Type:=8)
    If sumCell Is Nothing Then Exit Sub

    'turn off error handling
    On Error GoTo 0

    'initialize values
    comparisonValue = cell.Value
    topSumRow = cell.Row
    sumColumn = sumCell.Column

    'loop until a blank cell is encountered
    While Not IsEmpty(cell)

    'check to see if value has changed
    If cell.Value <> comparisonValue Then

    'if the value has changed, insert two rows and a sum formula
    Range(cell.Offset(1, 0), _
        cell.Offset(2, 0)). EntireRow.Insert
    Cells(cell.Row + 1, sumColumn).Formula = _
        "=Sum(" & Range(Cells(topSumRow, sumColumn), _
        Cells(cell.Row, _
            sumColumn)).Address(False, False) & ")"
'update the cell to be checked, the comparison value, and the
top row number

    Set cell = cell.Offset(3, 0)
    comparisonValue = cell.Value
    topSumRow = cell.Row
Else

    'if the same value, set cell to the next cell

    Set cell = cell.Offset(1, 0)
End If
Wend
End Sub

9.3.11 An Example Of Inserting Rows And Sum Formulas

This macro goes down a column of cells, and whenever the value changes, it inserts two blank rows and then puts in a sum formula underneath the numbers in the column D.

Sub InsertTwoRowsAndSum()
    Dim oldVal
    Dim rwTop As Long
    Dim colmn As Integer
    Dim offColmn As Integer
    If IsEmpty(ActiveCell.Value) Then
        MsgBox "You must select the top of the column " & _
        Chr(13) & "you want to work on"
        Exit Sub
    End If

    'store the active cell's value and row number
    'oldVal is used to compare to the active cell's value as it changes
    'rwTop is used to refer to the first cell in the sum function

    oldVal = ActiveCell.Value
    rwTop = ActiveCell.Row

    'this is the column that will be summed

    colmn = 4

    'determine the offset from the active cell's column

    offColmn = colmn - ActiveCell.Column
    If offColmn = 0 Then
        MsgBox "the column being checked for cell entry " & _
        "changes can not be the same as the one being summed."
        Exit Sub
    End If
End Sub
'select the next cell
ActiveCell.Offset(1, 0).Select

'loop until an empty cell is encountered
While Not IsEmpty(ActiveCell)

'if the value of the cell is not the same as oldVal, insert a row
'and sum formulas
If ActiveCell.Value <> oldVal Then

'add row
ActiveCell.EntireRow.Insert xlUp

'add sum formulas
ActiveCell.Offset(0, offColmn).Formula = "=sum(" & _
Range(Cells(rwTop, colmn), _
    Cells(ActiveCell.Row - 1, colmn)). _
    Address(False, False) & ")"

'go down a row and insert another line
ActiveCell.Offset(1, 0).Select
ActiveCell.EntireRow.Insert xlUp

'store this row number for use in the sum formula and update oldVal
rwTop = ActiveCell.Row + 1
oldVal = ActiveCell.Offset(1, 0).Value
End If

'go to the next row and loop if not blank
ActiveCell.Offset(1, 0).Select
Wend

'handle the sum and row inserts needed for the last set of values
ActiveCell.EntireRow.Insert xlUp
ActiveCell.Offset(0, offColmn).Formula = "=sum(" & _
Range(Cells(rwTop, colmn), _
    Cells(ActiveCell.Row - 1, colmn)).Address(False, False) _
& ")"
ActiveCell.Offset(1, 0).Select
ActiveCell.EntireRow.Insert xlUp
End Sub
9.3.12 Deleting Rows

If you record a macro that deletes a row, you will get code like the following:

```vba
Rows("7:10").Select
Selection.Delete Shift:=xlUp
```

This gives you a clue to how to write the code, but some of the tricks are not obvious. The following examples show how to delete rows a number of different ways.

If you wish it to delete the row containing the active cell and the next two rows, you can do this way:

```vba
Dim A As Long, B As Long
A = ActiveCell.Row
B = A + 2
Rows(A & ":" & B).Delete
```

The following is another approach that uses range variables instead.

```vba
Dim startCell As Range, endCell As Range
Set startCell = ActiveCell
Set endCell = startCell.Offset(2, 0)
Range(startCell, endCell).EntireRow.Delete
```

If the startCell and endCell refer to cells on a sheet that is not the active sheet, then use this approach:

```vba
Dim oWS As Worksheet
Set oWS = startCell.Parent
oWS.Range(startCell, endCell).EntireRow.Delete
```

This could also have been written:

```vba
```

The key is qualifying the `Range()` method with the worksheet when the range being deleted is not on the active sheet.

9.3.13 Deleting Sets Of Rows

The following example deletes every 2nd and 3rd row (i.e. rows 2, 3 and 5, 6 and 8, 9 and >11, 12, ...etc.) from a spreadsheet.

```vba
Sub DeleteEvery()
    Dim N As Integer
    Dim M As Integer
    With ActiveSheet

    'calculate last row not to delete
\[ M = \text{Int}((\text{UsedRange.Rows.Count} + 1) / 3) \times 3 + 1 \]

'step in threes as two rows are deleted and one kept

\[ \text{For } N = M \text{ To 4 Step } -3 \]

'delete the two rows above the row not to delete

\[
\text{.Rows}(N-1).\text{Delete}
\text{.Rows}(N-2).\text{Delete}
\text{Next } N
\text{End With}
\text{End Sub}
\]

9.3.14 Deleting Error Rows

The problem facing the user is the following: In a worksheet there are three adjacent columns with the results of some calculations (about > 50 rows). The user wants to delete the rows where in column 1 AND 2 AND 3 the value is DIV/0!. If one of the three columns has a valid solution, then the user just wants to replace DIV/0! by a single quote and the row must not be deleted.

The following code accomplishes the above task. The user must first select a range of cells just in the first column and then run the macro.

\[ \text{Sub DeleteErrors()} \]
\[ \text{Dim rng As Range, i As Long, j As Integer} \]
\[ \text{Dim eCount As Integer} \]

'set a the range variable rng equal to the selection

\[ \text{Set rng = Selection} \]

'step through the selection rows backwards as rows are to be deleted

\[ \text{For } i = \text{Selection.Rows.Count} \text{ To 1 Step } -1 \]
\[ \text{eCount} = 0 \]
\[ \text{For } j = 1 \text{ To 3} \]

'loop through the three cells on each row. rng(i, 1) refers to the 'cell in the selected range, rng(i, 2) the cell in the column to the right, 'and rng(i,3) the cell two columns to the right. rng(i,2) and rng(i,3) are 'not in the selection or in the range variable rng.

\[ \text{If IsError(rng(i, j)) Then} \]

'note that IsError does not distinguish what kind of error 'is in the cell
rng(i, j).Value = "-"
eCount = eCount + 1
End If
Next j
If eCount = 3 Then rng(i, 1).Resize(1, 3).Delete shift:=xlUp
Next i
End Sub

9.3.15 Deleting Duplicate Rows

The following code will delete duplicate rows if the cell values in the rows in the selection are duplicates. The data first must be sorted, as this code compares each row to the next row.

Sub Delete_Duplicates()
Dim iRows As Long
Dim iCols As Long
Dim RowMax As Long
Dim ColMax As Long
Dim bSame As Boolean
Dim rowMin As Long
Dim colMin As Long

' restrict range to check to just cells in the used range
With Intersect(Selection, ActiveSheet.UsedRange)

' make certain there are at least two rows
If .Rows.Count = 1 Then
    MsgBox "Pick more rows!"
    Exit Sub
End If

' make certain only one range is selected
If .Areas.Count > 1 Then
    MsgBox "Only a single area is allowed"
    Exit Sub
End If

' set min and max columns numbers
rowMin = .Cells(1).Row + 1
RowMax = .Cells(.Cells.Count).Row
colMin = .Cells(1).Column
End With

' check rows, starting from the bottom and working up
For iRows = RowMax To rowMin Step -1


'initialize each time

bSame = True

'check column values

For iCols = colMin To ColMax
    If Cells(iRows, iCols).Value <> _
        Cells(iRows - 1, iCols).Value Then

'if a difference is found set bSame to False

    bSame = False
    Exit For
    End If
Next

'if bSame still true, delete the row

    If bSame Then Rows(iRows).Delete
Next
End Sub

9.3.16 Remove/Highlight Duplicate Rows

Assuming your data is in columns A, B, and C, you can do something like this:

1)select the data

2) use macro code like the following. Note that it deletes from the bottom up.

Dim I As Integer
For I = Selection.Cells(Selection.Cells.Count).Row To _
    Selection.Cells(1).Row + 1 Step -1
    If Cells(I, 1).Value = Cells(I - 1, 1).Value And _
        Cells(I, 2) = Cells(I - 1, 2).Value And _
        Cells(I, 3).Value = Cells(I - 1, 3).Value Then _
            Rows(I).EntireRow.Delete
Next I

9.3.17 Conditionally Deleting Rows

When you are writing code to delete entire rows, you should always run your loops backwards,
from the bottom of the range to the top. This prevents Excel from skipping Rows. The following
illustrates a way to do this. In this example and the subsequent examples, range D8 to D15 is
checked and if the value in the cell is zero, then the row is deleted.

Dim RowNdx As Long
For RowNdx = 15 To 8 Step -1
    If Cells(RowNdx, "D").Value = 0 Then
        Cells(RowNdx, "D").EntireRow.Delete
Another way to have the job done would be a routine along these lines, which utilizes the ability of Excel to delete non-contiguous rows.

```vba
Dim DelRange As Range
Dim c As Range
For Each c In ActiveSheet.Range("D8:D15").Cells
    If c.Value = 0 Then
        If DelRange Is Nothing Then
            Set DelRange = c.EntireRow
        Else
            Set DelRange = Union(DelRange, c.EntireRow)
        End If
    End If
Next c

' turn on error handling in case no range is assigned
On Error Resume Next
DelRange.Delete
On Error GoTo 0
```

It takes up a couple of lines more, but is about 5 times as fast. 10000 lines in about 10 sec. instead of about 54 sec. (On a Pentium 200 Mhz)

The following is another way to select specify rows and delete them without looping. It makes use of the `SpecialCells` method.

```vba
' turn On Error handling in case no matching cells
On Error Resume Next
Range("D7:D15").AutoFilter Field:=1, Criteria:="0"
If Err = 0 Then _
    Range("D8:D15").SpecialCells(xlCellTypeVisible) _.EntireRow.Delete
ActiveSheet.AutoFilterMode = False
ActiveSheet.UsedRange

' turn off error handling
On Error GoTo 0
```

Another technique that is similar is again to use `SpecialCells`, but this time set a Boolean value in the rows to be selected:

```vba
For Each c In Range("D8:D15").Cells
    If c.Value = 0 Then c = True
Next c
```
'Delete all in 1 shot, turn on error handling in case no matching rows

On Error Resume Next
Range("D8:D15") . SpecialCells(xlCellTypeConstants, _
   xlLogical). EntireRow.Delete

'Turn off error handling

On Error GoTo 0

9.3.18 How To Delete Blank Rows

You can use the following statements to find the blank cells in a column, and then delete the rows:

On Error Resume Next
Intersect(Activesheet. UsedRange, Columns("A:A")). _
   SpecialCells(xlCellTypeBlanks). EntireRow.Delete
On Error GoTo 0

The On Error statements are needed in case there are no blank cells in the column and in the used range.

The following is an alternate way that checks each row and deletes the only one if there are no entries in the row:

Sub Example1()
   Dim Firstrow As Long
   Dim Lastrow As Long
   Dim Lrow As Long
   Dim CalcMode As Long

   With Application
      CalcMode = .Calculation
      .Calculation = xlCalculationManual
      .ScreenUpdating = False
   End With

   Firstrow = ActiveSheet.UsedRange.Cells(1).Row
   Lastrow = ActiveSheet.UsedRange.Rows.Count + Firstrow - 1

   'delete from the bottom up

   For Lrow = Lastrow To Firstrow Step -1
      If Application.CountA(Rows(Lrow)) = 0 _
         Then Rows(Lrow).Delete
   Next Lrow
End Sub
This will delete the row if the whole row is empty (all columns)

Next
Application.Calculation = CalcMode
End Sub

9.3.19 Examples That Delete Rows Based On A Cell's Value

The following is an example that deletes a row if the value in column A is zero:

Sub DeleteZeroLine()
    If Cells(ActiveCell.Row, 1).Value = 0 Then _
        ActiveCell.EntireRow.Delete
End Sub

the following is a slight modification that also makes certain that the entry in column A for the active row is not empty. As the value of an empty cell is considered to be zero. It also uses a With..End With statement to make the code simpler

Sub DeleteZeroLine()
    With ActiveCell
        If Cells(.Row, 1).Value = 0 And _
            Not IsEmpty(Cells(.Row, 1)) Then _
            .EntireRow.Delete
    End With
End Sub

The following checks all the cells in selection and deletes the row if the value in the row is less than 1. No check is made for blank cells as it is assumed in this example that such rows should also be deleted.

9.3.20 Auto Sizing Rows When Cells Are Merged

Merged cells won't auto size. Jim Rech has written a VBA procedure which will size the row in this instance.

Sub AutoFitMergedCellRowHeight()
    Dim CurrentRowHeight As Single
    Dim MergedCellRgWidth As Single
    Dim CurrCell As Range
    Dim ActiveCellWidth As Single
    Dim PossNewRowHeight As Single
    If Not ActiveCell.MergeCells Then Exit Sub
    With ActiveCell.MergeArea
        If .Rows.Count = 1 And .WrapText = True Then
            Application.ScreenUpdating = False
            CurrentRowHeight = .RowHeight
            For Each CurrCell In Selection
                MergedCellRgWidth = CurrCell.ColumnWidth + _
                MergedCellRgWidth
            Next
            PossNewRowHeight = CurrentRowHeight + MergedCellRgWidth
            For Each CurrCell In Selection
            If CurrCell.ColumnWidth > PossNewRowHeight Then _
                MergedCellRgWidth = CurrCell.ColumnWidth + _
                MergedCellRgWidth
            Next
            MergedCellRgWidth = MergedCellRgWidth
        End If
    End With
End Sub
9.4 COLUMN EXAMPLES

9.4.1 Making Certain That A Selection Is Only A Single Column Or Row Wide

When you ask the user to provide you with a selection that can only be a single row or column wide, you have no assurance that the user will make a selection that meets your criteria. The following examples show how to check the selection that the user made to insure that it meets these criteria.

Example 1 - verifying columns selected

'count the number of columns. If more than one, stop the macro.

If Selection.Columns.Count > 1 Then
    MsgBox "Select only a single column. No action taken."

' stop the macros

    End
    End If

' code to execute if selection is a single column

Example 2 - verifying rows selected

Dim userRange As Range

' get a range from the user via an input box

On Error Resume Next
Set userRange = Application.InputBox(
    prompt:="Select cells on a single row for processing", 
    Type:=8, default:=Selection.Address)
On Error GoTo 0
'if no range selected, stop

If userRange Is Nothing Then Exit Sub

'if cells on more than one row selected, display message and stop

If userRange.Rows.Count > 1 Then
    MsgBox "Select just cells on a single row. " & _
    "No action taken"

'stop the macros

End
End If

'if selection passes the above text run any code place here

9.4.2 Converting Column Letters To Column Numbers

The following converts columns letters to numbers. For example, supplying "F" to this statement would return the number 5.

Function ColumnNumber(AlphaColumn As String) As Integer
    ColumnNumber = Columns(AlphaColumn).Column
End Function

9.4.3 Converting Alphabetic Column Labels To Numeric Column Labels

There are several easy ways to convert column labels such as "CF" to a number and use in a Visual Basic statement

AlphaColumn = "CF"
numberColumn = Range(AlphaColumn & "1").Column

or

Cells(intRow, Range(AlphaColumn & "1").Column).Value = 5

numberColumn = Columns(AlphaColumn).Column

9.4.4 Getting The Letter Of A Column

The following illustrate several ways to get the column letter of a column:

Sub Approach1
    Dim colLetters As String
    Dim N As Integer
'prompt for a column number, exit if none entered

N = Val(InputBox("enter a column number"))
If N = 0 Then Exit Sub
With Worksheets(1).Columns(N)

'extract the column letters from the address
'first worksheet in the workbook is used for convenience

colLetters = Left(.Address(False, False), _
    InStr(.Address(False, False), ":") - 1)
End With

'display the letter

MsgBox colLetters
End Sub

Approach 2:

Public Function ColumnLetter(anyCell As Range) As String
ColumnLetter = Left(anyCell.Address(False, False), _
    1 - CInt(anyCell.Column > 26))
End Function

Sub Demo()
    MsgBox ColumnLetter(ActiveCell)
End Sub

9.4.5 Comparing Two Columns

The following code illustrates how to compare column 'A' and 'B' by Visual Basic code. The cells at each column contain figures. If the cell amount in column A is greater than the cell amount in column B, then the cell in column B should have a red color background.

Sample Data:

A1 100 B1 100
A2 150 B2 50 this cell should be 'red'
A3 40 B3 20
A4 50 B4 100

The following is the simple solution to this. Please note that it hard codes the ranges.

Sub CompareColumns()
    Dim CurrCell As Range
'loop through each cell in the selection

For Each CurrCell In Range("B1:B4")

'compare values; .offset(0,1) refers to the cell to the left of CurrCell

If CurrCell.Value < CurrCell.Offset(0, -1).Value Then

'color the cell red if true

CurrCell.Interior.ColorIndex = 3
Else

'remove any color if not true

CurrCell.Interior.ColorIndex = xlNone
End If
Next
End Sub

A more flexible solution is the following, which prompts the user for the first range and then the first cell in the second range.

Sub CompareColumns()
Dim cell As Range
Dim firstRange As Range, firstCell As Range
Dim I As Integer

'turn on error checking in case cancel selected in inputbox

On Error Resume Next

'use Application.InputBox with Type:=8 so that the user must input a range.
'Set the default equal to the current selection

Set firstRange = Application.InputBox(_
prompt:="Please select the first range of cells", _
Type:=8, _
default:=Selection.Address(False, False))

'exit if cancel selected or no range entered

If firstRange Is Nothing Then Exit Sub

'prompt for the comparison cell related to the first cell of the above range

Set firstCell = Application.InputBox(_
prompt:="Please select the cell related to " & _
firstRange.Cells(1).Address(False, False), _
Type:=8)
'exit if cancel selected or no range entered

If firstCell Is Nothing Then Exit Sub

'turn off error checking

On Error GoTo 0
I = 0

'rotate through each cell and compare to its related cell

For Each cell In firstRange
If firstCell.Offset(I, 0).Value > cell.Value Then
  firstCell.Offset(I, 0).Interior.ColorIndex = 3
Else
  firstCell.Offset(I, 0).Interior.ColorIndex = xlNone
End If

'increment I for next cell

I = I + 1
Next
End Sub

9.4.6 How To Convert Alphabetic Column Labels To Numeric

If you have a column letter and want to turn it into a number, you can use the following approach:

numberColumn = Worksheets(1).Columns(AlphaColumn).Column

9.4.7 How To Copy Multiple Columns At A Time

If you want to copy a group of columns that are together, then you can use a statement like the following:

Columns("A:G").Copy

However, if you want to copy just columns A and columns G, then you must use the following statement instead.

Range("A:A,G:G").Copy

If you used Columns("A:A,G:G").Copy, you would get an error.

9.4.8 How To Delete Columns In Multiple Sheets At One Time

the following deletes D column in all worksheets:
but the following does not work - only deleting column D from the active sheet:

```
Worksheets(1).Activate
Worksheets.Select
Columns("D:D").Delete
```

It appears to be one situation that VBA can only handle by selecting,

### 9.4.9 How To Insert Columns In Multiple Sheets At One Time

the following deletes D column in all worksheets:

```
Worksheets(1).Activate
Worksheets.Select
Columns("D:D").Select
Selection.Insert
```

but the following does not work - only deleting column D from the active sheet:

```
Worksheets(1).Activate
Worksheets.Select
Columns("D:D").Insert
```

It appears to be one situation that VBA can only handle by selecting,

### 9.4.10 An Insert A Column And Formula Example

In this example, the user needs a macro that inserts column D, puts a formula in D1 =sum(a1:c1) and copies it down as many rows as there is data in column C. The number of rows varies day to day. The following macro accomplishes this task:

```
Sub FillFormulas()
Dim myRng As Range
Dim lastRw As Long

'find the last row in column C, starting at cell C1
lastRw = Worksheets("Sheet1").Range("C1").End(xlDown).Row

'write a sum formula to cell D1
Set myRng = Worksheets("Sheet1").Range("D1")
myRng.Formula = "=SUM(A1:C1)"

'fill the formula down column D
```
The following is an alternate solution to the above problem:

```vba
Sub AlternateFill()
    Dim c As Range

    'insert a column
    Columns("D").Insert

    'set a range variable equal to the first cell in column C
    Set c = ActiveSheet.Range("C1")

    'loop until a blank is encountered
    Do While c <> ""

        'write a formula using R1C1 notation
        c.Offset(0, 1).FormulaR1C1 = "=Sum(RC[-3]:RC[-1])"

        'set c to the next cell down
        Set c = c.Offset(1, 0)
    Loop
End Sub
```

### 9.4.11 Deleting Columns

If you record a macro that deletes a column, you will get code like the following:

```vba
Columns("E:G").Select
    Selection.Delete Shift:=xlToLeft
```

As long as you know the column letters, you can use the above approach. And, you can simplify it:

```vba
Columns("E:G").Delete
```

However, because Visual Basic does not return column letters, the above approach is very limited when you are writing code. Instead, you must work around this limitation by using the **Cells()** method and the **Range()** method. The following examples show how to use these methods to delete columns a number of different ways.

If you wish it to delete the column containing the active cell and the next two columns, you can do it this way:
Dim A As Long, B As Long
A = ActiveCell.Column
B = A + 2
Range(Cells(1, A), Cells(1, B)).EntireColumn.Delete

The following is another approach that uses range variables to accomplish this task.

Dim startCell As Range, endCell As Range
Set startCell = ActiveCell
Set endCell = startCell.Offset(2, 0)
Range(startCell, endCell).EntireColumn.Delete

If the startCell and endCell refer to cells on a sheet that is not the active sheet, then use this approach:

Dim oWS As Worksheet
Set oWS = startCell.Parent
oWS.Range(startCell, endCell).EntireColumn.Delete

This could also have been written:


The key is qualifying the Range() method with the worksheet when the range being deleted is not on the active sheet.

9.4.12 Setting Column Widths

The following illustrates how to set the width of a column:

Columns("b").ColumnWidth = 25

If you have a range of cells and want to set the column range for this range, do it like the following:

Range("A1:D5").EntireColumn.ColumnWidth = 1

To AutoFit the column widths, use a statement like the following:

Columns("A:A").EntireColumn.AutoFit

or

Range("A1:D5").EntireColumn.AutoFit

9.4.13 Setting Column Widths And Row Heights

The following illustrates how to set row widths and column heights:
'this sets the column width and row width of the active cell

ActiveCell.EntireColumn.ColumnWidth = 12
ActiveCell.EntireRow.RowHeight = 15

'in this example, variables X, Y, M, and N have been determined by
earlier code, and are used to specify a cell range on the active sheet

Range(Cells(X, Y), Cells(M, N)).EntireRow.RowHeight = 15

If the range is on a different sheet, then qualify the range with the sheet name. If the sheet is not
in the active workbook, then qualify the sheet with the workbook book:

Workbooks("Some Book").Sheets("some sheet") _
.Cells(4, 4).EntireRow.RowHeight = 24

If the range variable "myRange" has been set to refer to a range of cells on a worksheet, then the
following would set the column width of the columns in this range to an automatic fit. Because
"myRange" is a range variable, you do not need to qualify it with the worksheet or workbook.

myRange.EntireColumn.ColumnWidth = AutoFit

The following example will auto fit columns A through Z of the active sheet:

Columns("A:Z").ColumnWidth = AutoFit

9.4.14 Setting Column Widths To A Minimum Width

The following code will auto-fit all columns, and then check all columns in the used range and if
their width is less than 13.5, set the width to 13.5

Sub gbColFit()
  Dim sh As Worksheet
  Dim col As Range

  'rotate through all worksheets in the active workbook

  For Each Sh In ActiveWorkbook.Worksheets

    'auto-fit all columns

    Sh.Cells.EntireColumn.AutoFit
    For Each col In sh.UsedRange.Columns

      'check the columns in the used range and set to a minimum width

      If col.ColumnWidth < 13.5 Then
        col.ColumnWidth = 13.5
      End If
    Next col
  Next Sh
End Sub
9.4.15 Setting Column Width And Row Height In Centimeters

The macros below to set the column width and row height in millimeters (print size/100% zoom):

Sub SetColumnWidthMM(ColNo As Long, mmWidth As Integer)
'Set the columnwidth in millimeters (approximately)
Dim w As Single
If ColNo < 1 Or ColNo > 255 Then Exit Sub
Application.ScreenUpdating = False
w = Application.CentimetersToPoints(mmWidth / 10)
While Columns(ColNo + 1).Left - Columns(ColNo).Left - 0.1 > w
  Columns(ColNo).ColumnWidth = _
  Columns(ColNo).ColumnWidth - 0.1
Wend
While Columns(ColNo + 1).Left - Columns(ColNo).Left + 0.1 < w
  Columns(ColNo).ColumnWidth = _
  Columns(ColNo).ColumnWidth + 0.1
Wend
End Sub

Sub SetRowHeightMM(RowNo As Long, mmHeight As Integer)
'Set the rowheight in millimeters
If RowNo < 1 Or RowNo > 65536 Then Exit Sub
Rows(RowNo).RowHeight = _
  Application.CentimetersToPoints(_
  mmHeight / 10)
End Sub

The sample macro below shows how you can change the column width of column C and the row height of row 3 to 3.5 cm:

Sub ChangeWidthAndHeight()
  SetColumnWidthMM 3, 35
  SetRowHeightMM 3, 35
End Sub

9.4.16 Determining The Populated Cells In A Column Of Data

The following illustrates how to find the range of cells in a column that contains data. In this example, the assumption is that the there are no blank cells in the column, and that the data cells contain no blanks:
'this example assumes that the sheet has not been modified by deleting rows and that only a uniform block of data is on the sheet:

```vba
Set ActiveColArea = Intersect(Columns("a"), _
  ActiveSheet.UsedRange)
```

OR

```vba
Set activeColArea = Range(Cells(1,1), Cells(10000,1).End(XlUp))
```

'assuming that row 10000 is well below the last entry in the column

### 9.5 FINDING THE FIRST BLANK CELL

#### 9.5.1 Determining Where The First Blank Is In A Column

The following illustrates how to find the first blank in a column, in this case in Column A, and assign it to a range variable. it assumes that both cell A1 and A2 have an entry.

```vba
Sub findLastCell()
  Dim firstBlank As Range

  'set a range variable to the first blank
  Set firstBlank = Range("A1").End(xlDown).Offset(1, 0)

  'display the cell found
  MsgBox firstBlank.Address
End Sub
```

The above will also scroll the screen if done on the active sheet. To avoid this scrolling, use code like the following:

```vba
Sub findLastCell_Without_Scrolling()
  Dim firstBlank As Range
  Dim scrollCol As Integer, scrollRow As Integer

  'store the scroll settings
  scrollCol = ActiveWindow.ScrollColumn
  scrollRow = ActiveWindow.scrollRow

  'set a range variable to the first blank
  Set firstBlank = Range("A1").End(xlDown).Offset(1, 0)

  'apply the stored scroll settings
  ActiveWindow.ScrollColumn = scrollCol
  ActiveWindow.scrollRow = scrollRow
End Sub
```
ActiveWindow.ScrollColumn = scrollCol
ActiveWindow.scrollRow = scrollRow

'display the cell found

MsgBox firstBlank.Address
End Sub

9.5.2 Finding The First Blank Cell In A Column

The following returns the first blank cell in a column. In this example, in column C.

Dim colNum As Integer
colNum = 3
Dim eCell As Range
Set eCell = Columns(colNum) _
    .SpecialCells(xlCellTypeBlanks).Cells(1)
eCell.Select

9.5.3 How To Find The Next Available Row In Column

Assuming you want the first blank row with no data below it.

NextRow = Cells(Rows.Count,"A").End(xlUp).Offset(1,0).Row

Will identify the first blank row in column A at the bottom of your data.

Cells(Rows.Count,"A").End(xlUp).Offset(1,0).EntireRow.Select

Will select the entire row.

Cells(Rows.Count,"A").End(xlUp).Offset(1,0).Select

will select just the cell in column A

If you have a cell selected in the column and there is data below it, but there may be a break and then more data, to find the first blank cell below the cell you have selected:

ActiveCell.End(xlDown).Offset(1,0).Select

9.6 SELECTING THE LAST CELL

9.6.1 The VBA Equivalents Of Ctrl-Shift-Down And Ctrl-Down

The visual basic equivalents of these two manual actions are

'this is the same as ctrl-shift-down
Range(ActiveCell, ActiveCell.End(xlDown)).Select

'this is the same as ctrl-down

ActiveCell.End(xlDown).Select

Please note that you may not get the results you expected with the above. For example, if the cell below the active cell is empty, the statement will select to the next cell with an entry or to the last cell in the sheet if there is no cell with an entry below the active cell.

You can also use the above statements and assign the resulting range or cell to a range variable versus selecting the cell. Doing so makes your code faster as you are not doing a select.

Dim tempRange As Range

Set tempRange = Range(ActiveCell, ActiveCell.End(xlDown))

or

Dim tempRange As Range

Set tempRange = ActiveCell.End(xlDown)

**9.6.2 Determining The Last Cell In A Column**

To set the variable lastCell to the last cell in a column do the following. If the variable topCell has been set to the first cell in a column containing an entry and there are no blanks cells in the column until one reaches the last cell.

Set lastCell = topCell.End(xlDown)

The above approach works even if topCell is not on the active sheet.

If you are uncertain if topCell is really the top cell, or if there are blank cells in the column, then use one of the following approaches

if you know that there are no cells below row 1000 with entries and the active sheet is the sheet containing topCell then do this

Set lastCell = Cells(1001, topCell.Column).End(xlUp)

If topCell is on a sheet other then the active sheet, then use this approach.

Set lastCell = topCell.Parent.Cells(1001, _
     topCell.Column).End(xlUp)

If you are uncertain how many rows may have data then use the following approach to get the last entry in the last column. Note the period in front of the key word Cells in the third row of this example
Dim maxRow As Long
With topCell.Parent.UsedRange
    maxRow = .Cells(.Cells.Count).Row + 1
End With
Set lastCell = topCell.Parent.Cells(maxRow, _
    topCell.Column).End(xlUp)

9.6.3 Finding The Last Entry In A Column

Their are several different ways to find the last entry in a column

If you know a row number which will always be below the last entry in the column, and you
know the column number, then the last cell can be found very easily:

Dim lastEntryCell As Range

'set variable equal to the last cell in column 3, assuming last cell is well
'above row 20000
Set lastEntryCell = Cells(20000, 3).End(xlUp)

If you're uncertain of a row well below the last entry, you can find it this way:

'declare a variable as long so this will work if more than 32,768 rows
Dim lastRow As Long
Dim lastEntryCell As Range

'use the count property to return the index number of the last cell in the
'UsedRange, then return its row number
With ActiveSheet.UsedRange

'add 100 to the last used row to get a number well below the last row
    lastRow = .Cells(.Count).Row + 100
End With

'set variable equal to the last cell in column 3
Set lastEntryCell = Cells(lastRow, 3).End(xlUp)

If you've got many entries in a column, and no blanks until the last entry, then you can find the
last cell by using code like the following:

Dim lastEntryCell As Range
Set lastEntryCell = Range("A1").End(xlDown)

Using End(xlDown) or End(xlUp) will cause the screen to scroll if used on the active sheet. To
prevent this, use code like the following.
'store the scroll settings

Dim scrollCol As Integer, ScrollRow As Integer
scrollCol = ActiveWindow.ScrollColumn
scrollRow = ActiveWindow.ScrollRow

'set a range variable to the last cell

'<code that finds the last cell>

'apply the stored scroll settings

ActiveWindow.ScrollColumn = scrollCol
ActiveWindow.ScrollRow = scrollRow

9.6.4 Finding The Last Non-Blank Cell In A Column

assuming that variable "c" has been assigned to the column number in question and the active sheet is the one containing the column in question then

Dim cell As Range
Set cell = Cells(65536, c).End(xlUp)

will do the trick.

If the column is on a sheet other than the ActiveSheet, then qualify cells with the sheet object:

Set cell = Sheets("some sheet name").Cells(65536, c).End(xlUp)

and so forth.

You can also use this approach, which eliminates the need for you to put in a row number:

Dim cell As Range

9.6.5 Finding The Last Entry In A Row

Their are several different ways to find the last entry in a row

The following is probably the easiest way to find the last entry in a row:

Dim lastRowEntryCell As Range

'set variable equal to the last cell in row 3

Set lastEntryCell = Cells(3, 256).End(xlLeft)
If you've got many entries in a row, and no blanks until the last entry, then you can find the last cell by using code like the following:

```vba
Dim lastEntryCell As Range
Set lastEntryCell = Range("A1").End(xlRight)
```

Using `End(xlDown)` or `End(xlUp)` will cause the screen to scroll if used on the active sheet. To prevent this, use code like the following.

'**store the scroll settings**

```vba
Dim scrollCol As Integer, ScrollRow As Integer
scrollCol = ActiveWindow.ScrollColumn
scrollRow = ActiveWindow.scrollRow
```

'**set a range variable to the last cell**

' `<code that finds the last cell>`

'**apply the stored scroll settings**

```vba
ActiveWindow.ScrollColumn = scrollCol
ActiveWindow.scrollRow = scrollRow
```

**9.6.6 Determining The Last Cell In A Row**

To set the variable `lastCell` to the last cell in a row do the following If the variable `topCell` has been set to the first cell in a row containing an entry and there are no blanks cells in the row until one reaches the last cell.

```vba
Set lastCell = topCell.End(xlToRight)
```

The above approach works even if `topCell` is not on the active sheet.

If you are uncertain if `topCell` is really the first cell in the row, or if there are blank cells in the row, then use one of the following approaches

1) if you know that there are no entries in the last column of the worksheet and the active sheet is the sheet containing `topCell` then do this

```vba
Set lastCell = Cells(topCell.Row, 256).End(xlToLeft)
```

2) If `topCell` is on a sheet other then the active sheet, then use this approach.

```vba
Set lastCell = topCell.Parent.Cells(topCell.Row, 256) End(xlUp)
```
9.6.7 Finding The Last Cell, Last Row, or Last Column

Here are some additional "last cell" examples:

'1) VERY Last in a worksheet:

```
LastRow = Rows.Count
LastCol = Columns.Count
Set LastCell = Cells(LastRow, LastCol)
```

'2) Last in Used Range

```
LastRow = ActiveSheet.UsedRange.Row
LastColumn = ActiveSheet.UsedRange.Column
Set LastCell = Cells(LastRow, LastCol)
```

'or

```
Set LastCell = _
```

'3) Last Non-Blank Cell In A Column

```
Set LastCell = Range(Rows.Count, 2).Offset(1, 0).End(xlUp)
```

'where 2 indicates column 2 ("B")

'4) Last Non-Blank Cell In A Row

```
Set LastCell = Range(2, Columns.Count).Offset(0, 1).End(xlLeft)
```

'where 2 indicates row 2.

The following is still another example. In this case the user wanted to select from cell H2 to the last entry in the column

```
Range(Range("H2"), Range("H" & Rows.Count).End(xlUp)).Select
```

In the above example, Rows.Count returns the number of the last row in the workbook. End(xlUp) is the same as pressing the End button and then up arrow.

9.6.8 Selecting from the ActiveCell to the Last Used Cell

The following statement will select from the active cell to the last cell in a worksheet's used range:

```
With ActiveSheet.UsedRange
    ActiveSheet.Range(ActiveCell, _
```
Note that Cells is qualified with a period, which means that it is qualified by the With object, the active sheet's used range. The code .Cells.Count returns the number of cells in the used range. .Cells(.Cells.Count)) returns the bottom right cell in the used range.

You can assign this range to an object variable, which you then use in your code instead of selecting it.

```vba
Dim someCells As Range
With ActiveSheet.UsedRange
    Set someCells = ActiveSheet.Range(ActiveCell, _
        .Cells(.Cells.Count))
End With
```

### 9.6.9 Determining The Last Cell When Multiple Areas Are Selected

Selecting the last cell in a multiple area selection is not difficult - if you have a clear definition of what you mean by the last cell. For example, is it the last cell selected, the cell with the highest row number, or the highest column number, or some other combination, or is it the cell that is the intersection of the largest row and column number (an may not be part of the selected range)? The following examples illustrate solutions to several of these possible last cell possibilities.

The following code returns the last selected cell, which may not be the one with the largest row or column number:

```vba
Dim Rng As Range, lastArea As Range, lastCell As Range
Set Rng = Selection

'determine the last area selected
Set lastArea = Rng.Areas(Rng.Areas.Count)

'determine the last cell in the last area
Set lastCell = lastArea.Cells(lastArea.Cells.Count)
MsgBox lastCell.Address
```

The following returns the last cell, which is the intersection of the largest row number and the largest column number. Typically, this cell is not in the selected range.

```vba
Dim lastCell As Range
Set lastCell = Selection.SpecialCells(xlLastCell)
MsgBox lastCell.Address
```

The following returns the cell that has the largest row number, and is in the right most column:
Dim lastCell As Range
Dim rng As Range
Dim area As Range
Dim tempR As Range

' store selection to a range variable
Set rng = Selection

' loop through each area
For Each area In rng.Areas

' set a variable to the last cell in an area
Set tempR = area.Cells(area.Cells.Count)
If lastCell Is Nothing Then

' initialize last cell the first time through
Set lastCell = tempR
ElseIf tempR.Row > lastCell.Row Then

' if the row number is bigger, update last cell variable
Set lastCell = tempR
ElseIf tempR.Row = lastCell.Row Then

' if rows numbers are the same, update if the column is higher
If tempR.Column > lastCell.Column Then
Set lastCell = tempR
End If
End If
Next
MsgBox lastCell.Address

9.6.10 Finding the Last Row and Column Numbers

The following code returns the row and column numbers of the last row with an entry and last column with an entry:

Dim rNum As Long
rNum = Cells.Find(What:="*", After:=Range("a1"), _
    SearchOrder:=xlByRows, _
    SearchDirection:=xlPrevious).Row
MsgBox "last row with an entry: " & rNum
Dim cNum As Integer
cNum = Cells.Find(What:="*", After:=Range("a1"), _
    SearchOrder:=xlByColumns, _
SearchDirection:=xlPrevious).Column
MsgBox "Last column with an entry: " & cNum

Once you have the above, you could then select the range from A1 to the bottom corner of the used range:

`Range(Range("A1"), Cells(rNum, cNum)).Select`

The above approach is often preferred to using the **UsedRange** property of a sheet, as **UsedRange** will sometimes return a much broader range due to formats beyond cells with entries.

### 9.6.11 Fill Down

If you have a column of cells and modify the first cell, double clicking on the fill handle on the cell you changed will copy the modified cell down the column, to the first empty cell. The following macro does the same action:

```vba
Sub Fill_Down()
    Dim cell As Range
    Set cell = ActiveCell
    If IsEmpty(cell.Offset(1, 0)) Then
        MsgBox "The active cell is the last cell"
    Else
        Set lastCell = cell.End(xlDown)
        cell.Copy Range(cell, lastCell)
    End If
End Sub
```

### 9.7 COLOR AND FORMAT EXAMPLES

#### 9.7.1 Color Every Other Row Gray And Bold Text

The following code illustrates a way to color every other row gray and to bold the text in those rows.

```vba
Sub Gray_Alt_Rows()
    Dim Cell As Range, my_Range As Range

    'eliminate screen flashing
    Application.ScreenUpdating = False

    'set the range to change - must be single column wide
    Set my_Range = ActiveSheet.Range("B1:B100")
    For Each Cell In my_Range

        'act on the entire row that the cell is on
```

```vba
    Select Case Cell.Row
        Case Is = 2
            Cell.Interior.Color = 15
            Cell.Font.Bold = True
        Case Else
            Cell.Interior.Color = 16
    End Select
    Next Cell
End Sub
```
With Cell.EntireRow

' Mod divides two numbers and returns only the remainder
' color only the rows when this value is zero
' and reset the other rows

If Cell.Row Mod 2 = 0 Then
  .Interior.ColorIndex = 15
  .Font.Bold = True
Else
  .Interior.ColorIndex = xlNone
  .Font.Bold = False
End If
End With
Next
End Sub

The following is another approach. It sets a conditional format on the range of selected cells. The advantage of this approach is that you can insert rows and the formatting automatically adjusts:

Sub SetConditionalFormatting()
  With Selection
    .FormatConditions.Delete
    .FormatConditions.Add Type:=xlExpression, Formula1:=
    "=MOD(ROW(),2)=0"
    With .FormatConditions(1).Interior
      .ColorIndex = 15
      .PatternColorIndex = 15
      .Pattern = xlGray50
    End With
  End With
End Sub

9.7.2 Coloring Cells Based On Their Value

The following routine will color a cell in a selection red if the cell's value is greater than 5

Dim cell As Range
For Each cell In Selection

' test for a numeric value first

If IsNumeric(cell.Value) Then
  ' test the value of the cell
  If cell.Value > 5 Then

    ' color if greater than 5

    cell.Interior.ColorIndex = 3
  Else
'clear color if not greater than 5

    cell.Interior.ColorIndex = xlNone
    End If
End If
Next

9.7.3 Coloring Cells Example

The following will change the background color of each row that has the text "account" in column 1.

Sub ColorThem()
    Dim Ndx As Long
    For Ndx = 1 To ActiveSheet.UsedRange.Rows.Count
        If LCase(Cells(Ndx, 1).Value) = "account" Then
            Rows(Ndx).Interior.ColorIndex = 3
        End If
    Next Ndx
End Sub

How to have your macro wait for a few seconds

The following statement,

    Application.Wait Now + TimeValue("00:00:05")

will wait for 5 seconds and then run the next statement.

9.7.4 Copying Formats From One Sheet To Another

The following code illustrates how to perform an action, in this case copying formats, on a group of selected sheets:

Sub Copy_Formats()
    Dim sh As Object
    Dim topCell As Range

    'set a range variable to the first cell in the selection
    Set topCell = Selection.Cells(1)

    'copy the selection for later use
    Selection.Copy

    'rotate through each selected sheet, and paste the formats if
    'the sheet is a worksheet
For Each  sh  In  ActiveWindow.SelectedSheets
   If  TypeName(sh)  =  "Worksheet"  Then  
      sh.Range(topCell.Address).PasteSpecial  
         Paste:=xlFormats
   Next
End Sub

9.7.5 Summing Cells Based On Cell Color

The following macro will sum all cells in the selection that have a background color of red.

Sub  SumOfColor()
   Dim  sumIt  As  Single
   Dim  cell  As  Range

   'initialize the sum variable to zero (not required, but good practice
   sumIt  =  0

   'check each cell in the selection that is also in the used range.
   'This avoids having to check empty cells if an entire column
   'or row is selected.
   For  Each  cell  In  Intersect(Selection,  ActiveSheet.UsedRange)

      'the interior refers to the background of a cell
      
         If  cell.Interior.ColorIndex  =  3  Then
            sumIt  =  sumIt  +  cell.Value
         End  If
   Next  cell

   'display the results
   MsgBox  "the sum is "  &  sumIt
End  Sub

9.7.6 Outlining A Selection

The macro recorder records far too much code when creating a border around a selection. The
following is the compact code:

Selection.BorderAround  xlContinuous,  xlThin

For more information on  BorderAround,  highlight it and press F1.

9.7.7 Getting The Formatted Contents Of A Cell
The range property **Text** returns the formatted content of a cell versus its fully calculated value. For example if a cell contains the equation =22/3, the calculated value is 3.14285714285714. But if you format the cell to two decimals, then **ActiveCell.Text** returns 3.14.

If the cell's column width is too small to display the formatted text, **ActiveCell.Text** returns what you see, which is a series of # signs.

### 9.8 WORKING WITH FORMULAS

#### 9.8.1 Writing Formulas That Require Double Quotes

If you need to have Visual Basic write a formula to a worksheet cell and quotes are needed in the resulting formula, use two double quotes instead of one. For example, if you want the following formula to appear in a cell:

```
= Text(A1,"0000")
```

Then use a statement like the following in your code:

```
Worksheets(1).Range("B1").Formula = "=Text(A1,""0000"")"
```

Notice the two double quotes in front and after the 0000.

You could also write the above statement using **Chr**(34), which returns a double quote:

```
Worksheets(1).Range("B1").Formula = _
  "=Text(A1, Chr(34) & "0000" & Chr(34) & ")"
```

#### 9.8.2 The Difference Between Formula And FormulaR1C1

If you care creating a formula for a cell or a range, you can use either **Formula** or **FormulaR1C1**. If you use **Formula**, then you must use A1 notation. If you use **FormulaR1C1**, then you must use R1C1 notation. A1 notation is the one you see when you are editing a worksheet. R1C1 allows you to reference cells by their relative distance from the formula cell or by absolute reference, and is what the macro recorder uses.

These two are the same:

```
ActiveCell.Formula = "= A1+B2"
```

```
ActiveCell.FormulaR1C1 = "= R1C1 + R2C2"
```

These two are the same:

```
ActiveCell.Formula = "'=" & ActiveCell.Offset(1,0).Address
```

```
ActiveCell.FormulaR1C1 = "=R[1]C"
```
Mixing A1 and R1C1 notation or using the wrong notation will cause an error or unpredictable results.

9.8.3 Modifying A Cell's Formula

The following examples illustrate how to modify a cell's formula. In this case the formulas are being modified to multiply the existing equation by the range name "BaseRate"

```vba
Sub CellTimesBaseRate()
    With ActiveCell
        .Formula = .Formula & " * baserate"
    End With
End Sub

Sub RangeTimesBaseRate()
    Dim cell As Range

    'loop through all the cells in the selection using a For..Next loop
    For Each cell In Selection

        'write back a formula, which requires an equal sign
        cell.Formula = cell.Formula & " * baserate"
    Next cell
End Sub

If the cells contain values instead of formulas, then use code like the following:

```vba
Sub CellTimesBaseRate()
    With ActiveCell

        'write back a formula, which requires an equal sign

        .Formula = "=" & .Value & "*baserate"
    End With
End Sub

Sub RangeTimesBaseRate()
    Dim cell As Range

    'loop through all the cells in the selection using a For..Next loop
    For Each cell In Selection

        'write back a formula, which requires an equal sign

        cell.Formula = "=" & cell.Value & "*baserate"
    Next cell
End Sub
```
9.8.4 Determining If A Cell Contains A Formula

You can use the HasFormula property to determine if a cell contains a formula. It returns True if the cell has a formula, False if it does not.

If ActiveCell.HasFormula Then

'code to run if cell has a formula

End If

9.9 WORKING WITH COMMENTS

9.9.1 Checking For Comments

The following is how to check to see if a cell has a comment:

Dim C As Comment
On Error Resume Next
Set C = ActiveCell.Comment
On Error Goto 0
If C Is Nothing Then MsgBox "No comment."

This macro should give you an idea of how to check if a range for comments:

Sub FindComments()
Dim(rngToSearch As Range
Dim rngToFind As Range

On Error GoTo ExitFindComments

'Set the range to check for comments

Set rngToSearch = Selection

For Each rngToFind In rngToSearch
    If Not rngToFind.Comment Is Nothing Then
        MsgBox "Comment found in " & rngToFind.Address
    End If
Next rngToFind

ExitFindComments:
End Sub

9.9.2 Commenting A Cell With A Macro

The sub routine "CommentTheCell" can be called by other routines to place a comment in a cell note, or to remove any comment by using the word "none" as the argument.
Sub CommentExample1()
    Dim cell As Range

    'create a range name to use when calling the routine CommentTheCell
    Set cell = Worksheets(1).Range("a1")

    'call the subroutine and pass a cell reference and a text string for a cell note
    CommentTheCell cell, "This is a cell note"
End Sub

Sub CommentExample2()
    Dim cell As Range

    'create a range name to use when calling the routine CommentTheCell
    Set cell = Worksheets(1).Range("a1")

    'call the subroutine, but this time use it to just remove any cell note
    CommentTheCell Worksheets(1).Range("B2"), "none"
End Sub

Sub CommentTheCell(xlCell As Range, strCommentText As String)
    Application.DisplayAlerts = False
    xlCell.ClearNotes
    If LCase(strCommentText) = "none" Then Exit Sub
    xlCell.NoteText Text:=strCommentText
End Sub

9.9.3 Working With Comments

The following code illustrates how to use Visual Basic to add a comment to a cell and then auto size it:

With ActiveSheet.Range("A1")
    .AddComment "This is my meaningful comment"
    .Comment.Shape.TextFrame.AutoSize = True
End With

The following code will auto size all comments on all worksheets in the active workbook:

Dim cmt As Comment, sh As Worksheet

'rotate through all the worksheets
For Each sh In Worksheets

    'rotate through all the comments in the worksheet
For Each cmt In ActiveSheet.Comments

'auto size the comment

    cmt.Shape.TextFrame.AutoSize = True

Next

Next

The next code will auto size any comment in the selection on the active sheet.

Dim cell As Range
On Error Resume Next

'rotate through each cell in the selection

For Each cell In Selection

'if the cell has a comment, resize it. The On Error keeps the code 'from crashing if there is no comment.

    cell.Comment.Shape.TextFrame.AutoSize = True

Next

'turn off error handling

On Error GoTo 0

9.9.4 How To Create Or Append A Comment On A Cell

Sub PromptingForCellComments()
    Dim xlCell As Range
    Dim sCommentText As String
    Dim sNewComment As String

    'set a reference to the active cell

    Set xlCell = ActiveCell
    With xlCell
        On Error Resume Next

    'get the existing comment

        sCommentText = .NoteText
        On Error GoTo 0

    'display the inputbox, setting the default value to the current comment

End Sub
sNewComment = InputBox(_
  prompt:="Please enter a comment. " & _
  "Selecting cancel will erase the comment", _
  default:=sCommentText)

'update the comment

  .NoteText sNewComment
End With
End Sub

### 9.9.5 Deleting Comments

The following code will delete all comments in a worksheet:

**Dim oCmt As Comment**
**For Each oCmt In ActiveSheet.Comments**
  **oCmt.Delete**
**Next Cmt**

And another approach:

**ActiveSheet.Cells.ClearComments**

### 9.9.6 Auto-Sizing Comments

The following code will auto-size all comments on a worksheet:

**Sub AutosizeComments()**
  **Dim cmt As Comment**
  **Dim cell As Range**

  'set on error in case there is not a comment in a cell
  **On Error Resume Next**

  'rotate through all cells in the used range
  **For Each cell In ActiveSheet.UsedRange**

  'get the comment if there is one
  **Set cmt = cell.Comment**

  'if a comment, resize it
  **If Not cmt Is Nothing Then**
    **cmt.Shape.TextFrame.AutoSize = True**
  **End If**
9.10 CELL EXAMPLES

9.10.1 Determining What Is In A Cell

There are a number of ways to determine what is in a cell. The following illustrate their use, using active cell. However, any cell reference or object variable set to refer to a cell can also be used in these statements.

Extract the active cell's value

Dim V

'Store the active cell's value in a variable
V = ActiveCell.Value

'test to see if a value was assigned, quit if not
If V ="" Then Exit Sub

Test to see if the active cell is empty

If IsEmpty(ActiveCell) Then
    MsgBox "the cell is empty"
End If

The following tests to see if the value in the active cell is numeric or can be converted to a number. Please note, that the value can be number if is a number entry, or if it is a formula that evaluates to a number. Also an entry such as '004, which begins with a single quote, will also return True in the following test as it can be converted to the number 4.

If IsNumeric(ActiveCell.Value) Then

    'code to execute if true

End If

You can always use the worksheet functions, which don't do a conversion to numeric if the entry is actually a string.

Application.IsNumber(ActiveCell.Value)

'only returns true if it is actually a number

Application.IsText(ActiveCell.Value)
'returns true if the argument is text.

The following tests to see if the active cell contains a formula. It uses the HasFormula property which returns True of all cells in the range have a formula, False if no cell in the range has a formula, and Null if some cells have formulas.

If ActiveCell.HasFormula Then

'actions to take if the active cell has a formula

End If

The following tests to see if the active cell contains an error value, for example #DIV/0! Or #REF!

If IsError(ActiveCell) Then

'actions to take if an error value

End If

The following returns the type entry in the active cell. For example, it returns "String" if a string entry, "Double" if a numeric entry, and "Boolean" if True or False.

MsgBox TypeName(ActiveCell.Value)

9.10.2 Determining Information About A Cell

Sub InfoExample1()

'This example displays the value of the active cell

    MsgBox ActiveCell.Value
End Sub

Sub InfoExample2()

'This example determines if the active cell is empty

    If IsEmpty(ActiveCell) Then
        MsgBox "The cell is empty"
    Else
        MsgBox "The cell is not empty"
    End If
End Sub

Sub InfoExample3()
'this example determines which cells in a selection are numeric
'both a numeric test and an IsEmpty test is needed because the
'numeric test will return TRUE if a cell is empty.

Dim cell As Range
For Each cell In Selection
If Not IsEmpty(cell) And IsNumeric(cell.Value) Then

'actions to do if true

End If
Next
End Sub

Sub InfoExample4()

'This example returns the text appearance of a cell, exactly as it appears on
'the screen and stores it in a variable called cellText. For example, if the
'actual cell value is 0.123 and the cell is formatted to one decimal, then the
'the following would return 0.1

cellText = ActiveCell.Text
End Sub

Sub InfoExample5()

'This shows how to determine the row and column number of a cell selected
'by some means. For example by displaying an input box for the user to make
'a selection or by using the Find command.
'code that sets the variable someCell to a cell reference

rowNum = someCell.Row
columnNum = someCell.Column
End Sub

Sub InfoExample6()
Dim someCell As Range
Dim sheetThatContainsTheCellRef As Worksheet
Dim workbookThatContainsTheCellRef As Workbook

'This shows how to determine the sheet and workbook of a cell selected by
'some means. For example by displaying an input box for the user to make a
'selection.
'code that sets the variable someCell to a cell reference

sheetThatContainsTheCellRef = someCell.Parent
workbookThatContainsTheCellRef = someCell.Parent.Parent
End Sub

Sub InfoExample7()
Dim someCell As Range
Dim cellFormat As String
Dim cellLeft As Integer, cellTop As Integer

' The following shows how to determine some properties of a cell:
' code that sets the variable someCell to a cell reference

If someCell.Locked Then

' code that runs if the cell is protected

End If

' this stores the cell's format for later use

cellFormat = someCell.NumberFormat

' this stores the cell's position in points. This is useful if one is creating
' buttons or charts on a sheet and want them started at a particular cell
' reference.

cellLeft = someCell.Left
cellTop = someCell.Top
End Sub

9.10.3 Reading And Writing Cell Values Without Switching Sheets

If you do it something like this it will switch sheets

Sub Approach1()
    Sheets("Sheet1").Select
    AnAmount = ActiveSheet.Cells(1, 1).Value
    Sheets("Sheet2").Select
    ActiveSheet.Cells(1, 1) = AnAmount
End Sub

If you do it like this it won't switch sheets, it involves less code, and if you have a lot of such statements, it will run faster.

Sub Approach2()
    Sheets("Sheet2").Cells(1, 1).Value = _
        Sheets("Sheet1").Cells(1, 1).Value
End Sub

9.10.4 Determining If A Cell Is Empty And Problems With IsEmpty

The function IsEmpty returns True if a cell is empty and False if is not:
If IsEmpty(ActiveCell) Then
    MsgBox "The active cell is empty"
Else
    MsgBox "The active cell is not empty"
End If

There are some instances when IsEmpty will not give you the correct results. For example, In cell A1, enter ="". (that's an equal sign and then two double quotes). Then copy and paste special values back into cell A1. The cell is clearly empty. If you run the above code it will tell you that the cell is not empty. An alternate test that overcomes this problem is Len(ActiveCell.Value) = 0:

If Len(ActiveCell.Value) = 0 Then
    MsgBox "The active cell is empty"
Else
    MsgBox "The active cell is NOT empty"
End If

9.10.5 Testing To See If A Cell Is Empty

There are several ways to see if a cell is empty. The following test the active cell. However, you could specify any cell on any worksheet.

If IsEmpty(ActiveCell) Then
    'actions to take if the cell is empty
End If

If ActiveCell.Value = "" Then
    'actions to take if the cell is empty
End If

If a cell can contain spaces but this would qualify as an empty cell in your code, you can use the following test:

If Application.Trim(ActiveCell) = "" Then
    'actions to take if the cell is empty
End If

If you wanted to test to see if the cell is not empty, then use the Not operator in your If statement:

If Not IsEmpty(ActiveCell) Then
    'actions to take if the cell is not empty
9.10.6 Assigning A Value To A Cell

The following code illustrates how to assign a value to a cell, in this case cell A1 of Sheet1

```vba
Dim a
'get value of a from your file here
A = 5
'put it into cell A1 on Sheet1
Worksheets("Sheet1").Range("A1").Value = a
```

9.10.7 Using Visual Basic To Extract Data From Cells

In this example, the user has imported data from another computer system, and each cell is an entry like the following:

Cell A1:

Smith, Henry (123456)

Where both the name of the individual and the individual's code number is in a cell. The code number begins with a left parentheses, and ends with a right parentheses.

What is desired is the following, which retains the original data and also splits it into the employee code and the name:

Cell A1 Cell B1 Cell C1

Smith, Henry (123456) 123456 Smith, Henry

The following is the code that achieves this result. It assumes that the data starts in cell A1 of the worksheet and the first blank defines the end of the data.

```vba
Sub Split()
Dim rngCell As Range
Dim strName As String
Dim OpenParen As Integer
Dim CloseParen As Integer

'define a For..Next loop from cell A1 to the last entry in column A
For Each rngCell In Range("A1", Range("A1").End(xlDown))
```

```vba
```
'store the entry in the rngCell

strName = rngCell.Value

'find the position of the parentheses

OpenParen = InStr(1, strName, "(")
CloseParen = InStr(1, strName, ")")

'extract the number inside the parentheses and write it to the
cell to the right

rngCell.Offset(0, 1).Value = Mid(strName, OpenParen + 1, CloseParen - OpenParen - 1)

'extract the employee name and write it two cells to the right

rngCell.Offset(0, 2).Value = Mid(strName, 1, OpenParen - 2)

'loop until done

Next rngCell
End Sub

9.10.8 Copying Values Without Using PasteSpecial

PasteSpecial is a fairly difficult command to use. Often, the only way to figure it out is to record a macro that uses this command. If all you want is to copy the values in a range, there is a slightly easier way. If you specify code that looks like the following, and the ranges are the same size, then only values will be copied:

destinationCells.Value = SourceCells.Value

The following example illustrates this, and also provides a subroutine that you may want to make part of your library of must have routines:

Sub CopyValuesExample()
Dim rangeToCopy As Range
Dim destCell As Range
On Error Resume Next
With Application

'get the input range, exit if no input

Set rangeToCopy = .InputBox( _
"Select the range whose values will be copied", _
Default:=Selection.Address, Type:=8)
If rangeToCopy Is Nothing Then Exit Sub
'get the destination cell

    Set destCell = .InputBox(_
        "Select the destination", Type:=8)
    If destCell Is Nothing Then Exit Sub
End With
On Error GoTo 0

'call the routine that copies the values

    CopyCellValues rangeToCopy, destCell
End Sub

Sub CopyCellValues(ByVal SourceCells As Range, _
    ByVal destCell As Range)

'make certain destination is a single cell

    Set destCell = destCell.Cells(1)

'resize destination to the same size as the source range

    With SourceCells
        Set destCell = destCell.Resize _
    End With

'set values to be the same

    destCell.Value = SourceCells.Value
End Sub

9.10.9 Checking For Division By Zero

The following code illustrates how to check if a cell's value is division by zero:

    If IsError(Range("A1").Value) Then
        If Range("A1").Value = CVErr(xlErrDiv0) Then
            MsgBox "#DIV/0! error"
        End If
    End If

Other possible values you can use with the CVErr function are: xlErrNA, xlErrName, xlErrNull, xlErrNum, xlErrRef, xlErrValue

9.10.10 Filling A Range With A Formula

The following procedure fills the cells in column D with a formula that sums the cells in column A, B, and C. For example, the formula in D1 would be =SUM(A1:C1) and the formula in D2 would be =SUM(A2:C2). The actual number of rows vary from use to use.
Sub fillFormula()
  Dim myRng As Range
  Dim lastRw As Long

  'get the last row with an entry (could have been done from A1 or B1)
  lastRw = Worksheets("Sheet1").Range("C1").End(xlDown).Row
  With Worksheets("Sheet1").Range("D1")
    .Formula = "=SUM(A1:C1)"
    .AutoFill Destination:=Worksheets("Sheet1") _
      .Range("D1:D" & lastRw&)
  End With
End Sub

The following is another approach

Sub Approach2()
  Dim C As Range
  Set c = ActiveSheet.Range("C1")
  Do While c <> ""
    c.Offset(0, 1).FormulaR1C1 = "=Sum(RC[-3]:RC[-1])"
    Set c = c.Offset(1, 0)
  Loop
End Sub

And still another approach:

Sub Approach3()
  Range("D1", Cells(Application.CountA(_
    ActiveSheet.Columns("C")), "D") _
    .FormulaR1C1 = "=SUM(RC[-3]:RC[-1])"
End Sub

or

Sub Approach4()
  Range("D1", Cells(Application.CountA(_
    ActiveSheet.Columns("C")), "D") _
    .Formula = "=SUM(A1:C1)"
End Sub

The different between the third and fourth approach is that Approach4 uses "Formula" and A1 notation instead of "FormulaR1C1 and R1C1 notation

9.10.11 Changing The Value Of Cells In A Range Based On Each Cell's Value

The following example shows how to operated on each of the selected cells in a range using a For..Next loop. It also shows you how to use Select Case instead of If/Then statements.

Sub Chg_all()
  Dim cell As Range
For Each cell In Selection
    With cell
        Select Case .Value
            Case 1 To 17
                'do this if value between 1 to 17
                .Value = .Value + 54
            Case 18 To 30
                'do this if value between 18 to 30
                .Value = .Value + 24
        End Select
    'note: if the value is between 17 and 18, less than 1 or greater than 30, then the value is not changed
    End With
Next cell
MsgBox "All done!"
End Sub

Note that the With...End With statements are used to avoid typing the word "cell" in front of Value in the above code. Also, there is a period in front of Value to link it back to cell.

9.10.12 Undoing The Last Manual Entry

You can undo the very last manual entry by using the following statement:

Application.Undo

9.10.13 Determining The Number Of Selected Cells

The following procedure displays the number of selected cells. It takes into account that the user could select multiple areas by holding down the control key.

Sub NumberOfCells()
    Dim number As Long
    Dim area As Range
    For Each area In Selection.Areas
        number = number + area.Cells.Count
    Next
    MsgBox number
End Sub

If you used the following, then you would get the count of cells only in the first selection. The Count property applies only to the first area in a selection.
Sub NumberOfCells2()
    MsgBox Selection.Cells.Count
End Sub

9.10.14 How To Determine If A Range Is Empty

One way to determine if a named range on a worksheet is empty would be to use something like the following:

If Application.CountA(Worksheets("Sheet1") .Range("NewData")) = 0 Then
    MsgBox "The new data section is empty"
Else
    MsgBox "There are entries in the new data section."
End If

In the above code, Application.CountA returns the number of cells that have entries.

The following will determine if a named range on the active sheet is empty:

If IsEmpty(Worksheets("Sheet1") .Range("theData")) Then
    MsgBox "The range is empty"
End If

9.10.15 Determining The Number Of Empty Cells In A Range

The following statement uses CountA to return the number of empty cells from the active cell through the next 199 cells (for a total of 200 cells being checked.

MsgBox Application.CountA(Range(ActiveCell, _
    ActiveCell.Offset(199, 0)))

9.10.16 Cell References And Merge Cells

Assume that cells C4 to E7 are merged together. Now, assume you need to refer to the cell 4 cells below the top left cell of the merged cells, which is identified as destCell (for example, destCell for C4:E7 would be C4). The 4th cell below C4 would then be destCell. Offset(4,0) right? Wrong! destCell. Offset(4,0) is C11 to VB! Instead, you have to use destCell. Offset(1,0) to refer to the 4th cell below C4 if C4:C7 is merged. There appears to be no way to refer to cell D8 by use of an offset from C4.

A more logical reference is to use destCell(5,1) to refer to the 4th cell below the first merged cell, which made a little more sense. To refer to D8, use destCell(5,2). Interesting, if one were to use Range("C1").offset(9,0), Visual Basic ignores the merged cells, and returns C10. It appears that the problem only occurs if you use one of the cells in the merged range in an offset reference.

9.10.17 Determining if there are Merged Cells in a Range
The following function will return True if there are merge cells in a range. Just supply it the range to be checked.

Function bMergedCells(anyR As Range) As Boolean
    Dim sheet_has_merged_cells As Variant
    On Error Resume Next
    sheet_has_merged_cells = anyR.MergeCells
    On Error GoTo 0
    If IsNull(sheet_has_merged_cells) Or sheet_has_merged_cells = True Then
        bMergedCells = True
    Else
        bMergedCells = False
    End If
End Function

9.10.18 Determining The Number Of Cells With Entries

The Excel spreadsheet function CountA can be used to determine the number of cells in a selection that has entries. If a value of zero is returned, then all cells in the selection are empty. CountA must be prefixed with Application since it is a worksheet function and not a Visual Basic function.

Dim N As Integer
N = Application.CountA(Selection)

or

Dim N As Integer
N = Application.CountA(Range("A1:B100"))

or

' this example deletes row 1 of sheet1 if it is empty

Dim N As Integer, anyR As Range
Set anyR = Sheets("Sheet1").Rows(1)
If Application.CountA(anyR) = 0 Then anyR.Delete

9.10.19 Modifying Cell Values Based On Two Tests

The following modifies a cell depending on the tests on two other cells:

Sub ModifyACell()

' use an If statement to check cells values

If (Range("A6").Value > 5 And Range("B6").Value <> 1) Then
'if above tests are both true, change the value in C6 to 1

    Range("C6").Value = 1
Else

    'if either test is false, change the value to 0

    Range("C6").Value = 0
End If
End Sub

The following is similar to the above, but it tests the values in column A and B for each cell in the selection, and modifies the cell in column 6 based on the outcome of the test:

Sub ModifyAllCellsInASelection()
    Dim cell As Range
    Dim r As Long

    'rotate through all cells in the selection; its best to select cells in just
    'a single column, and only the cells in rows to be checked

    For Each cell In Selection

        'get the row number

        r = cell.Row

        'use an If statement to check cells values on the row

        If (Cells(r, 1).Value > 5 And _
            Cells(r, 2).Value <> 1) Then

            'if above tests are both true, change the value in C to 1

            Cells(r, 3).Value = 1
        Else

            'if either test is false, change the value in column C to 0

            Cells(r, 3).Value = 0
        End If
    Next
End Sub

9.10.20 Replacing Characters in a String

If you are using Excel 2000 or higher, you can use the Replace function to replace characters in a string. For example:
Sub ReplaceCharacters()
    Dim S As String
    S = "ABCabc"
    S = Replace(S, "B", "_")
    MsgBox S
End Sub

If you have

Option Compare Text

at the top of your module, the above is case insensitive. If you do not, then it is case sensitive.

9.10.21 VBA Code for ALT-ENTER

If you type some text, press ALT-ENTER, and type more text, the entry will appear on two lines in your cell. Try this by typing "Hello" alt-Enter "World. To get the same effect with a macro statement, use CHR(10):

    ActiveCell.Value = "Hello" & Chr(10) & "World"

9.11 SELECTING AND SPECIFYING CELLS

9.11.1 Using Column Letters to Reference Cells

The function Cells(row number, column number) can also take column letters! For example

    Dim X
    X = Cells(1,"C").Value

will return the value of cell "C1".

With this approach, you do not need to get the column number to refer to a cell. You can also use the same approach for columns:

    Columns("C").Select

will select column C.

9.11.2 How To Reference The Selected Cells
The following is how one would reference each cell in a selection on a sheet and perform an operation on those cells:

```vba
Dim cell As Range
For Each cell In Selection

' your macro code goes here, using the variable "cell" which the
' For..Next loop changes to another cell in the cell each time the Next
' statement is encountered

Next
```

The following illustrates using the above approach. It is a simple macro that numbers the cells in the selection 1, 2, and so forth. Try it on a single selection and on a multiple selection (made by holding down the control key as you select ranges).

```vba
Sub NumberCells()
    Dim N As Integer
    Dim cell As Range, I As Integer
    For Each cell In Selection
        I = I + 1
        cell.Value = I
    Next
End Sub
```

### 9.11.3 Specifying Cells Relative To Other Cells

The **Offset** method allows you to specify cells relative to other cells. Its arguments are number of rows and columns, and the values can be positive, negative, or zero. For example, the following sets the range variable cell to the active cell, and then sets cells around it to certain values.

```vba
Dim cell As Range
Set cell = ActiveCell

' set value three cells to the right to 5:
cell.Offset(0, 3).Value = 5

' set value four cells down to 99
cell.Offset(4, 0).Value = 99

' set the value two cells up and one to the left to 12
cell.Offset(-2, -1).Value = 12
```

### 9.11.4 Referring To Cells And Ranges
Sub ReferringExample1()
    Dim myCell As Range

    'To refer to the cell that is the active cell, use ActiveCell
    'the following returns the active cell's value:
    MsgBox ActiveCell.Value

    'this sets a variable to refer to the active cell. Please note that if the active cell
    'changes, the variable still refers to the original cell, not the new active cell:
    Set myCell = ActiveCell
End Sub

Sub ReferringExample2()

    'this shows how to use row and column numbers to refer to a cell
    'code that sets the variables R and C to row and
    'column numbers
    'the following stores the value of the cell that is at this row and
    'column location.
    cellValue = Cells(R, C).Value
End Sub

Sub ReferringExample3()

    'the following shows how to ask the user for a number and then set the
    'value of a cell that is identified by the range name "monthNumber" and
    'located on a sheet called "Ref Info" in the active workbook to this value

    Dim N As Variant
    N = Application.InputBox( _
        prompt:="Please enter the month number", Type:=1)
    If TypeName(N) = "Boolean" Then Exit Sub
    Sheets("Ref Info").Range("monthNumber").Value = N
End Sub

Sub ReferringExample4

    'to refer to each cell in a range, use a For...Next loop and a range variable:
    Dim cell As Range, someRange As Range

    'code that sets someRange to a group of cells

    For Each cell In someRange

        'code that does something using the range variable cell
Sub ReferringExample5()
    Dim someCell As Range

    'to refer to a cell that is offset from a cell that you know, using
    'the Offset method:
    'code that sets the variable someCell to a cell reference
    'this sets the cell three columns to the right to the value 12
    someCell.Offset(0, 3).Value = 12

    'this stores the value of the cell that is two rows up to the variable tempVal
    tempVal = someCell.Offset(-2, 0).Value
End Sub

9.11.5 Using The Offset Function To Specify Cells

The Offset function allows you to specify cells relative to another cell. The basic format is:

    cell reference.Offset(# of rows, # of Columns)

Where the number of rows or columns offset can be positive or negative. The following illustrate its use:

'Change the value of the cell three cells to the right of the active cell

    ActiveCell.Offset(0, 3).Value = 5

'change the label in the cell two rows above the active cell:

    ActiveCell.Offset(-2, 0).Value = "New Label"

If the variable "cell" is a reference to a cell on some sheet, the following stores the value that is two cells down and one to the right in a variable.

    aValue = cell.Offset(2, 1).Value

9.11.6 Use The Offset Method To Specify Cells Relative To Other Cells

The following statement illustrates how to activate a cell that is in one row up from the active cell

    ActiveCell.Offset(-1, 0).Select
The **Offset** method’s arguments are rows offset and columns offset, and the numbers can be positive or negative.

*Any range reference.Offset(rows, columns)*

the cell object can be any range variable. The following are some additional examples:

'the following sets the cell that is three columns to the right of the cell
'referred to as anyCell to the value 3

`anyCell.Offset(0, 3).Value = 5`

'the following changes the range variable cell to the cell three rows up

`Set cell = cell.Offset(-3, 0)`

Remember that you do not need to activate or select a cell to use or alter its values and properties. All you need to do is to refer to it so that Excel knows how to identify it.

### 9.11.7 Scrolling To A Particular Cell

The following two routines show how you can scroll to a particular cell: In these examples, the target cell or range is selected, then the window positioned for viewing.

```vba
Sub scrollTest()
    Range("Z99").Select
    With ActiveWindow
        .ScrollRow = ActiveSheet.Range("Z99").Row - 1
        .ScrollColumn = ActiveSheet.Range("Z99").Column - 1
    End With
End Sub

Sub scrollTest1()
    Dim myRange As Range
    Set myRange = ActiveSheet.Range("C21:L33")
    myRange.Select
    With ActiveWindow
        .ScrollRow = myRange.Rows(1).Row - 1
        .ScrollColumn = myRange.Columns(1).Column - 1
    End With
End Sub
```

### 9.11.8 Controlling Cell Selection And The Scroll Area

The following two statement sets the selection property on the active sheet so that only unlocked cells can be selected:

```vba
ActiveSheet.EnableSelection = xlUnlockedCells
ActiveSheet.Protect Contents:=True
```
The **EnableSelection** property can be set to **xlNoRestrictions**, **xlNoSelection**, or **xlUnlockedCells**. This property takes effect only when the worksheet is protected: **xlNoSelection** prevents any selection on the sheet, **xlUnlockedCells** allows only those cells whose **Locked** property is **False** to be selected, and **xlNoRestrictions** allows any cell to be selected.

You can also change the **EnableSelection** property in the properties window for a sheet, which can be accessed in the VB Editor window.

You can alternatively set the sheet's **ScrollArea** property if you only want to restrict selection to a small rectangular area. This does not require protection to be on. For example:

```
ActiveWorkbook.Worksheets("Sheet1").ScrollArea = "A1:G15"
```

Be aware that the setting the scroll area will prevent you from adding rows/columns unless the scroll area includes entire rows/columns (ex: "A:G" or "1:5"). Even then, you'll only be allowed to insert within the specified area.

### 9.11.9 Selecting A Range For Sorting Or Other Use

If you need to sort a range, and that range can be manually selected by pressing CTL and the * key on the keypad at the same time, you can duplicate that selection behavior in a macro by using the following statements like the following in your code:

```
Range("A1").CurrentRegion.Select
```

or

```
Set rangeToUse = Range("A1").CurrentRegion
```

or

```
Set rangeToUse = _

any cell reference on any sheet.CurrentRegion
```

For example, if your code has set the variable "cell" to refer to a cell somewhere, this will set the variable "rangeToUse" to the current region around that cell.

```
Set rangeToUse = cell.CurrentRegion
```

A cell reference can be **ActiveCell**, **Cells**(row number, column number), **Range**("name or cell address"), etc. If the reference is not on the active sheet, then they can be qualified with the sheet and workbook to fully identify it.
9.11.10 Making Certain That A Selection Consists Of Only A Single Area

If your macro needs the user to make a selection, it is important to verify that it is a single area unless multiple areas can be used by your code. The following counts the number of areas and stops if more than two areas are selected:

'determine if more than one area selected

If Selection.Areas.Count > 1 Then
    MsgBox "Select only a single area. Activity halted."

'stop the macros

End
End If

9.11.11 Counting And Selecting Cells With Certain Characteristics

The following macro will not only count the number of cells that contain formulas, but also select all of the formula cells. In this macro, any cell whose entry begins with an equal sign is considered a formula. The following is written in a long fashion to better illustrate approaches you could use to select cells that meet a certain characteristic.

Sub CountAndSelectFormulaCells()
    Dim R As Integer, rng As Range, cell As Range

    'check each cell in the sheet's used range

    For Each cell In ActiveSheet.UsedRange

        'see if the cell's entry has an equal sign

        If Left(cell.Formula, 1) = "=" Then
            R = R + 1
            If rng Is Nothing Then
                Set rng = cell
            Else
                Set rng = Union(rng, cell)
            End If
        End If

    Next cell
End Sub
MsgBox "There are " & R & " formulas in the worksheet"

'select the cells containing formulas

If R > 0 Then rng.Select
End Sub

The short approach to do the above is the following which uses the `SpecialCells` function. For help on what all `SpecialCells` can select, place the cursor on `SpecialCells` in your module and press F1.

Sub CountAndSelectFormulaCells()
    Dim N As Integer

    'set on error resume next in case there are no matching cells

    On Error Resume Next
    N = ActiveSheet.UsedRange.SpecialCells(xlFormulas).Count
    On Error GoTo 0
    If N > 0 Then
        MsgBox "There are " & N & " formula cells"
        ActiveSheet.UsedRange.SpecialCells(xlFormulas).Select
    Else
        MsgBox "There are no formula cells on the sheet."
    End If
End Sub

The following is another example of selecting cells with certain characteristics. In this example, only cells with a value greater than 100 are selected. Also, the routine that does the work is called as a subroutine. It has one argument, the range of cells to be checked. The called routine changes the range variable to the cells that are greater than 100 if such cells exist, or to "Nothing" if no cells in the range to be checked are greater than 100.

Sub TryOutCellChecker()
    Dim myRange As Range

    'set a range variable to the cells to be checked. This allows the CheckCells
    'routine to change it to cells that are > 100

    Set myRange = Selection

    'call the subroutine and give it the needed cell range

    CheckCells myRange

    'check the range variable to see if it has been assigned a cell range

    If myRange Is Nothing Then
        MsgBox "There were no cells with values > 100"
    Else

143
myRange.Select
   End If
End Sub

Sub CheckCells(anyR As Range)
    Dim rng As Range, cell As Range

    'check each cell in the range passed to the subroutine
    For Each cell In anyR

    'see if the cell's entry is greater than 100
    If Val(cell.Value) > 100 Then
        If rng Is Nothing Then

        'initialize rng with first found cell
            Set rng = cell
        Else

        'expand the found range variable
            Set rng = Union(rng, cell)
        End If
    End If
    Next cell

    'change the range supplied to the matching cells
    'if no matching cells, anyR is set to Nothing as
    'that is the initial value of rng
    Set anyR = rng
End Sub

9.11.12 How To Expand Or Resize A Range:

If your range variable is named Rng and it has already been set, then


will reset the Rng variable to the a range with the same number of rows, and 1 more column to the right.

For example,

Dim Rng As Range
Set Rng = Range("B2:C10")
Rng.Select
At the end of the above code, Rng will be B2:D10.

9.11.13 Resizing Or Expanding A Range

The Resize method resizes a range based on its arguments, which are the number of rows and columns desired for the new range. The arguments are optional. If one is not provided, then the original value is used for the argument value.

Examples:

The following resizes the selection to be one row and three columns wide, starting from the top left cell

\[ \text{Selection.Resize}(1, 3) \]

This example resizes a range variable to be one additional row and column wider.

```vba
Dim anyRange As Range
Set anyRange = Selection
With anyRange
    Set anyRange = .Resize(.Rows.Count + 1, .Columns.Count + 1)
End With
anyRange.Select
```

The following example selects just the data that is below a title row. In this example, the title row is row 1 and the data and title rows start in cell A1.

```vba
Dim dataRange As Range
With Range("A1").CurrentRegion
    Set dataRange = .Offset(1, 0).Resize(.Rows.Count - 1)
End With
dataRange.Select
```

In the above example, CurrentRegion is equivalent to pressing CTL asterisk. The \text{offset}(1,0) statement refers to one cell below A1.

If your range object is Rng, then

```vba
```

will reset the Rng object to the a range with the same number of rows, and 1 more column to the right.

For example,

```vba
Dim Rng As Range
Set Rng = Range("B2:C10")
Rng.Select
```
At the end of the code, Rng will be B2:D10.

9.11.14 Selecting Just Blank Cells

The following statement selects all the blank cells in a selection:

\[ \text{Selection.SpecialCells(xlCellTypeBlanks).Select} \]

Please note that an error will occur if there are no blank cells. You can set an error trap to handle that situation.

9.11.15 Selecting Just Number Cells

A number cell is a cell that has just numbers in it or formulas that consist of just numbers. For example, the entries 4, 5, =12 are obvious all number cells. The entries "=12" and "=2+3" are also number cells. However, Excel does not recognize the last as numbers, but instead wants to view them as formula cells. Thus there is no easy way to select just number cells. If someone tells you that you can do this with the statement

\[ \text{Selection.SpecialCells(xlConstants, xlNumbers).Select} \]

try it on a cell containing "=2+3" or on a cell containing "=12"

It will not work.

The following code shows you how you can select the true number cells in a selection.

```
Sub NumberSelectDemo()
    Dim anyR As Range

    'set a range variable equal to the range to be searched for number cells
    Set anyR = Selection

    'call the routine that searches the range for number cells, passing it the
    'above range variable. The routine will modify the range variable,
    'returning just the number cells.
    Select_Number_Cells anyR

    'select the number cells
    anyR.Select
End Sub
```

```
Sub Select_Number_Cells(rangeToCheck As Range)
    'this routine selects the number cells in the range passed to it,
    'and modifies the range variable to be just those cells.
```
Dim numberCells As Range, cell As Range

' turn off screen updating
Application.ScreenUpdating = False

'restrict the range to just the used range of the sheet containing the range
Set rangeToCheck = Intersect(rangeToCheck, _
    rangeToCheck.Parent.UsedRange)

'use the function below this routine to further restrict the range to
' just cells that are formulas or constants
Set rangeToCheck = Cells_To_Check(rangeToCheck)

' check each cell in the range to see if it contains just numbers and
'math operators, using the function listed below this procedure.
For Each cell In rangeToCheck
    If IsANumber(cell) Then

    ' if the range variable numberCells has not been set to a range,
    ' then set it equal to the first found cell
        If numberCells Is Nothing Then
            Set numberCells = cell
        Else

    ' if numberCells has already been set, expand it with the next cell found
        Set numberCells = Union(numberCells, cell)
    End If
    End If
Next

'display a message if no number cells were found
If numberCells Is Nothing Then
    MsgBox "There are no number cells in the " & _
        "selection. Cells are either blank, " & _
        "contain formulas or text."
End If
Else

'halt further macro activity
    End
Else

' if cells found, change the range passed to this routine
Set rangeToCheck = numberCells
End If
End Sub

Function IsANumber(anyCell) As Boolean
Dim I As Integer, cellText As Variant

'if the cell being checked is empty, then exit the function.
'this returns a default value of False for the function
If IsEmpty(anyCell) Then Exit Function

'do the same if the cell is not numeric
If Not IsNumeric(anyCell) Then Exit Function
cellText = anyCell.Formula

'do the same if the cell contains an error value
If IsError(cellText) Then Exit Function

'check the cell for a letter. If a letter is found, then the cell can not contain
'just numbers and math operators. Exit the function in this case,
giving a False value for the function.
'convert the cell text to upper case for the following comparisons

cellText = UCase(cellText)
For I = 1 To 26
    If InStr(1, cellText, Chr(I + 64), 1) > 0 Then
        Exit Function
    End If
Next

'if the cell passes all the above tests, it must contain just numbers
IsANumber = True
End Function

Function Cells_To_Check(anyRange As Range) As Range

'this function modifies the range passed to it to potential number cells,
'which can be either constants or formula cells (ones that begin

'with an equal sign

Dim constantCellsExist As Boolean
Dim formulaCellsExist As Boolean

'use error traps as the SpecialCells causes an error if no matches are found
On Error GoTo constantTrap
Set Cells_To_Check = anyRange.SpecialCells(xlConstants)
'if no error occurred, then there must be constant cells in the range

constantCellsExist = True

formulaCheck:
  On Error GoTo FormulaTrap
  Set Cells_To_Check = anyRange.SpecialCells(xlFormulas)

'if no error occurred, then there must be formula cells in the range

formulaCellsExist = True

ContinueProcedure:
  On Error GoTo 0

'depending on the results of the above, which sets Boolean variables,

'set the range

If constantCellsExist And formulaCellsExist Then

'if both constants and formulas then do this

  Set Cells_To_Check = Union(anyRange._
   SpecialCells(xlConstants), _
   anyRange.SpecialCells(xlFormulas))
ElseIf constantCellsExist Then

if just constants, do this

  Set Cells_To_Check = anyRange.SpecialCells(xlConstants)
ElseIf formulaCellsExist Then

'if just formula cells, do this

  Set Cells_To_Check = anyRange.SpecialCells(xlFormulas)
Else

'if no qualifying cells, then display a message and halt macro activity

  MsgBox "There are no numeric cells in the " & _
  "selection. Activity halted."
End
End If
Exit Function

constantTrap:
  Resume formulaCheck

FormulaTrap:
  Resume ContinueProcedure
End Function
9.11.16 Setting Number Cells to Zero

The following example will set all number (input) cells on a worksheet to zero:

Sub ClearNumbers()
    Dim numCells As Range
    Dim formulaCells As Range
    Dim cellsToSetToZero As Range

    'on error needed in case no qualifying cells exist
    On Error Resume Next

    'get numeric cells without equal sign at start
    Set numCells = _
        ActiveSheet.Cells.SpecialCells(xlConstants, xlNumbers)

    'get all cells with an equal sign at start
    Set formulaCells = _
        ActiveSheet.Cells.SpecialCells(xlCellTypeFormulas, 1)

    On Error GoTo 0

    'exit if no matching cells
    If numCells Is Nothing Then
        If formulaCells Is Nothing Then
            Exit Sub
        Else
            Set inputcells = formulaCells
        End If
    ElseIf formulaCells Is Nothing Then
        Set inputcells = numCells
    Else
        Set inputcells = Union(numCells, formulaCells)
    End If

    'check for entries w/o a, b, c... which would be a non
    'input cell
    For Each cell In inputcells
        If Not cell.Formula Like "*[A-Za-z]*" Then
            If cellsToSetToZero Is Nothing Then
                Set cellsToSetToZero = cell
            End If
        End If
    Next cell
End Sub
Else
    Set cellsToSetToZero = Union(cell, cellsToSetToZero)
End If
End If

Next
'if no input cells, exit

If cellsToSetToZero Is Nothing Then Exit Sub

' set input cells to zero

cellsToSetToZero.Value = 0
End Sub

9.11.17 Selecting The Current Region

If you place the cell pointer among a group of cells containing entries and press CTL and Asterisk at the same time, Excel will select what is called the CurrentRegion. The following statement does the same thing in Visual Basic.

    ActiveCell.CurrentRegion.Select

You can also assign the current region of a sheet to an object variable without first selecting it or activating the sheet:

    Dim curReg As Range
    Set curReg = Sheets("Sheet1").CurrentRegion

9.11.18 Using the Used Range Property In Your Code

The UsedRange property of a worksheet refers to the "the range of the worksheet that is used". For example, the following statement will select the UsedRange on the active worksheet:

    ActiveWorksheet.UsedRange.Select

in Excel 5 and 7, if you delete rows or columns from the used range, Visual Basic does not recognize that the used range has changed. To reset the used range in Excel 5 and 7, you need to save the file.

You can also refer to the used range on another worksheet and set it to a variable for later use:

    Dim usedR As Range
    Set usedR = Workbooks("Book1.Xls").Sheets("Sheet1").UsedRange

Once important use of the UsedRange property is to restrict the selection to just cells in the used range. This is very important if the user can select an entire column or row and you must test or take action each cell in the selection:
Dim rangeToCheck As Range
Set rangeToCheck = Intersect(Selection, _
    ActiveSheet.UsedRange)

The above restricts the range to check to just cells in the used range, even if an entire column or row is selected.

### 9.11.19 Resetting the Used Range

**UsedRange** does return a range object to the **UsedRange** when used like

```
Set SomeRange = ActiveSheet.UsedRange
```

But calling it alone,

```
ActiveSheet.UsedRange
```

will force Excel to reset the range to the "real" used range.

### 9.11.20 Selecting The Used Range On A Sheet

The simplest way to select the used range on a sheet is to use the **UsedRange** function.

```
Sub UsedRangeExample1()
    ActiveSheet.UsedRange.Select
End Sub
```

or

```
Sub UsedRangeExample2()
    Dim tempR As Range

    'set a variable to the active sheet's used range
    Set tempR = ActiveSheet.UsedRange

    'statements that use tempR

End Sub
```

If the sheet whose used range is needed is not the active sheet, then you can use the following approach:

```
Sub UsedRangeExample3()
    Dim oSheet As Worksheet
    Dim oSheetUsedRange As Range

    'create a variable that refers to another sheet
```
Set oSheet = Workbooks("MYBOOK.XLS").Worksheets("Sheet1")

'quality the UsedRange function with the variable
Set oSheetUsedRange = oSheet.UsedRange

'statesments that use OSheetUsedRange
End Sub

One of the problems with the UsedRange function is that in Excel 5 and 7 it will return cells that don't have entries in the far right columns or the bottom rows. This is because Excel remembers that these rows or columns were once in use although you may have shrunk your sheet via deleting rows or columns. Saving the workbook will reset the used range, however this often is not desirable.

The following example overcomes this problem by using a function that checks for the last row and column with entries and then returns the range from cell A1 to the intersection of the last row and column.

Sub UsedRangeExample4()
    Dim tempRange As Range
    Set tempRange = RangeToUse(ActiveSheet)
    tempRange.Select
End Sub

Function RangeToUse(anySheet As Worksheet) As Range

    'this function returns the range from cells A1 to cell which is the intersection of the last row with an entry and the last column with an entry.

Dim I As Integer, c As Integer, r As Integer
With anySheet.UsedRange
    I = .Cells(.Cells.Count).Column + 1
    For c = I To 1 Step -1
        If Application.CountA(anySheet.Columns(c)) > 0 Then Exit For
    Next
    I = .Cells(.Cells.Count).Row + 1
    For r = I To 1 Step -1
        If Application.CountA(anySheet.Rows(r)) > 0 Then Exit For
    Next
End With
With anySheet
    Set RangeToUse = .Range(.Cells(1, 1), .Cells(r, c))
End With
End Function

9.11.21 Restricting A Selection To The Cells In The Sheet's Used Range
Sometimes your code may need to take some action on every cell that the user selects. For example, check each cell's value or modify each cell if it has an entry. However, it is very easy for the user to select an entire row or column. If a column, then your code will check many thousands of cells instead of the few that are in the sheet's used range. The following two examples illustrate ways to do this:

Example - if selection is always on the active sheet:

```vba
Dim rangeToChk As Range

' restrict the range to the intersect of the selection and the active  
'sheet's used range
Set rangeToChk = Intersect(Selection, ActiveSheet.UsedRange)

' code that uses the range variable rangeToChk, for example:
Dim cell As Range
For Each cell In rangeToCheck

' do something to each cell or check cell and do something

Next
```

Example 2 - if the selection can be on any sheet in any workbook

```vba
Dim anyRange As Range

' get a range from the user via an input box
On Error Resume Next
Set anyRange = Application.InputBox(  _
    prompt:="Select cells for processing", _
    Type:=8, default:=Selection.Address)  
On Error GoTo 0

' if no range selected, stop
If anyRange Is Nothing Then Exit Sub

' restrict the range to the intersect of anyRange and the used range  
'on the sheet containing anyRange
Set anyRange = Intersect(anyRange, anyRange.Parent.UsedRange)

' code that uses the range variable anyRange, for example:
Dim cell As Range
For Each cell In anyRange
```
'do something to each cell or check cell and do something

Next

9.11.22 Using The Intersect Method With Ranges

Once important use of the Intersect method is to restrict the selection to just cells in the used range. This is very important if the user can select an entire column or row and you must test or take action each cell in the selection:

Dim rangeToCheck As Range
Set rangeToCheck = Intersect(Selection, _
              ActiveSheet.UsedRange)

You can also use the Insert method to determine if a cell is within a range:

If Not Intersect(ActiveCell, Range("A1:G5")) Is Nothing Then

' action to take if cell is in the range

End If

or

If Intersect(ActiveCell, Range("A1:G5")) Is Nothing Then

' action to take if cell is not in the range

End If

9.11.23 Getting The Intersection Of Two Ranges

Assume that you have a table in Excel that looks like the following:

Sales Person Territory Main Customer

Bill Southwest K-Mart

Linda Northeast U-Haul

John West Coast Disney Land

And, that you have assigned the following range names:

TheTable the cells containing all the above

SalesPerson the cell containing the words "Sales Person"
Territory the cell containing the word "Territory"

Customer The cell containing the words "Main Customer"

Assuming that you are on the sheet containing the above information, the following assigns the range containing just the entries in the territory column but excluding the label "Territory" to a range variable and selects this range:

Sub IntersectExample1()
    Dim rTable As Range, rTemp As Range
    Dim rTerritoryList As Range
    Set rTable = Range("TheTable")

    'specify the intersection of the column containing the range name and
    'the table. '
    Set rTemp = Intersect(Range("Territory").EntireColumn, rTable)

    'set a variable to the range which is from the second cell of the
    'intersection to the last cell

    'select the range
    rTerritoryList.Select
End Sub

The following does the same thing, but uses the With statement to avoid repeating the references and also minimized the range variables. Notice that there are periods in front of the word Cells.

Sub IntersectExample2()
    Dim rTerritoryList As Range

    'specify the intersection of the column containing the range name and
    'the table. A "_" is used to continue the statement onto the next line
    With Intersect(Range("Territory").EntireColumn, _
        Range("TheTable"))

    'set a variable to the range which is from the second cell of the
    'intersection to the last cell
    'Notice that there are periods in front of the word Cells.
    Set rTerritoryList = Range(.Cells(2), _
        .Cells(.Cells.Count))

End With
'select the range

    rTerritoryList.Select
End Sub

The following illustrates how to do the same task, except that the active sheet is not the sheet containing the ranges. Also, this example displays the values of the individual cells instead of selecting the range:

Sub IntersectExample3()
    Dim rTerritoryList As Range, cell As Range
    Dim WS As Worksheet

    'set a variable to refer to the sheet containing the range names
    Set WS = Workbooks("MYBOOK.XLS").Sheets("sheet1")

    'qualify ALL the Range statements with the sheet reference
    With Intersect(WS.Range("Territory").EntireColumn, WS.Range("TheTable"))

        'set a variable to the range which is from the second cell of the intersection to the last cell
        'Notice that there are periods in front of the word Cells.
        Set rTerritoryList = WS.Range(.Cells(2), _
            .Cells(.Cells.Count))
    End With

    'use a For Each..Next loop to display each cell's value
    'note that cell is a range name and not a keyword to Visual Basic
    For Each cell In rTerritoryList
        MsgBox cell.Value
    Next
End Sub

If you have to do tasks like the above repeatedly, then you can write sub routines that do the task and return the range in question. In this example, we've colored the sub routine names red for easy reference

Sub IntersectExample4()
    Dim r Territories As Range, r Customers As Range

    'call subroutine to get the range. Note that this statement supplies
    'the empty range r Territory, and that the subroutine returns a range in it.
    'Also, the names of the book, sheet, and ranges are supplied to the routine
    Get_Range "MYBOOK.XLS", "sheet1", "Territory", _

"TheTable", rTerritories

'display each cell's value. In a real world application, action could be 'taken on each cell in the range

Display_Info rTerritories

'repeat again for the customer information

Get_Range "MYBOOK.XLS", "sheet1", "Customer", _

"TheTable", rCustomers

Display_Info rCustomers

End Sub

Sub Get_Range(wbName As String, wsName As String, _
    cellRangeName As String, bigRange As String,
    rangeToReturn As Range)
    Dim WS As Worksheet

    'set a variable to refer to the sheet containing the range names
    Set WS = Workbooks(wbName).Sheets(wsName)

    'qualify ALL the Range statements with the sheet reference
    With Intersect(WS.Range(cellRangeName).EntireColumn,
        WS.Range(bigRange))
        'set variable to the range which is from the second cell of the
        'intersection to the last cell
        'Notice that there are periods in front of the word Cells, which
        'cause them to refer back to the With object above
        'Also, this changes the range in the calling routine's variable
        Set rangeToReturn = WS.Range(.Cells(2), _
            .Cells(.Cells.Count))
    End With
End Sub

Sub Display_Info (anyRange As Range)

    Dim cell As Range
    For Each cell In anyRange
        MsgBox cell.Value
    Next
End Sub
**9.11.24 Union Method Problem**

The **Union** method does not do a truly logical "union". For example.......

The range A1:B10 is 20 cells.

The range B10:C19 is 20 cells.

There is a single cell, B10, common to both ranges.

The "logical" (and "Logical") definition of the union of these two ranges would be a range of 39 cells (20 cells in A1:B10 plus 20 cells in B10:C19 minus the 1 cell common to the two sets). However, **Union** returns a range of 40 cells:

```vba
Dim uRange As Range
Set uRange = Union(Range("A1:B10"), Range("B10:C19"))
MsgBox uRange.Cells.Count
```

returns 40 cells, not 39.

This "oddity" manifests itself with "**For Each**" loops, too.

However, the problem is slightly worse than what is described, as it is not consistent. The above does not appear to apply to single row and single column ranges. For example, the Union of A1:A5 and A5:A10 returns 10 cells, the number of cells in the logical union.

The work around is to do an intersection of the ranges, and then check each cell as you process it to see if it is already in the union and exclude it if it is – if there is a chance that the union of two ranges overlap. The following illustrates this

```vba
Dim tRange As Range, uRange As Range
Dim cell As Range

' do a normal Union into a temporary range variable
Set tRange = Union(Range("A1:b10"), Range("b10:c19"))

' initialize the Union range To the first cell In the above range
Set uRange = tRange.Cells(1)

'turn on error handling
On Error Resume Next
For Each cell In tRange

' see if the cell is already in the Union range
```
iRange = Intersect(cell, uRange)

'If it is not, add it

If iRange Is Nothing Then
    Set uRange = Union(cell, uRange)
Next
On Error GoTo 0
MsgBox uRange.Cells.Count

Please note that you end up with multiple range areas instead of one area.

9.11.25 Limiting Access To Cells

You can use the scroll area method to limit access to certain rows and columns:

Sub RestrictScrollArea()
    With ThisWorkbook.WorkSheets("Sheet1")
        .ScrollArea = .UsedRange.Address
    End With
End Sub

This routine restricts the users from scrolling outside the used area on sheet 1. It restricts both rows & columns. If you have data on the sheet that you do not want the users to see, then change ".UsedRange.Address" to the address you want (ex: "A1:M80" - including the quotes).

9.11.26 Hiding The Cursor Frame / Preventing Cell Selection

You can hide the cursor and prevent cell selection very easily. Simply set the worksheet property EnableSelection to xlNoSelection and turn protection on:

With Worksheets(1)
    .EnableSelection = xlNoSelection
    .Protect Contents:=True, UserInterfaceOnly:=True
End With

9.11.27 Preventing Cell Drag And Drop

To prevent a user from dragging an unprotected cell to another unprotected shell on a protected sheet use:

Application.CellDragAndDrop = False

9.11.28 Using The Merge Command In Your Code
You can merge a range of cells with statements like the following

```
Range("C3:E3").Merge
```

To merge the active cell and the two cells to the right, use

```
Range(ActiveSheet.Range(ActiveCell).Offset(0, 2)).Merge
```

To assign a value to merged cells or retrieve a value from merged cells, reference the left most cell in the range. Use the `UnMerge` method to undo merged cells.

**9.11.29 VBA and Validation List**

Unfortunately, if a cell is changed by using a validation list, the `Private Sub Worksheet_Change(ByVal Target As Excel.Range)` does not run. This has been corrected in Excel 2000. Meanwhile, if you're stuck in Excel 97 there really aren't a lot of good options.

The best option is to put some kind of inherently volatile formula in a hidden cell on the worksheet (=RAND(), for instance) and then use the calculate event to signal changes in the validation lists.

Of course a million other things besides selecting an item from the validation list will cause the calculate event to fire, so you have to write code to store the old value from the list and compare it to the current value to see if the dropdown list really changed. Definitely a pain.

**9.12 DETERMINING IF A RANGE IS IN ANOTHER RANGE**

**9.12.1 Determining If A Selection Is Within A Named Range**

The following code determines if a selection is within a named range. It will return true if any part of the selection is within a range

```
Sub Determine_If_Selection_Is_In_A_Range()
    Dim testRange As Range
    Dim nameRange As Range
    Dim nm

    'set a range variable equal to the selected range
    Set testRange = Selection

    'set an error trap in case the name is not assigned to a range or
    'the name is on a different sheet
    On Error GoTo errorTrap
    For Each nm In Application.Names
        `
'get the range assigned to the name

Set nameRange = Range(Mid(nm.Refersto, 2))
If Not nameRange Is Nothing Then

'see if the selected range is in the named range

If Union(nameRange, testRange).Address = _
   nameRange.Address Then
   MsgBox "The selected range is " & _
   "contained in " & nm.Name
   Exit Sub
End If
End If

Next nm

'if no match found above, then display this message

MsgBox "There is no range name that contains" & _
   " any part of the selected range"
Exit Sub

errorTrap:

'this handles an error when a named range is not valid or on a different sheet

Resume nextName
End Sub

9.12.2 Determining If A Range Is Within A Specific Range

The following code illustrates how to determine if a range (or any part of a range) is within another range.

Dim addr As Range

'set a range variable to the selection

Set addr = Selection

'check the intersect of the two ranges

If Application.Intersect(addr, _
   Range("B2:D5")) Is Nothing Then
   MsgBox "No Intersection"
Else
   MsgBox "Intersection"
End If

The following illustrates how to do this using a function that returns True or False. Also, this function will return True only if the entire selection is within the allowed range. Note that the
default value of the function is False, and this value is returned if the function is not explicitly set to True

Function InRange(rng As Range) As Boolean
  Dim tempR As Range

  'get the intersection of the supplied area and the range
  Set tempR = Application.Intersect(rng, _
    Range("B2:D5"))
  If tempR Is Nothing Then Exit Function

  'check to see if the address of the range supplied and the result of 'the intersection is the same. If they are then return True
  If rng.Address = tempR.Address Then InRange = True
End Function

Sub Test()

  'this illustrates how to use the above function
  Dim anyR As Range
  Set anyR = Selection
  If InRange(anyR) Then
    MsgBox "In range"
  Else
    MsgBox "Not in range"
  End If
End Sub

9.12.3 Determining If A Cell Is Within A Certain Range

If you know the row and column number of a cell, and want to test to see if it is within a certain range, then you can do so with code like the following:

If Union(Cells(row, col), Range("Range")).Address = _
  Range("Range").Address Then
  MsgBox "The cell is contained in the range"
Else
  MsgBox "The cell is not contained in the range"
End If

Please note that the above assumes that the cell and range name are on the active worksheet. If not, you need to qualify Cells and Range with the worksheet.

9.12.4 Determining When A Cell Is Within A Range

The easiest way to determine if a cell is within a range is to use the Intersect method. For example:
Dim TempR As Range

'initialize TempR to nothing in case it has been set to a range by prior code
Set TempR = Nothing

'set on error so the following Set statement does not display an error
'message if the range name does not exist
On Error Resume Next

'note that this test assumes that the range is on the ActiveSheet
Set TempR = Intersect(ActiveCell, Range("SomeName"))

'turn off error trapping
On Error GoTo 0
If TempR Is Nothing Then

'action to take if ActiveCell is not in the range
Else

'action to take if ActiveCell is within the range
End If

The following is a more complex example. In this example, the user wants to determine when the user selects a cell in any one of five ranges, and take action based on the range. In addition, if the user selects a second cell in a given range, then no action would be taken. I.e., action is taken only when the user first selects a cell in a given range. For simplicity in this example, the ranges are columns A, B, C, D, and E.

The following code goes into the worksheet code sheet

'declare this at the top of the module
Dim PrevRange As Integer

Private Sub Worksheet_SelectionChange(_
    ByVal Target As Excel.Range)
    Dim RangeList As Variant
    Dim I As Integer

    'assign list of array names to a variant variable
    RangeList = Array("A:A", "B:B", "C:C", "D:D", "E:E")
'check each range to see if the ActiveCell is within the range
'note that the array index begins at 0 and ends at 4, not 1 and 5.

For I = LBound(RangeList) To UBound(RangeList)
    If Not Intersect(ActiveCell, Range(RangeList(I))) _
        Is Nothing Then

'if the range is not the same as the last range then display a
'message based on the range

    If PrevRange <> I Then
        Select Case I
            Case 0: MsgBox 0
            Case 1: MsgBox 1
            Case 2: MsgBox 2
            Case 3: MsgBox 3
            Case 4: MsgBox 4
        End Select
    End If
Exit For
End If
Next I

'update the last range with an intersection

PrevRange = I

End Sub

9.12.5 Determining If One Range Is Within Another

The following function will compare two ranges to see if the first range is within the second range.

Function bR1_inside_R2(rang1 As Range, _
                        rang2 As Range) As Boolean
    Dim tempR As Range
    On Error Resume Next
    Set tempR = Intersect(rang1, rang2)
    If tempR Is Nothing Then Exit Function
    If tempR.Address = rang1.Address Then bR1_inside_R2 = True
End Function

Sub demo_bR1_inside_R2()
    MsgBox bR1_inside_R2(Range("a1:a10"), Range("a1:b10"))
End Sub

9.12.6 How To Determine If The ActiveCell Is Within A Named Range

The following function returns True if the cell object passed to the function is within the range name passed to it.
Function bInName(anyCell As Range, anyName As String, _
  oBook As Workbook) As Boolean
  Dim nM

  'rotate through each name in the workbook that is passed to the function
  For Each nM In oBook.Names

    'check for a matching name
    If LCase(nM.Name) = LCase(anyName) Then

      'if a match is found, see if the cell is in the range
      If Not Intersect(anyCell, _
        Range(Mid(nM.Value, 2))) Is Nothing Then

        'set function to True if found. Default value is False
        bInName = True
        Exit If

    End If

  Next
  End Function

The following subroutine shows how to use the above function to find out if the cell is within the range name "CurrentYear". Please note you can set the cell and workbook to any cell on any open workbook. ActiveCell and ActiveWorkbook were used for convenience.

Sub Test()
  Dim cell As Range, wBook As Workbook
  Set cell = ActiveCell
  Set wBook = ActiveWorkbook
  MsgBox bInName(cell, "CurrentYear", wBook)
End Sub

The following example determines which named range the active cell is within.

Sub ChkNamesForRange()

  Dim nM, bFound As Boolean

  'Rotate through all the names in the workbook
  For Each nM In ActiveWorkbook.Names
'restrict testing to range names and to names on the ActiveSheet
' the Value of a range name is the range it refers to,
'beginning with an equal sign. Ex: =sheet1!$A$1:$C$3

If InStr(nM.Value, "$") > 0 And _
   InStr(nM.Value, ActiveSheet.Name) > 0 Then
   If Not Intersect(ActiveCell, _
      Range(Mid(nM.Value, 2 ))) Is Nothing Then

   'If a match is found set Boolean flag and exit For

      bFound = True
   Exit For
   End If
End If
Next nM

'display a message based on the value of the Boolean flag

If bFound Then
   MsgBox "The active cell is within range name " & nM.Name
Else
   MsgBox "The active cell is not within a named range"
End If
End Sub

9.12.7 A Function That Determines If A Range Is Within
Another Range

The following function returns True if range 1 is within range 2, and False if it is not. The
arguments must be range references, not text strings.

Function IsInRange(Rg1 As Range, Rg2 As Range) As Boolean
   If Union(Rg1, Rg2).Address = Rg2.Address Then _
      IsInRange = True
End Function

The function returns False, its default value, if the two addresses are different.

The following illustrates how to use the above function.

Sub Test_Function()
   With Worksheets("sheet1")
      MsgBox (IsInRg(.Range("C5"), .Range("A1:D10")))
   End With
End Sub

9.13 WORKING WITH RANGE NAMES

9.13.1 Working With Range Names
If you have a range name in a worksheet and want to use it in your code, Excel will only recognize the range name if you are on the sheet containing the range name, or if you qualify the Range reference with the worksheet.

For example, if range name "MyRange" is on worksheet "Sheet12", and you are on "Sheet3" when you run the following statement, the code will crash:

```vba
Range("MyRange").Copy
```

The solution is very simple, but not obvious. The following approaches will work:

**Approach 1**

```vba
Range("MyRange").Worksheet.Activate
Range("MyRange").Copy
```

**Approach 2** - which does not require you to go to the worksheet

```vba
Range("myRange").Worksheet.Range("myrange").Copy
```

If all you want to do is to determine the worksheet containing the range, then you can do the following:

```vba
Dim oSheet As Worksheet
Set oSheet = Range("myRange").Worksheet
MsgBox oSheet.Name
```

Please note that the above approaches require that the active workbook be the workbook containing the range name, and that the range name is not a local sheet specific range name.

### 9.13.2 Creating Range Names

You can create a range name very simply:

```vba
Range("C4:R9").Name = "SomeName"
```

or

```vba
Set tempRange = Range("C4:R9")
tempRange.Name = "SomeName"
```

### 9.13.3 Creating Hidden Range Names

The following illustrates how to create a hidden range name:

```vba
ActiveWorkbook.Names.Add Name:="hiddenName", _
    RefersTo:="=1000", Visible:=False
```

Hidden range names are useful to store settings that are not visible to the user.

### 9.13.4 Referring To A Range Name In Your Code
Use of range names can make your work in Visual Basic easier. However, to get the full benefit of range names, you should use them to set a range variable to the range name's range. This is easy to do if the active sheet is the sheet containing the range name. For example, if the range name is "MyRange" and it refers to cells on the active sheet, then the following will work:

Dim MyVariable As Range

Set MyVariable = Range("MyRange")

However, the above will crash if the range name "MyRange" is not on the active sheet. The following is a workaround you can use:

If in the active workbook:

Dim myVariable As Range
Set myVariable = Names("MyRange").RefersToRange

If in a different workbook:

Dim myVariable As Range
Set myVariable = _
    Workbooks("book3.xls").Names("MyRange").RefersToRange

If in a different workbook

Sub GoToARange()
    Dim myRange As Range

    'set a variable to a range name's range by calling a user defined function
    Set myRange = RangeNameToRange(Workbooks("book3"), "myRange")

    'got to the range
    Application.Goto myRange
End Sub

Function RangeNameToRange(wBook As Workbook, _
    rName As String) As Range
    Dim I As Integer, J As Integer,
    Dim sName As String, tempStr As String

    'get full address excluding = sign
    tempStr = Mid(wBook.Names(rName).RefersTo, 2)
    I = InStr(tempStr, "'!")
    If I = 0 Then
        I = InStr(tempStr, ":")
    Else
        J = 1
    End If
'get only the sheet name in the address

sName = Mid(tempStr , 1 + J, I - J - 1)

'use the pieces to return a range reference

Set RangeNameToRange = wBook.Sheets(sName).Range(rName)

End Function

9.13.5 How To Refer To Range Names In Your Code

If you need to refer to ranges in your workbook, then you can use statements like the following:

If the range name is on the active sheet, then the following refers to the range:

    Range("range name")

If the range name is on a different sheet in the active workbook, then use statements like the following:

    Sheets("sheet name").Range("range name")
    or Range("'sheet name'!range name")

If the range name is not in the active workbook, then use a statement like the following:

    Workbooks("name").Sheets("sheet name").Range("range name")

If you have object variables set that refer to the workbook or sheet, you can use the object variables instead of the Workbooks("name") and Sheets("sheet name") statements.

9.13.6 Check For Existence Of A Range Name

The following function returns True if the range name exists and is not a local (sheet specify) range name, False if it has note.

Function NameExists(theName As String) As Boolean
    Dim S As String
    On Error GoTo EndFunction
    S = ThisWorkbook.Names(theName).RefersTo
    NameExists = True
    Exit Function
End Function:
NameExists = False
End Function

9.13.7 Determining If A Range Has Been Assigned A Range Name
The following function will return **True** if the range passed to it has been assigned a range name. It will return **False** if is not a named range.

**Function** bNamed(anyRange As Range) As Boolean

Dim nm
Dim nameRange

' set error trap in case name is not for a range
On Error GoTo errorTrap
For Each nm In anyRange.Parent.Parent.Names

' set a variable equal to the range of the name
Set nameRange = Range(Mid(nm.RefersTo, 2))

' compare complete addresses, including sheet name
If nameRange.Address(external:=True) = anyRange.Address(external:=True) Then

' if a match, set function to true and exit
    bNamed = True
    Exit Function
End If
nextName:
Next nm

' if no match found function exits and returns its default value of false
Exit Function
errorTrap:

' resets error trap and returns to process the next name
Resume nextName
End Function

For example, the following is one way of using this function:

**Sub** TestFunction()
    MsgBox bNamed(Selection)
End Sub

9.13.8 Determining The Name Assigned To A Cell

The following statement will return the name assigned to the cell begin referenced, in this case the active cell

MsgBox ActiveCell.Name.Name
ActiveCell.Name returns a name object, and then Name returns its name property.

9.13.9 Expanding A Range Name's Range

The following statement illustrates how to expand a named range to include new data:

Range("myName").CurrentRegion.Name = "myName"

9.13.10 Accessing A Named Range's Value In Another Workbook

To determine the value of a cell that is a named range in another open workbook, use a statement like the following:

cellValue = Workbooks("myworkbook.xls").Worksheets("mysheet") _
.Range("mycell").Value

To set the value of such a cell, use a statement like this one:

Workbooks("myworkbook.xls").Worksheets("mysheet") _
.Range("mycell").Value = "XXX"

9.13.11 Deleting Range Names

Every now and then, a workbook will end up with range names that are bad. I.e., they refer to #REF, as the sheet or range they refer to have been deleted. The following code will delete such names:

Sub DeleteBadNames()

Dim nm As Variant
Dim vTest As Variant

'rotate through all the names in the workbook
For Each nm In ActiveWorkbook.Names

'reset vTest each time through
    vTest = Empty
    On Error Resume Next

'evaluate the name reference
    vTest = Application.Evaluate(nm.Refersto)
    On Error GoTo 0

'if an error, delete the name
Next

End Sub
If TypeName(vTest) = "Error" Then nm.Delete
Next nm
End Sub

If you just want to delete all the range names in a workbook, try the following:

Sub DeleteAllNames()
    Dim I As Integer
    For I = ActiveWorkbook.Names.Count To 1 Step -1
        Range(ActiveWorkbook.Names(I).Name).Name.Delete
        Next
    End Sub

9.13.12 Deleting Range Names - Another Example

You can delete all the names in a workbook like this:

Sub DeleteAllNames()
    Dim Nm As Name

    'loop through the names in the workbook, deleting each
    For Each Nm In Names
        Nm.Delete
        Next
    End Sub

or just the ones that refer to items on the active sheet like this:

Sub DeleteActivesheetNames()
    Dim Nm As Name, SheetLen As Integer

    'get the length of the sheet name plus one for the = sign
    SheetLen = Len(ActiveSheet.Name) + 1
    For Each Nm In Names

        compare the sheet part of the name to the active SheetName
        If Left(nm.RefersTo, SheetLen) = "=" & ActiveSheet.Name Then
            Nm.Delete
        End If
        Next
    End Sub

9.13.13 Deleting All The Range Names In A Workbook

The following should delete all the range names in the active workbook.

Sub DeleteAllNames()
For i = ActiveWorkbook.Names.Count To 1 Step -1
    Range(ActiveWorkbook.Names(i).Name).Name.Delete
Next
End Sub

9.13.14 Deleting Bad Range Names With A Macro

The following code will delete all range names in a workbook that no longer refer to a valid range. An example would be a range name that refers to #REF!$A$!:$P$266. Such range name problems sometimes occur when sheets or ranges are deleted.

Sub DeleteBadNames()
    Dim nm As Excel.Name
    Dim vTest As Variant

    'loop through all the names in the active workbook
    For Each nm In ActiveWorkbook.Names

        'clear vTest of any value or reference
        vTest = Empty

        'use On Error to prevent the evaluation to an error value from halting the macro
        On Error Resume Next
        vTest = Application.Evaluate(nm.ReferTo)
        On Error GoTo 0

        'test the value of the range and if an error, delete the name
        If TypeName(vTest) = "Error" Then nm.Delete
    Next
End Sub

9.14 SORTING DATA

9.14.1 A Simple Sort Example

If you record a macro of clicking the A-Z button to sort your data, you will get a macro much like the following, where the data range is A1:H15 and the active cell when the button was clicked is B7:

Range("A1:H15").Sort Key1:=Range("B7"), _
    Order1:=xlAscending, _
    Header:=xlGuess, _
    OrderCustom:=1, MatchCase:=False, _
Orientation:=xlTopToBottom, _
DataOption1:=xlSortTextAsNumbers

Excel has sorted what is called the current region. This is the region that is selected if you first select a single cell and then press CTL * (Control and asterisk). It is all "connected cells, not separated by blank rows and columns.

The following is a modification of the above to be more useful, as one often expands a data range and may want to sort on a different cell.

ActiveCell.CurrentRegion.Sort Key1:=ActiveCell, _
   Order1:=xlAscending, _
   Header:=xlYes, _
   OrderCustom:=1, MatchCase:=False, _
       Orientation:=xlTopToBottom, _
   DataOption1:=xlSortTextAsNumbers

In the above, the region is now dynamic and set by ActiveCell.CurrentRegion. The sort is based on the active cell. And the header option has been set to xlYes versus xlGuess.

9.14.2 A Complex Data Sort Example

Quite often one wants to sort all the data on a worksheet and wants to sort on multiple columns. In this example, the data range starts at row 4 as there are three header rows. The trick in this is to first set a variable to the data range and then do the sort.

The first data cell is thus

    Dim firstCell As Range
    Set firstCell = Range("A4")

To find the last data cell,

    Dim lastCell As Range
    With ActiveSheet.UsedRange
        Set lastcell = .Cells(.Cells.Count)
    End With

In the above code, note that there is a period in front of Cells. This means it is referring to the the cells in ActiveSheet.UsedRange.

To specify a variable to refer to the data cells:

    Dim dataCells As Range
    Set dataCells = Range(firstCell, lastcell)
If one wants to sort on columns A, F, and G with a worksheet that has three title rows and then data cells, the macro would look like this:

```vba
Sub Sort_On_Three_Columns()
    Dim firstCell As Range
    Dim lastCell As Range
    Dim dataCells As Range
    Set firstCell = Range("A4")
    With ActiveSheet.UsedRange
        Set lastCell = .Cells(.Cells.Count)
    End With
    Set dataCells = Range(firstCell, lastCell)
    dataCells.Sort _
        Key1:=Range("A4"), Order1:=xlAscending, _
        Key2:=Range("F4"), Order2:=xlAscending, _
        Key3:=Range("G4"), Order3:=xlAscending, _
        Header:=xlNo, _
        OrderCustom:=1, _
        MatchCase:=False, _
        Orientation:= xlTopToBottom, _
        DataOption1:=xlSortNormal, _
        DataOption2:=xlSortNormal, _
        DataOption3:=xlSortNormal
End Sub
```

9.15 Using Worksheet Functions

9.15.1 Finding the Minimum Value in a Range

The following statement illustrates how to find the minimum value in a range:

```vba
minValue = Application.WorksheetFunction.Min(Range("A1:A4"))
```
10. TEXT AND NUMBERS

10.1 255 Character Limitations

255 Character Limitations

For most activities in Visual Basic, text strings can not exceed 255 characters. For example, if you have to pass an address to a property, the address must be less than 255 characters. Normally this is not a problem, but if you have long workbook names, long sheet names, and are writing an address that refers to multiple regions, it can be. Non-contiguous cell references can get very lengthy very quickly, because each individual address is also preceded by the sheet name. The same thing goes when creating an array reference to a group of sheets.

10.2 Adding Characters To The End Of A String

The following code adds a semi-colon to the end of the cells string values in the specified range.

```vba
Sub AddSemicolon()
    Dim Rng As Range
    For Each Rng In Range("A1:A200")
        If Rng.Value <> "." Then
            Rng.Value = Rng.Value & ";"
        End If
    Next Rng
End Sub
```

10.3 Adding Text To A Range Of Cells

The following example modifies all the cells in column A of the used range. The modification consists of putting the character "*" and a space in front of each cell with an entry.

```vba
Dim c As Range
For Each c In Intersect(ActiveSheet.UsedRange, Columns("A"))
    If Not IsEmpty(c) Then
        c.Value = "* " * c.Value
    End If
Next c
```

In the above example, the `Intersect` method is used to return just the cells in the used range of the active sheet and in column A. This is much faster than the following code, which checks every single cell in column A.

```vba
Dim c As Range
For Each c In Columns("A")
    If Not IsEmpty(c) Then
        c.Value = "* " * c.Value
    End If
Next c
```
In both examples, the cell is tested using `IsEmpty()` to determine if it should be modified. Please note the above examples assume that if the cell has an entry, that it is a text entry that should be modified. Also "c" is used as a range name to refer to the cell in the range being modified.

If you know the specific range to be modified, or can specify a range that encompasses the cells to be modified, then you can do something like the following instead. This example also specifies sheet so that the code is not restricted to working on the active sheet or changing to the sheet containing the range to be modified. It does assume that the sheet is in the active workbook. Lastly, this example tests the value of each cell versus testing to see if the cell is empty and uses a `With…End With` construction to make the code slightly faster. The periods in front of `Value` in this example are needed because of the `With…End With` construction. It makes the `Value` property refer to cell "c".

```vba
Dim c As Range
For Each c In Sheets("my data").Range("A1:A200")
    With c
        If .Value <> "" Then
            .Value = "* " & .Value
        End If
    End With
Next
```

### 10.4 Case Insensitive Comparisons

To make text comparisons in a module case insensitive, put the statement

```vba
Option Compare Text
```

at the top of your module.

Or, you can use `LCase()` and `UCase()` to convert your text strings to the same case and then do comparisons:

```vba
str1 = Range("A1").Value
str2 = Range("A1").Value
If UCase(str1) = UCase(str2) Then
    'actions to take if the same
Else
    'actions to take if different
End If
```

If you are using `InStr()` to search a string for a matching string, then you have two options to making the search case insensitive:

- Put `Option Compare Text` at the top of the module, or
Supply all the arguments to the InStr function:

**InStr**(starting position, text to search, text to search for, comparison type)

By supplying a 1 for the starting position and a 1 as the comparison type, a string will search from the first characters onward, and the search will be case insensitive. To make the search case sensitive, use 0 for the comparison type.

```vba
str1 = Range("A1").Value
str2 = Range("A1").Value
N = InStr(1, str1, str2, 1)
```

### 10.5 How to do A Date Comparison

If you want to see if the current date is after a specific date, then use the following approach:

```vba
If Date > #12/31/2004# Then
    'code if true
End If
```

### 10.6 Concatenating Strings

Excel uses the & operator to concatenate strings. For example

```vba
Dim A As String
Dim B As String
Dim C As String

A = "John"
B = "Smith"
C = A & " " & B
```

The above returns "John Smith", with a space between the words.

### 10.7 Converting Numbers That Appears As Text Back To Numbers

Every now and then Excel will treat numbers as text. This is most obvious when they appear left justified in a cell. And, clearing the cells format does not help. Or, you get multiple listings for the same number in a pivot table. This typically happens when importing data, but other events can cause it to happen. The following is a Visual Basic solution to this problem.

```vba
Dim R As Range
Set R = Selection
```
The following is one way to convert these entries back to numeric entries:

```vba
Dim c As Range
For Each c In Selection
    c.Formula = c.Formula
Next c
```

The following is still another way to fix this problem:

```vba
Dim tempR As Range

'find a blank cell
Set tempR = Cells(ActiveSheet.UsedRange.Count + 1)

'format the selection as a number
Selection.NumberFormat = "0"

'copy tempR, which is a blank cell
tempR.Copy

'add tempR's value to the cells in the selection
Selection.PasteSpecial Paste:=xlValues, operation:=xlAdd

'clear the clipboard
Application.CutCopyMode = False
```

Still another approach you can use is:

```vba
Dim cell As Range
For Each cell In Selection
    If Not IsEmpty(cell) Then cell.Value = cell.Value
Next
```

### 10.8 Converting Numbers To Strings

The statements `CStr(number)` and `Format(number)` return a string consistent with the Windows settings, while `Str(number)` returns a string in US format. If you don't do any explicit conversion, you get a string consistent with the language of Excel you're using.

### 10.9 Converting Text To Proper Case
The following code converts all text in the range A1 to A100 to proper case. It checks to make certain that the cell does not contain a formula.

```vba
Dim TheCell As Range
For Each TheCell In Range("A1:A100")
    'Make certain the cell does not have a formula and is not empty
    If TheCell.HasFormula = False And Not IsEmpty(TheCell) Then
    End If
Next TheCell
```

10.10 Creating A Fixed Length String

If you need to create strings of a certain length and all of the same character, then use a statement like the following:

```vba
tenAstericks = String(10, ")
```

or

```vba
ThreeQuotes = String(3, 34)
```

In the second example, the number indicates the ASCII character to use.

10.11 Determining If A Number Is Odd Or Even

Something like this will determine if a number is odd or even:

```vba
If MyVariable mod 2 = 0 Then
    MsgBox "Even"
Else
    MsgBox "Odd"
End If
```

10.12 Determining If A Value Is Text Or Numeric

VBA has the function `IsNumeric(value)` which returns true if the argument is a number or can be converted to a number

You can always use the worksheet functions, which are more exact:

- `Application.IsNumber(Value)` which only returns `True` if it is actually a number
- `Application.IsText(Value)` returns `True` if the argument is text.
10.13 Entering Special Characters With The Chr Function

The Chr() function allows you to enter special characters in message boxes that you may display or when you write entries to a cell. The following examples illustrate its use:

'message box with double quotes around a word in the message:

MsgBox "This is not in quotes and " & _
    Chr(34) & "This is in quotes" & Chr(34)

'To have text in a message box appear on a different line, use Chr(13)

MsgBox "The file name is " & Chr(13) & ActiveWorkbook.Name

'This does the same as the above, but adds a blank line and

'does a tab indent:

MsgBox "The file name is " & _
    Chr(13) & Chr(13) & Chr(9) & ActiveWorkbook.Name

'The following uses Chr() to the equivalent of alt-enter if you were to type

the following into a cell:

ActiveCell.Value = "January 1998" & Chr(10) & "Sales Forecast"

'The following writes out the characters generated by the Chr() Function. Characters 1-32,
which are not generated are not printable characters.

Sub Show_All_Characters()
    Dim I As Integer

    'write out column labels

    Worksheets(1).Cells(1, 1).Value = "Chr #"
    Worksheets(1).Cells(1, 2).Value = "Symbol"

    'fill in the character set

    For I = 33 To 255
        Worksheets(1).Cells(I - 30, 1).Value = I
        Worksheets(1).Cells(I - 30, 2).Value = Chr(I)
    Next

    'center the column entries
Worksheets(1).Columns("a:b").HorizontalAlignment = xlCenter
End Sub

10.14 Extracting Beginning Numbers From Text Strings

If you have text entries like 12A34, 789B2 and so forth, the following function will extract just
the beginning numbers:

Public Function ValString(ByVal anyS As String) As Double
 anyS = UCase(anyS)
 If InStr(anyS, "D") > 0 Then
 anyS = Application.Substitute(anyS, "D", "Z")
 ElseIf InStr(anyS, "E") > 0 Then
 anyS = Application.Substitute(anyS, "E", "Z")
 End If
 ValString = Val(anyS)
End Function

10.15 Extracting Numbers From The Left Of A String

You can use either Val() or CDbl() or CInt() to convert a string like "50 people" to the number
50.

AStr = "50 People"
num = Val(aStr)

' or

num = CDbl(aStr)

CDbl(string) assumes that the string is in a format consistent with the Windows regional settings,
while Val(string) assumes the string is in US format (i.e. with a period as the decimal separator).
When a possibility exists that different decimal separators may be used (for example, in
international applications), you should use CDbl instead to convert a string to a number. The
IsNumeric VBA function also uses the Windows settings (but doesn't like % signs).

If the string begins with a $ sign, then you must first remove the dollar sign. Otherwise the Val
function will return a 0 value:

Function NumOnly(alphaNum As String)
 If Left(alphaNum, 1) = "$" Then

 ' if a $ sign at the start, evaluate string to the right of it

 NumOnly = Val(Mid(alphaNum, 2))
 Else

 ' if no $ sign, evaluate the whole string
10.16 Extracting Numbers From The Right Side Of A String

The following function will extract the number value out of a string such as ABC-12. It returns the integer value of the number.

```vba
Function GetNumber(sStr As String) As Integer
    Dim NumPart As String

    'extract the portion of the string to the right of the dash
    'the InStr() function returns the number of characters into the string where
    'the "-" or whatever string you want to find is located
    NumPart = Right(sStr, Len(sStr) - InStr(sStr, "-"))

    'convert the extracted string to a number
    GetNumber = CInt(NumPart)
End Function
```

The following illustrates the above function:

```
MsgBox GetNumber("ABC-12")
```

10.17 Extracting Part Of A String

The functions, **Left**, **Right**, and **Mid** allow you to extract part of a cell's entry. If cell A1 contains "ABC567DEF", then

- **Right(Range("A1") .Value, 3)** returns "ABC"
- **Mid(Range("A1") .Value, 3)** returns C567DEF
- **Mid(Range("A1") .Value, 4, 2)** returns just "56", a text string. To convert it to a number use **Val(any string)**.
- **Right(Range("A1") .Value, 2)** returns "EF"
The **Mid** function allows you to extract part of a string. Its arguments are

\[ \text{Mid}(\text{string}, \text{start position}, \text{Numbers of Characters}) \]

The number of characters to extract are optional.

For example, \text{Mid}("ABCD", 2, 2) returns "BC". \text{Mid}("ABCD", 2) returns "BCD".

You can use **InStr** in conjunction with **Mid** to find the location of a string within a string and then extract just the needed text. For example:

```vba
Dim N As Integer
Dim lastName As String
Dim sName As String
sName = "John Smith"
N = InStr("John Smith", " ")
lastName = Mid(sName, N, )
```

Please note that the **InStr** function is case sensitive unless you have put **Option Compare Text** at the top of your module. For examples on making **InStr** case in-sensitive, please see the topic "Case Insensitive Comparisons".

The following illustrates the use of the above functions:

```vba
Dim aStr As String
aStr = "ABCDE"

'extract just AB
MsgBox Left(aStr, 2)

'extract just DE
MsgBox Right(aStr, 2)

'extract just CD (the string starting at position 3 that is 2 characters long)
MsgBox Mid(aStr, 3, 2)

'extract all characters to the right, starting at character 2
MsgBox Mid(aStr, 2)
```

For more examples, highlight the word **Mid**, **Left**, or **Right** and press F1 while in a module.

**10.18 Extracting Strings Separated By A /**

This is a solution for a user who had a column of cells whose entries looked like the following:
The user wanted to extract each string or number and write it to the cells to the left of the entry, splitting the up the value based on the "/"s and putting each value in a separate cell.

This is the routine that initiates the process. Select a range of cells and run it. Blank cells are ignored.

Sub ProcessCells()
    Dim cell As Range
    For Each cell In Selection
        'call the routine that processes a cell if the entry is not blank
        If Not IsEmpty(cell) Then
            ExtractValue cell
        Next
    End Sub

Sub ExtractValue(anyCell As Range)
    Dim s As String
    Dim N As Integer, I As Integer
    'set a variable equal to the value in the cell
    s = anyCell.Value
    'find the first occurrence of a / in the string
    N = InStr(s, "/")
    'if a / is found, process the string and loop until no more /'s
    While N > 0
        'index the counter used to specify the offset for the output
        I = I + 1
        'write the left portion excluding the / to the offset cell
        anyCell.Offset(0, I).Value = Left(s, N - 1)
        'remove the left portion and the / from the variable "s"
        s = Mid(s, N + 1)
        'search for the next occurrence of a /
N = InStr(s, "/")
Wend

'index I for the next offset cell
I = I + 1

'write the remaining string to this cell
anyCell.Offset(0, I).Value = s
End Sub

10.19 Finding The Number Of Occurrences Of A String In A Range

If you need to find the number of occurrences of a string in a range, there are several different approaches you can use. If the string is the only entry in the cells, then the following would work, if you wanted the count of occurrences of ABC filled cells in range A1:A10

Dim N As Long
N = Application.CountIf(Range(A1:A10), "ABC")

If the string can be embedded in a string, for example, "Part ABC", then you must either check each cell, or repeated use Find in a loop. The following illustrates using a Find loop:

Sub findABCs()
    Dim R As Range, startCell As Range
    Dim firstAddress As String
    Dim foundCell As Range
    Dim I As Long

    'set the range to search
    Set R = Range("A1:A10")

    'set the cell to be used in the After argument
    Set startCell = R.Cells(1)
    Do

    'do the search and set a range variable to the result.
    'Search is by part instead of whole cell entry.
    Set foundCell = R.Find(What:="ABC", _
        After:=startCell, _
        LookIn:=xlValues, LookAt:=xlPart, _
        SearchOrder:=xlByRows, _
        SearchDirection:=xlNext, _
        MatchCase:=False)
'exit if no matching string found

    If foundCell Is Nothing Then Exit Do
    If I = 0 Then

'the first time a match is found, store that cells address

    firstAddress = foundCell. Address
Else

'on future matches check the address, and exit if it is a repeat 
'of the first found cell, which indicates that the Find procedure 
'has looped back to the first cell.

    If foundCell. Address = firstAddress Then Exit Do
End If

'increase the count of found cells by one

    I = I + 1

'set the startCell to the foundCell so that the next search starts 
'after this cell, and does not continue to find the same cell

    Set startCell = foundCell
Loop

'display a message with the results

    MsgBox I & " found"
End Sub

Another way to check for the occurrence of a string within a string throughout a range is to check 
cell by cell. This is less efficient than the above approach but simpler to write. The following 
illustrates this approach.

Sub findABCsApproach2()
    Dim R As Range, cell As Range
    Dim I As Long

'set the range to search

    Set R = Range("A1:A10")
    For Each cell In R

'check each cell's value and see if ABC is in it. Convert the 
'cell's value to upper case so that the comparison is upper case

'to upper case

    If InStr(UCase(cell. Value), "ABC") > 0 Then
'increment the count by one when ABC is found.

    I = I + 1
    End If
Next

'display a message with the results

    MsgBox I & " found"
End Sub

If you need to check entire rows or columns for a string, then you should restrict the range to be searched to the sheet's used range. This will make both approaches above run faster. The following illustrates how you would do this:

' set R initially to the overall range

    Set R = Rows("1:3")

'redefine R to its intersection with the used range

    Set R = Intersect(R, ActiveSheet.UsedRange)

10.20 How To Get The Number Of Characters In A Selection

The following code will return the number of characters, including blanks in a selection of cells:

Sub CharacterCount()
    Dim cell As Range
    Dim I As Integer
    For Each cell In Selection
        I = I + Len(cell.Value)
    Next
    MsgBox "there are " & I & " characters and spaces in the selection"
End Sub

10.21 How To Test If A Cell Or Variable Contains A Particular Text String

To test if a cell or a string variable contains a particular text string, use the InStr function. It returns the starting position in a string of a string. If the string is not found a zero value is returned.

You can do a case insensitivity test by setting the 1st and last arguments of InStr to 1. The 1st argument is the starting position of the search and the last argument is a 1 for case insensitive testing and a 0 for case sensitive comparisons. If one is supplied, then the other is required.
If InStr(1, Range("A1").Value, "ABC", 1) > 0 Then

'actions to take if ABC is in cell A1's string

End If

Example:

Dim myString As String
myString = Ucase(Range("A1").Value)
If InStr(myString, "ABC") > 0 Then

'actions to take if ABC is in cell A1's string

End If

Another way to do case insensitive tests using InStr is to put

Option Compare Text

at the top of your module. This makes all text comparisons case insensitive unless overridden.

To override Option Compare Text, set the last argument of InStr to 0:

If InStr(1, Range("A1").Value, "ABC", 0) > 0 Then

'actions to take if ABC is in cell A1's string
'and ABC in the string is capitalized

End If

10.22 Numbers To Words

If you want to convert numbers to words, take a look at the following Microsoft Excel knowledge base articles: Q140704 and Q95640

10.23 Finding A Font

The following code will check to see if a given font is present on a machine:

Dim I As Long
With Application.CommandBars.FindControl(Id:=1728)
For I = 1 To .ListCount
    If .List(I) = "Arial" Then
        MsgBox "Arial font found!"
        Exit For
    End If
End For
If you want to list all the available fonts, you can do so with this code:

```vba
Dim I As Long
With Application.CommandBars.FindControl(Id:=1728)
    For I = 1 To .ListCount
        Cells(I, 1).Value = .List(I)
    Next
End With
```

### 10.24 Removing Alt-Enter Characters

If you have a cell where a user has used alt-enter, you can remove the alt-enter by using a statement like the following:

```vba
ActiveCell.Value = _
Application.WorksheetFunction.Substitute(ActiveCell.Value, _
vbLf, " ")
```

If you have Excel 2000 or Excel XP, you can use the following statement:

```vba
ActiveCell.Value = Replace(ActiveCell.Value, vbLf, " ")
```

### 10.25 Removing Text To The Right Of A Comma

The following code removes data to the right of a comma. More specifically, the cells in a column contain last names, a comma, and first names and middle initials. For example: SMITH, JOHN D or JOHNSON, BILL.

```vba
Sub OnlyLastName()
    Dim rangeToChange As Range
    Dim cell As Range
    Dim iLoc As Integer

    'rotate through each cell in the selection
    For Each cell In Selection

        'look for a comma using the InStr function
        iLoc = InStr(cell.Value, ",")

        'if comma found then iLoc will be greater than 0
        If iLoc > 0 Then
```

```vba
```
'remove the comma and any text to the right of the comma

    cell.Value = Left(cell.Value, iLoc - 1)
    End If
    Next cell
End Sub

10.26 Using The Chr Function To Return Letters

The Chr() function can be used to return upper or lower case letters. The following illustrates it use to fill cells on a the active sheet with the alphabet:

Sub AToZ()
    Dim iChar As Integer
    For iChar = 0 To 25

      'write upper case letters
      ActiveCell.Offset(iChar, 0) = Chr(65 + iChar)

      'write lower case letters
      ActiveCell.Offset(iChar, 1) = Chr(97 + iChar)
    Next
End Sub

10.27 Using The LIKE Operator To Do Text Comparisons

The Like operator can be used to do text comparisons, and is simpler than the InStr function. The following statement will display True if the word Asia is found in the ActiveCell's value. Note the use of Ucase to make the test case insensitive.

    MsgBox UCase(ActiveCell.Value) Like "*ASIA*"

The string following Like is the pattern that is searched for in the first string. For more information on using the Like function, place the cursor in the word Like and press F1 to display Visual Basic's help on this function.

If you put

    Option Compare Text

at the top of your module, then the Like test is case insensitive and you do not need to use Ucase to make the pattern string upper case.

10.28 Writing The Alphabet Out To A Worksheet
The following code will write the alphabet out to a worksheet:

```vba
Sub AToZ()
    Dim iChar As Integer
    For iChar = 0 To 25
        ActiveCell.Offset(iChar, 0) = Chr(65 + iChar)
    ' or 95 for lowercase
    Next
End Sub
```
11. MESSAGE BOXES

11.1 Displaying Message Boxes

The following displays a message box that just has an OK button on it

```vba
Sub MsgBoxExample1()
    MsgBox "This is an example of a message."
End Sub
```

The following displays a message box that displays the OK and Cancel buttons, allowing the user to provide a response:

```vba
Sub MsgBoxExample2()
    Dim iResponse As Integer
    iResponse = MsgBox("Select OK or Cancel", vbOKCancel)

    'check to see if cancel button selected
    If iResponse = vbCancel Then
        MsgBox "You selected Cancel"
    End If
    If iResponse = vbOK Then
        MsgBox "You selected OK"
    End If
End Sub
```

This example could also have been written in a much more condensed fashion, where selecting Cancel stops execution and selecting OK allows it to continue:

```vba
Sub MsgBoxExample3()
    If MsgBox("Select OK to Continue") = vbCancel Then End

    'Statement to execute if OK selected
End Sub
```

The following illustrates a message box that displays Yes, No, and Cancel buttons. If you want only Yes or No buttons, then use `vbYesNo` instead of `vbYesNoCancel`

```vba
Sub MsgBoxExample4()
    Dim iResponse As Integer

    'display message and store response for evaluation
    iResponse = MsgBox("Select a button", vbYesNoCancel)

    'check to see if cancel button selected
End Sub
```
If iResponse = vbCancel Then
    MsgBox "You selected Cancel"
End If

'check to see if Yes selected
If iResponse = vbYes Then
    MsgBox "You selected Yes"
End If

'check and see if No selected
If iResponse = vbNo Then
    MsgBox "You selected No"
End If
End Sub

11.2 Formatting in a Message Box

The following illustrates how to format entries in a MsgBox:

MsgBox " Equipment No.     " & EquipNum & Chr(13) 
   & " Description  -   " & Description & Chr(13) 
   & " Internal Due -   " & Format(partDate,"mmm-dd"

where the variables EquipNum, Description, and partDate have already been set.

In the above Chr(13) is a carriage return, letting you display the entry on the next line. Other useful Chr() values are 174 which is a copyright symbol and 34 which is a double quote.

11.3 Using Double Quotes In A Message Box

The primary use of the double quote in your code is to use it to delimit text strings. For example

MsgBox "Hello World"

or ActiveCell.Value = "Description"

If you wish to wish to use double quotes to highlight text in a message box, then use one of the following approaches:

MsgBox "The word " & Chr(34) & "WOW" & Chr(34) " is in double quotes"

or MsgBox "The word ""WOW"" is in double quotes"

The second example works because two double quotes in a row is interpreted as part of the text string, and not as the delimitation of a text string.
11.4 How To Format A Message In An InputBox Or Message Box

If you use `Chr(13)` in your MsgBox text, then this acts as a line feed, putting the text following the `Chr(13)` on a new line in a MsgBox.

```vba
Sub MultiLineExample()
    MsgBox "This is the first line." & _
        Chr(13) & "This is the second line." & _
        Chr(13) & "This is the third line"
End Sub
```

If you want a line to be indented, then use code like the following:

```vba
Sub IndentExample()
    MsgBox "This is the first line." & _
        Chr(13) & Chr(9) & "This is indented." & _
        Chr(13) & Chr(9) & "This is also indented"
End Sub
```

11.5 Adding A Help Button To A MsgBox

The help button on a message box can only be displayed using the following approach. It is possible that bug fixes by Microsoft may cure this, but this is needed for users who don't upgrade.

You have to add a help button explicitly with the buttons argument, like this:

```vba
MsgBox "Click Help for help?", _
    vbCritical + vbMsgBoxHelpButton, "Help", "runner.hlp", 10
```

This would display a help file named runner.hlp that is in the Windows directory (assuming it exists). Please note that the number used as the last argument is help file dependent.
12. GETTING USER INPUT

12.1 Pausing A Macro For Input

The following pauses a macro until the user enters a value. The value is then written to the active cell. If the user presses Cancel, the macro loops.

```vba
Sub GetInput()
    Dim str As String
    Do
        str = InputBox("Enter something")
        If str <> "" Then
            ActiveCell.Value = str
            Exit Do
        End If
    Loop
End Sub
```

You can also put the following step right before the Loop statement to see if the user wants to continue:

```vba
If MsgBox("Do you want to continue", vbYesNo, _
    "Continue?") = vbNo Then End
```

12.2 Restricting What Is Allowed In An InputBox

If you use Application.InputBox to display an input box, you can restrict what the user is allowed to enter by specifying a value for the type argument. For example, the following restricts the user to providing just numbers:

```vba
Dim inputAnswer As Variant
Dim lNum As Single

' use a variant variable to get the inputbox response as the response can
' be Boolean (false) or a value

inputAnswer = Application.InputBox(_
    Prompt:="Enter a number", Type:=1)
If inputAnswer = False Then Exit Sub

' store the user entry to a numeric variable.
' Please note that the inputbox returns a string even if the type is 1

lNum = Val(inputAnswer)
MsgBox "you entered " & lNum
```
The following uses `Application.InputBox` to get a user selected range, by setting the type argument to 8.

'The variable for the inputbox response must be type range 'or the Is Nothing test will crash!

```vbnet
Dim userSelection As Range

'turn on error handling in case user press cancel
On Error Resume Next
Set userSelection = Application.InputBox( _
    prompt:="enter a range", Type:=8)
On Error GoTo 0
If userSelection Is Nothing Then

    'display this message if cancel selected
    MsgBox "A range was not selected"
    Exit Sub
End If

'display range selected
MsgBox "You selected " & userSelection.Address(external:=True)

'go to selected range
Application.Goto userSelection, True
```

The most popular values for the *type* argument are:

1. A number
2. Text (a string)
4. A logical value (True or False)
8. A cell reference, as a Range object

For more information on the inputbox, highlight the word in the Visual Basic editor and press F1 for help.

### 12.3 Prompting The User To Enter A Number

You can get a user to supply you with a number using code like the following.

```vbnet
Sub GetANumber()
    Dim userInput As Variant
```
'display the application input box, with type set to 1 which allows
'
'only number input

userInput = Application.InputBox(_
   prompt:="Please enter a number", Type:=1)

'if cancel is selected, the TypeName is Boolean

If TypeName(userInput) = "Boolean" Then
   Exit Sub
End If

'convert the user input from a string to a number

userInput = Val(userInput)

'display the user's input and the TypeName

MsgBox userInput & " " & TypeName(userInput)
End Sub

If the Type argument is not specified, then the user could enter any string, not just numbers. If Application.InputBox is not used, but InputBox alone is used, there is no way to distinguish between a Zero and the user selecting cancel. Lastly, all input from an input box is a string, and must be converted to a number if it is a number. Otherwise, numeric tests will not work.

12.4 Using The Application InputBox Function To Specify A Number

The Application.InputBox function can be used to restrict user input to just numbers by specifying a type argument when you display the inputbox. The following illustrates this:

Sub Get_A_Number()
   Dim userResponse As Variant

   'display the input box

   userResponse = _
      Application.InputBox(Prompt:="Enter a number", _
         Type:=1)

   'check and see if cancel is selected. Exit if it was

   If userResponse = "False" Then Exit Sub

   'display a message showing what number was entered
   'use Val() to convert to a number instead of a "string number"
MsgBox "The number you entered was " & Val(userResponse)
End Sub

In the above example, the variable that receives the result of the Application.InputBox statement must be a Variant variable. That way, it can accept either a number or a False response if cancel is selected.

Another way to determine if cancel was selected in the Application.InputBox is to use a TypeName() test:

If TypeName(userResponse) = "Boolean" Then Exit Sub

12.5 InputBox to Ask For the Date

Try the following code:

Sub GetADate()
        Dim TheString As String
        Dim RowNdx As Integer
        Dim TheDate As Double

        TheString = Application.InputBox("Enter A Start Date")
        If IsDate(TheString) Then
            TheDate = DateValue(TheString)
        Else
            MsgBox "Invalid date"
        End If
End Sub

Please note that you should validate TheDate variable to confirm that it is within the date range you want.

12.6 Using The Visual Basic InputBox To Return A Range

There are two input boxes in Excel. There is the Visual Basic InputBox function and it only returns a text string. It is the simplest to use. There is also the Excel Application.InputBox which allows you to specify the type of input which is returned. The following illustrates using the VB InputBox function:

Simplest approach with no error checking - assumes the user won't screw up the selection or won't get upset if an error box appears.

'declare a Variant variable for the output of the InputBox

Dim myObject As Variant
myObject = InputBox("Enter a cell")
Range(myObject).Select
However, you should account for the user hitting the cancel key which returns `False` or entering an invalid address. The following shows how to do that:

```vba
Sub InputBoxExample()
    Dim myObject As Variant, cellSelected As Range

    'return label in case an invalid range is entered
    GetACell:
        myObject = InputBox("Enter a cell")

    'If no entry made in the box, even if OK canceled, exit procedure
    If myObject = "" Then
        Exit Sub
    Else
        'turn on error handling incase the entry is not a range
        On Error GoTo ErrorHandler
        'store the input in a range variable for later use
        Set cellSelected = Range(myObject)
    End If

    'turn off error handling
    On Error GoTo 0
End Sub
```

12.7 How To Get A Cell Address From A User

Use the `Application.InputBox` method, with the `Type` argument set to 8:

```vba
Dim TheRange As Range
On Error Resume Next
Set TheRange = Application.InputBox _
    (prompt:="Select A cell",type:=8)
On Error GoTo 0
If TheRange Is Nothing Then
    MsgBox "You didn't select a cell"
End If
```
12.8 Using InputBoxes To Get A Cell Range

The following two examples show how to get the user to pick just a single cell range. The two examples after these two allow the user to pick any number of cells.

Sub InputExample1()

' This example displays an input box and asks the user to select a cell

Dim cellSelected As Range

' set on error in case cancel selected

On Error Resume Next

' display inputbox and assign result of select to a variable

Set cellSelected = Application.InputBox(prompt:="Select a single cell", Type:=8)

' turn off error checking

On Error GoTo 0

' check if a range is selected

If cellSelected Is Nothing Then
    MsgBox "No cell selected"
    Exit Sub
ElseIf cellSelected.Cells.Count > 1 Then

' check and see how many cells were selected
' display messages giving the result

    MsgBox "You selected more than one cell"
    Exit Sub
Else
    MsgBox "You selected " & cellSelected.Address(external:=True)
End If

End Sub

Sub InputExample2()

' This example displays an inputbox and asks the user to select a cell
' It continues to loop until either cancel is selected or a single cell is selected

Dim cellSelected As Range

'this example displays an input box and asks the user to select a cell

'this example displays an input box and asks the user to select a cell
'set on error in case cancel selected

On Error Resume Next
Do

'display inputbox and assign result of select to a variable

Set cellSelected = Application.InputBox _
(prompt:="Select a single cell", Type:=8)

'turn off error checking

On Error GoTo 0

'check if a range is selected. Stop all action if none selected

If cellSelected Is Nothing Then
End

'check and see how many cells were selected

ElseIf cellSelected.Cells.Count > 1 Then
MsgBox "You selected more than one cell"
Else

'exit if one cell selected

    Exit Do
End If
Loop

'display the value of the cell selected

MsgBox "the value in the selected cell is " _
& cellSelected.Value
End Sub

Sub InputExample3()

'This example displays an inputbox and asks the user to select a range

Dim cellsSelected As Range

'set on error in case cancel selected

On Error Resume Next

'display inputbox and assign result of select to a variable

Set cellsSelected = Application.InputBox(_
prompt:="Select a range of one or more cells", Type:=8)
'turn off error checking

On Error GoTo 0

'check if a range is selected

If cellsSelected Is Nothing Then
    MsgBox "No cell selected"
    Exit Sub
Else

'display messages giving the cells selected

    MsgBox "You selected " &
    cellsSelected.Address(external:=True)
End If
End Sub

Sub InputExample4()

'This example displays an inputbox and asks the user to select a range
'It continues to loop until either cancel is selected or a range is selected

Dim cellsSelected As Range, cell As Range

'set on error in case cancel selected

On Error Resume Next
Do

'display inputbox and assign result of select to a variable

    Set cellsSelected = Application.InputBox _
        (prompt:="Select a single cell", Type:=8)

'turn off error checking

    On Error GoTo 0

'check if a range is selected. Stop all action if none selected

    If cellsSelected Is Nothing Then
        End
    Else

'exit as a range has been selected

        Exit Do
    End If
Loop
'display the value of the cells selected

For Each cell In cellsSelected
    MsgBox "the value of " & cell.Address(external:=True) & " is " & cell.Value
End Sub

12.9 An Application InputBox Example That Gets A Range

The following illustrates how to use the Application.InputBox function to get a range selection from the user.

Dim Rng As Range
On Error Resume Next
Set Rng = Application.InputBox(prompt:="Enter A Range", Type:=8)
If Rng Is Nothing Then
    MsgBox "No Range Selected"
Else
    Rng.Select
End If
On Error GoTo 0

12.10 Using The InputBox To Put A Value In A Cell

The following illustrates how to get a value from a user and place it in cell A3 of the active sheet:

Sub InputBoxExample()
    Dim cellValue As Variant
    reShowInputBox:
    cellValue = Application.InputBox("Enter value to go in A3")
    If cellValue = False Then
        Beep
        Exit Sub
    ElseIf cellValue = "" Then
        Beep
        GoTo reShowInputBox
    Else
        ActiveSheet.Range("A3").Value = cellValue
    End If
End Sub

12.11 Prompting The User For Many Inputs

The following is a simple way to prompt a user for a series of inputs and place the values into cells in the active sheet. It uses one main routine which calls a subroutine over and over again.
Sub MainProcedure()
    LoadData "A1", "Enter a value for something"
    LoadData "G1", "Enter a value for something else"
End Sub

'arguments are the cell address for the value and the message to be displayed

Sub LoadData(addr As String, msg As String)
    Dim response As Variant, iR As Integer

    'loop until the user enters a value or chooses to quite
    While response = ""

    'display a inputbox with the msg that was passed to this routine
    response = InputBox(prompt:=msg)

    'if cancel selected or no value entered see if the user wants to quit
    If response = "" Then
        iR = MsgBox("No value was entered. " & "Do you wish to quit?", vbYesNo)

    'this halts all activity
    If iR = vbYes Then End
    Wend

    'this loads the value in the cell
    Range(addr).Value = response
End Sub
13. USERFORMS

13.1 USERFORM EXAMPLES

13.1.1 How To Create And Display UserForms

To create a user form, first press ALT-F11 to go to the Visual Basic editor. Then select Insert, UserForm.

To display a userform, use a statement like the following:

    UserForm1.Show

Userforms stay in memory until they are unloaded by a Visual Basic statement. It is good practice to remove a userform from memory when you are done with it. This helps prevent you from running out of memory! To remove a userform from memory, use a statement like the following:

    UnLoad UserForm1

This statement resets the Userform back to its original setup.

13.1.2 How To Make UserForms Disappear When They Are Hidden

One of the bugs with userforms is that userforms may not disappear when told to by a statement like:

    UserForm1.Hide

until the code that called the form completes.

To make them go away, use the above statement, immediately followed by:

    Application.ScreenUpdating = True

If you wish to hide screen activity following the above statement, use the following:

    Application.ScreenUpdating = False

13.1.3 UserForm Display Problem

If you are using the Form Initialize event of a userform to display a dialog message and to run your routine, the userform may not paint completely until the Initialization procedure ends. There
are two ways that should get around this problem. One is to repaint the userform, and the other is to use an **OnTime** macro.

To repaint the userform, use the following three statements:

```vba
Application.ScreenUpdating = True
UserForm1.Repaint

'If you want to hide screen activity
Application.ScreenUpdating = False
```

The following illustrates how to use an OnTime macro:

```vba
Private Sub UserForm_Initialize()
    Application.OnTime Now, "DoWork"
End Sub
```

and place the DoWork procedure in a standard module:

```vba
Sub DoWork()
    'your main code here
End Sub
```

### 13.1.4 Initializing UserForms

There are two ways to initialize userforms:

- initialize the objects on the user form statements in your calling routine or from a sub-routine called by your calling routine, or

- Use the initialization event macro, **Private Sub** `UserForm_Initialize()`

Statements in either your calling routine or in the initialization event macro would have the form of:

user form name.object name.object property = value to use

For example:

```vba
UserForm1.Label1.Caption = "My Label"
```

or

```vba
UserForm1.ListBox1.RowSource = Worksheets(1).Range("A1:A10").Address
```
In these examples, the name of the user form is that the name that appears in the title row of the user form window. The name of the object is what appears in the Name field of the properties window when the object is selected. And the name of the property is what appears in below the name field in the properties window. The properties window can be displayed by pressing F4 or clicking on the properties window button.

To put the above examples in the initialization event requires you to do the following:

- Double click on the user form to get to its code sheet.
- Select the userform in the upper left drop down.
- Select Initialize in the upper right drop down

As you do these steps, the Visual Basic editor will create event code for events you won't need. For example it will first create a UserForm_Click macro when you double click on the user form. Its OK to leave these on the code sheet, or you can delete them if you do not need them.

The initial initialize event code looks like the following:

```vbnet
Private Sub UserForm_Initialize()
    'your code goes here
End Sub
```

There are advantages of using the user form initialize event macro and there are advantages of just putting the code in your procedures. You will have to be the judge of which is the best for your project.

### 13.1.5 Preventing UserForm Events from Running

If you are initializing optionbuttons, checkboxes, and other userform controls, you may not want the events associated with these controls from running. For example, you may have a option button change event that runs each time a user changes the value. But, when you initialize the button, you do not want the event to run.

The solution is to create a public variable in one of your regular modules. For example:

```vbnet
Public bDoEvents As Boolean
```

Then, when you are initializing controls, you first set this variable to false, initialize events, and then set to true:

```vbnet
bDoEvents = False
'code that initializes controls
bDoEvents = True
```
In your userform's code module, at the top of each event's code, check the value of this variable. If False, you would exit the subroutine so it does not run. For example:

```vba
Private Sub OptionButton1_Click()
    If bDoEvents = False Then Exit Sub
    'code you want executed when bDoEvents is True
End Sub
```

### 13.1.6 Unloading Versus Hiding A UserForm

If you use the following statement, assuming your userform is named UserForm1,

```vba
UserForm1.Hide
```

this will retain the form in memory until all macro execution is complete. This means that any changes made to the form will be retained and displayed the next time you show the form. It also means that the userform activate event will not run, as the form is still active, just hidden.

If you use the `Unload` statement, for example,

```vba
Unload UserForm1
```

then the form is removed from memory, and all changes to the form, either by the user or by your code is removed. It also means that you can not access the form to get these changes.

To get values from a form, do something like the following:

```vba
UserForm1.Show

'have some button issue the command "UserForm1.Hide
'store the values you need
'unload the form from memory

Unload UserForm1
```

### 13.1.7 Using Hide Instead Of Unload With UserForms

If you need a userform controls such as textboxes, listboxes, etc. to retain their values between showing while your code runs, do not UnLoad the form between showings. If you want the values reset to the defaults, then use UnLoad. In the following example, the UnLoad clears the form from memory, resetting it to its original settings.

```vba
Sub ResetUserForm()
    UserForm1.Show

'your code here that gets values from the form and does something
'unload from memory so form is reset
```

210
Unload UserForm1
UserForm1.Show

'additional user code
End Sub

On the userform is a button which has the following code in the userform's code module:

Private Sub CommandButton1_Click()
  'this hides the userform
  Me.Hide
End Sub

If you do not want to reset the userform, then do not use the statement "Unload UserForm1", assuming your form is named UserForm1.

13.1.8 Having UserForms Retain Settings Between Macro Runs

One of the problems with userforms is that once a userform is loaded, if the subroutine that loads it terminates, the form automatically unloads. A very undesirable behavior. However, it is possible to have a userform retain any user changes between subroutines, and even between macro runs.

To demonstrate this, create a userform with an edit box and a button. Assign the following code to the button, assuming it is named CommandButton1.

Private Sub CommandButton1_Click()
  'this hides the userform
  Me.Hide
End Sub

In a regular module, put the following code:

'declare a public variable at the top of the module
Public oFrm As UserForm1

Sub MainRoutine()
  'set variable to the userform
  Set oFrm = New UserForm1
'run subroutines which display the form

Sub ShowForm1()
'show the form; its hidden by a button's code
oFrm.Show
MsgBox "Finished with first showing"
End Sub

Sub ShowForm2()
'clears the Public variable, which unloads the userform
Set oFrm = Nothing
MsgBox "Finished with second showing"
End Sub

If you do not use the statement
Set oFrm = Nothing

Then the userform will continue to stay initialized for subsequent macro runs. You can demonstrate this by commenting out this statement in ShowForm2. Then run MainRoutine. Next, run ShowForm2 by itself. Notice that any changes to the edit box is retained.

You should see that any controls you place on the form will keep their values between showings until the global variable is reset, or the userform displayed by using a statement like UserForm1.Show.

13.1.9 Positioning a Form where it was Last Displayed

The following code will position a userform (in this case userForm1) at its last displayed position when it is redisplayed:

'declare public variables at the top of the module
Public formTop As Single
Public formLeft As Single

Sub PositionAndShowForm()
    With UserForm1
        'check to see if a location is stored
        If formTop > 0 Or formLeft > 0 Then
'StartUpPosition position must be set
to zero before setting top and left
.StartUpPosition = 0
'for top and left to be applied
'startup position must be set to zero
.Top = formTop
.Left = formLeft
End If
.Show
'save settings in case the user clicked a button
'that hides the form. Hiding does not trigger the
'query close event to run
SaveDialogSettings
End With
Unload UserForm1
End Sub

Sub SaveDialogSettings()
'only save settings if values > 0
'as clicking the "X" to close a dialog sets the
position to
'zero after the dialog closes
'however, as the procedure runs before it is truely
closed
'the top and left postions are > zero and can be stored
With UserForm1
  If .Top > 0 Or .Left > 0 Then
    'save the form location
    formTop = .Top
    formLeft = .Left
  End If
End With
End Sub

Place a statement calling the SaveDialogSetting procedure in the userform's query close event. To do this, right click on the userform in project explorer and select code. Then in the left dropdown at the top select userform. This will initially insert the code for the userform click event. Click into this code. Now, in the right dropdown select Query. Code for the QueryClose event will appear (we have added the statement to call the SaveDialogSetting procedure.

Private Sub UserForm_QueryClose(Cancel As Integer,
CloseMode As Integer)
'this even is triggered when the "X" is clicked and
'when the Unload command run
SaveDialogSettings
End Sub

13.1.10 Setting The Tab Order In An UserForm

The tab order of a userform is the order in which one moves from object to object when the userform is displayed. To change the tab order:

◆ Make sure no controls are selected.

◆ Right-click in the form or dialog, but not on a control.

◆ From the shortcut menu, choose Tab Order.

◆ Select the name of a control you want to reposition in the tab order.

◆ Choose Move Up or Move Down until the control name is in the appropriate position in the tab order.

13.1.11 Shortcut Variable Name For A UserForm

In a userform's code module, you can use the variable "Me" to refer the userform. This avoids having to type out the userform's full name to identify the form in your code. Typically, one would put code in the userform's Activate event to initialize edit boxes on the form.

Private Sub UserForm_Activate()
    Me.TextBox1.Text = ""
End Sub

Please note that the Activate event for a userform is run only when the form is loaded into memory. If you hide and then subsequently show the userform again, the Activate event is not run the second time. If however you unloaded the userform and then showed it again, the Activate event will run a second time.

13.1.12 Passing Information And Variables To UserForm Procedures
To demonstrate this technique, make a call to a subroutine in the UserForm's code module, passing the variable or value to that sub. Then, in the called subroutine, tell the form to show itself.

Here's an example, assuming that the userform is named UserForm1

In your module:

```vba
Sub TheCallingSub()
UserForm1.RoutineInUserFormCodeModule("Hello!")
End Sub
```

In the UserForm, which has a label on it:

```vba
Sub UserFormSubRoutine(sPassedIn As String)
Me.Label1.Caption = sPassedIn
Me.Show
End Sub
```

In the above example, the variable Me is a built in variable that can be used to reference the class where the code is being executed. In this case the class is a user form. If the variable Me were in a worksheet's code module, then Me would refer to the worksheet.

### 13.1.13 Putting Data On A Sheet From A Userform

To put data onto a sheet from a userform, use a statement like the following:

```vba
Worksheets("Data").Range("D6").Value = UserForm1.TextBox1.Value
```

Do not unload the userform until after you have stored the values you need from the form in variables or on a worksheet.

### 13.1.14 Getting Values From A UserForm

One way to get a value from a userform, such as the contents of an edit box, is to place code in the userform's UserForm_QueryClose procedure. The code in this procedure would in turn set the values of global variables and then unload or hide the procedure.

```vba
Private Sub UserForm_QueryClose(Cancel As Integer, CloseMode As Integer)
    gUserEntry = UserForm1.Editbox1.Text
End Sub
```

Please note that the global variables must be declared **Public** in a regular module, and not in the userform module. If declared in the userform module, they will not retain their values when the form is unloaded or hidden.
13.1.15 Displaying A Dialog To Get A Password

Just set the textbox property `PasswordChar` to "*". Do this by selecting the textbox in the Visual Basic editor, display the properties windows, and enter an asterisk (*) in the `PasswordChar` property field.

13.1.16 Removing The Quit/X Button On An UserForm

To remove the Quit/X button in the upper right corner of a userform can be done, but it requires some Windows API magic developed by Stephen Bullen.

'at the top of a module put the following function.
'each function must be on a single line

Private Declare Function FindWindow Lib "user32" Alias "FindWindowA" (ByVal lpClassName As String, ByVal lpWindowName As String) As Long

Private Declare Function GetWindowLong Lib "user32" Alias "GetWindowLongA" (ByVal hWnd As Long, ByVal nIndex As Long) As Long

Private Declare Function SetWindowLong Lib "user32" Alias "SetWindowLongA" (ByVal hWnd As Long, ByVal nIndex As Long, ByVal dwNewLong As Long) As Long

'place these two constant statements at the top of the module

Const GWL_STYLE = (-16)
Const WS_SYSMENU = &H80000

'place this in the userform's code module

Private Sub UserForm_Initialize()
    Dim hWnd As Long, a As Long
    hWnd = FindWindow("ThunderXFrame", Me.Caption)
a = GetWindowLong(hWnd, GWL_STYLE)
    SetWindowLong hWnd, GWL_STYLE, a And Not WS_SYSMENU
End Sub

'be sure to put a button on the form so that you can exit the form. For example

Private Sub CommandButton1_Click()
    Me.Hide
End Sub

If you forget to the above button on the form and you have the Visual Basic Editor active, you can close the form by pressing ALT-Tab to get to the VB editor and then clicking on the reset button.

13.1.17 Hiding The Exit X On A Userform
To hide the little X that appears at the top right of a userform, you can use the following code that you would put in the userform's module:

'this goes at the top of the module:

Private Declare Function FindWindow Lib "user32" _
    Alias "FindWindowA" (ByVal lpClassName As String, _
    ByVal lpWindowName As String) As Long
Private Declare Function GetWindowLong Lib "user32" _
    Alias "GetWindowLongA" (ByVal hWnd As Long, _
    ByVal nIndex As Long) As Long
Private Declare Function SetWindowLong Lib "user32" _
    Alias "SetWindowLongA" (ByVal hWnd As Long, _
    ByVal nIndex As Long, ByVal dwNewLong As Long) As Long
Const GWL_STYLE = (-16)
Const WS_SYSMENU = &H80000

Private Sub UserForm_Initialize()
    'this hides the X on the caption line
    Dim hWnd As Long, a As Long
    Dim V As Integer
    V = CInt(Left(Application.Version, _
        InStr(Application.Version, ".")))
    'If V = 8 this is Excel 97
    If V = 8 Then
        hWnd = FindWindow("ThunderXFrame", Me.Caption)
    Else
        hWnd = FindWindow("ThunderDFrame", Me.Caption)
    End If
    a = GetWindowLong(hWnd, GWL_STYLE)
    SetWindowLong hWnd, GWL_STYLE, a And Not WS_SYSMENU
End Sub

13.1.18 Disabling the Exit X on a Userform

To prevent a user from closing an userform by clicking on the small "x" at the top right of a userform, put the following code in the user form's code module:

Private Sub UserForm_QueryClose(Cancel As Integer, CloseMode As Integer)
    Cancel = True
End Sub

Please note this prevents you from ever unloading the userform from memory. An Unload statement will be ignored if Cancel is set to True. This is OK if your procedure is not large and you are not displaying too many userforms, as the userform is automatically unloaded when all your procedures are done. However, if you are displaying many userforms or are expecting the
user form to be unload (and thus reset) when you issue the **Unload** statement you must use the following code instead:

**Private Sub** UserForm_QueryClose(Cancel As Integer, CloseMode As Integer)

    If CloseMode = vbFormControlMenu Then Cancel = True

**End Sub**

An even simpler approach is the following, which uses the numeric value of **vbFormControlMenu** instead of its name. Its value obviously is much easier to remember.

**Private Sub** UserForm_QueryClose(Cancel As Integer, CloseMode As Integer)

    If CloseMode = 0 Then Cancel = True

**End Sub**

Or, you can use the following approach, which is also simple:

**Private Sub** UserForm_QueryClose(Cancel As Integer, CloseMode As Integer)

    If Not CloseMode Then Cancel = True

**End Sub**

Lastly, if you want to run a routine to tell the user not to click on the X, then do this:

**Private Sub** UserForm_QueryClose(Cancel As Integer, CloseMode As Integer)

    If CloseMode = vbFormControlMenu Then

        Cancel = True

        MsgBox "Click on a button to close!"

    **End If**

**End Sub**

### 13.1.19 Displaying A UserForm Without A Blue Title Bar

If you remove a userform's blue title bar, then the user will not be able to move the userform nor click on the Quit/X button and close the form. Such a user form could be displayed as a task selection bar, as a modeless userform, or just as a normal userform but without the title bar. Since it is displayed without a title bar, there is no need to turn **Application.ScreenUpdating** back on in case the users moves the userform around.
The first step is to create a class module. Name the class module "cTitleBarHider" and put the following code in it:

```vbnet
Private Declare Function FindWindow Lib "USER32" _
Alias "FindWindowA" (ByVal lpClassName As String, _
ByVal lpWindowName As String) As Long
Private Declare Function GetWindowLong Lib "USER32" _
Alias "GetWindowLongA" (ByVal hWnd As Long, _
ByVal nIndex As Long) As Long
Private Declare Function SetWindowLong Lib "USER32" _
Alias "SetWindowLongA" (ByVal hWnd As Long, _
ByVal nIndex As Long, ByVal dwNewLong As Long) As Long
Private Declare Function DrawMenuBar Lib "USER32" _
(ByVal hWnd As Long) As Long

Private Const GWL_STYLE As Long = (-16)

' The offset of a window's style
Private Const WS_CAPTION As Long = &HC00000

' Style to add a title bar

Public Property Set Form(oForm As Object)
    Dim iStyle As Long
    Dim hWndForm As Long
    If Val(Application.Version) < 9 Then
        'XL97
        hWndForm = FindWindow("ThunderXFrame", oForm.Caption)
    Else
        'XL2000
        hWndForm = FindWindow("ThunderDFrame", oForm.Caption)
    End If
    iStyle = GetWindowLong(hWndForm, GWL_STYLE)
    iStyle = iStyle And Not WS_CAPTION
    SetWindowLong hWndForm, GWL_STYLE, iStyle
    DrawMenuBar hWndForm
End Property

Then, in the code module of any userform you wish to be titleless, put the following code:

Dim oTitleBarHider As New cTitleBarHider

Private Sub UserForm_Activate()
    Set oTitleBarHider.Form = Me
End Sub
```
Lastly, change the userform's border style property to 1-fmBorderStyleSingle in the userform's property box. This will improve the appearance of the userform when it is displayed.

When the form is displayed, the UserForm_Activate procedure will run and in turn run the code in the class module which hides the title bar.

13.1.20 Showing And Getting Values From A UserForm

It is easy to show a user form. Just use a statement like this in your code:

```
user form name.Show
```

for example: `UserForm1.Show`

However, closing the form is a bit more difficult. To close a form, you should put one of the following statements in the click event code of a button on the user form:

```
user form name.Hide
```

or

```
UnLoad user form name
```

There is a major difference between using the `Hide` method and the `UnLoad` method. If you use the `Hide` method on the user form, you can query the objects on the user form for their values. If you use `UnLoad`, then the user form is removed from memory and any settings or values entered by the user are lost. You should always unload the user form from memory using the `Unload` method once you have obtained the information you need from it. Failure to do so and then displaying other user forms will result in a severe memory drain on Excel and your code crashing.

The click event code of a button is displayed when you double click on a button. Doing so will take you to the user form's code sheet, and display lines like the following:

```
Private Sub CommandButton1_Click()

End Sub
```

If your user form is named UserForm1, the you would modify the code to look like the following:

```
Private Sub CommandButton1_Click()

UserForm1.Hide

End Sub
```
or

```
Private Sub CommandButton1_Click()
```
To determine which button on a form was clicked, you can

- declare a **Public** variable in one of your modules (but not in a user form's code module)

- Include a statement in the button's click event code that sets this variable to a value indicating which button was clicked.

For example, assume that you create two buttons on your dialog, one labeled "OK" and the other labeled "CANCEL". You could put statements like the following in the click event code to indicate which button was selected:

If the OK button, you could include one of the following:

```vbnet
buttonNumber = 1
bOkSelected = True
buttonSelected = "OK"

bCancelSelected = False
```

In the cancel button's click event code, you could include one of the following:

```vbnet
buttonNumber = 2
bOkSelected = False
buttonSelected = "Cancel"

bCancelSelected = True
```

Your code that calls the user form could then check the value of the variable to determine what action you should take.

The following code are examples of what the final code may look like. It assumes your buttons are named CommandButton1 and CommandButton2

In the main module:

```vbnet
Dim bCancelSelected As Boolean

Sub MyProcedure
    UserForm1.Show
    If bCancelSelected Then
```

UnLoad UserForm1
Exit Sub
End If

'code that queries the user form for values set while displayed

UnLoad UserForm1

'additional code

End Sub

In UserForm1's code module:

'For the OK button:

Private Sub CommandButton1_Click()
    UserForm1.Hide
    bCancelSelected = False
End Sub

'For the Cancel button:

Private Sub CommandButton2_Click()
    UserForm1.Hide
    bCancelSelected = True
End Sub

If you want to halt the macro if the cancel button is selected, then you can simplify the above and eliminate the need for a Public variable:

In the main module:

Sub MyProcedure
    UserForm1.Show

'code that queries the user form for values set while displayed

UnLoad UserForm1

'additional code

End Sub

In UserForm1's code module:

'For the OK button:

Private Sub CommandButton1_Click()
    UserForm1.Hide
    bCancelSelected = False
End Sub
For the Cancel button:

Private Sub CommandButton2_Click()
    End
End Sub

In the above, the End statement in the cancel button's code halts all activity, and the user form is unloaded automatically when code execute is terminated. All the OK button code does is to hide the form, which returns control to the calling routine. If control is returned, then it is obvious that the OK button was clicked and for the calling procedure to query the objects for any values set when the user form was displayed. Once these values are stored, the user form is unloaded, which allows other user forms to be displayed without potential memory problems.

13.1.21 Making A Userform the Size Of the Excel Window

The following code, placed in the userform's code module, will make the userform the same size as the Excel window:

Private Sub UserForm_Activate()
    With Application
        Me.Top = .Top
        Me.Left = .Left
        Me.Height = .Height
        Me.Width = .Width
    End With
End Sub

13.1.22 Showing A Userform For Just A Few Seconds

If you use the following code, the userform will stay displayed for just three seconds and then automatically disappear:

In the userform code module put the following:

Private Sub UserForm_Activate()
    Application.OnTime (
        Now() + TimeSerial(0, 0, 3)), "CloseUserForm"
End Sub

In a regular module, you would put the procedure CloseUserForm

Sub CloseUserForm()
    UserForm1.Hide
    Unload UserForm1
End Sub

13.1.23 Date Validation For UserForm TextBoxes

The following code, assigned to TextBox1 on a userform, will not allow the user to leave the text box if a date is not entered.
Private Sub TextBox1_Exit(ByVal Cancel As MSForms.ReturnBoolean)
If IsDate(TextBox1.Text) = False And _
    Len(TextBox1.Text) = 0 Then
    MsgBox "This is a date field only."
TextBox1.Text = ""
Cancel = True
End If
End Sub

13.1.24 Preventing A User From Closing Excel

To prevent a user from closing Excel, put the following code in the workbook code module (accessed by double clicking on the workbook object in the VBE project explorer. For example,

Private Sub Workbook_BeforeClose(Cancel As Boolean)
    Dim userResponse
    userResponse = MsgBox("Select OK to close Excel", vbYesNo)
    If userResponse = vbNo Then
        Cancel = True
    End If
End Sub

You can also use a class module to prevent a user from closing any workbook and from closing Excel. This avoids the need to have to put the above code in each new workbook and gives you better control. Here's a real simple example.

1. In the Visual Basic Editor (VBE), add a class module to your code workbook (Insert-->Class Module). Note the name of the class module (which is probably called Class1)...

2. In the code window that appears, type: Public WithEvents xlAppTrap as Excel.Application

3. At the top of the code window, you'll see two drop down boxes. Click the one on the left (should say General) and select: xlAppTrap

4. A private sub should appear on the module, called Private Sub xlAppTrap_NewWorkbook

5. Click the Drop-down on the right and select WorkbookBeforeClose. That procedure should now appear in the code window. You can delete the first one (xlAppTrap_NewWorkbook)

6. To prevent them from closing any workbook as well as Excel, simply enter one line of code

    Cancel = True

Finally, you need to 'activate' the class by defining and initializing a variable

7. At the top of the module where you keep your global variables (or any regular module), type the following

    Public clsAppTrap As New Class1
Where Class1 is the name of the class module you inserted in Step 1

8. In your Auto_Open or Workbook_Open procedure, add the following line of code

```vba
set clsAppTrap.AppTrap = Excel.Application
```

9. Once this code is run, the user will not be able to close any workbook, or exit the application, using neither the Excel or control menus. You'll need to add the following line of code to your Auto_Close (or any routine that attempts to close the app or workbooks):

```vba
Set clsAppTrap.AppTrap = Nothing
```

That line will disable the trap. Any routine that closes workbook(s) will need to use the code in step 9 before the close statement, then use the code in step 8 after the statement so that the trap is still active.

### 13.1.25 Changing The Names Of UserForm Objects

You can put text labels, edit boxes, list boxes, and many other objects on a userform. These objects are assigned names like "EditBox 1", EditBox2", "ListBox1", "Label1" and so forth. If you want to assign these objects names that are more descriptive and make it easier for you to identify them in your code do the following:

- Click on the object you wish to rename to select it

- Either press the F4 key or click on the properties button to display the properties window

- In the Name property, change the name to a descriptive name. You can not use spaces, but you can use underscores .

You should rename your userform and its objects before you write any code referring to the form or its objects. Doing so saves you the trouble of having to edit all such code if you rename the form or its objects after you've written your code.

For example, if you have changed the name of an edit box from "EditBox1" to "Last_Name", then the following code would return the value of the edit box and store in a variable:

```vba
UserForm1.Show
lastName = UserForm1.Last_Name.Text
UnLoad UserForm1
```

In the above example, the name of the user form is "UserForm1". The Show method displays the user form. When the user form is hidden (not unloaded) by a control on the form, the next
line is executed. Then the last line, the `Unload` statement, is executed to remove the userform from memory.

### 13.1.26 Showing Another UserForm From A UserForm

If you want to have a button on a userform display another userform, then you can do so by assign code like the following to the click event of the button:

```vba
Private Sub CommandButton1_Click()

'unload the userform that is displayed
Unload Me

'display another userform
UserForm2.Show
End Sub
```

Please note that unloading the userform removes it from memory and you can not get any of the settings or entries the user may have made on the form. To retain it in memory, do the following instead.

```vba
Private Sub CommandButton1_Click()

hide the userform that is displayed, but keep it in memory
Me.Hide

'display another userform
UserForm2.Show
End Sub
```

This allows you to redisplay the form if necessary. You should `Unload` a form once you are done using it to minimize the memory impact.

### 13.1.27 UserForms Sometimes Reset Module-Level Public Variables

If you declare variables for your userforms and use those variables to control the userform rather than using the userform’s name directly, the problem seems to go away.

For instance, say you have a userform called MyForm. Rather than do this:

```vba
MyForm.Show
Unload MyForm
```
Do this instead:

```vba
Dim frmMyForm As MyForm
Set frmMyForm = New MyForm
frmMyForm.Show
Unload frmMyForm
Set frmMyForm = Nothing
```

### 13.1.28 Unreliable Events with UserForms

There are certain conditions where the UserForm_Terminate event refused to fire. Most often, this occurs if the form is hidden and then unloaded. The best way to cure this problem is to treat it as if it were a normal class module.

```vba
Dim frmMyForm As UserForm1

'fires Initialize event
Set frmMyForm = New UserForm1

'Show and hide the UserForm as many times as you want in here.
frmMyForm.Show

'This *should* fire the terminate event.
Unload frmMyForm

'If not, this *definitely* will
Set frmMyForm = Nothing
```

### 13.1.29 RowSource Property Bug

The `RowSource` property for a listbox in is quirky at best. Another annoying thing is that if you hide the worksheet where the `RowSource` is located, or make the workbook into an add-in, all the `RowSource` data disappears. The best approach if possible is to stick with using `AddItem`, or loading the `List` property with an array.

### 13.1.30 Force User Form To Top Right Of Screen

If you place the following code in the userform's code module, it will display the form in the upper right hand corner of the Excel screen.

```vba
Private Sub UserForm_Activate()
    With Me
        .Left = Application.Width - .Width
        .Top = 0
    End With
End Sub
```
End With
End Sub

13.1.31 Userform Controls

The following illustrates how to iterate through the objects on a userform and determine if they are an option button. And if so, to take whatever action is desired.

Dim C As Control

For Each C In UserForm1.Frame1.Controls
  If Left(C.Name, 6) = "Option" Then
    'do whatever you want
  Else
    'do something else
  End If
Next

The following is another example of looping through the controls on a userform. The following would go in the userform's code module.

Private Sub UserForm_Click()
  For Each oC In Me.Controls
    If TypeName(oC) = "CheckBox" Then
      MsgBox oC.Caption
    End If
  Next
  For Each oC In Me.Controls
    If TypeOf oC Is msForms.CheckBox Then
      MsgBox oC.Caption
    End If
  Next
End Sub

13.1.32 Accessing A Userform From Another Workbook

Book1.xls contains a userform (userform1) with a button (commandbutton1) with an event procedure, commandbutton1_click:

Private Sub CommandButton1_Click()
  MsgBox "button clicked"
  Me.Hide
End Sub

The WB also contains a standard module with one procedure:

Sub showform()
'unload form if it is still in memory

Unload UserForm1
UserForm1.Show
End Sub

Book2 contains a standard module with the test procedure:

Sub testBook1()
End Sub

After saving Book1.xls and establishing a reference via Tools, References in the Visual Basic editor from Book2 to Book1. Running testBook1 results in userform1 popping up. Clicking on the sole button run the correct event procedure running.

What are the consequences of this mumbo-jumbo? First, unless the designer(s) of Book1 thought of - and accommodated - this type of access, the side-effects could, potentially, be disastrous. Second, this sneaks around an intended VBA feature. If MS decides to block this "ability" those exploiting it are on their own.

13.1.33 Iterating Through Objects In A Frame

The following illustrates how to iterate through a collection of text boxes that are contained in a userform frame using the For Each...Next statement

Dim ctl As Control
For Each ctl In UserForm1.Frame1.Controls
  If TypeOf ctl Is MSForms.TextBox Then
    MsgBox ctl.Name
  End If
Next ctl

13.1.34 Looping Through Controls On A Userforms

The easy way it is to fully qualify the object you're looking for, telling VBA exactly which object library you want it to look in. This not only solves your problem but makes for faster code as well.

Private Sub UserForm_Initialize()
Dim ctl As Control
For Each ctl In Me.Controls
  If TypeOf ctl Is MSForms.CheckBox Then

""More code here.

  ElseIf TypeOf ctl Is MSForms.TextBox Then

""More code here.
13.1.35 Passing Values From A Userform To A Sub

First, declare a public variable at the top of a normal module. For example

```vba
Public bResponse As Boolean
```

Let's assume you have two buttons on the form, one an OK button and one a Cancel button. If you double click on the OK button in the form editor, it will display the button click code. In that subroutine put

```vba
bResponse = True
Me.Hide
```

Repeat on the Cancel button but put

```vba
bResponse = False
Me.Hide
```

When the form closes, check bResponse to see if the user clicked OK or Cancel. If they clicked Cancel, you will probably want to unload the form and exit your subroutine.

13.1.36 Useful Internet Articles On UserForms And DialogSheets

Here are some sources which might prove helpful on creating userform and dialogsheets.

- [http://support.microsoft.com/support/kb/articles/q164/9/23.asp](http://support.microsoft.com/support/kb/articles/q164/9/23.asp)

  How to Fill a UserForm ListBox with Database Values

- [http://support.microsoft.com/support/kb/articles/q161/5/98.asp](http://support.microsoft.com/support/kb/articles/q161/5/98.asp)

  XL97: How to Add Data to a ComboBox or a ListBox

- [http://support.microsoft.com/support/kb/articles/q183/1/83.asp](http://support.microsoft.com/support/kb/articles/q183/1/83.asp)

  XL98: How to Fill ListBox Control with Multiple Ranges (works for XL97 too)

- [http://support.microsoft.com/support/kb/articles/q165/5/70.asp](http://support.microsoft.com/support/kb/articles/q165/5/70.asp)

  XL97: How to Use the TextColumn Property
13.2 MULTIPAGE CONTROL

13.2.1 Specifying The Starting Page In A MultiPage Control

The default starting page on a MultiPage control is the first page. You can change pages by setting the Value property of the MultiPage control. The page indices start at 0, so to switch from the first page to the second page you do

```
MultiPage1.Value = 1
```

13.2.2 Setting The Displayed Page Of A MultiPage UserForm Object

The Value property of a MultiPage object defines the active page, where 0 is the first page, 1 is the second page and so forth. To activate the second page use a statement like:

```
Userform1.MultiPage1.Value = 1
```

To open on the first page:

```
Userform1.MultiPage1.Value = 0
```

13.2.3 How To Add Additional Pages To A MultiPage Tab In A UserForm

The default is 2 pages for a MultiPage tab on a userform. Right Click on the MultiPage and select "New Page" to add additional pages.

13.2.4 Activating Page On A UserForm's MultiPage
Use the **Value** property to specify which page should be displayed:

```vbnet
.MultiPage1.Value = 2
```

'The page value is zero based: page 1 = 0. The above code activates page 3

### 13.3 BUTTONS AND CHECKBOXES

#### 13.3.1 Putting OK and Cancel Buttons On UserForms

To put OK and Cancel buttons on your userform, do the following:

- Draw two command buttons on the form
- Change the text in the buttons to OK and Cancel
- Double click on one of the buttons to get to the userform’s code module
- Assuming that the OK button is named CommandButton1 and the Cancel button is named CommandButton2, put the following code in the userform’s module:

```vbnet
Private Sub CommandButton1_Click()

    'hide the userform so that code execution will continue
    UserForm1.Hide

    'set bContinue to indicate that the OK button was selected
    bContinue = True

End Sub

Private Sub CommandButton2_Click()

    'hide the userform so that code execution will continue
    UserForm1.Hide

    'set bContinue to indicate that the Cancel button was selected
    bContinue = False

End Sub
```

- In a regular module declare a public variable at the top of the module with the following statement

```vbnet
Public bContinue As Boolean
```
Put the following code in the module to display the form, assuming it is named UserForm1

```vba
Sub Using_OK_Cancel_Butons()

  'display the userform
  UserForm1.Show

  'test the value of bContinue to determine which button was selected
  If bContinue Then
    MsgBox "OK selected"
  Else
    MsgBox "Cancel selected"
  End If

  'remove the form from memory when done using
  Unload UserForm1

End Sub
```

13.3.2 How To Associate Code With A Button On A User Form

When you double click on a userform button while in the VBA editor, the editor will automatically create and display the 'click event' subroutine for that button. If you have not assigned a name to the button, you will see something like 'command button1 Click'. You can enter any code you wish within the click event, or call up another subroutine. The code will execute when the button is 'clicked' once you run your program. That's all there is to it!

For other controls on a userform, double click on them will display the default procedure associated with the control. You can then select other procedures that are associated with a given control from the drop downs that are displayed on the userform's code module.

If the code window doesn't come up when you first double click on it, click off the control and then double click on the control again.

In summary:

- Create the button
- Use the properties window to give it a good name
- Right click on the button and choose the View Code command. Code like the following will appear

```vba
Private Sub CommandButton1_Click()

End Sub
```
13.3.3 Making Buttons On UserForms Do What You Want

When you double click on the control while in the VBA editor, the editor will automatically create and display the 'click event' subroutine for that button. If the code window doesn't come up, click off the control and then double click on the control again.

For a button, the click event is the default and will be in the windows. Additional events are in the upper right dropdown. If you have not assigned a name to the button you will see something like 'command button1 Click'. You can enter any code you wish within the click event, or call up another subroutine. The code will execute when the button is 'clicked' once you run your program.

13.3.4 Grouping Option Buttons With or Without a Frame

You can group option buttons together by including them within a frame. All of the option button must be within the frame. Only one button within the frame can be on at any time. All buttons outside of a frame act as a group, with only one button being active.

You can also assign option buttons to groups. This allows buttons to act as if they were in a frame, without the frame. Only one button in a group can be active at a time.

To assign buttons to a group, right click on each option button and select properties. Then assign a unique name for each set of buttons in the group name property.

13.3.5 How To Check How Many CheckBoxes Are Clicked

Here's one way to do it (this code goes in the userform's code module):

```vba
Private Sub CommandButton1_Click()
    Dim ctlControl As Control
    Dim lNumChecked As Long
    For Each ctlControl In UserForm1.Controls
        If TypeOf ctlControl Is MSForms.CheckBox Then
            If ctlControl.Value Then
                lNumChecked = lNumChecked + 1
            End If
        End If
    Next ctlControl
    MsgBox Cstr(lNumChecked) & " checkboxes were checked."
End Sub
```

13.4 USING THE REFEREDIT CONTROL

13.4.1 Using The RefEdit Control On A Userform

The RefEdit control provides the functionality to fill an input box with the address of a range by clicking a worksheet and selecting the range.
To use the RefEdit control in your project, click References on the Tools menu, check Ref Edit Control, and then click OK. The RefEdit control appears on the toolbox in the Visual Basic Editor.

You have to be very careful with a RefEdit control. Avoid putting it in a container object (such as a Frame or MultiPage). If you do so, you run the risk of crashing Excel.

To validate input into a RefEdit control, you can use a VBA function like this:

```vba
Function IsRange(ref As String) As Boolean
    Dim x As Range
    On Error Resume Next
    Set x = Range(ref)
    If Err = 0 Then IsRange = True Else IsRange = False
End Function
```

Before the form is closed, execute some validation code attached to your OK button:

```vba
If Not IsRange(RefEdit1.Text) Then
    MsgBox "Invalid range."
    RefEdit1.SetFocus
    Exit Sub
End If
```

This will not allow an invalid range to be specified.

The following URL is an article on the MS Knowledge Base which describes the RefEdit control and provides sample code on how to use it:

http://support.microsoft.com/support/kb/articles/q158/4/02.asp

XL97: Using the RefEdit Control with a UserForm

Please note that there are problems with the RefEdit control. **If the worksheet is maximized, then only ranges from the active workbook can be selected.** Workarounds are:

- set all workbooks to a windowed state.
- Or, tell the user to select a cell on the active workbook, and then press CTL-TAB to cycle through the open workbooks.
- Another solution is to avoid the RefEdit control. Rather, use a Label with a button next to it. The button's caption is something like "Specify a Range." Clicking the button runs a sub that hides the userform and displays Excel's Application.InputBox, set up so the user can select a range (type = 8). Then, the code transfers the selected address to the Label.

### 13.4.2 Using A Ref Edit Form On A User Form To Select A Range
The following is another example of using a ref edit form.

If you need to have the user select a cell or a range of cells for you, you can do so using a ref edit box on a user form. The first step is obviously to create a user form. Do this by selecting in the Visual Basic editor Insert, User Form.

Next, click on the ref edit button of the toolbox controls and draw a ref edit box on the user form. The default name should be RefEdit1. Verify this by pressing F4 with the ref edit box selected. If it is different, then you will need to modify the code below accordingly.

Next, draw two command bar buttons on the sheet and change their labels to "OK" and "CANCEL". Select the OK button and press F4 to display the properties window. Change the name of the button from "commandbar1" to "OK". Select the Cancel button and press F4 to display its property window. Change its name to "Cancel".

Select the OK button and then double click on it. The following code will appear in a module with a name like "UserForm1 (code)"

```
Private Sub OK_Click()
End Sub
```

Modify this code to have the following line in it, assuming that the name of your user form is "UserForm1". If the name of your user form is different, use that name instead. This new line hides the user form when the button is clicked.

```
Private Sub OK_Click()
UserForm1.Hide
End Sub
```

Next, select the Cancel button on the user form. Then double click on it and modify the code that appears to be the following. If the name of your user form is different, use that name instead.

```
Private Sub Cancel_Click()
Unload UserForm1
End
End Sub
```

The **Unload** line removes the userform from memory. The **End** line halts all macro activity. If you do not want to halt activity, then set a Boolean Public variable to **False** and then have the routine that displayed the user form check this variable to determine if OK or Cancel was selected. In the OK button code, set the public variable to True.

Now copy the following code to a module in the workbook containing the user form. It should not be in the user form module. Please note that the above code and the following code assumes that your form is named "UserForm1". If it is a different name, then use that name instead.

```
Sub Get_A_Range()
    Dim selectedRange As Range

    'display the user form

```
UserForm1.Show

'set a variable to the range selected.

    Set selectedRange = Range(UserForm1.RefEdit1.Text)
    'code that uses the selected range
End Sub

The above code sets the range variable "selectedRange" to the range selected in the user form. If you want to demonstrate that this was done, use the following line of code:

    Application.Goto selectedRange, True

To select a range in a different workbook, the user will need to press CTRL-Tab if the current window is maximized. As they probably won't figure this out, you should make a comment to this effect on the userform.

13.4.3 Using Reference EditBoxes on DialogSheets

An alternate to the userform Ref Edit Control is an editbox on a dialogsheet. This has one very large advantage over a ref edit control: You can select a range on any worksheet or workbook from a dialogsheet's editbox.

To select a range in an editbox in a dialogsheet, you must first set the editbox's validation option to "reference". The validation options can be displayed by clicking on the Control Options button of the Forms toolbar. To assign this selection to a range variable for later use, use a statement like the following:

    Dim rangeToUse As Range
    Dim myDialog As DialogSheet
    Dim refBox As EditBox

    'use object variables to refer to the dialogsheet and edit boxes
    Set myDialog = ThisWorkbook.DialogSheets("Dialog1")
    Set refBox = myDialog.EditBoxes("Edit Box 4")

    'clear the edit box in case it has a prior entry
    refBox.Text = ""

    'loop until a selection is made or cancel selected
    While refBox.Text = ""

    'display the dialog, exit if cancel selected
    If Not myDialog.Show Then Exit Sub
    If refBox.Text = "" Then _
MsgBox "You did not specify a range"
Wend

'assign the range selected to a range variable

Set rangeToUse = Range(refBox.Text)

13.4.4 Sample Code On Using The RefEdit Box

A RefEdit control is an edit box on an user form that allows one to pick a range of cells. If it is not displayed on your toolbox commandbar, then click References on the Tools menu, check Ref Edit Control, and then click OK. The RefEdit control will then appear on the toolbox in the Visual Basic Editor.

To have a user select a range from a userform, you must put a RefEdit box on the userform. You must also put command buttons on the userform that act as OK and Cancel buttons. The following is code like the above that displays the userform and assigns the selected range to a range variable:

Dim rangeToUse As Range

'use a with statement to simplify the code. Note the periods in
'front of RefEdit1

With UserForm1

'clear the edit box in case it contains an entry

.RefEdit1.Text = ""

'loop until the cancel button selected or an entry made and
'the OK button selected

While .RefEdit1.Text = ""

' set focus to the refedit box as a loop back will put the focus
'on the button

.RefEdit1.SetFocus

'display the user form

UserForm1.Show

'if no selection, display a message

If .RefEdit1.Text = "" Then
    MsgBox "Please select a range"
'companion statement the While statement

    Wend
End With

Set rangeToUse = Range(UserForm1.RefEdit1.Text)

'remove userform from memory

Unload UserForm1

Please note on the userform above, you need two buttons that act as OK and Cancel buttons. When you draw the buttons, they will have labels like "Commandbutton1". You can edit this text to be OK or Cancel. You then need to double click on the buttons to display the userform's code module, and put the following code in for each button:

Private Sub CommandButton1_Click()

' the OK button, which hides the form and allows execution to continue

    UserForm1.Hide
End Sub

Private Sub CommandButton2_Click()

' the cancel button. In this case an End statement is used which
' halts macro execution

    End
End Sub

The option to use an End statement in the second button is to hide the form, set a global variable indicating which button was selected, and then in the code testing the value of the global variable and determining what action to take.

The following URL is an article on the MS Knowledge Base which describes the RefEdit control and provides sample code on how to use the RefEdit control:

http://support.microsoft.com/support/kb/articles/q158/4/02.asp

XL97: Using the RefEdit Control with a UserForm

13.5 LABELS AND TEXTBOXES

13.5.1 An Example Of Using A UserForm With A TextBox

The following illustrates how to obtain the user's textbox entry on a userform. The key is that you hide your userform rather than unload it, the values and results of the userform remain set and addressable by your code module. The following code provides a general approach, for a userform with two buttons and a text box.
'In the userform code module put the following for your 1st button
'which should be labeled OK

Private Sub CommandButton1_Click
    Me.Hide
End Sub

'Label the other button "Cancel" and put the following code in it, which halts all activity

Private Sub CommandButton2_Click
    Me.Hide
End sub

In your general code module:

Sub Test()
    Dim myData
    userform1.Show

    'Store the value on the userform
    myData = UserForm1.TextBox1.Text

    'now unload the form
    UnLoad UserForm1

    ' remainder of code
End Sub

13.5.2 Highlighting Entry In A Userform TextBox

In the Enter event in the text box control, placing the following code (in this case for TextBox1) will select the text in the text box when the user clicks in it.

Private Sub TextBox1_Enter()
    TextBox1.SelStart = 0
    TextBox1.SelLength = Len(TextBox1.Text)
End Sub

The above code goes into the userform's code module.

13.5.3 How To Select The Entry In A TextBox

The simplest way to force a specific control to have focus when you show a UserForm is to set that control's TabIndex property to 0 (top of the tab order). This only applies to controls sited directly on the userform. It won't work if the control is placed within another container, like a Frame or MultiPage control. Select View, Tab Order to set the tab order on a userform.
To have the text in a textbox elected when the userform is selected, use code like the following:

```vba
With UserForm1.TextBox1
    .SelStart = 0
    .SelLength = Len(.Text)
End With
```

The textbox should be at the top of the tab order.

If the text box has the focus, then the above will select the text that is in the box. For example

```vba
Sub SelectTextInTextBox()
    UserForm1.TextBox1.Text = "Some Text"
    UserForm1.Show
End Sub
```

You could also assign the code to a button's code so that when the button is clicked, the focus is transferred to the text box and the text in the text box selected. One additional line is needed, a set focus line:

```vba
Private Sub CommandButton1_Click()
    With UserForm1.TextBox1
        .SelStart = 0
        .SelLength = Len(.Text)
        .SetFocus
    End With
End Sub
```

13.5.4 How To Clear and Set TextBox Entries

You must refer to each textbox individually to clear or set their entries. For example,

```vba
UserForm1.TextBox1.Text ="
```

If your textboxes are named TextBox1, TextBox2, etc, you can use the following technique instead:

```vba
With UserForm1
    For i = 1 To 3
        .Controls("textbox" & i).Text = ""
    Next
End With
```

If the Textbox is used to get a range reference, and you want to set it to the current selection on the sheet then do the following:

```vba
UserForm1.RefEdit1.Text = Selection.Address
```

13.5.5 Cursor Position In A UserForm TextBox
You can use the `SelStart` property to set a textbox in a userform so that the cursor goes to the beginning and not to the end of text when the userform is activated:

```vba
Private Sub UserForm_Activate()
    UserForm1.TextBox1.SelStart = 0
End Sub
```

### 13.5.6 How To Format A Number On A Label In A UserForm

If you have a label on a user form and you want to change the text to a formatted value, you can do so using the `Format()` function. The following illustrates the syntax to use if you wanted to format the number 1000 to appear as $1,000.00 in a label and have the phrase “Amount paid” prefixing the number:

```vba
UserForm1.Label1.Caption = _
"Amount paid: " & Format(1000, "$#,##0.00")
```

If the value to be placed in the user form is first obtained and stored in a variable, then you can replace the 1000 with the name of the variable.

### 13.5.7 Multiple TextBoxes with Same Validation

Here is an example of how to trap the Change event for all text boxes on a UserForm so that you can apply the same validation tests to the entry. First create a class module and name it `clsControlEvents`. Enter the following code:

```vba
Public WithEvents txt As MSForms.TextBox

Private Sub txt_Change()
    If Len(txt.Text) > 6 Then
        MsgBox "Do not enter more than 6 characters", vbCritical
        txt.Text = Left(txt.Text, 6)
    End If
End Sub
```

As soon as you have entered the first line, you can use the drop downs at the top of the class module to generate the event procedure's first and last lines. Unfortunately, text boxes do not expose their most useful events in the class module (Enter, Exit, BeforeUpdate, AfterUpdate), but you can get to the Change event.

In the userform class module, enter the following code:

```vba
Dim colTextBoxes As New Collection

Private Sub UserForm_Initialize()
    Dim ctl As MSForms.Control
    Dim ctlEvents As clsControlEvents

    For Each ctl In Me.Controls
        If TypeOf ctl Is MSForms.TextBox Then
```
This code assigns instances of the class module to the text boxes in the UserForm and stores the instances in a collection.

### 13.5.8 Formatting Textbox Entries

It is possible to format a numeric entry in a text box. The **BeforeUpdate** event is the best way to do this, but the format is not applied until the textbox loses focus.

To try this, enter code like the following in the userform's code module (reached by double clicking the textbox). Please note that the name of this subroutine is dependent on the name of your text box. If you are uncertain, select the text box from the left drop down and the **BeforeUpdate** event from the right drop down when you are on the userform's code sheet. This will create first and last lines of that event's code.

```vba
Private Sub TextBox1_BeforeUpdate( Val Cancel As MSForms.ReturnBoolean)
    TextBox1.Text = Format(TextBox1.Text, "$#,##0.00")
End Sub
```

When you try the above code, pressing tab or enter after making an entry formats the text box.

### 13.5.9 Formatting Textboxes on UserForm

The textbox only contains a string. You can format the text in the textbox by using the VBA format function and putting the following in the UserForm code module.

```vba
Private Sub TextBox1_Exit(ByVal Cancel As MSForms.ReturnBoolean)
    TextBox1.Text = Format(TextBox1.Text, "$ #,##0.00")
End Sub
```

### 13.5.10 Formatting Numbers In A UserForm Textbox

The following statement shows how to format an entry in a userform's textbox to a given number of decimal places using the **Format** function.

```vba
UserForm1.TextBox1.Text = Format(30, "0.00")
```

The above statement will display "30.00" in the textbox.

### 13.5.11 Bulk Clearing Of Text Boxes

If you have many text boxes on a userform, the following is an easy way to clear all of them:
Sub ClearTextboxes()

Dim CurrCtrl As Control
For Each CurrCtrl In UserForm1.Controls
  If TypeName(CurrCtrl) = "TextBox" Then
    CurrCtrl.Text = ""
  End If
Next
End Sub

If you wanted to clear just some of them perhaps you could use their Tab Indexes. Something like if TaxIndex > X and < Y.

13.5.12 Validating UserForm Textbox Entries

You can assign code to a userform's textbox's exit event to check the entry. If the entry is not valid, then you can retain the focus in the text box so that the user can not leave the text box until a valid entry has been made. The following illustrates this, which requires the user to type either A or B in the textbox, and accepts no other entry:

Private Sub TextBox1_Exit(ByVal Cancel As MSForms.ReturnBoolean)
  Dim sEntry As String

  'store the textbox entry in a variable for later use
  sEntry = UCase(TextBox1.Text)

  'check to see if the entry is valid
  If Not (sEntry = "A" Or sEntry = "B") Then

    'if the entry is not valid, display a message
    MsgBox "Your entry was incorrect"

    'if the entry is not valid, set the Cancel variable to True so that
    'the user can not leave the edit box
    Cancel = True
  End If
End Sub

If you need to use the same code to validate many textboxes, then you can do so by using code like the following:

Private Sub TextBox1_Exit(ByVal Cancel As MSForms.ReturnBoolean)

  'call the function that validates the text box, and pass the text box
  'entry to the function, which will return either True or False
  'if False returned, do not allow the user to exit the textbox

  'call the function that validates the text box, and pass the text box
  'entry to the function, which will return either True or False
  'if False returned, do not allow the user to exit the textbox

244
If Not ValidateEntry(TextBox1.Text) Then Cancel = True
End Sub

Private Sub TextBox2_Exit(ByVal Cancel As MSForms.ReturnBoolean)
    'call the function that validates the text box, and pass the text box
    'entry to the function, which will return either True or False
    'if False returned, do not allow the user to exit the textbox
    If Not ValidateEntry(TextBox2.Text) Then Cancel = True
End Sub

Function ValidateEntry(ByVal anyString As String) As Boolean
    'this function returns True if the entry is an A or B, False otherwise
    'the text box entry is passed ByVal to keep the following code from
    'changing the supplying variable's value to upper case
    'convert the string to upper case
    anyString = UCase(anyString)
    'see if the entry is OK
    If Not (anyString = "A" Or anyString = "B") Then
        'if the entry is not valid, display a message
        MsgBox "Your entry was incorrect"
        'if the entry is not valid, set the function to False and exit the function
        ValidateEntry = False
        Exit Function
    End If
    'if execution gets to here, the entry is valid, so pass back a True value
    'to the function
    ValidateEntry = True
End Function

13.5.13 Validating UserForm TextBox Input

The easiest way I have found is to do something like the following:

Do
    myform.Show

End Do
'code or subroutine that validates the entries and
'exits the loop if OK
Loop

You could put code in your userform that validates it when the exit button is clicked. However,
that frequently has one displaying a MsgBox on top of a userform, which is confusing.

13.5.14 Validating A TextBox Entry As A Number
Visual Basic does not provide any validation for userform text box entries. Thus, you need to
use code to prevent a user from entering non-numeric entries when a number is required. The
following code will validate an entry and only allow numeric entries:
'pass to this sub the textbox object (see example below)
Sub Validate_Number_Entry(oBox)
Dim tempS As String

'store the text in a variable for later use
tempS = oBox.Text

'if user has removed all entries, just exit
If tempS = "" Then Exit Sub

'if the entry is numeric, then exit
If IsNumeric(tempS) Then Exit Sub

'if the entry is not numeric, remove the last entry
oBox.Text = Mid(tempS, 1, Len(tempS) - 1)
End Sub

In the userform's code module, assuming that your text box is named textBox1, add the
following code:
Private Sub TextBox1_Change()

'call the validate routine and pass to it the textbox object
Validate_Number_Entry Me.TextBox1
End Sub

If you need to validate the entries and only allow two entries to the right of the decimal or need
to further validate the numeric entry, then use code like the following:

246


'pass to this sub the textbox object (see example above)

Sub Validate_Number_Entry(oBox)
    Dim tempS As String

    'store the text in a variable for later use
    tempS = oBox.Text

    'if user has removed all entries, just exit
    If tempS = "" Then Exit Sub

    'if the entry is numeric, check for a decimal
    If IsNumeric(tempS) Then
        If InStr(tempS, ".") > 0 Then

            'if a decimal found check entry count following it
            If Len(tempS) - InStr(tempS, ".") > 2 Then

                'if more than two entries, remove the third and advise user
                MsgBox "Only two entries are allowed to " & _
                    
                        "the right of the decimal place"
                oBox.Text = Mid(tempS, 1, Len(tempS) - 1)
            End If
        End If
    End If

    'exit sub as this is the end of the numeric testing
    Exit Sub
End If

'if the entry is not numeric, remove the last entry
oBox.Text = Mid(tempS, 1, Len(tempS) - 1)
End Sub

13.5.15 Automatically Adding Hyphens To Phone Numbers In Text Box

You can automatically add hyphens to an entry in a text box as the user is typing in a number. Assuming that your textbox is named TextBox1, you can place the following code in the userform's code module. It checks the entry for hyphens, and adds as needed the code checks for hyphens, and inserts new ones if necessary, to allow for editing of a previous entry

Private Sub TextBox1_Change()
    Dim txt As String

Dim J As Integer
Dim I As Integer
Dim c
Dim hold

'get the entry in the textbox

txt = TextBox1.Text
J = 1
For I = 1 To Len(txt)

'extract character at position i

c = Mid(txt, I, 1)
If J = 4 Or J = 8 Then
    If c = "-" Then

        'if already a "-" just add to hold string
        hold = hold & c
        J = J + 1
    Else

        'if not a "-" add a hyphen and increment J by 2
        hold = hold & "-" & c
        J = J + 2
    End If
ElseIf c <> "-" Then

    'if character position not 4 or 8, just add character to hold string
    hold = hold & c
    J = J + 1
End If
Next I

'if the length is 3 or 7, add a hyphen to the end

If Len(hold) = 3 And Right(hold, 1) <> "-" Then _
    hold = hold & "-"
If Len(hold) = 7 And Right(hold, 1) <> "-" Then _
    hold = hold & "-"

'update the text in the textbox

TextBox1.Text = hold
End Sub

13.5.16 Forcing A Textbox to Accept Only Numbers
Code like the following in a userform's code module will force a textbox to accept only numbers or the negative sign:

 Declare this at the top of the module
 Dim previousEntry As String

 Private Sub TextBox1_Change()
   TextBox1.Text = Trim(TextBox1.Text)
   If Not IsNumeric(TextBox1.Text) And _
   TextBox1.Text <> "-" And _
   TextBox1.Text <> "" Then
     TextBox1.Text = previousEntry
   End If
 End Sub

 Private Sub TextBox1_KeyDown(ByVal KeyCode As MSForms.ReturnInteger, ByVal Shift As Integer)
   previousEntry = TextBox1.Text
 End Sub

13.5.17 Reading A Date From A Textbox

If you have a user type in a date into a textbox, you can read it as a date using a statement like the following:

 Dim inputDate As Date
 InputDate = CDate(UserForm1.TextBox1.Text)

If you want to display a date in the textbox as an initial suggestion for the user, use a statement like the following:

 UserForm1. = Format(ActiveCell.Value, "mm/dd/yy")

13.6 COMBO, DROPDOWN, AND LIST BOXES

13.6.1 ListBox Differences

There are two list box controls in Excel. There is the old list box, associated with the Forms toolbar and dialogsheets, and the new list box associated with the Control Toolbox toolbar and userforms. You can see both in the Object Browser window if you right click the Classes list and choose "Show hidden members".

The old listbox has a RemoveAllItems method. The new listbox has a Clear method,

13.6.2 Populating A ComboBox or ListBox With External Data
The following articles discuss filling list boxes and combo boxes with data from an external source such as Access. The first article addresses the issue specifically (list box / combo box - same approach) and the second gives you access to a list of multiple articles on using DAO. The first article pertains to the XL95 style list box, but can be easily adapted to XL97 style.

http://support.microsoft.com/support/kb/articles/Q149/2/54.asp

XL7: How to Return DAO Query Results Directly to a List Box

http://support.microsoft.com/support/excel/dao.asp

Using Data Access Object (DAO) in a Microsoft Excel Macro

13.6.3 Populating A List Box With Unique Entries

IF you wish to populate a listbox or a combobox with just the unique entries in a range, then use code like the following:

```vba
Sub PopulateListWithUniqueItems()
    Dim anyR As Range
    Dim listCollection As New Collection
    Dim cMember
    Dim I As Integer
    Dim J As Integer
    Dim lCount As Integer
    Dim cell As Range
    Dim tempS
    Dim myList()
    'in this example, the entries are assumed to be in
    'cells A1 to A20.
    Set anyR = Range("A1:A20")
    'on error must be set as an error is created
    'when a duplicate item is added to the listCollection
    'and we want the macro to continue
    On Error Resume Next
    For Each cell In anyR
        If Not IsEmpty(cell) Then
            listCollection.Add cell.Value, CStr(cell.Value)
        End If
    Next
    'turn off error handling
    On Error GoTo 0
    'assign to a list for easier use
    lCount = listCollection.Count
    ReDim myList(1 To lCount)
    For Each cMember In listCollection
        I = I + 1
```
myList(I) = cMember
Next

'sort the list
For I = 1 To lCount - 1
    For J = I + 1 To lCount
        If myList(I) > myList(J) Then
            tempS = myList(I)
            myList(I) = myList(J)
            myList(J) = tempS
        End If
    Next
Next

'assign to listbox
UserForm1.ListBox1.List = myList

display form
UserForm1.Show
End Sub

13.6.4 Assigning A Range To A ListBox

There are many different ways to assign a range to a listbox. You can assign the range when you create the list box or have your code assign the list to the box.

If you want to use a fixed range on a worksheet for the list of a listbox, then you can set the range for the list by setting it as a property of the listbox

◆ Click on the listbox

◆ Press F5 to display the properties menu

◆ In the RowSource property, you would type in the sheet and range. Enclose the sheet name in single quotes, and then follow it with an exclamation point before typing in the cell range:

'My Lists'!A1:A10

If you want to have your code assign the range to the listbox, then you can do the following:

Use statements like the following, which set the RowSource property of the listbox to the full address of the range to use for the list:

UserForm1.ListBox1.RowSource = Worksheets("sheet1")_('a1:a10').Address(external:=True)

You can also do the following:

Dim X As Variant
'set a variant variable equal to a range, which makes the variant variable 'an array

X = Worksheets("sheet1".Range("a1:a10")

'set the list property by assign it to the value of X

UserForm1.ListBox1.List = X

' or

UserForm1.ListBox1.RowSource = X.Address(External:=True)

You can use the AddItem property to assign a list to a listbox. The following illustrates this by populating a list with a list of the open workbooks

    Dim lBox
    Dim wb As Workbook
    Set lBox = UserForm1.ListBox1

    ' cycle through the open workbooks

    For Each wb In Workbooks

    ' add the name of the workbook to the list

        lBox.AddItem wb.Name

    Next

You can not use both the RowSource property and the AddItem approach for the same list.

If you need to remove an item from a list box, you can do so with the RemoveItem property. The following removes all of the items in a list box:

Note that the list index starts at 0:

    Set lBox = UserForm1.ListBox1

    ' remove all items one at time, working backward through the list

    For I = lBox.ListCount To 1 Step -1
        lBox.RemoveItem (I - 1)
    Next

The last way to set the list for a list box is to first create an array, and then assign the list property of the listbox to that array

    UserForm1.ListBox1.List = SomeArray()
13.6.5 Linking A List Box On A UserForm To Cells On A Worksheet

To specify the data to be shown in the list box, you can manually type a sheet and cell or range address in the RowSource property of the listbox. With the listbox selected, press F4 in the VBE window to see the properties window or right click the listbox and choose Properties to display the properties windows. Unlike dialogsheets, you can't point to the range. It must be typed as an external reference such as:

Sheet1!A1:A12

If the sheet name has spaces, included the sheet name in single quotes:

'My Sheet'!A1:A12

If you want a link cell in the worksheet, define the ControlSource property in a similar way. Unlike dialogsheets, the cell will display the selected value of the list, not the list's index number.

A listbox also has a ListFillRange property. However, the ListFillRange property of a listbox is only available when the listbox is located on the worksheet. When the control is on a userform, you must use the RowSource property. They work the same.

13.6.6 Filling A Listbox With Month Names

The following code illustrates how to fill a listbox with the names of the month

Sub FillListBoxWithMonths()
    Dim I As Integer
    Unload UserForm1
    With UserForm1
        For I = 1 To 12
            .ListBox1.AddItem Format((I * 30) - 15, "MMMM")
        Next
        .Show
        MsgBox .ListBox1.ListIndex
        MsgBox .ListBox1.Value
    End With
End Sub

13.6.7 Determining What Is Selected In A ListBox

There are a number of ways to determine what is selected in a listbox. If the cell that is assigned the index number from the list box is A1 on sheet "sheet1" of "myworkbook.xls" then you can do this

Dim listNum As Integer
listNum = _
    Workbooks("myworkbook.xls") .Sheets("sheet1") .Range("A1")
If the cell has been assigned a range name, you can use the range name instead of the cell reference.

Also, you can get the list box index value and the name of the item in the list box without using a cell reference. For example, assuming that the dialogsheet is in the same workbook as your macro code:

```vba
numOfItemSelected = _
    ThisWorkbook.Dialogsheets("my dialog") ._
    .ListBoxes("listbox name").Value

TextOfItemSelected = ThisWorkbook _
    .Dialogsheets("my dialog") _
    .ListBoxes("list box name").List(numOfItemSelected)
```

### 13.6.8 Determining What Was Selected In A Multi-Select List Box

Since a multi-select list box can have numerous items selected, a single cell cannot contain the number(s) of the selected item(s). Instead you will need to write some code to handle the list.

In a listbox the property ListCount returns the number of items in a list. The index numbers start at zero and go to the ListCount value minus one. As an example, a ListBox contains 3 items and the ListCount is 3. The index numbers are 0, 1, and 2.

The following shows how to determine what was selected in a multi-select list.

```vba
For i = 0 To ListBox1.ListCount - 1
    If ListBox1.Selected(i) Then
        'the selected value is true if the item is selected
        MsgBox ListBox1.List(i) & " selected"
    End If
Next i
```

### 13.6.9 Auto Word Select In ComboBoxes

ComboBoxes will automatically word select to the first matching entry if the combobox AutoWordSelect property is set to True (its default). Setting it to False will eliminate AutoFill.

### 13.6.10 How To Make A ComboBox A Dropdown Box

The toolbox set of controls not provide a dropdown box. Instead, it provides a combination edit / dropdown box that can be set to be just a dropdown box. You do this by selecting the combobox, displaying the properties window, and then setting the MatchRequired property to True.
13.6.11 How To Make A ComboBox Be Just A Drop Down ListBox

The default setting for an ComboBox allows the user to either select from the list or to type an entry into the edit box. To turn the ComboBox into just a drop down list box and disable the edit box capability do the following:

- select the ComboBox
- press F5 to display the properties dialog
- change the Style setting to fmStyleDropDownList

Now, Any typing is funneled towards looking for a match with an existing entry

13.6.12 Removing the Selection From A ComboBox

One of the problems with combo boxes is that it is difficult to blank or remove the previous selection from appearing in the combo box. The following statements will do that:

```
With UserForm1.ComboBox1
  .AddItem ""
  .ListIndex = .Listcount-1
  .RemoveItem .Listcount-1
End With
```

13.6.13 Have UserForm ComboBox Drop Down When It Is Selected

If the ComboBox is on a userform, then do the following to have the dropdown drop down when the userform appears:

Go to the userform's code module (select the form, right click on it and select view code)

In the left dropdown, select UserForm and in the right drop down select Activate. That will create the following code:

```
Private Sub UserForm_Activate()
  Me.ComboBox1.DropDown
End Sub
```

Add the statement `Me.ComboBox1.DropDown` to the above, assuming that your ComboBox is named ComboBox1. The resulting code is:

```
Private Sub UserForm_Activate()
  Me.ComboBox1.DropDown
End Sub
```
13.6.14 Problems With Dropdowns And Split Windows

If you have dropdowns on a worksheet and also split the windows, you run the risk that the dropdowns won't work. This is a bug, and a very hard one to recognize.

13.6.15 ComboBox.RowSource Returns Type Mismatch

ComboBox.RowSource Returns "Type Mismatch"

If you try something like the following, you will get a type mismatch error message:

```
UserForm1.ComboBox1.RowSource = _
   shSheet.Range(shSheet.Cells(1, 1), shSheet.Cells(nCount, 2))
```

The above happens because the RowSource property needs a string as its input. Instead, use the following approach:

```
UserForm1.ComboBox1.RowSource = _
   shSheet.Range(shSheet.Cells(1, 1), _
   shSheet.Cells(nCount, 2)).Address
```

13.6.16 How To Assign Column Headings In ListBoxes

According to MS KB article, you can only have column headings populated when you use the ListFillRange or RowSource and fill the box from the worksheet. Then the column headings are populated with the row above the specified fill range.

13.6.17 Getting Column Headings In A ListBox

You can only get column headings in a listbox if you define the RowSource property of the listbox - meaning you bind your list to a worksheet. The source range not to include your headers. The headers will be picked up from the row above your data on the spreadsheet. For example, if your headers are in cells A1 and B1, and your values are in A2 to B5 of Sheet2, you would enter Sheet2!A2:B5 in the RowSource property of the listbox. You also need to set the ColumnHeads property to True and define the ColumnCount property to the number of columns.

13.6.18 Displaying A List box With Multiple Columns

You can display a multi-column list box in several ways:

- assign the list's RowSource property to a multi-column range of cells
- Assign the list's List property to a two dimensional array

Lastly, you need to set the listbox's ColumnCount property to the number of columns that will be displayed. If you want to change the size of the columns, then set the listbox's
**ColumnWidths** property. For help on setting this property, place the cursor in the properties edit box in the property window and press F1.

### 13.6.19 Displaying Worksheet Names In A ListBox

The following illustrates how to populate a listbox with the names of the active workbook's chart and worksheet names. It also shows how to get back the sheet name that was selected, how to assign this sheet to an object variable, and how to activate the sheet.

```vba
Sub Sheet_Names_In_Dialog()
    Dim J As Integer, N As Integer
    Dim sName As String
    Dim oSheet As Object
    With UserForm2.ListBox1
        'rotate through the sheets
        For Each oSheet In Sheets
            'check the type of sheet
            If TypeName(oSheet) = "Worksheet" Or _
                TypeName(oSheet) = "Chart" Then
                'if a worksheet or chart, add to the list
                .AddItem oSheet.Name
            End If
        Next oSheet
    End With

    'display the userform
    UserForm2.Show

    'get the number of the item selected in the box
    N = UserForm2.ListBox1.ListIndex

    'get the name of the sheet selected
    sName = UserForm2.ListBox1.Value

    'assign the sheet to an object variable
    Set oSheet = Sheets(sName)

    'activate the sheet
    oSheet.Activate
```
'unload the form from memory

Unload UserForm2
End Sub

13.6.20 Printing Out What Is Selected In A ListBox

In this example, the user needed to printout the items selected in a multi-select list box when the user clicks on a button on the dialog. This approach prints what is in the listbox rather than assuming a list is already on a worksheet somewhere (but could be modified to do that). As constructed, it adds a scratch sheet, puts the items in the listbox on the sheet, prints the items from the sheet, and deletes the sheet.

This code is the click event for the button and should be put on the userform's module.

Dim bScreenSetting As Boolean
Dim bAlertSetting As Boolean
Dim sh As Worksheet
Dim I As Integer
Dim rng1 As Range

'store settings so they can later be reset

bScreenSetting = Application.ScreenUpdating
bAlertSetting = Application.DisplayAlerts

'turn off the following

Application.ScreenUpdating = False
Application.DisplayAlerts = False

'store a reference to the active sheet so it can be re-activated

Set sh = ActiveSheet

'add worksheet and give it a name. It becomes the active sheet

Worksheets.Add(After:=ActiveSheet).Name = "MyScratch"

'put a title on the sheet

Cells(1, 1) = "Items Needed:"

'write the selected items in the list to the scratch sheet

For I = 1 To Me.ListBox1.ListCount
    Cells(I + 1, 1) = Me.ListBox1.List(I - 1)
Next

'select the range containing the data and print it out
Set rng1 = Cells(1, 1).CurrentRegion
rng1.PrintOut

'delete the scratch sheet. No alert occurs since DisplayAlerts is False
Worksheets("MyScratch").Delete

'activate the original sheet and reset settings
sh.Activate
Application.DisplayAlerts = bAlertSetting
Application.ScreenUpdating = bScreenSetting

13.6.21 Referring To ListBoxes On Worksheets

Excel has two ListBox controls. One is from previous versions and is associated with the Forms Toolbar. The other is an ActiveX control and associated with the Control Toolbox Toolbar. If you are using the newer type, you need to specify it's object library:

Dim ListBoxX As MSForms.ListBox

An OLEObject is a container for the ActiveX control. If you want to refer to the control, you use the Object property of the OLEObject:

Set ListBoxX = Sheets("sheet1").OLEObjects("listbox1").Object

l$ = Sheets("sheet1").OLEObjects("Listbox1").Object.ListIndex

You don't need to create object variables to refer to these ActiveX controls. They have already been created for you:

l$ = Sheets1.ListBox1.ListIndex

will do the job,

To set the list in a list in Excel 2000, use

listdata = Sheets("sheet1").Range("a1"b20")
listbox1.List = listdata

or just:

listbox1.List = Sheets("sheet1").Range("a1"b20").Value

13.6.22 Unselect in ListBox

Set the listbox’s ListIndex to 0 (for an Excel listbox on a worksheet) or -1 (for an ActiveX listbox on a userform)
13.6.23 Initializing One ListBox Based On Another ListBox

Assume that you have the following entries on the first worksheet in your workbook:

Col A Col B Col C

Category Cars TVs

Cars Ford Sony

TV's Toyota RCA

BMW Misc Brands

Nissn

And, assume that you have two listboxes on your userform. The first listbox will allow the user to select a category in the first listbox based on the cells in column A. Once the user has selected a category, you want the appropriate list of selections for that category to appear in the second list box.

To achieve this result, you will need to put code in the 1st listbox's click event code that populates the 2nd list. In the following example, the listboxes are named ListBox1 and ListBox2. Also, the ColumnHeading property of each listbox has been set to True. This allows the headings (Category, Cars, and TVs) to be automatically displayed in the list boxes.

This example also illustrates how to store the selections made in the second listbox and use them if the user wants to see what he or she selected before exiting.

In a regular module, enter the following code:

Option Explicit

Public listChoice(0 To 1) As Integer, bOk As Boolean

Sub Populating_List2_Based_On_List1()
    Dim I As Integer

    'initialize array that holds choices from listbox2

    listChoice(0) = -1
    listChoice(1) = -1

    'unload form in case it is in memory

    Unload UserForm1
    With UserForm1

    End With
End Sub
'initialize 1st listbox from cell range  
'note that the column heading is not included in the range

.ListBox1.RowSource = Sheets(1).Range("a2:A3")  _
.Address(external:=True)  _
.Show
End With

'remove form from memory

Unload UserForm1

'See if Cancel selected

If Not bOK Then Exit Sub

'display choices

With Sheets(1).Range("a2")
For I = 0 To 1

'if no choice made so indicate

    If listChoice(I) = -1 Then
        MsgBox "No choice for " & .Offset(I, 0).Value

    'if choice made get the value from the cell

    Else
        MsgBox .Offset(I, 0).Value & " choice: " & _
        .Offset(listChoice(I), I + 1).Value
    End If
Next
End With
End Sub

In the code module for the userform (assumed to be named UserForm1), enter the following code:

Private Sub CommandButton1_Click()

'used to close and hide the userform

    Me.Hide
    bOK = True
End Sub

Private Sub CommandButton2_Click()

'used to close and hide the userform
Me.Hide
bOK = False
End Sub

Private Sub ListBox1_Click()

'only run the following if a choice is made in the listbox

If ListBox1.ListIndex > -1 Then
With ListBox2
  Select Case ListBox1.ListIndex
  Case 0:

    'initialize the 2nd list for choice 1 in first list
    'Note range is hard coded but can be a range name
    'or determined by the code
    .RowSource = Sheets(1).Range("b2:b5") _
    .Address(external:=True)
    .ListIndex = listChoice(0)
  Case 1:

    'initialize the 2nd list for choice 2 in first list
    'Note range is hard coded but can be a range name
    'or determined by the code
    .RowSource = Sheets(1).Range("c2:c4") _
    .Address(external:=True)
    .ListIndex = listChoice(1)
  End Select
End With
End If
End Sub

Private Sub ListBox2_Click()

'store selection from listbox2

listChoice(ListBox1.ListIndex) = ListBox2.ListIndex
End Sub

When you run the subroutine Populating_List2_Based_On_List1, selections in the first listbox change the list in the second listbox. When you exit, the name of the selected items are displayed. Also, the index value of the selection choices in listbox2 are stored in the array listChoice.

13.6.24 Putting Listbox Selection Into A TextBox Or Cell

To put what is selected in a listbox or a combobox into a textbox on the form when one clicks on a selection in the box, use code like the following:
Private Sub ComboBox1_Change()
    Me.TextBox1.Text = Me.ComboBox1.Value
End Sub

Private Sub ListBox1_Click()
    Me.TextBox1.Text = Me.ListBox1.Value
End Sub

To put the value in a cell when the user clicks on a selection, use code like the following:

Private Sub ComboBox1_Change()
    Sheets("Sheet1").Range("A1").Value = Me.ComboBox1.Value
End Sub

Private Sub ListBox1_Click()
    Sheets("Sheet1").Range("A1").Value = Me.ListBox1.Value
End Sub

13.6.25 Using A Horizontal Range For A List Box's Item

If you wish to specify a horizontal range for a listbox's items, you can do so with code just before you display the userform:

UserForm1.ListBox1.List = Application.Transpose(Range("A1:Z1"))

13.6.26 Having A Macro Run When A Selection Is Made In A List Box

To have a macro run when a selection in a combo box is made display the Visual Basic toolbar, click on the design button and then double click on the combo box. This will create the following code, which is stored in the worksheet's code sheet:

Private Sub ComboBox1_Change()
    End Sub

Edit the above to add in code that does what you want to happen based on what is selected in the combo box. For example, the following will run the macro DoIfValueIs5 if the value selected in the list or typed into the edit box is 5.

Private Sub ComboBox1_Change()
    If ComboBox1.Value = 5 Then DoIfValueIs5
End Sub

Sub DoIfValueIs5()
    MsgBox "hello"
End Sub
If you have a list box and want a macro to run when a particular selection is made in the list, then do the same thing to create the initial code in the worksheet's code sheet. Then edit the code for the action you want. For example, the following will display a message telling you what was selected in the list

```vbnet
Private Sub ListBox1_Click()
    MsgBox ListBox1.Value
End Sub
```

You could also make the action as complex as you want, for example setting different cells to specific values or running special macros.

### 13.6.27 Modifying An ActiveX Combobox On A Worksheet

OLEformat is the one you want for an ActiveX combobox

```vbnet
ActiveSheet.Shapes("Combobox1").OLEFormat._Object.ListfillRange = $A$5:$A$9
```

The **ListfillRange** actually belongs to the **OLEObject** and not the control itself. While the normal combobox properties are one level below that - thus the object.object.

Using the OLEObjects collection is shorter

```vbnet
ActiveSheet.OleObjects("Combobox1").ListfillRange = $A$5:$A$9
```

### 13.6.28 Internet Articles On ComboBoxes, EditBoxes, And ListBoxes

Here are a couple of MS KB articles which should get you headed in the correct direction for this question and those which might arise in the immediate future reference ComboBoxes and ListBoxes on UserForms in XL97:

http://support.microsoft.com/support/kb/articles/q164/9/23.asp

How to Fill a UserForm ListBox with Database Values

http://support.microsoft.com/support/kb/articles/q161/5/98.asp

XL97: How to Add Data to a ComboBox or a ListBox

http://support.microsoft.com/support/kb/articles/q183/1/83.asp

XL98: How to Fill ListBox Control with Multiple Ranges (works for XL97 too)

http://support.microsoft.com/support/kb/articles/q165/5/70.asp
13.7 OTHER USERFORM OBJECTS

13.7.1 Drawing Lines On UserForms

The best approach for putting a line on a userform is to use a textbox and making it very short (horizontal line) or very thin (vertical line) and then setting the background color to a darker gray. Set the height and width from the control property dialog. You can play with the SpecialEffect property to make it raised or to suite your purposes. You should also set the enabled property of the edit box to False so users can not tab into it.

13.7.2 How To Show A Chart, Map, WordArt, Shape Etc On A UserForm

Quite a few users want to show a chart, map, WordArt, shape etc. on an userform. A work around for charts was to export them to a GIF or JPG, then read them into an image control. This is not possible for the other object types.

Stephen Bullen has created code called PastePicture that does this! Its available at his web site in PastePicture.zip. This is a fairly small routine which uses API calls to convert whatever is on the clipboard into a standard Picture object, which can then be assigned to a Label, Image control, or the UserForm background itself. Displaying a chart from sheet1 is now:

```vbnet
Set Image1.Picture = PastePicture 'download code per above
```
The latest version of PastePicture supports pasting the picture as a metafile instead of a bitmap. The bitmap format gives a more exact image when copying same-size images, but the metafile gives a much better result when zooming/stretching the image.

13.7.3 Putting Background Graphics On A UserForm

You can put a background picture on a userform by:

- Selecting the userform
- Displaying the properties window
- Clicking in the picture property and then clicking on the little button that appears
- Selecting a picture
- Setting the picture properties which affect how it appears on the userform

13.7.4 Pasting Images To A UserForm Image Control

1. Create the Word-Art in a worksheet or anywhere else but in a userform
2. Select and Edit/Copy it
3. Switch to the userform and add the image control
4. Find the Picture property of the image control, click in it and press Ctrl+V

If you wish to do it with code, then use a statement like the following:

UserForm1.Image1.Picture = LoadPicture("C:\my picture.bmp")

With the second approach, you may need to repaint the form if it is being displayed when you change the image. Use

UserForm1.Repaint

To repaint the form.

13.7.5 Using A Calendar Control On A UserForm

Firstly you need the calendar control installed on your machine. This is done during the installation of Access and is one of the install options (Just run the Access setup program). It is also installed if you have Excel 2000.
To see if you have the calendar control, open the Visual Basic Editor in Excel and create a userform. Right click on the toolbox dialog and select "Additional Controls". Put an "x" in the calendar control box to make the control available and select OK to close the window. The control is now shown as an icon in the Toolbox/Controls window. If you do not have it, you may be able to find it on the Microsoft web site.

Once you do the above, you can put the calendar control on a userform. It has easy to use properties visible in the properties dialog.
14. FILES, CHARTS, AND WORKSHEETS

14.1 WORKING WITH WORKSHEETS

14.1.1 Adding Worksheets

To add a new worksheet, use the following statements:

Sheets.Add

or

Sheets.Add After:=ActiveSheet

The above statement will work fine except in early versions of Excel. The following is the workaround for adding a worksheet to the end of a workbook:

Sheets.Add.Move After:=Sheets(Sheets.Count)

14.1.2 Adding And Naming A New Sheet At The Same Time

The following will add a new sheet after the sheet name "Result" and name the sheet "Samples"

Worksheets.Add(After:=Worksheets("Result")).Name = "Samples"

14.1.3 Adding A Worksheet As The Last Sheet In A Workbook

You cannot add a sheet as the last sheet. You can however add it and then immediately move it. The following illustrates this in a workbook that initially has just one sheet.

With ActiveWorkbook.Worksheets.Add(before:=Worksheets(1))
    .Move after:=Worksheets(2)
End With

The following example duplicates the active worksheet and then moves it after that sheet. A two step approach is used in case there is just one sheet in the workbook

Dim sh As Worksheet
Set sh = ActiveSheet

' this Makes a copy of the sheet

sh.Copy sh
'this moves it after that sheet.

ActiveSheet.Move after:=sh

14.1.4 Renaming a worksheet

All sheets have a name property. Thus you can use statements like the following to rename a sheet

ActiveSheet.Name ="New Name"

or

Sheets("Sheet1").Name ="New Name"

14.1.5 How To Copy A Sheet And Make It The Last Sheet

The following statement illustrates how to copy a sheet and at the same time relocate the copy to the end of the workbook.

Sheets("My Sheet").Copy After:=Sheets(Sheets.Count)

If you wanted to copy it after a specific sheet, then you would put that sheet’s name in place of Sheets.Count. Please note that you do not have to select the sheet in order to copy it.

14.1.6 Sheet Copy Limit And The Cure

If you copy a sheet, and then copy the copy, etc, Excel 97 will ultimately crash after between 25 to 35 copies, even over multiple Excel sessions. If you check in the Visual Basic project explorer, you will see that the object names assigned to the sheets are Sheet1, Sheet11, Sheet111 and so forth. Sooner or later, this name gets to large, and Excel's reaction is to crash.

The cure is to run the following macro, which renames the object names for the worksheets. Please note if you have code that refers to the sheets by object name, you will have to fix the code.

Sub RenameObjectNames()

Const 1worksheet As Long = 100 'modules are 1

Dim objCode As Object, objcode2 As Object
Dim objComponents As Object
Dim J As Integer
Dim bNameOk As Boolean
Set objComponents = ActiveWorkbook.VBProject.VBComponents
For Each objCode In objComponents

If objCode.Type = lworksheet Then
  Do
    bNameOk = True

  'find un-used name

    J = J + 1
    For Each objcode2 In objComponents
      If objcode2.Name = "sheet_" & J Then
        bNameOk = False
        Exit For
      End If
    Next

  'rename and exit do loop if name can be used

    If bNameOk Then
      objCode.Name = "sheet_" & J
      Exit Do
    End If
  Loop
End If
Next
End Sub

14.1.7 How To Copy A Sheet To A New Workbook

The following illustrates how to copy a worksheet and rename it:

Worksheets("Sheet1").Copy after:=Worksheets("Sheet2")
ActiveSheet.Name = "NewSheet"

The following line of code copies a sheet to a new workbook and makes that workbook the active workbook:

Sheets("Sheet2").Copy

The following illustrates how to save each worksheet to its own workbook.

Dim wB As Workbook
Dim wS As Worksheet
Set wB = ActiveWorkbook
For Each wS In wB.Worksheets
  wS.Copy
  ActiveWorkbook.SaveAs wS.Name
  'if you want to close the new workbook:
  ActiveWorkbook.Close False
Next WS
'return to original workbook

WB.Activate

Please note that the above saves the files to the current directory. You can either set this to the desired directory before running, or specify the path. For example, in the following, if fPath has been set to C:\Temp, the files would be saved to that directory, not the current directory.

ActiveWorkbook.SaveAs fPath & wS.Name

14.1.8 Worksheet.Copy Bug - Public Variables Reset

If you do a worksheet copy, you may lose the settings of public variables. Apparently this is a known bug. The problem is with tracing through a worksheet copy and it also resets all module level variables.

The fix is to turn off the "Require Variable Declaration" option under Tools, Options from the Visual Basic Editor. You can still have Option Explicit in the module itself. However, you will need to manually type it in.

For more information, see the following Microsoft article:

http://support.microsoft.com/support/kb/articles/q177/8/34.asp

XL: Public Variables May Be Lost When You Copy a Worksheet

14.1.9 How To Delete Sheets

The best way to delete sheets is by name. For example:

'turn off error handling in case the sheet does not exist

On Error Resume Next
Application.DisplayAlerts = False
Sheets("D_Location").Delete
Sheets("D_Samples").Delete
Sheets("D_Results").Delete
On Error GoTo 0
Application.DisplayAlerts = True

On Error GoTo 0 causes normal error checking to resume,

If you delete sheets using index numbers, you should delete from the last sheet to the first. If you do it from 1 to N, this will most likely cause Excel to crash as the index numbers will ultimately refer to sheets that do not exist. In the least, it will crash the code

The wrong way to delete all empty sheets:
Sub Delete_All_Empty_Sheets_The_Wrong_Way()
    Dim I, J
    I = Worksheets.Count
    For J = 1 To I
        If Application.CountA(Worksheets(J).Cells) = 0 Then _
            Worksheets(J).Delete
    Next J
End Sub

The right way to delete all empty sheets:

Sub Delete_All_Empty_Sheets_The_Right_Way()
    Dim I, J
    I = Worksheets.Count
    For J = I To 1 Step -1
        If Application.CountA(Worksheets(J).Cells) = 0 Then _
            Worksheets(J).Delete
    Next J
End Sub

In the above Application.CountA returns the number of cells with entries. Also, the above will delete sheets that contain objects such as embedded charts.

14.1.10 Deleting Sheets Without Confirmation

To prevent the Excel warning message from appearing when you delete a sheet, do the following:

    Application.DisplayAlerts = False
    Worksheets("sheet1").Delete
    Application.DisplayAlerts = True

As the above example illustrates, you should set DisplayAlerts back to True. Otherwise Excel may not prompt you to save a modified file when you manually close Excel.

14.1.11 Getting The Exact Number Of Worksheets In A Workbook

Worksheets.Count returns the number of worksheets in the active workbook:

Sub CountSheets()
    MsgBox "Number of Worksheets: " & Worksheets.Count
End Sub

If you qualify Worksheets with a workbook, then you will get the number of worksheets in that workbook versus the number in the active workbook.

14.1.12 How To Determine If A Sheet Exists In A Workbook

The following functions returns True if a specific worksheet exists, and False if it does not. It takes two arguments: the name of the workbook, and the name of the sheet.
Function WorksheetExists(WBName As String, _
    WSName As String) As Boolean
On Error GoTo EndMacro
    If Workbooks(WBName).Worksheets(WSName).Name <> "" Then
        WorksheetExists = True
        Exit Function
    End If
EndMacro:
    WorksheetExists = False
End Function

For example

    If WorksheetExists("book1.xls", "Contracts") Then
        'code to execute if sheet does not exist
    End If

Or

    If WorksheetExists("book1.xls", "Contracts") Then
        'code to execute if sheet does not exist
    Else
        'code to execute if sheet exists
    End If

The following is an even simpler version of the above function:

Function bWorksheetExists(WBName As String, _
    WSName As String) As Boolean
On Error Resume Next
    bWorksheetExists = _
        (Workbooks(WBName).Worksheets(WSName).Name = WSName)
End Function

14.1.13 How To Determine If A Worksheet Is Empty

Application.CountA(Sheets("Sheet1").Cells) will return zero if all the cells on the sheet is empty. To test on the active sheet, just remove Sheets("Sheet1").

ActiveSheet.ChartObjects.Count will return the number of embedded charts on the worksheet. It is possible for all cells to be empty but for a sheet to contain embedded charts.

14.1.14 How To See If Worksheet Is Empty

The following determines if a worksheet is empty:
If Application.CountA(ActiveSheet.UsedRange) = 0 Then
  'empty
Else
  'Not empty
End If

14.1.15 Clearing A Worksheet On Open

The following macros run when a workbook is manually opened and clear ranges on a sheet:

Sub Auto_Open
  WorkSheets("sheet1").UsedRange.Clear
End Sub

Or

Sub Auto_Open()
  CertainRow = 15
  With WorkSheets("Sheet1")
    .Range(.Cells(CertainRow, 1), _
    .Cells(Rows.Count, 1)).EntireRow.Clear
  End With
End Sub

14.1.16 How To Loop Through Your Sheets

A worksheet is a member of the Worksheets collection. A sheet is a member of the more general Sheets collection. The Worksheets collection is a subset of Sheets collection. Other subsets of the Sheets collection are Charts, Dialogsheets, and Modules.

The following examples illustrate how to loop through these collections. All use a For...Next loop construction.

To loop through just the worksheets in the active workbook use code like the following. Each time through the loop, the variable ws is assigned to a new worksheet. This variable can be used to refer to that worksheet in the loop.

Dim ws As Worksheet
For Each ws In Worksheets
  'do something with sheet ws. For example the following statement
  'displays the sheet's name
  MsgBox ws.Name
Next
The following loops through all sheets in a workbook and prints them out. In this case the workbook is not the active workbook. Placing the variable \( sh \), which is used in the \emph{For} statement, after the \emph{Next} statement is optional but allows Visual Basic to match it up with the \emph{For} statement as a form of error checking.

```vba
Dim sh As Sheet
For Each sh In Workbooks("Estimate.Xls")
    sh.Printout
Next sh
```

Please note the above will printout all sheets, including modules and Dialogsheets. To restrict it to just worksheets or charts, do the following, which uses the \emph{TypeName} function to return the type of sheet. Note that \emph{LCase} is used to return an all lower case string so that the comparison is lower case to lower case.

```vba
Dim sh As Sheet
For Each sh In Workbooks("Estimate.Xls")
    If LCase(TypeName(sh)) = "worksheet" Or _
        LCase(TypeName(sh)) = "chart" Then sh.Printout
Next sh
```

### 14.1.17 Sorting Sheets By Name

The following macro that will sort the worksheets in the active workbook in ascending order:

```vba
Sub SortSheetbyName()
    Dim numberOfSheets As Integer
    Dim sheetPosition As Integer
    Dim I As Integer
    numberOfSheets = ActiveWorkbook.Worksheets.Count
    sheetPosition = numberOfSheets
    Do
        If sheetPosition = 1 Then Exit Do
        For I = 1 To sheetPosition - 1
            If Sheets(I).Name > Sheets(I + 1).Name Then
                Sheets(I + 1).Move before:=Sheets(I)
            End If
        Next I
        sheetPosition = sheetPosition - 1
    Loop
End Sub
```

### 14.1.18 Creating A List Of Sheets In A Workbook

The following code creates a list of all the worksheets in a workbook and writes it to a sheet in a new workbook. The new workbook only has one sheet in it
Sub ListSheets()
    Dim originalSetting As Integer
    Dim wb As Workbook
    Dim oS As Object
    Dim I As Integer

    'create an object variable that refers to the active workbook
    Set wb = ActiveWorkbook

    'create a new workbook with just one sheet
    'the new workbook becomes the active workbook.
    'The variable wb now refers to the previous active workbook
    originalSetting = Application.SheetsInNewWorkbook
    Application.SheetsInNewWorkbook = 1
    Workbooks.Add
    Application.SheetsInNewWorkbook = originalSetting

    'rotate through each sheet in the original workbook
    For Each oS In wb.Sheets
        increment I by one each time through
        I = I + 1

        'write the name of the sheet to a cell, starting with cell A1 and
        'going down column A
        Cells(I, 1).Value = oS.Name
    Next
End Sub

14.1.19 Protecting And UnProtecting Worksheets

It's easy to use Visual Basic to protect and unprotect your sheets. For example, you may want to
leave a sheet protected and only allow your macro to modify the sheet. The following illustrates
the basic technique to use:

Sub ProtectSheet()
    ActiveSheet.Protect password:="anystring"
End Sub

Sub UnProtectSheet()
    ActiveSheet.Unprotect password:="anystring"
End Sub

To determine if a worksheet is protected, check the value of the ProtectContents property:
If Sheets("mysheet").ProtectContents Then
    MsgBox "Protected"
End If

14.1.20 Using Controls On A Worksheet

You can assign a name to a command button (created from the Forms toolbar) by typing a name into the Name box when the shape is selected. (The Name box is at the left side of the formula bar – it normally displays the address of the active cell.)

Or, in VBA, you can assign a name with code like

```vba
ActiveSheet.Shapes(1).Name = "SomeName"
```

Once you've assigned a name, you can use that name as the index into the Shapes collection, which contains the buttons:

```vba
ActiveSheet.Shapes("SomeName").TextFrame.Characters.Text = "BB Button"
```

14.1.21 Protecting All The Sheets In A Workbook

The following code shows how to protect all the sheets in a workbook:

```vba
Sub ProtectingSheets()
    Dim S As Object
    For Each S In Sheets
        'unprotect the sheets first
        S.Unprotect password:="password"
        'protect the sheets and any objects on the sheet
        S.Protect password:="password", DrawingObjects:=True
    Next
End Sub
```

The following is a more elaborate example that prompts the user for a password.

```vba
Sub AskForPasswordAndProtectSheets()
    Dim S As Object
    Dim oldWord As String, pWord1 As String, pWord2 As String
    'prompt for old password, but do not exit if none; allows for first use
    oldWord = InputBox("Enter the old password")
    'prompt for new passwords, exit if not entered
```
pWord1 = InputBox("Enter a new password")
If pWord1 = "" Then Exit Sub
pWord2 = InputBox("please re-enter the new password")
If pWord2 = "" Then Exit Sub

'make certain passwords are identical
If InStr(1, pWord2, pWord1, 0) = 0 Or _
    InStr(1, pWord1, pWord2, 0) = 0 Then
    MsgBox "You entered different passwords. No action taken"
    Exit Sub
End If
For Each S In Sheets

'first unprotect the sheet using the old password
On Error GoTo errorTrap1
If oldWord <> "" Then S.Unprotect password:=oldWord

'protect the sheet using the new password
On Error GoTo errorTrap2
S.Protect password:=pWord1, DrawingObjects:=True
Next
MsgBox "All sheets protected."
Exit Sub

errorTrap1:
    MsgBox "sheet " & S.Name & _
    " could not be unprotected. Activity halted."
    Exit Sub
errorTrap2:
    MsgBox "sheet " & S.Name & _
    " could not be protected. Activity halted."
    Exit Sub
End Sub

14.1.22 A Simple Modify All Worksheets Example

The following is a simple example that modifies each worksheet in the active workbook. It
inserts a column in each sheet, and then inserts a formula in that column that sums the cells to the
left in row 5.

This example illustrates using the With..End With statement, and using Worksheets instead of
Sheets so that only worksheets are modified.

Sub Modify_Sheets()
    Dim I As Integer

    'rotate through all the worksheets by getting the count of worksheets

    For I = 1 To Worksheets.Count
        'insert a column in each sheet
        S.Cells(5, I).Formula = "=SUM(LEFT)

        'insert a formula in that column that sums the cells to the left in row 5
        S.Cells(5, I).Formula = "=SUM(LEFT)

    Next
End Sub
For I = 1 To Worksheets.Count
With ActiveWorkbook.Worksheets(I)
    ' please note the periods in front of Columns and Range.
    ' These connect these keywords back to the object that
    ' follows the above With keyword.
    .Columns("D").Insert
    .Range("d5") = "=sum(a5:c5)"
End With
Next I
End Sub

14.1.23 Inserting The Current Date In All Worksheets

If you need to update a particular cell on each worksheet in a workbook, then you can use code like the following to accomplish this task:

Sub ApplyDateValue()
    Dim mydate As Date
    Dim a As Integer
    mydate = Date
    For a = 1 To ActiveWorkbook.Worksheets.Count
        Worksheets(a).Range("K4").Value = mydate
    Next a
End Sub

You can also write the code this way, which is a bit more efficient, and refers to the sheets directly

Sub ApplyDateValue()
    Dim ws As Worksheet
    Dim myDate As Date
    myDate = Date
    For Each ws In Worksheets
        ws.Range("K4").Value = myDate
    Next ws
End Sub

Notice that neither example activates or selects the sheet. It is not necessary to do so, and in fact doing so would slow down the macro.

14.1.24 Making All Sheets Visible

The following code shows how to make all sheets in a workbook visible.

Sub MakeAllVisible()
    Dim oSheet As Object
    For Each oSheet In Sheets
        oSheet.Visible = True
    Next
End Sub
14.1.25 Preventing A User From Adding A Sheet

If you put the following code in the workbook code module, it will prevent the user from adding a new worksheet.

Private Sub Workbook_NewSheet(ByVal Sh As Object)
Application.DisplayAlerts = False
MsgBox "New sheets may not be added. " & _
"Sheet will be deleted"
Sh.Delete
Application.DisplayAlerts = True
End Sub

14.1.26 Using A Worksheet's Code Name

To always activate or work with the same sheet regardless of its tab name, you can use its Code Name which doesn't change when the sheet is renamed. For example:

Assuming you have a sheet with Name "SheetName" and CodeName "SheetCodeName", these commands work identically

Worksheets("SheetName").Activate

'this is the same as:

SheetCodeName.Activate

To change the code name of a worksheet, go into Visual Basic, display the project explorer (Ctrl+R), select the worksheet, and edit the name property (F4) of the worksheet.

14.1.27 Changing A Worksheet's CodeName

You can change a worksheet's CodeName using:

ActiveWorkbook.VBProject.VBComponents(ActiveSheet.CodeName) _
.Properties("_CodeName") = "NewCodeName"

or:

ThisWorkbook.VBProject.VBComponents("Sheet1").Name = "AnyName"

14.1.28 Checking If A Control Exists On A Worksheet

When ActiveX controls are on the worksheet, then they part of the OLEObject collection and the shapes collection.

Sub FindImagecontrol()
For Each OleOb In ActiveSheet.OLEObjects
    If TypeOf OleOb.Object Is MSFORMS.Image Then
Or, if you know the OLEObject name of the image control (default something like image1, image2), you can just test for it.

Function IsImage(sName As String)
Dim vtop
On Error Resume Next
vtop = ActiveSheet.OLEObjects(sName).Top
If Err = 0 Then
IsImage = True
Else
IsImage = False
Err.Clear
End If
End Function

Sub TestImage()
MsgBox IsImage("Image1")
End Sub

14.2 WORKING WITH CHARTS

14.2.1 Loop Through All Charts

The following will loop through all the chart objects in all the sheets and present a MsgBox displaying the name of each chart. Obviously it can be modified to do whatever you need in place of the MsgBox.

Sub FindCharts()
Dim oChart As ChartObject
Dim nSheet As Integer
Dim nChart As Integer
nSheet = 1

Do While nSheet <= Sheets.Count
nChart = 1

Do While nChart <= Sheets(nSheet).ChartObjects.Count
Set oChart = Sheets(nSheet).ChartObjects(nChart)
MsgBox (Sheets(nSheet).Name & " " & oChart.Name)
    nChart = nChart + 1
Loop
nSheet = nSheet + 1
Loop
End Sub

14.2.2 Relocating Embedded Charts By Code

The following code will set the top left of chart 1 to the top left of the D10 cell:
The following relocates embedded charts to specified cells and makes them a certain size. This code uses the name of the embedded chart which you can get by selecting the chart and checking the name that appears in the name box to the left of the formula edit box. The easiest way to select a chart is to display the Drawing toolbar and use the Select Objects button to select the chart.

Sub RelocateAllGraphs()

'call the RelocateAGraph routine three times, passing to it different chart names and destinations
RelocateAGraph "Chart 1", "A1"
RelocateAGraph "Chart 2", "a11"
RelocateAGraph "Chart 3", "a21"
End Sub

Sub RelocateAGraph(gName As String, cellRef As String)
With ActiveSheet.Shapes(gName)

' set the position of the chart based on the cell reference
.Left = Range(cellRef).Left
.Top = Range(cellRef).Top

' set the width and height of the chart
.Width = 150
.Height = 90
End With
End Sub

14.2.3 Creating A Chart On A New Sheet

The following code will prompt for the data source for a pie chart and then draw it on a new sheet:

Sub MakePies()
Dim dataRange As Range

' set error trap in case cancel selected
On Error GoTo errorTrap

' prompt for input range
Set dataRange = Application.InputBox _
("Where is your data for chart?", Type:=8)

'add a pie chart, supplying the above range

Charts.Add
ActiveChart.ChartWizard Source:=dataRange, _
Gallery:=xlPie, Format:=1, PlotBy:=xlColumns, _
CategoryLabels :=0, SeriesLabels:=0, HasLegend:=2, _
Title:="", CategoryTitle :="", _
ValueTitle:="", ExtraTitle:=""
Exit Sub
errorTrap:
 Exit Sub
End Sub

14.2.4 Deleting All Embedded Charts On A Worksheet

The following code will delete all embedded charts on a worksheet:

Dim oCht As ChartObject
For Each oCht In ActiveSheet.ChartObjects
 oCht.Delete
Next

14.2.5 Making Charts Using Visual Basic Code

The following is an example of creating a pie chart

Sub MakePies()
 Dim myRange As Range
 Set myRange = Application.InputBox _
 (Prompt:="Where is your data for chart?", Type:=8)
 Charts.Add
 ActiveChart.ChartType = xlPie
 ActiveChart.SetSourceData Source:=myRange, _
 PlotBy:=xlColumns
 ActiveChart.Location Where:=xlLocationAsNewSheet
End Sub

14.2.6 Changing The Size Of Embedded Charts

The following will re-size all the embedded charts on the active worksheet to the same size:

Sub Resize_charts()
 Dim cObj As ChartObject

 'rotate through each chart

 For Each cObj In ActiveSheet.ChartObjects
'set the height and width

    cObj.Height = 200
    cObj.Width = 250
    Next
    End Sub

14.2.7 Replicating Charts

Assume that you have a large number of embedded charts on a hidden worksheet, not displayed to the user, and a single embedded chart on a visible worksheet, which acts as a place holder to display whichever of the charts the user selects. Due to problems with copying/pasting charts between sheets (i.e. crashing after 80 or so), you can't just copy the selected chart from the hidden sheet to the visible sheet.

To avoid the crashes, copy just the ChartArea, rather than the ChartObject as illustrated by the following example. This example has the user enter the number of the chart. You could display a small userform with a selection list instead. The example loops until the user selects cancel (or enters zero or fails to enter a number)

Sub DisplayCharts()
    Dim chtTarget As Chart
    Dim chtSource As ChartObject
    Dim iChart As Integer
    Do
        iChart = Val(InputBox("Enter the number of a chart"))
        If iChart = 0 Then Exit Sub
        On Error GoTo eTrap
        Set chtTarget = ActiveSheet.ChartObjects(1).Chart
        Set chtSource = Sheets("Sheet1").ChartObjects(iChart)
        chtTarget.ChartArea.ClearContents
        chtSource.Chart.ChartArea.Copy
        chtTarget.Paste
    Loop
    eTrap:
    MsgBox "The chart you requested was not found."
    Resume doAgain
End Sub

14.2.8 How To Export Charts To GIF Files

The following will convert a chart sheet to a GIF file:

ActiveChart.Export FileName:="c:\Mychart.gif", FilterName:="GIF"

and save it to c:\Mychart.gif.

If the chart is an object in a worksheet, then you can use the following code to convert it to a GIF file:
The trick in the second approach is determining the embedded chart's name. To do that, do the following:

- select a cell on the worksheet
- display the drawing toolbar
- click on the select objects button
- click on the chart - the name of the chart will be displayed in the Name box, which is to the left of the formula box

You can also re-name the chart by clicking in the name box with the chart selected per the above procedure and typing in a new name.

### 14.2.9 Value Of A Point On A Line

To get the value of a point on a line, you need to use the series object. Use something like the following to process the first series in a bar chart, coloring each bar greater than 100 yellow, and those less than or equal to 100 blue:

```vba
'Use following line for embedded chart
Set cht = ActiveSheet.ChartObjects("chart 1").Chart

'Use following line for Chart sheet (note the quote making the next line a comment)
'Set cht = Charts("Chart1")
```

```vba
Set pts = cht.SeriesCollection(1).Points
varVals = cht.SeriesCollection(1).Values
For Each pt In pts
    i = i + 1
    If varVals(i) > 100 Then
        pt.Interior.ColorIndex = 6
    Else
        pt.Interior.ColorIndex = 5
    End If
Next pt
```

### 14.2.10 An Add An Embedded Chart Example

The following example adds a chart on the active sheet
Sub AddChartTest()
Dim cht As ChartObject
Dim wks As Worksheet
Set wks = ActiveSheet

'add an embedded chart and set it equal to an object variable
Set cht = wks.ChartObjects.Add(0, 75, 400, 250)

'specify the data and the chart type
With cht.Chart
  .SetSourceData wks.Range("B2:F4")
  .ApplyCustomType xl3DColumnClustered
End With
End Sub

Where:

(0, 75, 400, 250) sets the Left, Top, Width, and Height respectively.

("B2:F4") identifies the range you are plotting, including the column and row headings.

Because the code sets the cht variable equal to the added chart, you can refer to it directly by it's variable name and not by it's location and shape/chart name.

14.2.11 Changing A Chart's Size And Position

A chart embedded in a sheet is contained in a ChartObject, which is the parent of a Chart. If you want to position the embedded chart, you position the ChartObject. the following positions a chart over A10:H25:

Set rng = Range("A10:H25")
With ActiveSheet.ChartObjects(1)
  .Top = rng.Top
  .Left = rng.Left
  .Width = rng.Width
  .Height = rng.Height
End With

14.2.12 Determining If A Series Is Selected In A Chart

The following statement will return True if a series in a chart is selected, and False if one is not.

  Ucase(TypeName(Selection)) = "SERIES"

For example,

  If Ucase(TypeName(Selection)) = "SERIES" Then
'code to do if true

End If

14.2.13 Changing The Title On An Embedded Chart

The following illustrates how to change the title on an embedded chart to the text in a cell. In this example, cell A1 is used.

ActiveSheet.ChartObjects(1).Chart.ChartTitle.Text = _
  Range("a1").Text

or

ActiveSheet.ChartObjects("Chart 1").Chart.ChartTitle.Text = _
  Range("a1").Text

14.2.14 Relocating A Chart - Another Example

If you have charts that you wish to selectively display on a worksheet, then you will need to move them in and out of view. The following code will set the top left of your chart to the top left of the D10 cell:

ActiveSheet.Shapes("Chart 1").Left = Range("D10").Left
ActiveSheet.Shapes("Chart 1").Top = Range("D10").Top

You can name the chart with:

ActiveSheet.Shapes("Chart 1").Name = "Data"

or

ActiveSheet.Shapes(1).Name = "Data"

14.2.15 Determining What A User Has Selected In A Chart

if you have a situation where the user is to select a line in a chart and then click on a button to run a macro, you need to determine what is selected before taking any action. The following statement returns what is selected and sets it to a variable.

Dim sSel As String

sSel = ExecuteExcel4Macro("SELECTION()"

which returns, for example S2 if the entire second series is selected, or S2P3
if the third point on the second series is selected. Please note that there are double quotes around Selection() in the above statement.

You can also use TypeName(Selection()) to return the type of the selection.

14.2.16 Converting Chart Series References to Values

The following code will convert the formulas in a chart series to values. It is designed to loop through all series of all charts, whether they are embedded charts or chart sheets:

```vba
Sub Check_And_Remove_Links()
    Dim oSheet As Object
    Dim cObj As Object
    If MsgBox("Check for and remove chart links?", _
               vbOKCancel) = vbOK Then
        'rotate through all sheets
        For Each oSheet In ActiveWorkbook.Sheets
            'if a chart sheet, pass the sheet object
            If TypeName(oSheet) = "Chart" Then
                RemoveChartLinks oSheet
            Else
                'if a worksheet, check for chart objects and pass
                'the chart object to the remove macro
                For Each cObj In oSheet.ChartObjects
                    RemoveChartLinks cObj.Chart
                Next
            End If
        Next
    End If
End Sub

Sub RemoveChartLinks(oObj As Object)
    On Error Resume Next
    Dim seriesCount As Integer
    Dim I As Integer
    With oObj
        'get number of series and rotate
        seriesCount = .SeriesCollection.Count
        For I = 1 To seriesCount
            With .SeriesCollection(I)
                'change to values
            End With
        Next
    End With
End Sub
```
14.2.17 Labeling The Points On A Line

The following code illustrates how to label the points on a line or a scatter plot:

```vba
Sub AddLabelsToPoints()
    Dim oPoint As Point
    Dim dataRange As Range
    Dim I As Long
    On Error Resume Next
    Set dataRange = Application.InputBox(
        _
        "Select the series data label range",
        Type:=8)
    If dataRange Is Nothing Then Exit Sub
    On Error GoTo 0
    For Each oPoint In ActiveChart.SeriesCollection(1).Points
        I = I + 1
        oPoint.HasDataLabel = True
    Next oPoint
End Sub
```

14.2.18 Putting Charts On UserForms

Unfortunately, you can not put a chart directly on a UserForm. However, you can save the chart as a GIF file and then load the GIF file onto an image object that is on the userform. Image objects are created by using the image button on the VBE toolbox.

The following example uses an input box to get the number of the chart. The picture is saved as C:\temp.GIF and then loaded to the userform.

```vba
Sub ChartOnUserForm()
    Dim iChart As Integer
    Dim oChart As Chart
    Do
        iChart = Val(InputBox("Enter the number of the chart"))
        If iChart = 0 Then Exit Sub
        Set oChart = Sheets("Sheet1").ChartObjects(iChart).Chart
    'save the chart to a file
    oChart.Export Filename:="C:\temp.gif", FilterName:="GIF"
    'load onto userform image
```
UserForm1.Image1.Picture = LoadPicture("C:\temp.gif")
serForm1.Show
Loop
End Sub

The following approach uses a spinner button on a userform to rotate through the charts. In this example, a global constant is used to specify the maximum number of charts. Clicking on the spinner button will rotate continually through the charts:

' set to max number of charts

Global Const maxCharts = 3

Sub ChartOnUserForm()

' make sure form is unloaded

Unload UserForm1

' display the form

UserForm1.Show
End Sub

Private Sub UserForm_Activate()

' activating the form runs this line which initializes the spin button
' this causes the Change event below to be run which loads the first chart

Me.SpinButton1.Value = 1
End Sub

Private Sub SpinButton1_Change()

Dim iChart As Integer

' make certain the number of the chart is between 1 and the max number

With Me.SpinButton1
    If .Value > maxCharts Then
        .Value = 1
    ElseIf .Value = 0 Then
        .Value = maxCharts
    End If
    iChart = .Value
End With

' export the selected chart

With Sheets("Sheet1").ChartObjects(iChart).Chart
    .Export Filename:="C:\temp.gif", FilterName:="GIF"
End With
14.3 WORKING WITH FILES

14.3.1 GENERAL WORKBOOK EXAMPLES

14.3.1.1 How To Open A Workbook

The simple way to open a file is with a statement like the following:

```vba
Workbooks.Open filename:="c:\data\myfile.xls", _
    UpdateLinks:=False
```

Setting the `UpdateLinks` value to `False` prevents Excel from asking if you want to update links (and links are not updated). If you want to open the workbook as read only, then add "ReadOnly:=True" to the above statement.

You could also write the above statement slightly shorter, by not specifying the names of the arguments. If you do this, then the values supplied must be in the order that the method expects them.

```vba
Workbooks.Open c:\data\myfile.xls", False
```

To find the argument order for any method, simply put the cursor in the keyword and press F1 for Visual Basic help. If you supply one argument name, you must supply argument names for all arguments used.

If you want to prompt the user for the filename, then use statements like this to open the filename and open the file.

```vba
fName = Application.GetOpenFileName _
    ("Excel Files (*.xls), *.xls")
If fName <> False Then
    Workbooks.Open fName, False
End if
```

14.3.1.2 Just Opened Workbook Not The Active File

In some instances, a workbook that was just opened is not the active workbook, although it should be. The following sets an object variable to the opened workbook so that you can reference it directly and set it to be the active workbook.

```vba
Dim wkb As Workbook
Set wkb = Workbooks.Open(fileName:="C:\MyBook.xls")
wkb.Activate
```
14.3.1.3 Determining If A Workbook Is Open

The following function checks to see if a particular workbook is open

```vba
Function bFileOpen(wbName As String) As Boolean
    Dim wb As Workbook
    'check each open workbook's name, in lower case, and set the function
    'to true if a match is found. If not match the function defaults to False
    For Each wb In Workbooks
        If LCase(wb.Name) = LCase(wbName) Then
            bFileOpen = True
            Exit Function
        End If
    Next
End Function
```

An alternate and shorter function is:

```vba
Function bBookOpen(wbName As String) As Boolean
    On Error Resume Next
    bBookOpen = Len(Workbooks(wbName).Name)
End Function
```

This works because If Len() returns an error, bBookOpen stays at it default value, which is false.

The following illustrates how to use the above function:

```vba
If bFileOpen("MyBook.XLS") Then
    'Actions to take if open
End If
```

or

```vba
If Not bFileOpen("MyBook.XLS") Then
    'Actions to take if file is not open
End If
```

The following code illustrates how to determine if a particular file is open or not.

```vba
Dim wb As Workbook
For Each wb In Workbooks
    'do a upper case comparison, go to the label nextStep if the file is found
```
If UCase(wb.Name) = Ucase("somename.xls") Then GoTo nextStep
Next

'if the code gets to here, the above code did not find the file, which
'indicates it is not open

MsgBox "somename.xls is not open."
Exit Sub

nextStep:

'code to execute if file found

The following is a variation of the above approach that uses a function to return True if the file is open, and False if it is not.

Function bOpen(fName As String) As Boolean
    Dim wb As Workbook

    'check each wb to see if its name is the one passed to the function
    For Each wb In Workbooks

        'do a upper case comparison, go to the label nextStep if the file is found
        If UCase(wb.Name) = UCase(fName) Then GoTo nextStep
    Next

    'if the code gets to here, the above code did not find the file, which
    'indicates it is not open. Exiting without setting bOpen returns a False value

    Exit Function

nextStep:

'if code comes here, then file must be open. Set the function to True
'in this case.

    bOpen = True
End Function

The following illustrates how to use the above function:

Sub OpenExample()
    If bOpen("Book2.xls") Then

    'code to run if file open

    End If
    If Not bOpen("Book2.xls") Then
'code to run if file is not open

End If
End Sub

The following illustrates a means to test to see if a workbook is open without having to loop through the workbooks that are open. It also avoids having to a text comparison.

Sub test()
  Dim wkName As String

'make certain the string is empty

  wkName = ""

'set a string variable equal to the name of the workbook
'if the workbook is open
'turn On Error Resume Next on as the following statement would
'otherwise cause an error

  On Error Resume Next
  wkName = Workbooks("BOOK1.XLS").Name

'turn off error checking

  on Error GoTo 0

'if the workbook is open, the above will assign a string to the variable.
'If this happens the Len() function will return a value greater than 0

  If Len(wkName) = 0 Then
    MsgBox "the workbook is not open"
  End
  End If

'code to run if the workbook is open

End Sub

The following is a function that returns True if the workbook is open, and False if it is not open. It uses the same technique as illustrated above.

Function bIsOpen(anyName As String) As Boolean
  Dim wkName As String

'set a string variable equal to the name of the workbook
'if the workbook is open
'turn On Error Resume Next on as the following statement would
'otherwise cause an error


On Error Resume Next
wkName = Workbooks(anyName).Name

'if a string name returned, set the function equal to true
'otherwise just exit, leaving the function to its default False value

If Len(wkName) > 0 Then bIsOpen = True End Function

The following illustrates the above function:

Sub TestFunction()
    Dim wkBkName As String
    wkBkName = "book1.xls"
    If bIsOpen(wkBkName) Then
        MsgBox wkBkName & " is open"
    Else
        MsgBox wkBkName & " is not open"
    End If
End Sub

The following function returns True if a file is already open, and False if it is not.

Function IsOpen(anyName As String) As Boolean
    Dim wb As Workbook
    'set error trap in case no workbooks are open
    On Error GoTo noFilesOpen
    'compare the name of each open workbook to the name in question
    'convert all letters to lower case in the test
    For Each wb In Workbooks
        If LCase(wb.Name) = LCase(anyName) Then
            'if a match is found set the function to true and exit the function
            IsOpen = True
            Exit Function
        End If
    Next
    'if no match is found the function exits and returns a default value of False
    Exit Function
    noFilesOpen:
    'return a false value if no files open
    IsOpen = False
End Function
The following shows two ways to use the above function:

Sub Test1()
 If IsOpen("book1.xls") Then
    'do this if open
 End If
End Sub

Sub Test2()
 If Not IsOpen("Book1.xls") Then
    'do this if not open
 End If
End Sub

14.3.1.4 How To Determine If A File Is Open

The following function returns True if a workbook or add-in is already open, False if it is not open.

Function IsOpen(wbName As String) As Boolean
    Dim wb As Workbook
    On Error Resume Next
    Set wb = Workbooks(wbName)
    If Err = 0 Then IsOpen = True
End Function

For example

MsgBox IsOpen("My workbook.xls")

14.3.1.5 Testing For File Or Workbook Existence Before Opening

Use the Dir function to test for the existence of a file:

Sub OpenFileIfItExists()
    Dim FName As String
    FName = "C:\Test.xls"
    If Dir(FName) = "" Then
        MsgBox FName & " does not exist"
    Else
        Workbooks.Open FileName:=FName, updateLinks:=False
    End If
End Sub

Updating A Saved When Date In A File
You can use the before save event in the workbook module to update any cell in a workbook with the current date and time:

```vba
Private Sub Workbook_BeforeSave(ByVal SaveAsUI As Boolean, _
Cancel As Boolean)
ThisWorkbook.Sheets(1).range("a1").Value = Now()
End Sub
```

### 14.3.1.6 Adding Or Opening Workbooks

When you add or open a workbook via Visual Basic, Excel does not always make it the active workbook. This can happen if the workbook was saved when its window was minimized. If you code assumes this to be the case, you may be in for a rude surprise. The following is one way to solve this:

If adding a new workbook

```vba
Dim newBook As Workbook
Set newBook = Workbooks.Add
newBook.Activate
```

or

```vba
Dim oBook As Workbook
Set oBook = Workbooks.Open("MyBook.xls")
oBook.Activate
```

### 14.3.1.7 Adding Workbooks

To add a new workbook, use the following statement:

```
Workbooks.Add
```

You can also assign an object variable to refer to the new workbook when you create it.

```vba
Dim wb As Workbook
Set wb = Workbooks.Add
MsgBox wb.Name
wb.SaveAs FileName:="Fred.xls"
```

Using an object variable makes it easier to refer to the new workbook in your code.

You can also control how many sheets the workbook has with code like the following:

```vba
Dim originalSetting As Integer
Dim wb As WorkBook

'store the user's preferred number of new sheets in a workbook
originalSetting = Application.SheetsInNewWorkbook
```
'set the number to one so the new workbook will have only one sheet

Application.SheetsInNewWorkbook = 1
Set wb = Workbooks.Add

'set the number of sheets in a new workbook back to the original setting

Application.SheetsInNewWorkbook = originalSetting

14.3.1.8 Determining If A File Is Open In ReadOnly Mode

If Workbooks("workbookname").ReadOnly Then
  MsgBox "read-only"
End If

14.3.1.9 Changing the ReadOnly Status of a File

You can change the ReadOnly status of a file by using the ChangeFileAccess method:

    ActiveWorkbook.ChangeFileAccess xlReadWrite

makes the file writeable.

    ActiveWorkbook.ChangeFileAccess xlReadOnly

makes the file read only. If there have been changes, you will be prompted to first save the file. To avoid this, set the Application.DisplayAlerts property to False before changing to read only. Then immediately change back.

14.3.1.10 Finding Workbook Links

The following will list all the workbook links in a workbook:

Sub ListLinks()
  Dim aLinks As Variant
  aLinks = ActiveWorkbook.LinkSources(xlExcelLinks)
  If Not IsEmpty(aLinks) Then
    Sheets.Add
    For i = 1 To UBound(aLinks)
      Cells(i, 1).Value = aLinks(i)
    Next i
  End If
End Sub

14.3.1.11 How To Retrieve Names Of Workbooks, Sheets, Etc.

Description: VBA code Example

Sheet name: Application.ActiveSheet.Name Sheet1
File name only: test

Application.Substitute(ActiveWorkbook.Name, ".xls", "")

Or

Left(ActiveWorkbook.Name, Len(ActiveWorkbook.Name) - 4)

File name and extension: Application.ActiveWorkbook.Name  test.xls

Path name: Application.ActiveWorkbook.FullName  D:/TestFolder/test.xls

Path: Application.Path  C:/MSOffice/Excel

14.3.1.12 Saving A Workbook With Its Name Equal To The Current Date

The following statement

ActiveWorkbook.SaveAs Filename:=Format(date,"MMDDYY") & ".xls"

will save a workbook with a filename that is the current date, e.g. "092998.xls".

14.3.1.13 How To Not Save A Workbook When It Closes

If you've modified a workbook and do not want to save it when you close it manually, use the following Auto_Close procedure:

Sub Auto_Close()
   ThisWorkbook.Saved = True
End Sub

If you are closing a workbook with your code and do not want to save it or be prompted to save it, use a statement like the following:

ActiveWorkbook.Close False

If the file is not the active file, you can use a statement like the following:

Workbooks("MyBook.xls").Close False

14.3.1.14 Closing All But the Active Workbook

The following code closes and saves all workbooks but the active workbook:
For Each wb In Workbooks
    If Not wb Is ThisWorkbook Then
        wb.Close SaveChanges:=True
    End If
Next wb

If you do not want to save the workbooks being closed, then change SaveChanges to False.

14.3.1.15 Returning The Full Path And Name Of A WorkBook

The function FullName will return the path and the name of the workbook as a string. For example:

    FullPath = ActiveWorkbook.FullName

or, if the code is in the workbook in question:

    FullPath = ThisWorkbook.FullName

If a workbook has not been saved, then only the name of the workbook is returned.

14.3.1.16 Determining The Date And Time A File Or Workbook Was Last Saved

The following statement illustrates how to return a Variant (Date) value that indicates the date and time when a file was created or last modified.

    dModified = FileDateTime("C:\auto_open.bat")

The required argument is a string expression that specifies a file name. The argument may include the directory or folder, and the drive. The file must be closed for this to work or it will return the current time or a value close to it.

14.3.1.17 Open The Last Modified File In A Directory

Sub LastModifiedFile()
    Dim dirName As String
    Dim fName As String
    Dim fileTime As Date
    Dim fileName As String
    Dim latestFile As String

    'set the directory to be checked
    dirName = "C:\My Documents\"

    'query for a file in the directory
    fName = Dir(dirName & "*.*")
End Sub
'set values that can be changed later

latestFile = fName
fileTime = FileDateTime(dirName & fName)
While fName <> ""

'loop until no more files

If FileDateTime(dirName & fName) > fileTime Then

'if more recent file found, update variables

latestFile = fName
fileTime = FileDateTime(dirName & fName)
End If

'query for next file

fName = Dir()
Wend
If latestFile = "" Then

'advise if no file found

MsgBox "There are no files in the directory"
Else

'Open the file

Workbooks.Open "C:\My Documents\" & latestFile
End If
End Sub

14.3.1.18 Counting Visible (Non-Hidden) Workbooks

The following function returns the number of visible (non-hidden) workbooks. Please note it assumes that there is only one window per workbook.

Function VisibleWorkbookCount()
Dim lngNumberOfVisibleWorkbooks As Long
Dim bookCounter As Workbook

For Each bookCounter In Application.Workbooks
    If Windows (bookCounter.Name).Visible = True Then
        lngNumberOfVisibleWorkbooks = _
        lngNumberOfVisibleWorkbooks + 1
    End If
Next bookCounter

VisibleWorkbookCount = lngNumberOfVisibleWorkbooks
End Function
14.3.1.19 Extracting Values From Closed Workbooks

It is possible to extract values from closed workbooks by using Excel 4 macro statements like the following:

\[ x = \text{ExecuteExcel4Macro}('C:\[NotOpen.xls]Sheet1'!R1C1') \]

\[ x = \text{ExecuteExcel4Macro}('C:\[NotOpen.xls]Sheet1'!R1C1:R3C3') \]

\[ x = \text{ExecuteExcel4Macro}('C:\[NotOpen.xls]Sheet1'!R1C1:R2C2') \]

As indicated in help, there is no "=" in the argument string, you must use RC notation, and any embedded quotes must be doubled.

The following is another way, which does not rely on Excel 4 macro code, but is somewhat kludgy:

```vba
Sub GetClosedBookValue()
    Dim theValue

    'write a formula that refers to a cell on the closed workbook
    Range("A1").Formula = "='D:\[MMULT_EX.XLS]Sheet1'!$A$1"

    'update links
    ActiveWorkbook.UpdateLink ("D:\MMULT_EX.XLS")

    'extract the updated value and clear the formula
    theValue = Range("A1").Value
    Range("A1").ClearContents
End Sub
```

Additional extraction examples:

This extracts the value of a single cell

\[ x = \text{ExecuteExcel4Macro}('C:\[NotOpen.xls]Sheet1'!R1C1') \]

This extracts the average of a cell range

\[ x = \text{ExecuteExcel4Macro}('C:\[NotOpen.xls]Sheet1'!R1C1:R2C2') \]

As indicated in help, there is no "=" in the argument string, you must use R1C1 notation. Also, the single quotes and the brackets are needed.

14.3.1.20 Error 'The File Is Already Open...' - Re-Registering Excel
If you get the above error message when you open an Excel file from the Windows Explorer, you may need to re-register Excel. You can force Excel to re-register it self by using /regserver on the Run command.

C:\path_to_excel\Excel.exe /regserver

Excel will rewrite all of the registry keys with their default values, then quit.

14.3.2 SELECTING AND OPENING WORKBOOKS

14.3.2.1 Using The Built-In GetSaveAsFilename Dialog

The following illustrates how to use the GetSaveAsFilename built-in dialog to return the full name and path, just the filename, and just the path. Two functions are used to extract the filename and the path. In this case, the user is restricting the displayed files to just those with a CSV filetype.

Sub Getting_FileName_And_Path()
    Dim fileAndPath As String

    'display the built-in dialog, with an initial filename suggested
    'and with only CSV files being displayed for selection
    fileAndPath = Application.GetSaveAsFilename( _
            InitialFilename:="facility", _
            fileFilter:="Comma Separated Files (*.csv), *.csv", _
            Title:="Choose a destination directory")

    'if cancel selected, halt the macro
    If fileAndPath = False Then End

    'display the different values
    MsgBox "Files will be saved to " & fileAndPath
    MsgBox "The destination directory is " & _
        PathOnly(fileAndPath)
    MsgBox "The file name is " & NameOnly(fileAndPath)
End Sub

Function PathOnly(ByVal anyStr) As String
    Dim I As Integer, J As Integer
    I = Len(anyStr)

    'check each character for a \ working backwards
    For J = I To 1 Step -1
        If Mid(anyStr, J, 1) = "\" Then
            Exit For
        End If
    Next J

    'when a match is found set the function to the text to the left and exit
    PathOnly = Left(anyStr, J - 1) & Mid(anyStr, J, I - J + 1)
End Function
PathOnly = Left(anyStr, J - 1)
Exit Function
End If
Next
End Function

Function NameOnly(ByVal anyStr) As String
Dim I As Integer, J As Integer
I = Len(anyStr)

'check each character for a \ working backwards
For J = I To 1 Step -1
If Mid(anyStr, J, 1) = "\" Then

'when a match is found set the function to the text to the right and exit
NameOnly = Mid(anyStr, J + 1)
Exit Function
End If
Next
End Function

The following is another way to get just the path if the full path and filename are passed to a function.

Function FilePathOnly(FullFileName As String) As String
Dim Pos As Integer
Dim OldPos As Integer
On Error GoTo EndFunction

Pos = 1
While Pos <> 0
OldPos = Pos
Pos = InStr(Pos + 1, FullFileName, "\")
Wend
FilePathOnly = Left(FullFileName, OldPos - 1)
Exit Function
EndFunction:
FilePathOnly = ""
End Function

14.3.2.2 Displaying The Built-In File Selection Dialog To Have The User Select A File

The following displays the built-in file selection dialog and returns the name of the file and the path to a module level variable called fName

'declare this variable at the top of the module

Dim fName As String

Sub Main_Routine()
Sub Select_A_File()
  Dim sFile, I As Integer

  'display dialog asking user to select a file
  sFile = Application.GetOpenFilename _
    ("Files (*.xls),*.xls", , "Select A File")

  'check to see if cancel selected in the box
  If sFile = "False" Then
    MsgBox "No file selected. Activity halted."
  End
  End If
  fName = sFile
End Sub

The following variation of the above allows the user to select multiple files:

'declare this variable at the top of the module. As it is Variant,
it can hold an array of filenames
Dim fileList As Variant

Sub Main_Routine()
  Dim I As Integer

  'call subroutine that displays dialog
  Select_A_File

  'display the file list
  For I = 1 To UBound(fileList)
    MsgBox fileList(I)
  Next
End Sub

Sub Select_A_File()
  Dim sFile, I As Integer

  'display dialog asking user to select multiple files by setting
  'the last argument to True
  fileList = Application.GetOpenFilename _
    ("Files (*.xls),*.xls", , _
    "Select An Invoice File", , True)
'check to see if cancel selected in the box, which cause this to be an error

On Error Resume Next
If fileList(1) = "" Then
    MsgBox "No file selected. Activity halted."
End If

'turn off error checking

On Error GoTo 0
End Sub

14.3.2.3 Using The Excel File Open Dialog To Open Multiple Files

You can use the MultiSelect option of the Application.GetOpenFileName function to display the Excel open file dialog and allow multiple files to be selected.

When the MultiSelect option is invoked, the function returns an array of filenames, even if only one is selected. The dialog does not open any files - it just returns the array of filenames. You could then loop through the list of filenames, performing whatever actions are necessary. The following example illustrates this:

Sub Open_Files()

'file to store list must be of type variant

Dim fList As Variant, I As Integer

'displays the dialog.

fList = Application.GetOpenFileName(MultiSelect:=True)

'check and see if cancel selected, which returns a boolean variable

If TypeName(fList) = "Boolean" Then
    MsgBox "No files selected. Activity halted."
    Exit Sub
End If

'Loops through every file that is selected and open

For I = 1 To UBound(fList)

'open the workbook, but do not update links by setting the 2nd argument to false

    Workbooks.Open fList(I), False
Next
End Sub
14.3.2.4 Setting The Default GetOpenFilename Directory

The default drive that appears when you use Application.GetOpenFilename dialog is the default directory when appears when you select File, Open. The following sets this to a alternate drive and directory, displays the Application.GetOpenFilename dialog, and then sets the drive and directory back to the original setting.

Sub ExampleOfSettingDirectoryAndDrive()
    Dim fNameAndPath As String
    Dim currentDir As String
    Dim currentDrive As String

    'store the current settings
    currentDir = CurDir()
    currentDrive = Left(currentDir, 1)

    'set a drive and directory you want to appear when the dialog is displayed
    ChDrive "C"
    ChDir "C:\Temp"
    fNameAndPath = Application.GetOpenFilename

    'set the drive and directory back to the original setting
    ChDrive currentDrive
    ChDir currentDir
End Sub

14.3.2.5 Finding The Last Modified File In A Directory

The following will return the last modified file in a directory:

Sub test()
    MsgBox LastModifiedFile("C:\", "*")
End Sub

Function LastModifiedFile _
    (sDirPath As String, fType As String) As String
Dim sFile As String
Dim fileDate As Date
Dim fName As String
fName = Dir(sDirPath & ".*" & fType)
While fName <> ""
    If FileDateTime(sDirPath & fName) > fileDate Then
        sFile = fName
        fileDate = FileDateTime(sDirPath & fName)
    End If
    fName = Dir()
Wend
    If sFile <> "" Then
        LastModifiedFile = sFile
    End If
End Function
End If
End Function

14.3.2.6 Chdrive And Network Paths

If you want to change the current directory to a UNC specified location, the following API function will do this:

```vba
Private Declare Function SetCurrentDirectoryA Lib "kernel32" (ByVal lpPathName As String) As Long
```

```vba
Sub SetUNCPath(sPath As String)
    Dim lReturn As Long
    lReturn = SetCurrentDirectoryA(sPath)
    If lReturn = 0 Then _
        MsgBox "Error setting path."
End Sub
```

14.3.2.7 Using FileSearch Instead Of The DIR Command

The `FileSearch` feature is an alternative to the Dir command and having to recurs through subdirectories.

```vba
Sub GettheWholedirectory()
    With Application.FileSearch
        .NewSearch
        .LookIn = "C:\My Documents"
        .SearchSubFolders = True
        .FileName = "*.*"
        .MatchTextExactly = False
        .FileType = msoFileTypeAllFiles
        .Execute

        'Run a second time to insure correct listing
        'As there is a bug with FileSearch on its first run
        If .Execute > 0 Then
            MsgBox "There were " & .FoundFiles.Count & " file(s) found."
            For i = 1 To .FoundFiles.Count
                Cells(i, "A").Value = .FoundFiles(i)
            Next i
        Else
            MsgBox "There were no files found."
        End If
    End With
End Sub
```

If you use `FileSearch` and sort by the last modified date, `FileSearch` needs to be run once in order to "wake it up" to run correctly. First do a dummy `FileSearch`. This dummy `FileSearch` helps insure that the results of the second `FileSearch` are correct. Do not do a dummy
**FileSearch** on a large directory. This will speed up the process by not wasting time (finding & sorting). The following is an example of a dummy search.

```vba
With Application
  .NewSearch
  .LookIn = "C:\"
  .FileName = "*.jnk"
  .Execute SortBy:=msoSortBySize
End With

'Now do your real search
```

Even though there are no matching files to find in the above routine, it is good enough to get the function working correctly the second time around.

### 14.3.2.8 Opening All The Files In A Directory

When you call Dir() with a file spec, it starts looking at file names from the beginning. When you call it a 2nd time with no file spec, it uses the file spec that you supplied on the first call. When there are no more files that match the spec (assuming it contains wild cards), it returns an empty string instead of a file name. But you may find that Dir() still doesn't work correctly if you open a workbook, change it, and save it before going on to the next one. The save operation causes confusion re which files have been read and which haven't.

The usual approach is to use the loop to get the file names and put them in an array. You don't open and process them in this loop. Then you use a 2nd loop to retrieve each name from the array and process the file.

The following code will open all the files in a directory:

```vba
Sub Open_Files_In_A_Directory()
  Dim fileList() As String
  Dim fName As String
  Dim fPath As String
  Dim I As Integer

  'define the directory to be searched for files
  fPath = "C:\my files\"

  'build a list of the files
  fName = Dir(fPath & "*.xls")
  While fName <> ""
    'add fName to the list
    I = I + 1
    ReDim Preserve fileList(I To I)
    fileList(I) = fName
    fName = Dir(fPath & "*.xls")
  Wend

End Sub
```
'get next filename

    fName = Dir()
    Wend

'see if any files were found

    If I = 0 Then
        MsgBox "No files found"
        Exit Sub
    End If

'cycle through the list and open
'just those with the letter Z in the filename
'insure the following way is a case insensitive test

    For I = 1 To UBound(fileList)
        If InStr(1, fileList(I), "Z", 1) > 0 Then
            Workbooks.Open fPath & fileList(I)
        End If
    Next
End Sub

14.3.2.9 Getting Values From A Closed Workbook

The following will return the values on a sheet in a closed workbook and write them to the same range on the active worksheet:

Sub test()
    GetValuesFromAClosedWorkbook "C:", "Book1.xls", _
        "Sheet1", "A1:K30"
End Sub

Sub GetValuesFromAClosedWorkbook(fPath As String, _
    fName As String, sName, cellRange As String)
    With ActiveSheet.Range(cellRange)
        .FormulaArray = "='" & fPath & "\" & fName & "]" _
            & sName & "!'" & cellRange
        .Value = .Value
    End With
End Sub

14.3.2.10 Changing To A Floppy Drive

The following example illustrates how to change the current drive to the floppy drive and handle several likely errors:

Sub Drive_Change()
    On Error GoTo eTrap
    ChDrive "A"
End Sub
Exit Sub
eTrap:
    If Err.Number = 68 Then
    MsgBox "Please Close Drive Door or put in disk"
    Resume
    ElseIf Err.Number = 70 Then
    MsgBox "Permission denied. Please remove write-protection."
    Resume
    ElseIf Err.Number = 71 Then
    MsgBox "Disk not ready. Please insert a disk."
    Resume
    Else
    MsgBox "Drive can not be accessed. Activity halted."
    End
End If
End Sub

14.3.2.11 Open A File Only If No One Else Has It Open

The following will open a file only if it not open in write mode by someone else.

Function OpenFile(fName As String) As Boolean
    On Error GoTo NotOpen
    Set wb = Workbooks.Open(FileName:=fName, notify:=False)
    On Error GoTo 0
    OpenFile = True
    Exit Function
NotOpen:
End Function

14.3.3 COPYING, MOVING, RENAMING, AND DELETING

14.3.3.1 Copying, Moving, And Renaming A File Without Opening It

The following illustrates how to copy a file without first opening it:

Dim oldFile As String
Dim newFile As String

'define the source file and the destination file
oldFile = "c:\temp\book2.xls"
newFile = "c:\active\book2.xls"

'this does the actual copying
FileCopy oldFile, newFile

Please note that you do not get any warning when you are over-writing an existing file with FileCopy. If you want to insure that you are not over-writing a file then use the following before the FileCopy statement:
If Dir(oldFile) <> "" Then
    MsgBox "File " & oldFile & " already exists. " & _
        "File not copied."
End
End If

If you want to move the file, then use the following code:

    Dim oldFile As String
    Dim newFile As String

    'define the files
    oldFile = "c:\temp\book2.xls"
    newFile = "c:\active\book2.xls"

    'this does the move
    Name oldFile As newFile

The directories for the file must be on the same drive. If they are on different drives, then you
must use FileCopy and then delete the original using a Kill statement as illustrated later in this
section.

If the file exists in the destination directory, you will get an error message. You can determine if
the file exists by using statements like the following:

    If Dir(newFile) Then
         'actions to take if file exists in destination directory
    End If

For example, you could delete the existing copy of newFile before moving oldFile by using the
following statement in the above If..End If statement

    Kill newfile

If you want to rename a file, then just specify the same path when you use the Name statement:

    Dim oldName As String
    Dim newName As String
    oldName = "c:\temp\book2.xls"
    newName = "c:\temp\studies.xls"
    Name oldFile As newName

14.3.3.2 Using FileCopy To Copy Files Or Workbooks

The following routine shows how to copy all the files in one directory to another without
opening and re-saving the files
Sub CopyAllXLS()
    Dim fName As String
    fName = Dir("d:/*.xls")
    Do
        FileCopy "d:" & fName , "d:\temp" & fName
        fName = Dir()
    Loop Until fName = ""
End Sub

14.3.3.3 How To Delete A File

If you want to delete a file once you are done with it, then use a Kill statement. Please note that you can not undo such a delete, and the file must not be open. If you want to delete a file but be able to undo the delete, then see the topic "Deleting Files And Directories So That You Can Undelete Them"

The following illustrates how to use a Kill statement.

If Dir("C:\Fidelio\Quotes\temp.xls") <> "" Then _
    Kill "C:\Fidelio\Quotes\temp.xls"
In the above example, the Dir() statement is used to confirm that the file exists. If you don't do this and the file does not exist, you will get an error message. You could also use statements like the following, which would handle an error result with displaying any message:

'Turn on error handling so that error messages are not displayed
On Error Resume Next
'delete the file
Kill "C:\Fidelio\Quotes\temp.xls"
'turn off error handling
On Error GoTo 0
The advantage of the first approach is that you do not have to turn back on an error handling statement that may already be set.

You can also use wildcards with the Kill statement:

Kill "C:\MyFiles\*.Prn"

14.3.3.4 Getting Document And File Properties

Visual Basic provides a number of functions that return information on a file:

FileDateTime("C:\Assistnt\book1.xls")
'returns the date and time last saved (if the file is open it returns that time)

FileLen("C:\Assistnt\book1.xls")

'returns the file size in bytes

The following code will list the different properties of a workbook and their values, if they are numeric or text. Typical properties are the Author, subject, comments, creation date, etc. Some properties don't apply but will appear anyway.

Sub ListProperties()
    Dim rw, p
    rw = 1

    'turn on error handling as some properties can't be written to a cell
    On Error Resume Next

    'loop through each property and list it in the worksheet
    For Each p In ActiveWorkbook.BuiltinDocumentProperties
        Cells(rw, 1).Value = rw
        Cells(rw, 2).Value = p.Name
        Cells(rw, 3).Value = p
        rw = rw + 1
    Next

    'auto fit the columns for easy reading
    Columns("a:c").AutoFit
End Sub

14.3.3.5 Deleting Files And Directories So That You Can Undelete Them

The Kill method of Visual Basic deletes a file, but you can not undelete them as the file does not go to Windows 95's Recycle Bin - it is permanently removed.

If you want to put the file in the Recycle Bin you have to use APIs. The following approach even deletes entire directory structures.

'place the following at the top of a module

Const FO_DELETE = &H3&
Const FOF_ALLOWUNDO = &H40&
Const FOF_NOCONFIRMATION = &H10&

Private Type SHFILEOPSTRUCT
    hWnd As Long
    wFunc As Long
    pFrom As String
End Type
pTo As String
fFlags As Integer
fAnyOperationsAborted As Long
hNameMappings As Long
lpszProgressTitle As String
End Type

'the following two statements should each be on a single line

Private Declare Sub CopyMemory Lib "KERNEL32" Alias "RtlMoveMemory"
(hpvDest As Any, hpvSource As Any, ByVal cbCopy As Long)

Private Declare Function SHFileOperation Lib "Shell32.dll" Alias "SHFileOperationA" (lpFileOp As Any) As Long

'all of the above code should be at the top of your module

'the following illustrates how to call the delete procedure and delete a file

Sub Test()
    ShellDelete "c:\test.txt"
End Sub

'this is the procedure that does the deletion

Sub ShellDelete(SrcFile As String)
    Dim result As Long
    Dim lenFileop As Long
    Dim foBuf() As Integer
    Dim fileop As SHFILEOPSTRUCT
    lenFileop = LenB(fileop)
    ReDim foBuf(1 To lenFileop)
    With fileop
        hwnd = 0
        wFunc = FO_DELETE
        pFrom = SrcFile & Chr(0) & Chr(0)
        fFlags = FOF_NOCONFIRMATION & FOF_ALLOWUNDO
        lpszProgressTitle = "" & Chr(0) & Chr(0)
    End With
    Call CopyMemory(foBuf(1), fileop, lenFileop)
    Call CopyMemory(foBuf(19), foBuf(21), 12)
    result = SHFileOperation(foBuf(1))
End Sub

If you want to delete the active file, you must first close it. The following illustrates this approach

Sub Delete_Active_File()
    Dim iResponse As Integer
    Dim fPathName As String

    'get the path to the file
fPathName = ActiveWorkbook.Path

'if not path, the file has not been saved

If fPathName = "" Then
    MsgBox "The file has not been saved"
    Exit Sub
End If

'add the file name to the path

fPathName = fPathName & "\" & ActiveWorkbook.Name

'get user confirmation to delete the file

iResponse = MsgBox("Do you wish to delete " & fPathName & "?", vbOKCancel)

'exit if cancel selected

If iResponse = vbCancel Then Exit Sub

'delete the file

ShellDelete fPathName

'confirm the deletion back to the user

If Dir(fPathName) = "" Then
    MsgBox fPathName & " was deleted"
Else
    MsgBox fPathName & " was not deleted"
End If
End Sub

14.3.4 SAVING FILES AND WORKBOOKS

14.3.4.1 Getting Just the File Name

If you have run a directory listing, you will end up with full file names. For example: C:\My Documents\ABC.xls

If you want to get just the file name, then use a function like the following:

Function JustFileName(strFullPath As String) As String
    JustFileName = Mid(strFullPath, InStrRev(strFullPath, "\") + 1, 255)
End Function
14.3.4.2 Eliminating The File Exists... Message When Using The SaveAs Method

To eliminate the above message when you use SaveAs to replace an existing file, put the following statement ahead of the SaveAs statement

```
Application.DisplayAlerts = False
ActiveWorkbook.SaveAs "MyBook.Xls"
```

If you want future alert messages to appear as your macro runs, use the following after the SaveAs statement:

```
Application.DisplayAlerts = True
```

14.3.4.3 Saving A File In A New Directory With A New Name

The following code can be used to save a file either with a new name or in a new directory, or both. It a file exists in the directory by the name specified, it is replaced without an alert. The following brings up the standard Excel file save as dialog box. This also allows selection of the directory using the mouse.

```
Dim fileSaveName As Variant
fileSaveName = Application.GetSaveAsFilename( _
    fileFilter:="Excel Files (*.xls), *.xls")
If fileSaveName <> False Then
    Application.DisplayAlerts = False
    ThisWorkbook.SaveAs Filename:=fileSaveName
    Application.DisplayAlerts = True
End if
```

If you want to make certain the user wants to replace an existing file and that the file is saved, then use code like the following, which loops until the user specifies an acceptable file name. The following procedure uses a function to determine if the user wants to replace an existing file

```
Sub SaveExample()
    Dim number As Long
    Dim fileSaveName As Variant

    'loop until the file is saved
    Do

    'get a name and directory from the user
    fileSaveName = Application.GetSaveAsFilename( _
        fileFilter:="Excel Files (*.xls), *.xls")

    'if cancel not selected check if file can be replaced
    'bReplace returns either True or False
```

317
If bReplace(fileSaveName) and fileSaveName <> False Then

'turn off warning and then turn back on after saving

Application.DisplayAlerts = False
ThisWorkbook.SaveAs Filename:=fileSaveName
Application.DisplayAlerts = True

'exit loop since file has been saved

    Exit Do
End If
End Loop
End Sub

Function bReplace(fName) As Boolean
If Dir(fName) <> "" Then

'if the file exists confirm replacement

    If MsgBox("Do you wish to replace " _ 
        & fName & "?", vbYesNo) = vbYes Then

'if Yes clicked, return a True value

        bReplace = True
    Else

'if No clicked return a False value

        bReplace = False
    End If

Else

'If file doesn't exist, return a True value

    bReplace = True
End If
End Function

14.3.4.4 How To Save A Workbook In Excel 5 Format

Workbooks saved in Excel 5 format from Excel 97/2000 will result in the following message being displayed when a user tries to open them in Excel 5: "Can't find project or library". The following routine illustrates how to solve this problem:

Sub SaveWithoutLibraryReference()
    Dim R As Object
    For Each R In ActiveWorkbook.VBProject.References
        If R.Description = "OLE Automation" Then
            ActiveWorkbook.VBProject.References.Remove R
        End If
    Next R
End Sub
14.3.4.5 Make File Saving Mandatory

You can do this with the Workbook_BeforeClose event procedure. Put this code in the ThisWorkbook code module (not a standard code module).

```vba
Private Sub Workbook_BeforeClose(Cancel As Boolean)
    If Not ThisWorkbook.Saved Then
        Cancel = Not ThisWorkbook.Saved
        MsgBox "You must save the file before closing it"
    End
End Sub
```

14.3.4.6 Saving A Backup Copy Of A File

If you use the SaveCopyAs method, you can save a backup copy of the file but not modify any file links that exist to the file:

```vba
ActiveWorkbook.SaveCopyAs "C:\book2.xls"
```

14.3.5 CSV AND ASCII FILES

14.3.5.1 Displaying A Dialog To Have The User Select A CSV File

The following example shows how to display the built-in file open dialog box to have the user select a CSV file, have the procedure open the file and then process the file.

```vba
Sub SelectCSV()
    Dim sFilter As String
    Dim sTitle As String
    Dim sFile As Variant

    'set variables to use in the file open dialog box
    sFilter = "Comma-delimited Files (*.csv),*.csv"
    sTitle = "Please Select a CSV File"

    'display dialog asking user to select a CSV file
    sFile = Application.GetOpenFilename(sFilter, , sTitle)

    'check to see if cancel selected in the box
    If sFile = "False" Then End
```

319
'make certain a CSV file is selected, stop if it is not a CSV file

If LCase(Right(sFile, 4)) <> ".csv" Then
    MsgBox "You did not select a CSV file", _
    vbCritical, "Error"
End If

'Do your processing here.

End Sub

14.3.5.2 Creating CSV Files

The simplest way to create a CSV file is to make a copy of the active sheet and save it as a CSV file:

Sub CreateCSVFile()
    'turn off warning messages
    Application.DisplayAlerts = False
    'copy the active sheet to its own workbook
    ActiveSheet.Copy
    'save the sheet as a CSV file
    ActiveWorkbook.SaveAs Filename:="C:\myData.CSV", _
    FileFormat:=xlCSV, CreateBackup:=False
    ActiveWorkbook.Close False
End Sub

If you just want to save a range on a worksheet as a CSV file, then you can use the following macro instead. It also allows you to change the data separator from a comma to any other text character.

Sub Create_Csv_File()
    Dim F As Long, fName
    Dim J As Long, I As Long
    Dim rng As Range, outputLine As String
    Dim entrySeparator As String
    Dim fCol As Long, lCol As Long, fRow As Long, lRow As Long
    Dim nResponse As Integer
    entrySeparator = ",,"
    'get the next file number available for use
F = FreeFile(0)

' set variable to the current selection and define the starting and ' ending row and column numbers

Set rng = Selection
fCol = rng.Columns(1).Column
fRow = rng.Rows(1).Row

' get a filename from the use

fName = InputBox("Enter the filename with Path. " & _
"The current selection will be written as a CSV file.", _
"Please Enter Output File Name (ex: C:\MYDATA.CSV):")

' see if cancel selected, stop if it was

If fName = False Then
    MsgBox "no action taken"
    End
End If

' check to see if the file exists

If Dir(fName) <> "" Then
    nResponse = MsgBox(fName & " exists. " & _
    "Select OK to replace it", Buttons:=vbOKCancel)

' see if cancel selected, stop if it was

If nResponse <> vbOK Then
    MsgBox "no action taken"
    End
End If

' open the file for output

Open fName For Output As #F

' rotate through all the rows in the selection

For I = fRow To lRow

' initialize outputLine each time through

  outputLine = ""
  For J = fCol To lCol
    If J <> lCol Then
'if not the last column, do this, which puts a separator on the end

    outputLine = outputLine & Cells(I, J).Value & _
    entrySeparator

Else

'no separator put after the last column's value

    outputLine = outputLine & Cells(I, J).Value
End If
Next J

'write the row information to the file

    Print #F, outputLine
Next I

'close the file

    Close #F

MsgBox "Output written to: " & fName
End Sub

14.3.5.3 How To Save As Text File Without Quotations Marks

You will get quotation marks in text files even when you save manually if entry in a single cell has list separator (generally comma). You can avoid quotation marks while saving manually, by changing the list separator in your computer (Control Panel | Regional Settings | Number | List Separator) to some other character other than the character you have in your cell content.

When VBA is involved in saving, the list separator is always considered as comma, irrespective of the setting in control panel and hence, there is no way of saving a text file without quotations as long as commas are present in the content of a single cell. You can consider that VBA is involved even if you use send keys to display the save dialog box which manually is clicked OK to save the text file.

14.3.5.4 Save As CSV Using A Semi-Colon, Not A Comma

Visual Basic (unlike Excel itself) doesn't take often the regional settings in account. It will always save CSV files with the delimiter used in the US (comma). Although disk I/O with text files is very slow in Visual Basic, you should use an appropriate macro. An example is below.

Function Exporte(Wksht As Worksheet, NomFic As String, _
    Optional Remplace As Boolean = True) As Long

    Dim UsedRange As Range
    Dim NbCols As Integer, NbLignes As Long
    Dim Stat As Integer, Incr As Integer
    Dim I As Integer, J As Integer
    Dim Progr As Integer

On Error GoTo Erreur
If Dir(NomFic) < "" And Not Remplace Then
Exporte = -1
   Exit Function
End If
Open NomFic For Output As #1
Set UsedRange = Wksht.UsedRange
NbCols = UsedRange.Columns.Count - 1
NbLignes = UsedRange.Rows.Count
Stat = NbLignes / 40
Incr = Stat
For I = 1 To NbLignes
   If I = Stat Then
      Stat = Stat + Incr
      Progr = Progr + 1
      ApplicationStatusBar = _
         "Exportation " & String(Progr, ".")
   End If
For J = 1 To NbCols
   Print #1, Cstr(UsedRange(I, J)) & ";
Next J
Print #1, Cstr(UsedRange(I, J))
Next I
Close 1
GoTo Fin
Erreur:
Exporte = Err
Fin:
   ApplicationStatusBar = False
End Function

Sub Test()
   Dim Result As Long
   Result = Exporte(ActiveSheet, "Test.csv", False)
   Select Case Result
      Case -1
         MsgBox "File does Not exist", vbExclamation
      Case 0
         MsgBox "File exported."
      Case Else
         MsgBox Error(Result)
   End Select
End Sub

14.3.5.5 Reading A Text File Line By Line

An alternate way to read a text file is line by line. This allows you to treat each line individually. The following shows how to do this:

Sub Read_Ascii_File_Line_By_Line()
   Dim I As Long, myString
   'reset any file accidentally left open
Reset

' Open file for input.

Open "C:\TESTFILE.TXT" For Input As #1
i = 1

' Loop until end of file.

Do While Not EOF(1)

' Read data into a variable

Input #1, myString

'write output to a cell on the active sheet

ActiveSheet.Cells(i, 1) = myString
i = i + 1
Loop

' Close file.

Close #1
End Sub

14.3.5.6 Issues With Reading CSV Files

VBA uses the default delimiter for the International Version of Excel when it reads in CSV files regardless of what the list delimiter is set to in Win 95 Regional Settings. If you have an US version of Excel, it will use a comma as the delimiter. If you have a localized version and assuming the semi-colon is your list delimiter, then it would use that. Bringing the file in manually through file open, this behavior can be changed (by changing the regional settings) - but not in VBA. If you must read in the file using VBA, you probably need to write your own File reading routine and interpret the delimiters yourself. Your code segment appears to set the delimiter properly as Semi-Colon, but Excel may be "intelli-sensing" the fact that it is a CSV file. You could also try putting the extension ".txt" on the file rather than ".csv" and see if Excel will behave. Also, be aware that Excel Visual Basic behaves differently for reading CSV files in the debugger versus when actually run.

14.3.5.7 Writing Directly To An ASCII File

The following illustrates writing to an ASCII file:

Sub MyOutput()
    Dim iMyFreeFile As Integer
    Dim string1 As String
    Dim var1
    string1 = "hello"
    var1 = 3
' get a free file number
IMyFreeFile = FreeFile

' open the text file - please note this replaces the file
Open "c:\xltext.txt" For Output As #IMyFreeFile

' write whatever you want
Write #IMyFreeFile, string1, var1

' as long as you write on one line, the data is comma delimited
' if you add another write statement, the data will be on the next line
Write #IMyFreeFile, string1, var1

' now close the file
Close #IMyFreeFile
End Sub

14.3.5.8 Sheet/Range Extract To ASCII Files

This VBA procedure will save the active worksheet as space delimited text:

Sub SaveAsText()
    ActiveWorkbook.SaveAs _
    FileName:="C:\My Documents\MyFile.txt", _
    FileFormat:=xlTextPrinter
    'above saves as space delimited text files (normal text file)
End Sub

This VBA procedure will save the active worksheet as comma delimited:

Sub SaveAsText()
    ActiveWorkbook.SaveAs _
    FileName:="C:\My Documents\MyFile.csv", _
    FileFormat:=xlCSV
End Sub

To use an extension other than txt or csv, specify it as part of the filename.

If you wish to return to the previous workbook and not have to save and reopen it, then first copy the active sheet to a new workbook, run one of the two above routines, and then close the active workbook (which is the copy of the sheet;
ActiveSheet.Copy
SaveAsText
ActiveWorkbook.Close (False)

To replace the file without prompting if it exists, use

Application.DisplayAlerts = False

Be sure to set Application.DisplayAlerts back to True immediately as this is a permanent change during the Excel session, and stays in effect even after the macro completes.

14.3.5.9 Importing Text Files

Excellent ways to import text files are found on Chip Pearson's web site

http://www.cpearson.com

go to Importing Text Files in the News section. The code is brilliant and simple to modify.

If you use the import wizard, problems can occur by one's regional settings in windows, being different than in UK / US. This makes the text import wizard confused when American VBA code tries to open text delimited in regional format used by the rest of excel.

14.3.5.10 CSV Files And Non-U.S. Settings

One of the problems with Excel is that it assumes that everyone is using American or U.S. settings! Obviously not true. However, because of this assumption, one that not always import CSV files correctly – the dates and numbers get screwed up. One way that frequently solves is to re-name the CSV file to TXT and import into Excel using an OpenText statement. If you use the macro recorder, you can record the OpenText macro statements.

In Excel 97, if you save a file manually, Excel will use your regional settings to separate the entries. However, if you use a macro statement to save a worksheet as a CSV file, Excel uses the U.S. separator character, which is the comma. The following code will help you create a CSV file using any separator that you want. This code also allows you to create a CSV file from any range on your worksheet without having to copy to a new worksheet and saving the worksheet as a CSV file

Sub Create_A_CSV_File()
    Dim anyRange As Range
    Dim rowRange As Range
    Dim cell As Range
    Dim anyEntry As String
    Dim sSeparator As String
    On Error Resume Next
    Set anyRange = Application.InputBox(_
                                          "Select A Range", _
                                          "Select A Range", _
                                          Default:=Selection.Address, Type:=8)
    On Error GoTo 0
End Sub
If anyRange Is Nothing Then Exit Sub
'set regional separator
sSeparator = Application.International(xlListSeparator)
'Change the filename to any name you want
Open "c:\TEMP.CSV" For Output As #1
'rotate through each row
For Each rowRange In anyRange.Rows
    anyEntry = ""
    'rotate through the cells on the row
    For Each cell In rowRange.Cells
        anyEntry = anyEntry & cell.Value & sSeparator
    Next
    'remove un-needed closing separator
    If Right(anyEntry, 1) = sSeparator Then
        anyEntry = Left(anyEntry, Len(anyEntry) - 1)
    End If
    Print #1, anyEntry
Next
Close #1
End Sub

14.3.5.11 Read Text File With Variable Length Records

You can use Line Input and EOF to read in a text file with a varying number of records:

Sub Get_A_Line_At_A_Time()
    Dim F As Integer
    Dim ARecord As String
    F = FreeFile
    Open MyFile For Input As #F
    Do While Not EOF(F)
        Line Input #F, ARecord
        'code here to test the length and parse it, as needed
        Loop
    Close #F
End Sub

14.3.5.12 Importing Text File With Any Delimiter

Chip Pearson's code at

http://www.cpearson.com/excel/imptext.htm

lets you import any text file with the delimiter of your choice.

14.3.5.13 Saving The Active Sheet As A Comma Delimited File

Here is a simple procedure that exports the active sheet into a comma-delimited text file.
Sub ExportActiveSheetAsText()
    ActiveSheet.CCopy
    ActiveWorkbook.SaveAs "TextFile.csv", xlCSV
    ActiveWorkbook.Close False
End Sub

If the text file exists, then add the following code before the SaveAs so that you are not prompted with a dialog asking if you want to replace the text file.

    Application.DisplayAlerts = False

To turn alerts back on, set the above to True.
15. PRINTING

15.1 A Fast Way To Set The Page Setup

VBA's PageSetup method is extremely slow. Many programmers resort to using the Excel 4 macro language equivalents which are much faster (but only work in the English version of excel). Here's an example that sets the header and footer:

```vba
Application.ExecuteExcel4Macro _
("PAGE.SETUP("&CTest Header","Page &P")")
```

The Excel 4 Page.Setup function can set several settings in one shot. If you're not familiar with the Excel 4 macro syntax you should get the Excel 4 macro help file from MS's web site. Search Microsoft for MacroFun.Exe, download, and install.

15.2 How To Speed Up Changing Print Settings

When you record code that changes print settings, the macro recorder will record about 30 print setup properties. Each change in properties causes a slight execution delay. And, if the printer is a system or network printer, as opposed to a PC printer, the delay becomes very noticeable.

There are two basic tricks that will speed up your code:

- Only change the print settings that need changing

- Change the active printer to a PC printer, change the settings, and change the printer back to the original printer. The easiest way to get the code that changes the printer is to record it.

15.3 How To Set The Print Area

To set the print area of a sheet, you need to set the PageSetup.PrintArea property equal to the address of a range.

```vba
ActiveSheet.PageSetup.PrintArea = Selection.Address
```

You can set the print area on any worksheet, not just the active worksheet. For example, the following sets the print area on all worksheets equal to the selection on the active sheet, and prints all the worksheets:

```vba
Dim rng As String
Dim ws As Worksheet
rng = Selection.Address
For Each ws In Worksheets
    ws.PageSetup.PrintArea = ws.Range(rng) _
        .Address(external:=True)
```
The key to the above code is specifying the address with the `external` argument set to `True`, which returns the full address, including sheet and workbook name.

The following are several examples of setting the print area

\[
\text{Worksheets("Sheet1").PageSetup.PrintArea} = "A$1:B5"
\]

\[
R = 5 \\
C = 2 \\
\text{ActiveSheet.PageSetup.PrintArea} = _ \\
\text{Range(Cells(1, 1), Cells(R, C)).Address}
\]

\[
\text{ActiveSheet.PageSetup.PrintArea} = \text{Selection.Address}
\]

### 15.4 Determining The Print Area

The following statements will return a string which represents the print area on a worksheet:

\[
\text{MsgBox ActiveSheet.PageSetup.PrintArea}
\]

or

\[
\text{MsgBox Worksheets(index).PageSetup.PrintArea}
\]

This returns a string as an address, e.g. "A2:D55".

If you want to set a variable to this range, then use statements like the following:

\[
\text{Dim printR As Range}
\]

\[
\text{Set printR = Range(ActiveSheet.PageSetup.PrintArea)}
\]

OR

\[
\text{Dim printR As Range}
\]

\[
\text{With Worksheets("Sheet1")}
\]

\[
\text{Set printR = .Range(.PageSetup.PrintArea)}
\]

\[
\text{End With}
\]

In the last example, notice that there are periods in front of `Range` and `PageSetup`. This links a method or property back to the object specified in the `With` statement.
Please note the above examples assume that there is a print area on the worksheet. If there is not, the above will crash. To avoid the crash, either test to see if there is a print area, or use an On Error statement to handle the error:

```
Sub Select_The_Print_Area()
    Dim printR As Range

    'test to see if there is a print area set
    If ActiveSheet.PageSetup.PrintArea <> "" Then

        'if there is an area set printR to refer to this area
        Set printR = Range(ActiveSheet.PageSetup.PrintArea)
    End If

    'test to see if printR is set to a range, display a message if it is not
    'and select the range if it is set.
    If printR Is Nothing Then
        MsgBox "No print area"
    Else
        printR.Select
    End If
End Sub
```

### 15.5 Enlarging A Print Area Range

There are many methods to do this. Here are a couple that use the Resize(rows, columns) function. The Resize function allows you to specify the new number of rows and columns. It defaults to the existing number if an argument is not supplied.

```
Set rng = Range("print_area")
rng.Resize(rng.Rows.Count + 3).Name = "Print_Area"
```

or

```
Set rng = Range("print_area")
rng.Resize(rng.Rows.Count + 3, _
rng.Columns.Count - 2).Name = "Print_Area"
```

### 15.6 Add Or Exclude An Area From Print_Area

It looks as if Excel has a little feature that is not activated and must be activated via a macro. It is the ability to add a range to the current print area or exclude an area from the print area via a menu choice.

To see this feature in action first make sure the active sheet has a print area defined on it (File, Print Area, Set Print Area). Then run this macro:
Sub MakePrnAreaTB()
    On Error Resume Next
    CommandBars("Temp").Delete
    CommandBars.Add "Temp", , True
    CommandBars("Temp").Controls.Add _
        msoControlButton, 1583, , True
    CommandBars("Temp").Controls.Add _
        msoControlButton, 1586, , True
    CommandBars("Temp").Visible = True
End Sub

Now this toolbar can have 3 states (but, again, only if a print area has been set): If the current
selection is not in or partly in the print area the tool button "Add to Print Area" will appear. If the
current selection comprises all of one of the areas that make up the print area the tool button
"Exclude from Print Area" will appear (e.g., the print area is A1:B10 and A1:B10 is selected or
the print area is A1:B10,D5:F15 and D5:F15 is selected. If neither condition is met no tool
button will appear on the toolbar.

The macro above creates a new toolbar just for demonstrating the feature. The natural place for
this is on the Cell popup (right clicking on a cell). To add it there run this macro:

Sub AddPrnAreaCtrls()
    RemovePrnAreaCtrls
    With CommandBars("Cell")
        .Controls.Add msoControlButton, 1583
        .Controls.Add msoControlButton, 1586
    End With
End Sub

Sub RemovePrnAreaCtrls()
    On Error Resume Next
    With CommandBars("Cell")
        .Controls("Add To Print Area").Delete
        .Controls("Exclude from Print Area").Delete
    End With
End Sub

You only have to run this once as the addition is "permanent" at least until you reset your
toolbars or run the remove macro. The next time you run Excel it should be there.

15.7 Updating The Header Or Footer Before Printing

You can automatically update the print footer, header or other print settings before a sheet is
printed. To do this, you would put the code to update the settings in the worksheet's module
before print event. The following example changes the left header to the value in cell A1 of the
sheet:

Private Sub Workbook_BeforePrint(Cancel As Boolean)
    ActiveSheet.PageSetup.LeftHeader = Range("A1").Value
End Sub

The following prompts the user for a description for the footer before printing the active sheet:
Private Sub Workbook_BeforePrint(Cancel As Boolean)
Dim userResponse

'turn off EnableEvents so this event is not triggered again by the printout statement below
Application.EnableEvents = False

' set Cancel to True to cancel the printout request from the user
Cancel = True

' get a description from the user for the footer
userResponse = Application.InputBox( _
    prompt:="Enter a description for the footer", _
    Default:=ActiveSheet.PageSetup.LeftFooter, _
    Type:=2)

' if cancel is selected, bypass printing the sheet and changing the footer
If userResponse <> False Then
    ActiveSheet.PageSetup.LeftFooter = userResponse
    ActiveSheet.PrintPreview
End If

'turn event handling back on
Application.EnableEvents = True
End Sub

15.8 Restricting Options in PrintPreview

If you use the statement

Activesheet.PrintPreview False

in your code, then the Margins and Setup buttons are disabled. If you first immediately follow it with

ActiveWindow.View = xlNormalView

then the user can not convert to page break mode by use of that button on the preview screen.

15.9 Memory Problems With Page Setup

If you set the page setup using VB code, this causes a memory link. Using the Excel 4 equivalent macro works does not appear to cause this problem:
Application.ExecuteExcel4Macro("PAGE.SETUP(...)"

The following illustrates the syntax and its use (all must be on one line):

`PAGE.SETUP(Header, Footer, LeftMargin, RightMargin, TopMargin, BottomMargin, RCHeadings, Gridlines, HorizCenter, VertCenter, Orientation, PaperSize, Scale, PageNum, PageOrder)`

Where:

Orientation: 1 = Portrait; 2 = Landscape

PaperSize: 1 = Letter; 5 = Legal

PageOrder: 1 = TopToBottom, then Right; 2 = LeftToRight, then Down

Scale: TRUE = Fit to a page; To specify a percentage of reduction or enlargement, set scale to the percentage

To format the Header and Footer text, precede it with the following formatting codes:

&L to left-align the characters that follow

&C to center the characters that follow

&R to right-align the characters that follow

&B turns bold on or off

&I turns italics on or off

&U turns underlining on or off

&D date

&T time

&P page number

&F the filename

&& inserts an ampersand

&"fontname" prints the characters that follow in the font specified by 'fontname' (use double quotation marks)

&nn prints characters that follow in the font size specified by 'nn' (use 2 digit number)
&N prints the total number of pages in the document

The same function for chart sheets is a little different (again, all arguments are optional):

\[
\text{PAGE.SETUP (Header, Footer, LeftMargin, RightMargin,}
\text{ TopMargin, BottomMargin, ChartSize,}
\text{ HorizCenter, VertCenter, Orientation,}
\text{ PaperSize, Scale, PageNum)}
\]

ChartSize 1 = ScreenSize; 2 = FitToPage; 3 = FullPage

The following illustrates the code:

\[
\text{Sub PageSetupXLM4()}
\]

'Fill Header using XLM

' Note in the next line the use of the double sets of quotation marks (""")
' also, DO NOT USE a line continuation character – make all on one line

\[
\text{ExecuteExcel4Macro "PAGE.SETUP(""&LThis is the left header&CThis is the center header&RThis is the right header"",""&LThis is the left footer&CThe center footer&RThis is the right footer"")"}
\]

End Sub

15.10 How To Fit The Printout To One Page

If you record a macro that changes the page setup, you will get about 40 lines of code. If all you need to do is to fit the printout to one page, then just use the following statements:

\[
\text{With Worksheets("Sheet1").PageSetup}
\text{.Zoom = False}
\text{.FitToPagesTall = 1}
\text{.FitToPagesWide = 1}
\text{End With}
\]

15.11 Controlling Printing

Excel allows you to control printing by the Workbook_BeforePrint event. This event is one that is located in the workbook's code module. To access this module, select the workbook object in the project explorer and click on the view code button at the top of the project explorer. Note that you need to turn off Application.EnableEvents before printing out.

The following code illustrates using the Workbook_BeforePrint event.

\[
\text{Private Sub Workbook_BeforePrint(Cancel As Boolean)}
\]
'turn off EnableEvents so this event is not triggered again by the printout 'statement below

Application.EnableEvents = False

'set Cancel to True to cancel the printout request from the user

Cancel = True

'add a page break before printing

With ActiveSheet
 .HPageBreaks.Add Before:=Range("B10")
 .PrintOut
 .HPageBreaks(1).Delete
 End With

'turn EnableEvents back on

Application.EnableEvents = True
End Sub

15.12 Printing Directly To A Printer

You can open the printer as though it is a file.

Sub TestPrintToPrinter()
    Dim F As Integer
    F = FreeFile()
    Open "Lpt1:" For Output As #F
    Print #F, "abcde"; Space$(5); "fghij"
    Print #F, Chr$(12);
    Close #F
End Sub

15.13 How To Have The User Change The Active Printer

The following statement will display a dialog that lets the user change the active printer. If Cancel is selected, then it returns False.

bResponse = Application.Dialogs(xlDialogPrinterSetup).Show
If TypeName(response) ="Boolean" Then Exit Sub

15.14 How To Determine The Number Of Pages That Will Print
There is no VBA function to do this but you can use an Excel 4 macro function to find that information for the active sheet:

```vba
Dim nPages As Integer
nPages = ExecuteExcel4Macro("GET.DOCUMENT(50)")
```

The following is a more elaborate example that gets the number of pages for each worksheet in the active workbook and displays that number, and the total number of pages

```vba
Sub MainRoutine_GetPages()
    Dim W As Workbook
    Dim Pages() As Integer
    Dim I As Integer, J As Integer
    Set W = ActiveWorkbook

    'call subroutine that returns the number of pages in an array
    GetPageCounts W, Pages()

    'display the number of pages by worksheet
    For I = LBound(pages) To UBound(Pages)
        MsgBox W.Worksheets(I).Name & ": " & Pages(I) & ": pages"
    Next I

    'count the total pages
    J = J + Pages(I)
    Next I

    'display the total number of pages
    MsgBox "Total pages: " & J
End Sub

Sub GetPageCounts(WB As Workbook, PageCounts() As Integer)
    Dim SheetName As String
    Dim NumSheets As Integer
    Dim I As Integer
    With WB

        'get the number of sheets
        NumSheets = .Worksheets.Count

        'resize the array to this number
        ReDim PageCounts(1 To NumSheets) As Integer

        'get the number of pages by worksheet using an excel 4 statement
```
For I = 1 To NumSheets
    SheetName = .Worksheets(I).Name
    PageCounts(I) = Application.ExecuteExcel4Macro("Get.Document(50,"" & SheetName & "")")
Next I
End With
End Sub

15.15 Getting The Number Of Pages That Will Print

The following statements will return the number of pages that will print on the active worksheet. It uses Excel 4 macro code to do this.

Dim Cmd As String
Dim pagesToPrint As Integer

Cmd = "GET.DOCUMENT(50,"" & ActiveSheet.Name & "")"
pagesToPrint = Application.ExecuteExcel4Macro(Cmd)

If you want, you can replace the ActiveSheet.Name with a reference to any sheet in the active workbook. This can be by a variable which just replaces ActiveSheet.Name, or by hard coding in the name as shown below:

Cmd = "GET.DOCUMENT(50,"" & ""Sheet2"" & "")"
pagesToPrint = Application.ExecuteExcel4Macro(Cmd)

15.16 Printing Using Range Names

If you record your actions of going to a defined range and then setting the print area and printing, you will get code like the following:

Application.Goto Reference:="Definitions"
ActiveSheet.PageSetup.PrintArea = "$A$2:$AH$158"
ActiveWindow.SelectedSheets.PrintOut Copies:=1

The problem with the above recording is that the second line refers to a fixed cell range, which may not in the future be the same as the range "Definitions" because of addition or deletion of rows or columns.

However, you can print ranges out directly, without having to set the print area:

Range("Definitions").PrintOut
Range("Routes").PrintOut

If the ranges are on worksheets other than the active sheet, then qualify the range with the worksheet and if in a different workbook, the workbook:

Worksheets("Definitions").Range("Definitions").PrintOut
WorkBooks("Book1.Xls").Sheets("Trip").Range("Routes").PrintOut
The advantage of using range names instead of range references such as "B4:D5" is that the range will refer to the section you want to print out even if you add or delete rows or columns.

Please note that is best to qualify the Range statement with the sheet containing the range name.

15.17 Adding Page Breaks To Your Code

The following statements add horizontal page breaks:

```vba
Rows(9).PageBreak = xlManual
```

The following statements add vertical page breaks:

```vba
Columns("D").PageBreak = xlManual
Columns(3).PageBreak = xlManual
```

15.18 Determining PageBreaks Locations

In Excel, you can use the HPageBreaks collection to determine your horizontal page break Rows. The following (by John Green) displays the row number of all the manually inserted horizontal page breaks and the automatic ones in the print range. It is necessary to run this in Page Break View, so that the breaks are set.

```vba
For Each hpb In ActiveSheet.HPageBreaks
    MsgBox hpb.Location.Row
Next hpb
```

The following code puts borders around the pages to be printed.

```vba
Sub BorderPages()

    'Places borders around each page to be printed
    Dim hpb As HPageBreak
    Dim vpb As VPageBreak
    Dim rngPrintArea As Range

    'Clear all existing borders
    Cells.Borders.LineStyle = xlNone

    'If no Print Area set, set Print Area to used range
```
If ActiveSheet.PageSetup.PrintArea = "" Then
    Set rngPrintArea = ActiveSheet.UsedRange
    ActiveSheet.PageSetup.PrintArea = _
        ActiveSheet.UsedRange.Address
Else
    Set rngPrintArea = Range(ActiveSheet.PageSetup.PrintArea)
End If

'Put border around Print Area

With rngPrintArea
    .Borders(xlEdgeTop).LineStyle = xlContinuous
    .Borders(xlEdgeLeft).LineStyle = xlContinuous
    .Borders(xlEdgeRight).LineStyle = xlContinuous
    .Borders(xlEdgeBottom).LineStyle = xlContinuous
End With

'Use PageBreak View to ensure PageBreaks collection is populated

ActiveWindow.View = xlPageBreakPreview

'Place borders at all Page Breaks

For Each hpb In ActiveSheet.HPageBreaks
    hpb.Location.Offset(-1).EntireRow._
        .Borders(xlEdgeBottom).LineStyle = xlContinuous
    hpb.Location.EntireRow.Borders(xlEdgeTop)._
        .LineStyle = xlContinuous
Next hpb

For Each vpb In ActiveSheet.VPageBreaks
    vpb.Location.Offset(0, -1).EntireColumn._
        .Borders(xlEdgeRight).LineStyle = xlContinuous
    vpb.Location.EntireColumn.Borders(xlEdgeLeft)._
        .LineStyle = xlContinuous
Next vpb
ActiveWindow.View = xlNormalView
End Sub

15.19 Locating Page Breaks

The first 2 of the following functions returns the row numbers and column numbers of the horizontal and vertical page breaks using the HPageBreaks and VPageBreaks collections. The last 2 do it the old-fashioned way, with XLM functions.

Function PageBreakRows() As Variant
    Dim V As Variant, N As Long, i As Long

    Application.ScreenUpdating = False
    ActiveWindow.View = xlPageBreakPreview
    With ActiveSheet
        N = .HPageBreaks.Count
        ReDim V(0 To N)
    End With
    V(0) = 1
For i = 1 To N
    V(i) = .HPageBreaks(i).Location.Row
Next i
End With
ActiveWindow.View = xlNormalView
Application.ScreenUpdating = True
PageBreakRows = V
End Function

Function PageBreakColumns() As Variant
    Dim V As Variant, N As Long, i As Long
    Application.ScreenUpdating = False
    ActiveWindow.View = xlPageBreakPreview
    With ActiveSheet
        N = .VPageBreaks.Count
        ReDim V(0 To N)
        V(0) = 1
        For i = 1 To N
            V(i) = .VPageBreaks(i).Location.Column
        Next i
    End With
    ActiveWindow.View = xlNormalView
    Application.ScreenUpdating = True
    PageBreakColumns = V
End Function

Function ColBreaks() As Variant
    Dim V As Variant
    Dim i As Long, b As Long
    ReDim V(0 To 0)
    V(0) = 1
    On Error Resume Next
    i = 1
    Do
        b = _
        ExecuteExcel4Macro("INDEX(GET.DOCUMENT(65)," & i & ")")
        If Err Then Exit Do
        ReDim Preserve V(0 To i)
        V(i) = b
        i = i + 1
    Loop
    ColBreaks = V
End Function

Function RowBreaks() As Variant
    Dim V As Variant
    Dim i As Long, b As Long
    ReDim V(0 To 0)
    V(0) = 1
    On Error Resume Next
    i = 1
    Do
        b = _
        ExecuteExcel4Macro("INDEX(GET.DOCUMENT(64)," & i & ")")
        If Err Then Exit Do
        ReDim Preserve V(0 To i)
        V(i) = b
        i = i + 1
    Loop
    RowBreaks = V
End Function
If Err Then Exit Do
ReDim Preserve V(0 To i)
V(i) = b
i = i + 1
Loop
RowBreaks = V
End Function

15.20 How To Find Next Automatic Page Break

First, define a name in the worksheet, like PBs with a refers to as =GET.DOCUMENT(64). This will give an array of rows where the pagebreaks are. You can now use INDEX(PBS,1) to get the first row, INDEX(PBs,2) for the 2nd. A number 4 means the page break is between rows 3 & 4.

Several other approaches are:

For j = 1 To ActiveSheet.HPageBreaks.Count
MsgBox ActiveSheet.HPageBreaks(j).Location.Row
Next

For j = 1 To ActiveSheet.VPageBreaks.Count
MsgBox ActiveSheet.VPageBreaks(j).Location.Column
Next

or

For Each pb In ActiveSheet.VPageBreaks
MsgBox pb.Location.Column
Next

15.21 Removing Page Breaks

You can remove all page breaks on a sheet with the following statement in all versions of Excel:

Cells.PageBreak = xlNone

The documentation in Excel says that you should use xlPageBreakNone, but no such constant exists.

To remove both horizontal and vertical page breaks at a particular location use a statement like the following:

ActiveCell.PageBreak = xlNone

To remove just vertical page breaks, use a statement like the following:

ActiveCell.EntireColumn.PageBreak = xlNone

to remove just a horizontal page break, use a statement like the following:
Cells(3,4).EntireRow.PageBreak = xlNone

15.22 Printing Each Row In A Selection Onto A Separate Page

The following code will print out each row in a selection onto a separate page. For example, if you select the range A1:D9, the page one would have A1:D1, page two would have A2:D2, and so forth:

`Dim rng As Range
Dim r As Range

'restrict the range to the used range so that entire rows are not printed
Set rng = Intersect(ActiveSheet.UsedRange, Selection)

'rotate through each row in the selected range
For Each r In rng.Rows

'printout the row
r.PrintOut
Next`

15.23 Printing From A Dialogsheet

You can print directly from a userform. However you can not print directly from a dialogsheet. Instead, you must dismiss the dialog before printing. This example shows how to write your code so that you can display a dialog with buttons, with each set to print a different report. When the user clicks on a button, the dialog is dismissed, the report printed, and the dialog redisplayed.

In this example, the dialog sheet is named "Report Dlg", and it has two buttons in addition to the OK and Cancel buttons. Button 1 is for report 1 and Button 2 is for report 2. Both buttons have had their dismiss properties turned on. To do this, select a button, and click on the control properties buttons. Click on the dismiss property. When the dismiss property is set on, it causes the dialog to close when the button is clicked.

Also, button 1 has been assigned macro "Report1Flag" and button 2 has been assigned macro "Report2Flag". This is done by selecting a button and right clicking on it. In the pop-up that appears, select the assign macro option and select a macro. Macros "Report1Flag" and "Report2Flag" do a very simple task: They set a module level variable to a value of 1 or 2. This variable is then used in the main macro to print the appropriate report.

For the following code to work, you will need to create a worksheet named "My Reports" and name two ranges "Report1" and "Report2"

The following is the code:
'this variable is declared at the top of the module, before any macros

Dim reportNumber As Integer

Sub Print_Reports()

'create a loop that continually displays the dialog

Do

'initialize the reportNumber variable before each use

reportNumber = 0

'display the dialog; exit loop if cancel selected

If Not DialogSheets("Reports Dlg").Show Then Exit Do

'run print report macro based on reportNumber value

Select Case reportNumber
  Case 1: Print_Report1
  Case 2: Print_Report2
End Select

'loop to redisplay the dialog

Loop
End Sub

Sub Report1Flag()

'set flag variable for report 1

'assign this macro to the report 1 button

reportNumber = 1
End Sub

Sub Report2Flag()

'set flag variable for report 2
'set this macro to the report 2 button

reportNumber = 2
End Sub

Sub Print_Report1()

'set the print area using a range name on the worksheet
Sub Print_Report2()

' this is like the above except it uses a With..End With construction to
'make the code more efficient

With Sheets("My Reports")
  .PageSetup.PrintArea = .Range("report_2").Address
  .PrintOut
End With
End Sub

15.24 How To Printout A Sheet Or An Entire Workbook

If you want to print a sheet, use the PrintOut method: For example:

ActiveSheet.PrintOut

or use an object variable

Dim oSheet As Worksheet
Set oSheet = Workbooks("MyBook.Xls").Sheets("sheet1")
oSheet.Printout

If you want to print an entire workbook, then specify the workbook as the object to the Printout method. For example:

ActiveWorkbook.Printout

or

Workbooks("MyBook.Xls").Sheets("sheet1").Printout

15.25 Printing All The Files In A Directory

The following code illustrates how to open all the files in a directory, print the entire workbook, and then close the files.

Dim FName As String
Dim WB As Workbook
'get the first workbook to open

FName = Dir("C:\Tmp\*.xls")
While FName <> ""

'open the workbook and set to a variable

Set WB = Workbooks.Open(Filename:="c:\tmp\" & FName, _
    updatelinks:=False)

'print and then close the workbook

WB.PrintOut
WB.Close False

'get the next workbook

FName = Dir()
Wend

The following code opens all the files in a directory, prints all the sheets in the files, saves the file to a new directory, and then deletes the file from the original directory.

Sub PrintFiles()
    Dim XLSFiles() As String
    Dim NumFiles As Integer
    Dim N As Integer
    Dim oS

    'use user defined function to populate the array of files to open
    'and to return the number of files to open

    NumFiles = GetFileNames(XLSFiles())

    'only open files if there are any to open

    If NumFiles > 0 Then
        For N = 1 To NumFiles

            'open workbook, do not update links

            Workbooks.Open XLSFiles(N), False

            'print all the worksheets in the workbook

            For Each oS In WorkSheets
                oS.PrintOut
            Next

            'save the workbook to another directory
            ' the with statement avoids having to type ActiveWorkbook in front

        Next
    End If
End Sub
'of Save and in front of Name.
'Periods are required in front of Save and Name.

With ActiveWorkbook
  .Save "c:\New dir" & .Name
End With

'delete the original file. Note, file is not recoverable

Kill XLSFiles(N)
Next fName
End If
End Sub

'this function's argument is a string array and
'the function returns an integer value

Function GetFileNames(FileNames() As String) As Integer
  Dim fName As String
  Dim N As Integer

  N = 0
  fName = Dir$("C:\First Directory\*.xls")
  Do While fName <> ""

  'increase the number of files found each time through

  N = N + 1

  'expand the array and retain existing values

  ReDim Preserve FileNames(1 To N) As String
  FileNames(N) = "C:\First Directory\" & fName
  fName = Dir$()
  Loop

  'set the function equal to the number of files found

  GetFileNames = N
End Function

15.26 Printing Embedded Charts

If you use the recorder to record the actions of printing a series of embedded charts, you will get a lot of lines. The following illustrates the recorder code to print just two embedded charts:

Sub RecorderCodeToPrintEmbeddedCharts()
    ActiveSheet.DrawingObjects("Chart 1").Select
    ActiveSheet.ChartObjects("Chart 1").Activate
    ActiveChart.PrintOut
    ActiveWindow.Visible = False
Windows("Book2").Activate
ActiveSheet.DrawingObjects("Chart 2").Select
ActiveSheet.ChartObjects("Chart 2").Activate
ActiveChart.PrintOut
ActiveWindow.Visible = False
Windows("Book2").Activate
Range("H28").Select
End Sub

The same code can be condensed to just two statements:

Sub PrintEmbeddedCharts()
    ActiveSheet.DrawingObjects("Chart 1").Chart.PrintOut
    ActiveSheet.DrawingObjects("Chart 2").Chart.PrintOut
End Sub

As information, the name of the drawing object is displayed in the name window when you select the drawing object. You can edit this name to any name you want.

15.27 Case Of The Disappearing PageBreak Constant

Visual Basis doesn't recognize the constant xlPageBreakNone, even though the online documentation says this is the one to use. This is a known bug in Excel. Use xlNone rather than xlPageBreakNone.

The following is how you can use this constant to clear all the page breaks from a worksheet.

Sub ClearBreaks()
    WorkSheets("Template").Cells.PageBreak = xlNone
End Sub

15.28 Changing the Paper Type on each Sheet in a Workbook

If you share workbooks with people in other countries, you may find that you need to change the paper type on all the sheets in the workbooks before you can print. For example, the standard paper size in the U.S. is 8 ½ by 11, and in Europe it is A4, which is 210 by 297 mm.

The following code changes all the sheets to the same paper type:

Sub ChangePaperType()
    Dim oS

' use a For..Next loop to modify all the sheets.
' Note that the sheets do not have to be activated to change the setting

    For Each oS In Sheets
        oS.PageSetup.PaperSize = xlPaperA4
    Next
End Sub
15.29 File Path In Footer

In the before print command for the workbook, you can update it for the sheet being printed:

Private Sub Workbook_BeforePrint(Cancel As Boolean)
    ActiveSheet.PageSetup.LeftFooter = ActiveWorkbook.FullName
End Sub

15.30 Hiding the Windows Print Dialog

The following code by Stratos Malasiotis will hide the Windows print dialog when you do a printout. If you try to play with it too much you may get into problems. Use Alt+Ctrl+Del and then Cancel. The fncScreenUpdating sets the repainting window flag to false and therefore no WM_PAINT reaches the winproc.

Put the following code in its own module:

Option Explicit

Private Declare Function SendMessage _
    Lib "user32" _
    Alias "SendMessageA" _
    ByVal hwnd As Long, _
    ByVal wMsg As Long, _
    ByVal wParam As Long, _
    lParam As Any _
    As Long

Private Declare Function IsWindow _
    Lib "user32" _
    ByVal hwnd As Long _
    As Long

Private Declare Function InvalidateRect _
    Lib "user32" _
    ByVal hwnd As Long, _
    lpRect As Long, _
    bErase As Long _
    As Long

Private Declare Function UpdateWindow _
    Lib "user32" _
    ByVal hwnd As Long _
    As Long

Private Declare Function GetDesktopWindow _
    Lib "user32" () _
    As Long
Public Function fncScreenUpdating _
    ( _
        State As Boolean, _
        Optional Window_hWnd As Long = 0 _
    )

    Const WM_SETREDRAW = &HB
    Const WM_PAINT = &HF

    If Window_hWnd = 0 Then
        Window_hWnd = GetDesktopWindow()
    Else
        If IsWindow(hWnd:=Window_hWnd) = False Then
            Exit Function
        End If
    End If

    If State = True Then
        Call SendMessage _
            ( _
                hwnd:=Window_hWnd, _
                wMsg:=WM_SETREDRAW, _
                wParam:=1, _
                lParam:=0 _
            )
        Call InvalidateRect _
            ( _
                hwnd:=Window_hWnd, _
                lpRect:=0, _
                bErase:=True _
            )
        Call UpdateWindow(hwnd:=Window_hWnd)
    Else
        Call SendMessage _
            ( _
                hwnd:=Window_hWnd, _
                wMsg:=WM_SETREDRAW, _
                wParam:=0, _
                lParam:=0 _
            )
    End If
End Function

Sub PrintDirect()
    fncScreenUpdating State:=False
    ActiveSheet.PrintOut
    fncScreenUpdating State:=True
End Sub
16. DIRECTORIES

16.1 Displaying The Windows 95 Folder Dialog To Select A Directory

The following function, sDirectory(), will display the Windows folder selection dialog. This is useful if one needs the user to specify an output directory that does not contain any files. The display starts with C:A. To have a Windows folder dialog start in a specific folder, use the code in the next example.

'Place this type declaration and the next two functions at 'the top of the module

Public Type BROWSEINFO
    hWndOwner As Long
    pidlRoot As Long
    sDisplayName As String
    sTitle As String
    ulFlags As Long
    lpfn As Long
    lParam As Long
    iImage As Long
End Type

Declare Function SHGetPathFromIDList Lib "shell32.dll" _
    Alias "SHGetPathFromIDListA" _
    (ByVal pidl As Long, ByVal pszPath As String) As Long

Declare Function SHBrowseForFolder Lib "shell32.dll" _
    Alias "SHBrowseForFolderA" _
    (lpBrowseInfo As BROWSEINFO) As Long

Sub DemoGetPath()
    Dim anyPath As String
    'to use DirectoryName, just provide a msg for the dialog
    'to display
    anyPath = DirectoryName("Select destination folder")
    If anyPath = "" Then
        MsgBox "No directory selected"
    Else
        MsgBox anyPath
    End If
End Sub

Function DirectoryName(browseHeading As String) As String
    Dim browserInfo As BROWSEINFO
    Dim r As Long
    Dim iList As Long
    Dim wPos As Integer

    browserInfo.pidlRoot = 0&
'Title in the dialog

browserInfo.sTitle = browseHeading

browserInfo.ulFlags = &H1

'Display the dialog

iList = SHBrowseForFolder(browserInfo)

'Parse the result
DirectoryName = Space$(512)
r = SHGetPathFromIDList( ByVal iList , ByVal DirectoryName)
If r Then
wPos = InStr(DirectoryName, Chr$(0))
DirectoryName = Left(DirectoryName, wPos - 1)
Else
DirectoryName = ""
End If
End Function

16.2 Specifying the Windows Dialog Starting Directory

The following code will display the Windows directory dialog and lets you specify the starting directory. The user can not go above the starting directory.

Sub GetaDirectory()
    Dim strMessage As String
    Dim startDirectory
    strMessage = "Select a directory"
    startDirectory = "c:\program files"
    Dim objFF As Object
    Set objFF = _
        CreateObject("Shell.Application").BrowseForFolder(_
            0, strMessage, &H1, startDirectory)
    If Not objFF Is Nothing Then
        GetDirectory = objFF.items.Item.Path
        MsgBox GetDirectory
    Else
        GetDirectory = vbNullString
        MsgBox "No directory selected"
    End If
    Set objFF = Nothing
End Sub

16.3 Specifying A Starting Directory

To specify the starting directory and the file open dialog at the same time, use a statement like the following:
Dim bResponse As Boolean
bResponse = Application.Dialogs(xlDialogOpen).Show("c:\my documents")
If Not bResponse Then Exit Sub
'else code to process open file

16.4 Getting A Directory Using The File Open Dialog

If the directory the user needs to specify has a file in it, then you can use the following code to get the directory.

Sub DirNameExample()
    Dim dirName As String
    dirName = Directory_Name
    MsgBox dirName
End Sub

Function Directory_Name() As String
    Dim fName, I As Integer

    'display dialog asking user to select a file
    fName = Application.GetOpenFilename("Files (*.xls),*.xls", , "To Specify The Directory, Select A File")

    'check to see if cancel selected in the box
    If fName = "False" Then
        MsgBox "No selection made. Activity halted."
        Exit Function
    End If

    'extract just the directory name and store in the module level variable dirName
    For I = Len(fName) To 1 Step -1
        If Mid(fName, I, 1) = "\" Then
            Directory_Name = Left(fName, I)
            Exit Function
        End If
    Next
End Function

16.5 How To Have The User Select A Directory

The following code illustrates how to display the built in Excel file selection dialog and then extract the directory name from the user selection.

'declare this at the top of the module
Dim dirName As String

Sub Main_Routine()
    Select_A_Directory
    MsgBox dirName
End Sub

Sub Select_A_Directory()
    Dim sFile, I As Integer

    'display dialog asking user to select a file
    sFile = Application.GetOpenFilename _
            ("Files (*.xls),*.xls", , "Select A File")

    'check to see if cancel selected in the box
    If sFile = "False" Then
        MsgBox "No file selected. Activity halted."
        End
    End If

    'extract just the directory name and store in the module level variable dirName
    For I = Len(sFile) To 1 Step -1
        If Mid(sFile, I, 1) = "\" Then
            dirName = Left(sFile, I)
            Exit Sub
        End If
    Next
End Sub

16.6 Setting The Directory For UnMapped Network Drives

If the path to a file is \anyserver\directory\anotherDir\ then the ChDir function will not change the directory to this path. You need to instead use code like the following:

Private Declare Function SetCurrentDirectoryA Lib "Kernel32" _
            (ByVal sCurDir As String) As Long

Function bSetDir(anyDir As String) As Boolean
    If SetCurrentDirectoryA(anyDir) = 0 Then
        bSetDir = True
    End If
End Function

16.7 Getting A List Of Subdirectories
The following illustrates how to get a list of the subdirectories of a directory. It also illustrates using a main routine to call subroutines and passing variables to subroutines and having the subroutines change or use the variables/

Sub MainProgram()
    Dim dirList() As String

    'call the first subroutine and pass the main directory name and a string
    'array variable. The subroutine will populate the array with the
    'directories. (The array names do not have to be the same)
    ListDirectories "c:\analyzer\", dirList()

    'pass the populated array to a subroutine that will print the list on the
    'active worksheet
    PrintList dirList()
End Sub

Sub ListDirectories(anypath As String, dirList() As String)

    'this subroutine receives as its arguments a directory string
    'ending in a \, and a string array. It then populates the array
    'with the subdirectories
    Dim dirOutput As String, i As Integer

    'get the first subdirectory in the passed directory
    dirOutput = Dir(anypath, vbDirectory)

    'loop until the Dir functions returns ""
    Do While dirOutput <> ""

        'Ignore the current directory and the encompassing directory
        If dirOutput <> "." And dirOutput <> ".." Then

            'make sure dirOutput is a directory.
            If (GetAttr(anypath & dirOutput) _
            And vbDirectory) = vbDirectory Then
                i = i + 1

            'expand the array size, preserving the existing entries
            ReDim Preserve dirList(1 To i)

        dirOutput = Dir()
    End If

End Sub
'add the directory to the array

    dirList(i) = anypath & dirOutput
End If
End If

'get the next output from the Dir function. it may be a file or
'a directory so the above code checks for what it is

dirOutput = Dir()
Loop
End Sub

Sub PrintList(anyList() As String)
    Dim i As Integer, J As Integer

    'loop through the array. Since its size is not passed, get its lower
    'and upper limits for the For..Next loop. Print to the active sheet
    For i = LBound(anyList) To UBound(anyList)

        'set J equal to 1 if Lbound is zero. This is done so that the
        'next statement does not use zero as a row number
        If LBound(anyList) = 0 then J = 1
        Cells(i + J, 1).Value = anyList(i)
    Next
End Sub

16.8 Listing Sub Directories In A Directory

The following routine lists subdirectories in a directory.

Sub List_Sub_Directories_In_A_Directory()
    Dim anyPath As String
    Dim dirList() As String
    Dim dirOutput As String
    Dim I As Integer, J As Integer

    'the following variable is set to the path to be searched
    anyPath = "C:\Central\"

    'query the directory for any sub directories
    dirOutput = Dir(anyPath, vbDirectory)
    While dirOutput <> ""

        'check to see if the string is a directory
If (GetAttr(anyPath & dirOutput) _
    And vbDirectory) = vbDirectory Then
    If dirOutput <> "." And dirOutput <> ".." Then

    'if a directory add the string to the array

        I = I + 1
        ReDim Preserve dirList(1 To I)
        dirList(I) = dirOutput
    End If
End If

'query the directory and loop back if a string returned

    dirOutput = Dir()
    Wend

'print the directory listing

    For J = 1 To I
        Cells(J, 1).Value = anyPath & dirList(J)
    Next
End Sub

16.9 Determining If A Directory Exists

The following is a function that returns True if a directory exists, and False if it does not. The function is:

Function DirExists(sSDirectory As String) As Boolean
    If Dir(sSDirectory, vbDirectory) <> "" Then DirExists = True
End Function

The following illustrates it use:

If DirExists("C:\Program Files\Common Files") Then

    'actions to take if the directory exists

Else

    'actions to take if it does not exist

End If

You could also do the code this way:

If Not DirExists("C:\Program Files\Common Files") Then

    'actions to take if the directory does not exists
End If

16.10 Listing Files In A Directory And/Or Its Subdirectories

The routine Create_File_List lists all the files of any filetype in a directory and its subdirectories. You have the option of just searching the directory specified or all the subdirectories. You also have the option of returning the full path as part of the array fileList. The array fileAndPathList stores the path and file names separate. You can sort the resulting arrays using the QuickSort routine found in this example book.

'Place at the top of the module

Public fileList() As String
Public fileAndPathList()
Public fileCount As Long

'fileList is a one dimensional array of the files
'fileAndPathList list stores the path in the first dimension
'and the filename in the second dimension
'fileCount is the number of matching files

'note: array lists are un-sorted

Sub DemoDirListing()
    Dim I As Long

    'reset fileList, fileAndPathList, and count
    Erase fileList
    Erase fileAndPathList
    fileCount = 0

    'call search routine and pass arguments
    Create_File_List "C:\My Documents", "xls", True, True

    'display results
    If fileCount = 0 Then
        MsgBox "No files found"
    Else
        If MsgBox(fileCount & " files found. " & _
           "Select OK to list in a new worksheet", _
           vbOKCancel) = vbOK Then
            Worksheets.Add
            For I = 1 To UBound(fileList)
                Cells(I, 1).Value = fileList(I)
                Cells(I, 2).Value = fileAndPathList(1, I)
            Next I
    End If
End Sub
Cells(I, 3).Value = fileAndPathList(2, I)
Next

' set for viewing

Columns("A:C") . EntireColumn . AutoFit
Range("A1:C1") . Select
ActiveWindow . Zoom = True
Application . Goto Range("a1"), True
End If
End If
End Sub

Sub Create_File_List(directoryPath As String, _
    sFileType As String, _
    bDoSubs As Boolean, _
    bPrefixFileNameWithPath As Boolean)
Dim subDirectories() As String
Dim subDirCount As Long
Dim searchString As String
Dim fName
Dim I As Long

subDirCount = 0

'make certain the directory path ends in a \

If Right(directoryPath, 1) <> "\" Then _
    directoryPath = directoryPath & "\"

searchString = directoryPath & Dir(directoryPath & "*.*", _
    vbDirectory)
Do While searchString <> directoryPath
    If Not (Right(searchString, 2) = "." Or _
        Right(searchString, 3) = ".\") Then
        If GetAttr(searchString) = vbDirectory Then

        ' do this if a subdirectory

        If bDoSubs Then

        ' add to array of directories

        subDirCount = subDirCount + 1
        ReDim Preserve subDirectories(1 To subDirCount)
        subDirectories(subDirCount) = searchString
        End If
    Else

    ' do this if a file of the correct filetype

    If UCase(Right(fName, Len(sFileType))) = _
        UCase(sFileType) _
        Or sFileType = "*" Then
fileCount = fileCount + 1
ReDim Preserve fileList(1 To fileCount)
ReDim Preserve fileAndPathList(1 To 2, _
1 To fileCount)
fileAndPathList(1, fileCount) = directoryPath
fileAndPathList(2, fileCount) = fName

'include path in fileList if option to do so set

    If bPrefixFileNameWithPath Then
        fileList(fileCount) = searchString
    Else
        fileList(fileCount) = fName
    End If
End If
End If
End If

fName = Dir()
searchString = directoryPath & fName
Loop

'call recursively to process subdirectories

    If subDirCount > 0 And bDoSubs Then
        For I = 1 To subDirCount
            Create_File_List subDirectories(I), _
            sFileType, _
            bDoSubs, _
            bPrefixFileNameWithPath
        Next
    End If
End Sub

16.11 Counting The Number Of Files In A Directory

The following shows how to count the number of XLS files in a directory. It uses a function to do this. The advantage of this is that the function can be called from many places in your code versus having to repeat the code each time it is needed.

Sub Count_Files_In_A_Directory()
    Dim I

    'get the number of files by calling a user function

    I = CountFiles("C:\MyFiles")

    'display the results in a message box

    MsgBox "There are " & I & _
    " xls files in the directory you specified"
End Sub
Function CountFiles(tgtDir As String) As Integer
Dim fName As String

'Retrieve the first entry, handle error if directory not found
On Error GoTo badDirectory
fName = Dir(tgtDir & "\*.xls")
On Error GoTo 0

'loop through all files in the directory and increment the function's value
Do While fName <> ""

' Ignore the current directory and
' the encompassing directory.
    If fName <> "." And fName <> ".." Then
        CountFiles = CountFiles + 1
    End If

' Get next entry.
    fName = Dir()
Loop
Exit Function
badDirectory:

'come here if directory can not be accessed
MsgBox "The directory you specified does not exist or " & _
    "can not be accessed. Activity halted."
End
End Function

The following is another function that counts the number of files in a directory. The subroutine below runs this function and counts the number of files in the C:\Temp directory.

Sub Test1()
    MsgBox CountFiles("C:\TEMP\") & " files"
End Sub

Function CountFiles(tgtDir As String)
Dim fName As String
Dim cnt As Integer

' Retrieve the first entry.
    fName = Dir(tgtDir)
cnt = 0

' Start the loop.
Do While fName <> ""

' Ignore the current directory and
' the encompassing directory.

If fName <> "." And fName <> ".." Then
cnt = cnt + 1
End If

' Get next entry

fName = Dir()
Loop

'set the function equal to the number of files counted

CountFiles = cnt
End Function

16.12 How To Obtain The User's Temp Directory

The following returns the user's Windows Temp directory

Sub TempDir()
    MsgBox Environ("Temp")
End Sub

16.13 Getting The Windows Directory

One way to get the Windows directory is to use the Environ function:

MsgBox Environ("Windir")

Another way is to use the following function (author unknown), will return the Windows directory:

'32-bit API declaration

Declare Function GetWindowsDirectoryA Lib "KERNEL32" _
    (ByVal lpBuffer As String, ByVal nsize As Integer) _
    As Integer

'16-bit API declaration

Declare Function GetWindowsDirectory Lib "KERNEL" _
    (ByVal lpBuffer As String, ByVal nsize As Integer) _
    As Integer

Function WindowsDir()
'Returns the Windows directories

Dim WinDir As String * 255

'Determine if Excel is 16-bit or 32-bit

Select Case Left(Application.Version, 1)
  Case "5"
    '16-bit
    WLen = GetWindowsDirectory(WinDir, Len(WinDir))
  Case Else
    '32-bit
    WLen = GetWindowsDirectoryA(WinDir, Len(WinDir))
End Select
WindowsDir = Left(WinDir, WLen)
End Function

16.14 Getting File Information From A Directory

The following example shows how to return not only all the files and sub directories in a directory, but also the file size, last modified date, and file and directory attributes.

Sub DirectoryToSheet()
  Dim sh As Worksheet, lstAttr As Integer
  Dim mypath As String, myName As String
  Dim rw As Integer, fattr, strAttr As String

  'Add a new workbook for the information
  Workbooks.Add
  'set a variable to refer to the active sheet in this workbook
  Set sh = ActiveSheet

  'set key values
  lstAttr = vbNormal + vbReadOnly + vbHidden
  lstAttr = lstAttr + vbSystem + vbDirectory
  lstAttr = lstAttr + vbArchive

  ' Set the directory to be analyzed
  mypath = "c:\"

  ' Retrieve the first entry.
myName = Dir(mypath, lstAttr)

' put labels on the sheet at the top of the columns

sh.Cells(1, 1) = "Path:"
sh.Cells(1, 2) = mypath
sh.Cells(2, 2) = "Name"
sh.Cells(2, 3) = "Date"
sh.Cells(2, 4) = "Time"
sh.Cells(2, 5) = "Size"
sh.Cells(2, 6) = "Attr"

'set the output row to a variable, which is index below as entries are made

rw = 3

' Start the loop.

Do While myName <> ""

' Ignore the current directory and
' the encompassing directory.

If myName <> "." And myName <> ".." Then

'write file name to output file

sh.Cells(rw, 2) = myName

'write date to the output file

sh.Cells(rw, 3) = Int(FileDateTime(mypath & myName))

'write time to the output file

sh.Cells(rw, 4) = _
FileDateTime(mypath & myName) - _
Int(FileDateTime(mypath & myName))

'write file size to the output file

sh.Cells(rw, 5) = FileLen(mypath & myName)

'get the attributes of the file or directory

fattr = GetAttr(mypath & myName)
strAttr = ""
If fattr <> vbNormal Then
    If (fattr And vbReadOnly) Then
        strAttr = strAttr & "R"
    End If
End If
End If
If (fattr And vbHidden) Then
    strAttr = strAttr & "H"
End If
If (fattr And vbSystem) Then
    strAttr = strAttr & "S"
End If
If (fattr And vbDirectory) Then
    strAttr = strAttr & "D"
End If
If (fattr And vbArchive) Then
    strAttr = strAttr & "A"
End If
End If

'write the file or directory attribute to the output file
sh.Cells(rw, 6) = strAttr

'index the output row variable to the next blank row
rw = rw + 1
End If

'Get next entry.
myName = Dir()
Loop

'format columns that need formatting
sh.Columns("D:D").NumberFormat = "h:mm AM/PM"
sh.Columns("E:E").NumberFormat = "#,##0"
sh.Columns("B:B").EntireColumn.AutoFit
sh.Columns("F:F").HorizontalAlignment = xlRight
sh.Range("c2:e2").HorizontalAlignment = xlRight
End Sub

16.15 Creating A New Directory

The MkDir command allows you to create a new directory. For example:

    MkDir "C:\MyDirectory"

If the path is not specified, then the directory is created as a sub directory of the current directory.

16.16 Creating A Multi-Level New Directory

The subroutine MakeDirectory shown below by Rob Bovey will create a new subdirectory and any new directories needed in the specified path.
Sub MakeDirectoriesExamples()
MakeDirectory "c:\aaa\bbb\ccc\"
End Sub

Sub MakeDirectory(ByVal szDirectory As String)
Dim lPosition As Long
Dim szDir As String

' Ensure that the directory string to be processed has a trailing backslash.
If Right$(szDirectory, 1) <> "\" Then _
szDirectory = szDirectory & "\"

'If szDirectory doesn't exist, then create it.
If Len(Dir$(szDirectory, vbDirectory)) = 0 Then

'Each subdirectory in the string must be created
'one at a time from left to right.
lPosition = InStr(szDirectory, "\")

'Loop through each subdirectory level
Do While lPosition > 0

'Get the next subdirectory level.
    szDir = Left$(szDirectory, lPosition - 1)

'If the current level does not exist then create it.
    If Len(Dir$(szDir, vbDirectory)) = 0 Then MkDir szDir

' Increment the starting point for the next backslash search.
    lPosition = lPosition + 1

'Find the next slash position.
    lPosition = InStr(lPosition + 1, szDirectory, "\")
Loop
End If
End Sub

16.17 List Of Available Drives

The following routine, by Jim Rech, returns the available drives. It writes them out to the first worksheet in the active workbook.
'place at the top of the module

Declare Function GetDriveType Lib "KERNEL" _
(ByVal DriveNumber As Integer) As Integer

Declare Function GetDriveTypeA Lib "KERNEL32" _
(ByVal DriveNumber As String) As Integer

'this returns the drives, starting with drive C

Sub ListAvailDrives()
    Dim DrvCtr As Integer, Success As Integer
    Dim ListCtr As Integer
    Worksheets(1).Range("A1:A26").ClearContents

    'do this if Excel 7 or higher
    For DrvCtr = Asc("C") To Asc("Z")
        Success = GetDriveTypeA(Chr(DrvCtr) & ":")
        If Success <> 0 And Success <> 1 Then
            ListCtr = ListCtr + 1
            Worksheets(1).Cells(ListCtr, 1) = Chr(DrvCtr)
        End If
    Next
End Sub

16.18 Getting The Amount Of Free Disk Space On A Drive

You can use API calls to get the amount of free disk space on a drive. Place the following at the top of a module (either Excel 7 or 97):

Declare Function GetDiskFreeSpace Lib "kernel32" Alias _
"GetDiskFreeSpaceA" (ByVal lpRootPathName As String, _
lpSectorsPerCluster As Long, lpBytesPerSector As Long, _
lpNumberOfFreeClusters As Long, _
lpTotalNumberOfClusters As Long) As Long

Function GetFreeSpace(ByVal Drive$) As Double
    Dim SecPerCluster&, BytesPerSector&
    Dim NumFreeClusters&, NumClusters&
    Dim lRet&
    Dim dVal#

    lRet& = GetDiskFreeSpace(Drive$, SecPerCluster&, _
        BytesPerSector&, _
        NumFreeClusters&, NumClusters&)
    dVal# = SecPerCluster& * BytesPerSector&
dVal# = dVal# * NumFreeClusters&

GetFreeSpace = dVal#
End Function

The following illustrates how to use the above code to return the space on the C drive:

Sub DiskSpace()
    MsgBox GetFreeSpace("c:\") / 1048576 & " MB"
End Sub
17. PROGRESS MESSAGES

17.1 Creating A Splash Screen While Your Code Runs

If your code runs for a long time, you may want to create a splash screen that displays a message saying something like "Working...". To create a splash screen, you would need to do the following:

- add a new temporary workbook
- write a message to it
- turn screen updating off
- run your code
- delete the temporary workbook

Any time you wish to change the message on the screen, you would need to activate the temporary workbook, turn screen updating on, change the message, and turn screen updating back off.

The following illustrates the above procedures.

'declare this at the top of the module

Dim splashCell As Range

Sub CreateSplashWorkbook()
    Dim curBook As Workbook, splashBook As Workbook
    Dim splashSheet As Worksheet
    Dim originalSetting As Integer

    'store the active workbook so it can be reactivated
    Set curBook = ActiveWorkbook

    'create a one sheet workbook - be sure to close when done
    Application.ScreenUpdating = False
    originalSetting = Application.SheetsInNewWorkbook
    Application.SheetsInNewWorkbook = 1
    Set splashBook = Workbooks.Add
    Application.SheetsInNewWorkbook = originalSetting

    'set references to the sheet and cell

Set splashSheet = splashBook.Sheets(1)
Set splashCell = splashSheet.Cells(10, 3)

'format the cell, column, and sheet

With splashCell
  .Font.Bold = True
  .Font.FontStyle = "Bold"
  .Font.Size = 16
  .WrapText = True
End With
splashSheet.Columns("C:C").ColumnWidth = 43.86
ActiveWindow.DisplayGridlines = False
curBook.Activate
End Sub

Sub DisplayMsg(ByVal anyText As String, _
  ByVal nSeconds As Integer)
  Dim curBook As Workbook, waitTime
  Set curBook = ActiveWorkbook

  'activate the splash workbook and maximize the sheet
  On Error GoTo errorTrap
  ActiveWindow.WindowState = xlMaximized

  'insert the message and display it
  splashCell.Value = anyText
  Application.ScreenUpdating = True
  Application.ScreenUpdating = False

  'if a wait time is supplied, wait that number of seconds
  If nSeconds > 0 Then

    'calculate the time to wait until
    waitTime = Now() + TimeSerial(0, 0, nSeconds)
    Application.Wait waitTime

  End If
  curBook.Activate
  Exit Sub
errorTrap:
  MsgBox "the splash workbook has not been created"
End Sub

Sub SplashTest()

  'this illustrates using the above code
Dim I As Integer
CreateSplashWorkbook
For I = 1 To 5

'display a simple message and wait three seconds

    DisplayMsg "message " & I, 3
Next
DisplayMsg "Closing Splashworkbook", 2

'remove the temporary workbook

End Sub

Please note that there are alternatives to the above splash screen approach:

◆ Use the Application.StatusBar (examples are in this help file)

◆ Display a userform and write messages to it. This technique is also in this help file.

17.2 Displaying A Status Bar Message

At the very bottom the Excel screen is the status bar. When you save a workbook, you will see a progress bar in this area, showing that the worksheet is being saved. You can display a message in the status bar using code like the following:

    Application.StatusBar = "Working...."

'your code

    Application.StatusBar = "Still Working...."

'more code

    Application.StatusBar = False

'the above removes the text you wrote to the status bar,
'tallowing Excel's status messages to appear

Because some users have turned off the status bar, you can redisplay it via code, and then re-set it to the original setting, as illustrated by the following:

    Dim oldStatusBar As Boolean

'store the status bar setting

    oldStatusBar = Application.DisplayStatusBar
The following is another illustration on how to display a message on the status bar:

```vba
Sub Status_Message_Example()
    Dim I As Integer

    'loop 10 times, displaying a different message each time
    For I = 1 To 10
        Application.StatusBar = "Processing loop " & I

    'wait one second then continue loop - demo purpose only
    Next

    'clear the status bar message
    Application.StatusBar = False
End Sub
```

17.3 Rather Cool Non Modal Progress Dialog

Andred Baker has created a non-modal progress screen that shows a progress bar, title and caption. Although the install file is 1.4mb the actual DLL is tiny and efficient. It's free and [available from his web site](http://www).
In Excel 2000 and XP, user forms can be modeless, so you can use

```vbnet
Sub Excell200ModelessExample()
    UserForm1.Show vbModeless
    For I = 1 To 5
        UserForm1.Label1.Caption = "status message " & I
        Application.Wait (Now() + TimeValue("00:00:01"))
    Next
    Unload UserForm1
End Sub
```

Please note that Excel 97 does not support this feature.

### 17.5 Resetting The Status Bar

If you use the status bar to display messages to your users you will need to reset it by using the following statement:

```vbnet
Application.StatusBar = False
```

If your code crashes before this statement is reached, then the status bar will contain the last message sent. It is a good idea to start your code with the above statement, and include it in any error handing routines that halt your macros.

### 17.6 Display Status Messages In A Modeless UserForm

To display processing messages in a userform while your code is running, do the following:

1) Create a userform with a single label box on it and no buttons. Change the font on the label box to a large font. Change the caption on the userform to "Status" (or some other caption that you prefer)

2) On the userform's code module, select userform in the left drop down and Activate in the right dropdown. It will create the following two lines of code in the module:

```vbnet
Private Sub UserForm_Activate()
End Sub
```

3) In the above procedure, put the name of your main procedure. For example:

```vbnet
Private Sub UserForm_Activate()
    'runs your main procedure. It can be any name you want
    Main_Procedure
    'unloads the form with the above procedure is done
End Sub
```
Unload Me
End Sub

4) Assuming that the name of the userform is UserForm1, and that the name of the label is Label1, use statements like the following to display messages in the userform:

**change the message in the user form**

UserForm1.Label1.Caption = "any message you want"

' *repaint the form so that the message is displayed*

UserForm1.Repaint

5) To run your main code, create a procedure in a regular module that shows the user form:

6) Make certain that your error handling routines unload the userform if they stop execution.

To illustrate the above, put the following code in a regular module. The first procedure, called "Start_Up" is the one that you run. The second procedure, called "Main_Procedure" is the primary procedure.

```vba
Sub Start_Up()
    UserForm1.Show
End Sub

Sub Main_Procedure()
    Dim I As Integer

    ' *write a message to the label*
    UserForm1.Label1.Caption = "Step one being done......."

    ' *repaint the form so that the above message is displayed*
    UserForm1.Repaint

    ' *your code would go here; we've used Application.StatusBar to demonstrate that the code is executing*
    For I = 1 To 2000
        Application.StatusBar = I
    Next

    ' *display a second status message in the dialog*
    UserForm1.Label1.Caption = "Step two being done......."
    UserForm1.Repaint
    For I = 1 To 2000
```
Application.StatusBar = I
Next

'display a third status message in the dialog

UserForm1.Label1.Caption = "Final step being done......"
UserForm1.Repaint
For I = 1 To 2000
   Application.StatusBar = I
Next

'clear the status bar

Application.StatusBar = False
End Sub

Please note that it is very difficult to debug code while using the above code. The best approach is to only implement when you are done debugging.

The following is another way to use this approach:

Sub Main_Procedure()
   StatusForm.Show

   'If you need to step through your code, then put the statement "Unload StatusForm" at the point in your code
   'where you wish to start stepping through it. Also put a breakpoint or Stop statement at that location.
End Sub

Sub Continuation_Procedure()

'called by activate procedure in StatusForm

Show_Message "Processing step 1"

'your code here

'more code here

Show_Message "Processing step 3"

'more code here

End Sub

Sub Show_Message(ByVal anyText)
   StatusForm.Label1.Caption = anyText
17.7 Modeless Dialogs - Web Examples

There are a number of Microsoft articles on processing and status messages:

http://support.microsoft.com/support/kb/articles/q162/2/57.asp
OFF97: How to Show a "Now Processing" Dialog While Macro Runs

http://support.microsoft.com/support/kb/articles/q136/2/22.asp
Excel: How to Use a Custom Dialog Box as a Startup Screen (xl95/5.0)

XL97: How to Create a Startup Screen with a UserForm

http://support.microsoft.com/support/kb/articles/Q148/2/09.asp
XL: How to Create a Temporary Message Box While Macro Runs

Excel 95/5.0

http://support.microsoft.com/support/kb/articles/q162/2/57.asp
OFF97: How to Show a "Now Processing" Dialog While Macro Runs

17.8 Displaying A MsgBox for X Seconds

If you have Windows 98 or higher, then you can use the following to display a message box for a few seconds:

Sub StatusMsgBox()
    CreateObject("WScript.Shell").Popup _
    "Macro Started...Please", 2, "ATTENTION"
End Sub

The above uses the Windows Scripting Host files. To determine if the above will work, confirm that the file WSHOM.OCX is in the Windows Systems directory. If you do not have them, search Microsoft’s web site for download links.
18. FUNCTIONS

18.1 A Function That Uses Multiple Ranges As Input

The following function allows the use to select multiple ranges for its input. This function calculates the total error value of a range of numbers.

```
Function ErrorSum(ParamArray anyRange() As Variant) As Double
    Application.Volatile
    Dim cell As Variant
    Dim I As Integer
    Dim y As Double, x As Double
    y = 0

    'rotate through each range selected
    For I = LBound(anyRange) To UBound(anyRange)
        'rotate through all cells in the range
        For Each cell In anyRange(I)
            y = y + cell.Value ^ 2
        Next cell
        Next I

    'set the function equal to the above calculation
    ErrorSum = y ^ 0.5
End Function
```

Application.Volatile is used to insure that the function will recalculate if any cell on the worksheet changes. This does slow Excel down. If the data does not change, then you can exclude this statement. Or, you can exclude it, and just press ALT-CTRL-F9 to force the function to recalculate.

18.2 An Example Function

The following is a function that calculates the total error value of a range of numbers, given that the user selects a range of cells:

```
Function ErrorSum(anyRange As Range) As Double
    Application.Volatile
    Dim cell As Range
    Dim y As Double, x As Double
    y = 0

    'rotate through all cells in the range
    For Each cell In anyRange
        y = y + cell.Value ^ 2
    Next cell

    ErrorSum = y ^ 0.5
End Function
```
For Each cell In anyRange
y = y + cell.Value ^ 2
Next cell

'set the function equal to the above calculation

ErrorSum = y ^ 0.5
End Function

Application.Volatile is used to insure that the function will recalculate if any cell on the worksheet changes. This does slow Excel down. If the data does not change, then you can exclude this statement. Or, you can exclude it, and just press ALT-CTL-F9 to force the function to recalculate.

18.3 Determining Which Cell, Worksheet, And Workbook Is Calling A Function

The following statements

Dim cellAddress As String, shName As String, wbName As String
cellAddress = Application.Caller.Address
shName = Application.Caller.Parent.Name
wbName = Application.Caller.Parent.Parent.Name

Return the cell Address, the worksheet name, and the workbook name respectively and store in a variable for later use. The following statements set an object variable to the cell, the worksheet, and the workbook:

Dim oCell As Range, oSheet As Worksheet, oBook As Workbook
oCell = Application.Caller
oSheet = Application.Caller.Parent

18.4 Finding The Maximum Value In A Column

The following example finds the maximum value in column D, and then selects the first cell in that column with that value. This would be useful if you need to select the first cell in a range with the latest date. This example uses two worksheet functions, Match and Max to achieve its results.

Sub Find_Max()
Dim searchRange As Range
Dim cell As Range
Dim maxValue, maxRow As Integer

'define the search range, in this case it is column D

Set searchRange = Columns("d")
'get the maximum value in the column

maxValue = Application.Max(searchRange)

'get the row that the first maximum value is in. If your search range is
'not all cells, then you would add the row number of the start cell
'to the max row and subtract one to get the row number
'use Application.Match to do this. The third argument must be 0

maxRow = Application.Match(maxValue, searchRange, 0)

'set a range variable to refer to the first cell with the max value

Set cell = Cells(maxRow, 4)

'select the cell to prove the maximum cell value was found

cell.Select
End Sub

18.5 Forcing A Function To Recalculate When A Change Is Made

User defined functions will only recalculate when the cells that they directly refer to are changed. However, other cells on a sheet may change which changes the values in user defined functions' cells. For example, a range may be supplied to a function, but the function will only recalculate if the end cells are changed. Thus Excel will not detect this change and user defined functions will return their old, now incorrect values. To force a user defined function to recalculate, add the line "Application.Volatile = True" as the first line in your function. If it is not the first line of code, then it will not force the needed recalculation:

Function MyFunction()

    Application.Volatile = True

    'remainder of code

End Function

The drawback on this approach is that if you have used such a function many times in your worksheet, each cell's equation must be calculated each time you make a change. You could instead press ALT-CTRL-F9 and force a re-calculation as needed.

18.6 Getting The Maximum Value In A Range

The statement Application.Max(range) returns the maximum value in a range. DO NOT use Application.WorksheetFunction.Max(range). The WorksheetFunction qualifier can cause errors.
The same applies to MIN, AVERAGE, and MEDIAN.

The following sub will select the maximum value in the column of the active cell.

```vba
Sub SelectMax()

' set the range to check, and limit it to cells in the used range
Set rng1 = Intersect(ActiveSheet.UsedRange, _
Columns(ActiveCell.Column))

' use Match and Max to identify the largest value
Set Rng2 = rng1.Cells(Application._
Match(Application.Max(rng1), rng1, 0))

' select the cell that was found
Rng2.Select
End Sub
```

18.7 Tricks On Using Find

After using Find to search for a cell, you should use a test to see if it found what you were searching for. For example:

```vba
Dim Found As Range
Set Found = Cells.Find(What:="100", After:=ActiveCell, _
LookIn:=xlValues, LookAt:=xlWhole, _
SearchOrder:=xlByColumns, SearchDirection:=xlNext, _
MatchCase:=False)
If Found Is Nothing Then

' actions to take if not found

Else

' actions to take if found

End If
```

If you are searching for a date, then you need to convert the date to a value and supply it to the What argument of the Find function:

```vba
what:=DateValue("07/18/98")
```

You do not need to use the argument "After:=ActiveCell" with the Find command. Eliminating it allows you to use Find on a non-active sheet.
Set Found = Sheets(5).Cells.Find(What:="100", _
LookIn:=xlValues, LookAt:=xlWhole, _
SearchOrder:=xlByColumns, SearchDirection:=xlNext, _
MatchCase:=False)

You can also specify the After argument for a non-active sheet, if the starting position is important. The following searches down column B on a sheet, starting at cell B1.

With Sheets(5).Columns(2)
Set Found = .Cells.Find(What:="100", _
After:=.Cells(.Cells.Count) _
LookIn:=xlValues, LookAt:=xlWhole, _
SearchOrder:=xlByColumns, SearchDirection:=xlNext, _
MatchCase:=False)
End With

18.8 VLookUp Example

Here's an example to get you started with VlookUp

Sub Vlookup_Example()
Dim sTxt As String
Dim sFound As String
Dim sRange As Range
Dim vCol As Integer
Dim bPerfectMatch As Boolean
Dim eVal As Integer

' Set vlookup Values
sTxt = "mystring"
Set sRange = Sheets(1).Columns("a:b")
vCol = 2
bPerfectMatch = True
On Error Resume Next

'do lookup
sFound = Application.VLookup _
    (sTxt, sRange, vCol, Not bPerfectMatch)

'store error value and turn off error handling
eVal = Err
On Error GoTo 0

' check error value to see if match found
If eVal <> 0 Then
    MsgBox sTxt & " Not found"
Else
    MsgBox "vlookup of " & sTxt & " is " _
& sFound & " in column " & vCol
End If
End Sub

18.9 User Defined Functions - General Comment

If you know the C/C++ languages and have a compiler, see the Microsoft Excel Developer's Kit, published by MS-Press (Book + CD-ROM). It explains how to build XLL add-in functions, which you can better document in the function-wizard than VBA functions (and run ***MUCH FASTER***).

You can change the descriptive text that appears when the UDF appears for selection through the Function Wizard. To do that, use Alt-F8 from the Excel window, type the name of the function into the macro name line (it won't appear as a choice, and you may need to include the workbook name) and then choose Options. You can then enter the description into the text box labeled Description.

18.10 Using Worksheet Functions In Visual Basic Macros

You can use Excel worksheet functions for which there are no Visual Basic equivalents in your code by prefixing the name of the function with "Application". For example,

MsgBox Application.Round(ActiveCell.Value)

There are some worksheet functions that you can use without qualifying with Application, such as Sin() and Cos(). Such functions will not work if you qualify them with Application. The following are some of the functions you can use directly, without the Application qualifier

Abs Atn
Cos Exp
Fix Int
Log Rnd
Sgn Sin
Sqr Tan

One way to tell if you need to qualify the function with Application is to place the cursor on the function's name and press F1. If Visual Basic's help on the function appears then you do not qualify the function. If however the worksheet help appears then you do need to qualify the function.

The help in Excel says to use Application.WorksheetFunction instead of just Application. However, there have been a number of Excel news group reports of this causing problems. The
solution has been to not use **WorksheetFunction**, and just use **Application**. The only advantage of using **Application.WorksheetFunction** is that it automatically displays the arguments for the function.

Its possible to use many of Excel's functions in your code. The easiest way to get code is to do the following:

choose record a macro,

perform the function you need code for.

Stop recording

Now edit the macro and viola, there is the code you need.

### 18.11 Using The Worksheets Functions In Your Code

The following shows how to use the **Sum** function in a macro:

```vba
Dim dblResults As Double
dblResults = Application.Sum(Selection)
```

The following illustrates using the **MAX** function:

```vba
Dim X As Single
X = Application.Max(Worksheets("Sheet1").Range("A1:A100"))
```

The following illustrates using **AVERAGE**:

```vba
Set myRange = Worksheets(1).Range("A1:A10")
MsgBox Application.Average(myRange)
```

### 18.12 Using Match To Return A Row Or Column Number

**Application.Match** is an alternate to using **Application.Find** to search for a matching string. The following illustrates its use to search for the string "XYZ" in column A. **Match** is not case sensitive. If it does not find a match, then it can be set to return an error value.

In this example, an error will occur when this error is assigned to the variable R, which is allowed to have only whole number values. An alternate way to handle this is to make R a variant variable and then use the test IsError(R) to see if an error value is returned and handle that situation.

```vba
Sub Find_A_Match_In_A_Column()
    Dim R As Long

    'turn on in case a match is not found
```
On Error Resume Next

'search just a single column. Use the "0" argument in the match function
'to get an exact match

R = Application.Match("xyz", Columns(1), 0)

'turn off error handling

On Error GoTo 0

'check the results of the Match and display a message

If R = 0 Then
  MsgBox "No match found"
Else
  MsgBox "Match found on row " & R
End If
End Sub

If XYZ is somewhere in row 1, then you could do the following:

Sub Find_A_Match_In_A_Row()
  Dim C As Long
  On Error Resume Next
  C = Application.Match("xyz", Rows(1), 0)
  If C = 0 Then
    MsgBox "No match found"
  Else
    MsgBox "Match found in column " & C
  End If
End Sub

In the above examples, since you know the column and row being searched, you can refer to the
cell containing the match by using Cells(R, 1) or Cells(1, C) respectively.

18.13 User Defined Functions And The Function
Wizard

When you use the function wizard to fill in the arguments for a user defined function, the
function wizard is always recalculating the answer as you fill in an argument. Unfortunately,
most user defined functions will crash unless all arguments are supplied. Since the wizard runs
the functions before all the arguments are supplied, the wizard will crash unless you take
precautions to avoid a crash!

To get around this problem, put an error trap at the top of your function in your function:

Function Somefunction(ARG1, ARG2, ETC)
  On Error GoTo exitFunctionLabel
'function code
'normal exit

Exit Function
exitFunctionLabel:
    SomeFunction = "#N/A"
End Function

The On Error statement causes the function to jump to the label "exitFunctionLabel" at the end of the code when an error occurs. It then exits the function, and returns the default value of the function. For example, if the function is declares as a Boolean function, it would return False.

18.14 Using Find In Visual Basic Code

The Find function returns a range object. Once you have a range object you can copy it or assign its value to another cell, etc. Here is an example of copying and pasting the cell one to the right of the cell on Sheet1, column A that contains "ABC":

Dim R As Range
Set R = Worksheets("Sheet1").Columns(1).Find("ABC")

'confirm something was found, stop if not
If R Is Nothing Then
    MsgBox "ABC not found. Activity halted."
End If

'if ABC found copy the value next to it to sheet 2, cell A1
R.Offset(0, 1).Copy Worksheets("Sheet2").Range("A1")

Most people new to VBA think in terms of selecting a cell and then doing something to it. As these examples show that is not necessary (usually anyway) and in fact just slows things down.

18.15 Using Find In Your Macros

If you use the macro recorder to search the range A1:A10 for the string "ABC" on Sheet1, the following is what you would get:

Sub Macro1()
    Sheets("Sheet1").Select
    Range("A1:A10").Select
    Selection.Find(What:="abc", After:=ActiveCell, _
    LookIn:=xlFormulas, _
    LookAt:=xlPart, SearchOrder:=xlByRows, SearchDirection:= _
    xlNext, MatchCase:=False).Activate
End Sub
The following achieves the same result, but does not select the sheet, the range, nor activate the found cell. It also sets a range variable to the found cell and halts the code if the text is not found

Sub FindABC()
    Dim fCell As Range
    Dim R As Range

    'set a range variable equal to the range to be searched
    R = Worksheets("sheet1").Range("A1:A0")

    'search the range. Note that Find is qualified with the range, and 'that the After argument is specified as the last cell in the range, 'not with ActiveCell. This makes the search start with the first cell.
    Set fCell = R.Find(What:="ABC", _
        After:=R.Cells(R.Cells.Count), _
        LookIn:=xlValues, LookAt:=xlPart, _
        SearchOrder:=xlByRows, SearchDirection:=xlNext, _
        MatchCase:=False)

    'check to see if a match is found
    If fCell Is Nothing Then
        MsgBox "What You looked for couldn't be found"
        Exit Sub
    End If
End Sub

The key in modifying the macro code is to qualify Find with the search range, and to specify for the After argument a cell in this range.

If you wanted the search to start searching from the first cell in the search range, then the After argument must be specified as the LAST cell in the search range:

Set fCell = R.Find(What:="ABC",
    After:=R.Cells(R.Cells.Count), _
    LookIn:=xlValues, LookAt:=xlPart, _
    SearchOrder:=xlByRows, SearchDirection:=xlNext, _
    MatchCase:=False)

Please note that the above assumes that the range is one area, and not multiple areas. If there are multiple areas, then use the following:

Dim lastArea As Range
Set lastArea = R.Areas(R.Areas.Count)
Set fCell = R.Find(What:="ABC", _
    After:=lastArea.Cells(lastArea.Cells.Count), _
    LookIn:=xlValues, LookAt:=xlPart, _
18.16 Using The Find Command To Find A Particular Cell

When you record a macro that uses the Find command, the following is typical of what you get:

```vba
```

In the above recording, the find is for an exact match to the letters "ABC" and the recording selects the cell. The following is a simple function that does a find and returns the cell reference for use in your code. In this function, the search range is specified by the range variable `searchRange`, and the `After` argument is specified as the last cell in this range. This makes Excel start the search from the first cell of the range. Also, `Application.DisplayAlerts` has been set to `False` so that an alert box is not displayed if a match is not found. It is set back to true as this is the default setting, and it does not rest automatically.

```vba
Function FindCell(searchFor As String, searchRange As Range) As Range
    Application.DisplayAlerts = False
    With searchRange
    End With
    Application.DisplayAlerts = True
End Function
```

The following illustrates how to use this function to search for the words "ABC" on sheet 2 of the active workbook.

```vba
Sub FindExample1()
    Dim cell As Range
    Set cell = FindCell("ABC", Sheets(2).Cells)
    If cell Is Nothing Then
        'action to take of no match
    Else
        'action to take if a match
    End If
End Sub
```
If you wished to restrict the search to a specific range, such as a column and also specify the workbook and sheet by name, you could do the following:

Sub FindExample2()
    Dim cell As Range, someSheet As Worksheet
    Set someSheet = Workbooks("book1").Sheets("sheet1")
    Set cell = FindCell("ABC", Sheets(2).Cells)
    If cell Is Nothing Then

        'action to take of no match

    Else

        'action to take if a match

    End If
End Sub

18.17 Using VLookUp In Your Code

The following is an example of using VLookUp, an Excel function, in your Visual Basic code. In this example, the user has one worksheet with descriptions in column A and values in column B. The user has a second worksheet with just a list of descriptions. The user wants the code to read the descriptions in the second worksheet, look for a matching description in the first workbook, and if a match is found copy the value in column B to the right of the description in the second workbook.

Please note that if you have the last argument set to False, as in this example, VLookUp will return an error value if it cannot find an exact match. Thus you have to check for the error value to see if it was returned. And use a variant variable to capture the output of VLookUp.

Sub Vlookup_Example()
    Dim lookupRange As Range
    Dim srceRange As Range
    Dim cell As Range
    Dim lookupValue As Variant

    'the lookup range is the range to be searched
    Set lookupRange = Worksheets("Sheet1").Range("A1:B100")

    'the srceRange is the range that contains values that are searched for
    'in the lookupRange
    Set srceRange = Worksheets("Sheet2").Range("A1:A10")

    For Each cell In srceRange
        With cell
'Note that the third argument of VLookUp is False.  
'This means an exact match is required or an error value is returned.

lookupValue = _  
  Application.VLookup(.Value, lookupRange, 2, False)

'if a match is found the above returns a value  
'if a mach is not found, it returns an error value

If Not IsError(lookupValue) Then  
  .Offset(0, 1).Value = lookupValue  
Else  
  .Offset(0, 1).Value = "No Match"  
End If
End With
Next cell
End Sub

You should use Application.VLookup, and not Application.WorksheetFunction.VLookUp.  
The second approach has been known not to work correctly.

The following illustrates using VLookUp to lookup a date value and get an exact match:

Sub VLookupTest()  
  Dim theDate As Long  
  Dim theRange As Range  
  Dim Value As Variant

  'set variable equal to a date value
  theDate = #2/1/97#

  'set a variable to the range to be searched with VLookUp
  Set theRange = Worksheets("Contracts").Range("G1:H30")

  'Note that the third argument of VLookUp is False.  This means an exact  
  'match is required or an error value is returned.
  Value = Application.VLookup(theDate, theRange, 2, False)

  'display the results
  If IsError(Value) Then  
    MsgBox "No match found"  
  Else  
    MsgBox Value  
  End If
End Sub

If the third argument of VLookUp is changed to True or omitted then an exact match is not necessary.  VLookUp would search until it found a matching date or a value greater than the date
18.18 Using Application.Caller To Determine What Called A Function

You can use Application.Caller in a function to determine information about what called a function:

Application.Caller.Address
Application.Caller.Parent.Name

Returns Cell Address, Worksheet Name, Workbook Name, Application Name respectively.

As information, there is one report that that if you use Application.Caller when the caller is a Shape object, the returned name is truncated to 31 characters.

18.19 Why Functions Can't Change Cells

Functions cannot change other cell values; they can only return a value. The reason for this is that Excel must track the precedents and dependents of a cell to properly calculate the worksheet. Since Excel cannot know what goes on in your functions, it prohibits them from changing other cells.
19. WINDOWS

19.1 Determining The Visible Range In A Window

The following will set a range variable equal to the visible cells in a window

```vba
Sub visibleRange()
    Dim visRange As Range
    Set visRange = ActiveWindow.VisibleRange
    MsgBox "the visible range is " & visRange.Address
End Sub
```

The first row number would be:

```vba
firstRow = visRange.Rows(1).Row
```

The last row number would be

```vba
lastRow = visRange.Rows(visRange.Rows.Count).Row
```

Please note that if just a tiny fraction of a row or a column is visible, then it will be included in the visible range.

How to set the top left visible cell

The simplest way to set the top left visible cell is to use Application.Goto:

```vba
Application.Goto Range("E5"), True
```

However, the above approach changes the active cell. The following achieves the same result (setting the top left cell to E5), but sets the active cell to what it was before the statement ran:

```vba
Dim cell As Range
'store the active cell in a variable
Set cell = ActiveCell
Application.Goto Range("E5"), True

'reactivate the cell

cell.Select
```

Another way to change the top cell is to set the scroll properties of the window:

```vba
ActiveWindow.ScrollRow = 5
ActiveWindow.ScrollColumn = 3
```

The above sets the top cell to cell C5.
19.2 How To Make A Range The Visible Range In A Window

To set what is visible on the screen, do this, use Application.GoTo and turn screen updating on and off: Also set the scroll property of the GoTo method to True.

This sets a range variable to the desired range

```vba
Set myRange = Workbooks("xlmisc10h.xls").Worksheets("NewName").Range("M200")
```

'this goes to the above range and makes it the top left range on the screen

```vba
Application.Goto myRange, True
Application.ScreenUpdating = True
```

'this makes certain only a single cell is selected

```vba
ActiveCell.Select
Application.ScreenUpdating = False
```

The above will automatically switch workbooks and windows.

If you want just the range visible on the screen, use the following statement,

```vba
ActiveWindow.Zoom = True
```

which will fit the screen as closely as possible to the range.

19.3 Automatically Displaying A Sheet In Full Screen Mode

You can have Excel automatically go into full screen mode when a particular sheet is activated, and return to normal mode when it is deactivated. To do this:

Go to the worksheet’s code module by right clicking on the sheet tab and selecting view code or by double-click the worksheet object in the Visual Basic project window

Select worksheet in the left dropdown box and Activate in the right dropdown box.

Place the following statement in the macro that appears

```vba
Application.DisplayFullScreen = True
```

In the right dropdown select deactivate and place the following statement in it

```vba
Application.DisplayFullScreen = False
```
19.4 Disabling Window Minimization

The following statement will prevent windows from being minimized:

ActiveWindow.EnableResize = False

19.5 Displaying The Full Screen Without The Full Screen Toolbar

The following will do the equivalent of selecting View, Full Screen and then hiding the full screen toolbar if it appears. Since Excel works out which toolbars it should display AFTER your routine has finished. You therefore need something like:

Sub Show_Full_Screen()
    Application.DisplayFullScreen = True
    Application.OnTime Now, ContinueIt
End Sub

Sub ContinueIt()
    Toolbars("Full Screen").Visible = False
End Sub

19.6 Determining The Window State

The following returns the window state – maximized, minimized or normal of the Excel application.

Sub ReturnWindowState()
    Dim szState As String
    Select Case Application.WindowState
        Case xlMaximized
            szState = "Maximized"
        Case xlMinimized
            szState = "Minimized"
        Case xlNormal
            szState = "Normal"
    End Select
    MsgBox "Window state is: " & szState
End Sub

19.7 Finding Out Which Cell Is In The Upper Left Corner

ActiveWindow.VisibleRange(1, 1).Address

will do it.
19.8 Getting a Window's Handle and Other Information

The following two articles will help you find a Window's handle and other attributes of the window:

http://support.microsoft.com/support/kb/articles/Q147/6/59.ASP

HOWTO: Get a Window Handle Without Specifying an Exact Title

http://support.microsoft.com/support/kb/articles/Q112/6/49.ASP

Howto: Get a Window's Class Name and Other Attributes

19.9 Getting The Monitor's Screen Resolution

The following code will return the monitor's settings:

'place these statements at the top of your code

Declare Function GetSystemMetrics Lib "user32" _
     (ByVal nIndex As Long) As Long

Const SM_CYSCREEN As Long = 1
Const SM_CXSCREEN As Long = 0

Sub GetScreenDimensions()
    Dim lWidth As Long
    Dim lHeight As Long

    lWidth = GetSystemMetrics(SM_CXSCREEN)
    lHeight = GetSystemMetrics(SM_CYSCREEN)

    MsgBox "Screen Width = " & lWidth & " " & _
            "Screen Height = " & lHeight
End Sub

19.10 Getting The Screen Resolution

To get the screen resolution, use the following functions:

'These three functions need to be declared at the top of the module
 'each function needs to be on a single line

Declare Function GetDeviceCaps Lib "gdi32" (ByVal hdc As Long, ByVal nIndex As Long) As Long

Declare Function GetDC Lib "user32" (ByVal hwnd As Long) As Long
Declare Function ReleaseDC Lib "user32" (ByVal hwnd As Long, ByVal hdc As Long) As Long

Const HORZRES = 8

Const VERTRES = 10

' this subroutine illustrates using the function

Sub GetScreenSize()
    MsgBox ScreenResolution()
End Sub

Function ScreenResolution()
    Dim lRval As Long
    Dim lDc As Long
    Dim lHSize As Long
    Dim lVSize As Long
    lDc = GetDC(0&)
    lHSize = GetDeviceCaps(lDc, HORZRES)
    lVSize = GetDeviceCaps(lDc, VERTRES)
    lRval = ReleaseDC(0, lDc)
    ScreenResolution = lHSize & "x" & lVSize
End Function

19.11 Hiding A Worksheet While A Dialog Or UserForm Is Displayed

If you do not want your user to see the active worksheet while he or she is filling out a dialog or userform, then you can minimize that worksheet's window by using the statement:

ActiveWindow.WindowState = xlMinimized

To restore the window, set WindowState to either xlMaximized or xlNormal.

If you have multiple workbooks open and want to minimize all windows, then you can use the following code instead:

Dim w As Window

' set on error to ignore errors in case there are hidden workbooks

On Error Resume Next
For Each w In Windows
    w.WindowState = xlMinimized
Next

' turn on error off
On Error GoTo 0

Please note that this changes the active workbook, and obviously the active sheet. To reactivate
the originally active workbook and sheet, use the following code:

Dim w As Window
Dim wb As Workbook
Dim ws As Object
Set wb = ActiveWorkbook
Set ws = ActiveSheet

'above code would go here

    wb.Activate
    ws.Select

19.12 Hiding And Showing Windows

It is fairly easy to hide a workbook if it has just one window and that window is the active
window:

    ActiveWindow.Visible = False

However, this requires that you activate the workbook, and assumes that there is just one window
for that workbook. The following is a much safer way hide a workbook, does not require that you
activate it, and it can handle multiple windows on the workbook.

Dim w As Window
For Each w In Workbooks("My book.XLS").Windows
    w.Visible = False
Next

To show the first window on a workbook, and close all other windows on a workbook, use code
like the following

Dim I As Integer

'loop backwards to the first window as windows will be closed

For I = Workbooks("My book.XLS").Windows.Count To 1 Step -1
    If I > 1 Then
        'if not the first window, close the window
        Workbooks(I).Windows(I).Close
    Else
        'if the first window, make it visible
        Workbooks(1).Windows(I).Visible = True
    End If
Next
Workbooks(1).Windows(I).Visible = True
End If
Next

19.13 How To Change The Excel Window Caption

to change the caption that says "Microsoft Excel at the top when Excel is open, use a statement like the following:

Application.Caption = "My Title"

19.14 How To Keep The Workbook Window Maximized

You can do this with the Workbook_WindowResize event procedure. Put the following code in the workbook code module.

Private Sub Workbook_WindowResize(ByVal Wn As Window)
    Wn.WindowState = xlMaximized
End Sub

19.15 How To Maximize The Window

The following illustrate two ways to maximize the workbook window:

Windows("book1.xls").WindowState = xlMaximized

or

ActiveWindow.WindowState = xlMaximized

19.16 Positioning The Excel Window

The following code positions the Excel window on the left half of the screen:

Sub PositionOnTheLeftHalf()
    Dim dWidth As Double, dHeight As Double
    With Application
        .ScreenUpdating = False
        '.get maximum size of the application window
        .WindowState = xlMaximized
        dWidth = .Width
        dHeight = .Height
        '.make Excel a window

        Workbooks(1).Visible = True
    End If
Next

Sub PositionOnTheLeftHalf()
The following code positions the Excel window in the middle of the screen:

```
SubInTheMiddle()
    Dim dWidth As Double, dHeight As Double
    With Application
        .ScreenUpdating = False
        .WindowState = xlMaximized
        dWidth = .Width
        dHeight = .Height
        .WindowState = xlNormal
        .Top = dHeight / 4
        .Left = dWidth / 4
        .Height = dHeight / 2
        .Width = dWidth / 2
    End With
End Sub
```

19.17 Setting All Worksheets To The Same Scroll Position

The following will set all worksheets in a workbook to the top scroll position:

```
Sub top()
    Dim wksht As Worksheet
```
Application.ScreenUpdating = False
For Each wksht In ActiveWorkbook.Worksheets
    wksht.Activate
    Range("A1").Select
    ActiveWindow.ScrollRow = 1
    ActiveWindow.ScrollColumn = 1
Next
End Sub

The following will allow you to control screen scrolling through a spinner button on a userform.
You will first need to set the Min property of the spin button to -1 and the Max property to 1.
Then use the following code in the userform's code module:

Private Sub SpinButton1_Change()
    Application.EnableEvents = False
    SpinButton1.Value = 0
    Application.EnableEvents = True
End Sub

If you want to use scrollbar controls instead of SpinButtons, this technique works very well and
allows horizontal and vertical scrolling while a UserForm is displayed.

1. Add a horizontal scrollbar control, name it ScrollBarColumns, set its Min to 1 and its Max to 256.

2. Add a vertical scrollbar control, name it ScrollBarRows, set its Min to 1 and its Max to 65536.

3. Add these two subs to the code module of the UserForm:

Private Sub ScrollBarColumns_Change()
    ActiveWindow.ScrollColumn = ScrollBarColumns.Value
End Sub

Private Sub ScrollBarRows_Change()
    ActiveWindow.ScrollRow = ScrollBarRows.Value
End Sub

19.18 Sizing A Worksheet To Fit The Screen

Because of different screen resolutions and differences in monitor settings, no two monitors are
just alike. If you need to fit part of a worksheet to fit exactly to the visible screen, then do the
following:

'select the range to be viewed:

Range("A1:H10").Select

'set the window's zoom property to fit this range. Please note that Excel
'll normally be able to fit the screen to match either the columns or the
'rows, which ever demands the smallest zoom setting.
ActiveWindow.Zoom = True

'select a single cell so that the entire range is not selected:

Range("A1").Select

Other Visual Basic statements that are also useful are:

'maximize the size of Excel in Windows

Application.WindowState = xlMaximized

'maximize the window

ActiveWindow.WindowState = xlMaximized

19.19 Synchronizing Windows On Different Sheets

The following code will set all windows to the same scroll setting, effectively synchronizing them:

Sub SynchEm()
    Dim sr As Long, sc As Long, I As Integer
    sr = Windows(1).ScrollRow
    sc = Windows(1).ScrollColumn
    For i = 2 To Windows.Count
        Windows(i).ScrollRow = sr
        Windows(i).ScrollColumn = sc
    Next i
    Application.ScreenUpdating = True
End Sub

You can run the above code from the workbook SheetSelectionChange event procedure so that synchronization occurs automatically as new cells are selected. If you have a prior version of Excel, you would need to run the macro from a toolbar button or Ctrl+key shortcut to achieve synchronization after a move in the active window.

19.20 Unhiding Hidden Workbooks

If you have a hidden workbook, the following is the way to unhide it if you are uncertain how many windows have been opened on the file:

Workbooks("Book1.xls").Windows(1).Visible = True

If you are unsure how many windows have been opened on a workbook, you can find out with a statement like the following:

WindowCount = ActiveWorkbook.Windows.Count
Or

WindowCount = Workbooks("Book1.xls").Windows.Count
20. FILTERING DATA

20.1 AutoFilter's Range

If you turn AutoFiltering on, you will find that Excel creates a hidden sheet specific name on that sheet by the name of "_FilterDataBase". If the name of the sheet is "Sheet1" then

Range("Sheet1!_FilterDatabase").Select

will select this range.

The following will return the number of rows in a filtered selection:

```vbscript
Dim rVisible As Range
Dim rArea As Range
Dim R As Long
With Range("Sheet1!_FilterDatabase")
    Set rVisible = Intersect(.Cells, .SpecialCells(xlVisible))
End With
For Each rArea In rVisible.Areas
    R = rArea.Rows.Count + R
Next
MsgBox R - 1
```

20.2 Determining Filter Settings

The following function takes a worksheet cell and returns the AutoFilter criteria for the column containing that cell. It returns an empty string if there is no filter applied to that column (i.e. you're showing "All") or if the range is not inside an AutoFilter range.

```vbscript
Function FilterCriteria(oRng As Range) As String
    Dim sFilter As String
    On Error GoTo NoMoreCriteria
    With oRng.Parent.AutoFilter
        'Is it in the AutoFilter range?
        If Intersect(oRng, .Range) Is Nothing Then _
            GoTo NoMoreCriteria
        'Get the filter object for the appropriate column
        With .Filters(oRng.Column - .Range.Column + 1)
            'Does this column have an AutoFilter criteria?
        End With
    End With
    FilterCriteria = sFilter
    Exit Function
NoMoreCriteria:
    FilterCriteria = sFilter
    Exit Function
End Function
```
If Not .On Then GoTo NoMoreCriteria

' It has one!

sFilter = .Criteria1

'Does it have another (i.e. the "Custom" filter)?

Select Case .Operator
   Case xlAnd
       sFilter = sFilter & " AND " & .Criteria2
   Case xlOr
       sFilter = sFilter & " OR " & .Criteria2
End Select
End With
End With

NoMoreCriteria:
   FilterCriteria = sFilter
End Function

20.3 How To Select The Data In A Filtered List

The following code will copy just the filtered cells from one sheet to another sheet.

Sub CopyVisible()
   Set rng1 = Worksheets("Sheet2").Range("A1").CurrentRegion
   Set rng1 = rng1.SpecialCells(xlVisible)
   rng1.Copy Destination:=Worksheets("Sheet3").Range("a1")
End Sub

20.4 How To Turn AutoFilter Off And On

The following shows how to turn AutoFilter off or on. These examples are procedures that are called by another procedures. This allows you to use these procedures over and over again and not have to duplicate the lines.

Sub MainProcedure()
   Dim tempR As Range
   Set tempR = ActiveSheet.UsedRange
   FilterOn tempR

   ' code that works with sheet in filter mode

   FilterOff tempR
End Sub

Sub FilterOff(rng1 As Range)

   ' check to see if AutoFilter is on
End Sub
If rng1.Parent.AutoFilterMode Then

' Filter is applied - next line turns it off

    rng1.AutoFilter
End If
End Sub

Sub FilterOn(rng1 As Range)

' check to see if AutoFilter is on

If Not rng1.Parent.AutoFilterMode Then

'turn on AutoFilter if it is off

    rng1.AutoFilter
End If
End Sub

20.5 Determining If AutoFilter Is Turned On

ActiveSheet.AutoFilterMode returns True if the filter is turned on.

20.6 Determining The AutoFilter's Settings

Stephen Bullen supplied this bit of code to the Excel news group. I think even Microsoft KB articles say the AutoFilter is not queriable as to the filter criteria, but Stephen proved that it is. This function returns a string with the criteria for the column of the cell passed as an argument. It returns an empty string if no criteria exists for that column.

Function FilterCriteria(oRng As Range) As String
Dim sFilter As String
On Error GoTo NoMoreCriteria
With oRng.Parent.AutoFilter

' Is it in the AutoFilter range?

    If Intersect(oRng, .Range) Is Nothing Then _
        GoTo NoMoreCriteria

' Get the filter object for the appropriate column

    With .Filters(oRng.Column - .Range.Column + 1)

' Does this column have an AutoFilter criteria?

        If Not .On Then GoTo NoMoreCriteria

End With
End With
End Function

NoMoreCriteria:
'It has one!

sFilter = .Criteria1

'Does it have another (i.e. the "Custom" filter)?

Select Case .Operator  
Case xlAnd
   sFilter = sFilter & " AND " & .Criteria2
Case xlOr
   sFilter = sFilter & " OR " & .Criteria2
End Select

End With
End With

NoMoreCriteria:
   FilterCriteria = sFilter
End Function

As far as applying filters, this is fairly straightforward. Turn on the macro recorder and apply the AutoFilter and criteria. Then turn off the macro recorder and generalize the code.

20.7 Working With Just The Filtered Cells On A Sheet

A data filter allows one to display only certain rows. However, if you select the cells in the filtered list and run a macro against those cells, the macro will act on all cells in the selection, filtered or not. The trick is to test to see if the cell's row is visible.

Sub Filtered_Cells_Only()
   Dim cell

   'rotate through all the cells in the selection
   For Each cell In Selection

      'check to see if the cell is hidden
      If Not cell.EntireRow.Hidden Then

         'Do your thing here

      End If
   Next cell
End Sub
21. PIVOT TABLES

21.1 Expanding Pivot Table Ranges

When you record a macro that creates a pivot table, Excel will record code like the following, with the range being a fixed range in R1C1 notation.

```vba
ActiveSheet.PivotTableWizard SourceType := xlDatabase, SourceData:= " 'Pivot' !RiC1:R47C6", _ TableDestination:="", TableName:="PivotTable2"
```

The problem is that when you use the above with other data that had more rows, it doesn't include the additional rows.

You can modify this code so that it expands automatically to get additional rows and columns of data using the `CurrentRegion` method.

```vba
Dim dataRange As Range
Set dataRange = Worksheets("Pivot").Range("A1").CurrentRegion

ActiveSheet.PivotTableWizard _
 SourceType :=xlDatabase, _
 SourceData :=dataRange, _
 TableDestination :="", _
 TableName :="PivotTable2"
```

21.2 Clearing Incorrect Field Names in PivotTable Field Dialog Box

Excel does a poor job of removing old field names. They can stay visible to visual basic code, even though you may not be able to see them manually. The following deletes these "ghost" names:

```vba
Dim pField
Dim pItem
On Error Resume Next
For Each pField In ActiveSheet.PivotTables(1).PivotFields
    For Each pItem In pField.PivotItems
        pItem.Delete
    Next
Next
ActiveSheet.PivotTables(1).RefreshTable
```

The following is another approach that cleans all pivot tables in all worksheets

```vba
Sub DeleteOldFieldsWB()
'gets rid of unused fields in a PivotTable
```

406
'had to go through the procedure twice
Dim ws As Worksheet
Dim pt As PivotTable
Dim pf As PivotField
Dim pi As PivotItem
Dim i As Integer

On Error Resume Next
'do twice to insure removal
For i = 1 To 2
  For Each ws In ActiveWorkbook.Worksheets
    For Each pt In ws.PivotTables
      For Each pf In pt.PivotFields
        For Each pi In pf.PivotItems
          pi.Delete
        Next
      Next
      pt.RefreshTable
    Next
  Next
End Sub

21.3 Pivot Table Events

Prior to Excel XP there are no events specific to pivot tables. Excel XP introduced the PivotTableUpdate event which fires only when a pivot table is changed. This event is available in the sheet code module. Access a sheet's code module by right clicking on the sheet tab and selecting view code. Once you add any event, you can then add the Pivot TableUpdate event routine. Or you can copy the following and paste into the sheet's code module:

Private Sub Worksheet_PivotTableUpdate( _
  ByVal Target As PivotTable)

End Sub
22. DATE AND TIME

22.1 Converting The Date To A Day's Name

The following function will return a day's name if you supply a date value:

```
Function WeekDayName(dtDate As Date) As String
    WeekDayName = Choose(WeekDay(dtDate), "Sunday", "Monday", 
                        "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday")
End Function
```

For example, the entry of "=WeekDayName(11/7/98)" in a cell will return Saturday. You can 
also use the above function in your code. The following inserts the name of the current day into 
the active cell.

```
ActiveCell.Value = WeekDayName(Now())
```

22.2 Converting Now() To Hours, Minutes, Day, Month 
And Year

The following statements illustrate how to convert Now() to different values:

```
theHour = Hour(Now())
theMinute = Minute(Now())
theDate = Day(Now())
theMonth = Month(Now())
theYear = Year(Now())
```

To get the name of the day or month:

```
dayName = Format(Now(), "dddd")
monthName = Format(Now(), "MMMM")
```

22.3 Getting A Date Input From A User

A simple way to get a date from a user is to put three edit boxes on a userform and ask the user to 
enter in the month, day, and year. You can then convert it to a date using DateSerial(year, 
month, day). The biggest headache is insuring that the user does not put in bad data, such as 14 
months, or 31 days in February.

You would put the following in the userform code sheet, assuming that your OK button is 
CommandButton1, and your cancel button is CommandButton2:

```
Private Sub CommandButton1_Click()
    'set bResponse to true since the OK button was selected
```
bResponse = True

'hide, do not unload the userform
Me.Hide
End Sub

Private Sub CommandButton2_Click()

' set bResponse to false since the cancel button was selected
bResponse = False

'hide, do not unload the userform
Me.Hide
End Sub

At the top of a regular module, you would put the following two variables:

Public bResponse As Boolean

The variable bResponse stores the result of the OK or Cancel button. The variable dlg is assigned
to the userform. Be declaring it as a module level variable, the userform retains it values from
run to run.

The following is the procedure you would then run to display the form and get the date.

Sub GetDate()
    Dim dDate As Date
    Dim m As Integer, d As Integer, y As Integer

    'assign userform to module level variable so that values are redisplayed
    'if loop back occurs or if procedure is run again
    showdialog:

    'display dialog, (code assigned to button would halt if cancel selected)
    UserForm1.Show

    'check the value assigned to bResponse by the command buttons
    If Not bResponse Then End

    'assign variables the values in the boxes; convert to numbers using Val()
    m = Val(UserForm1.TextBox1.Text)
    d = Val(UserForm1.TextBox2.Text)
    y = Val(UserForm1.TextBox3.Text)
'make certain values are entered in all boxes

If m = 0 Or d = 0 Or y = 0 Then
    MsgBox "The date fields must contain numbers." & _
    "Zero is not allowed."

'loop back to display dialog

GoTo showdialog
End If

'check the month

If m > 12 Then
    MsgBox "The month value is too large"

'loop back to display dialog

GoTo showdialog
End If

'make certain 4 digit year entered

If y < 1000 Then
    MsgBox "You must enter all 4 digits of the year."

'loop back to display dialog

GoTo showdialog
End If

'convert the month, day, and year to a date value

dDate = DateSerial(y, m, d)

'check the day

If Day(dDate) <> d Then
    MsgBox "You have too many days in the month"

'loop back to display dialog

GoTo showdialog
End If

'Remove userform from memory

Unload UserForm1
'display result if all the above tests are OK

MsgBox "You entered " & Format(dDate, "mmm d, yyyy")
End Sub

22.4 How To Find A Specified Time In A Specific Range

Sub fndtime()
    Dim timetofind As Double
    Dim anyRange As Range
    Dim userInput
    Dim iLoc As Long

    'set the range to search for a matching time
    Set anyRange = Range("A1:A100")

    'get the time to find
    userInput= InputBox("What time do you want to find?")

    'exit if no value entered
    If userInput= "" Then
        Exit Sub
    End If
    If IsDate(userInput) Then

        'if the entry is a date then do this
        'convert value to number
        timetofind = CDbl(TimeValue(userInput))

        'get the closest match
        iLoc = Application.Match(timetofind, anyRange, 1)

        'select that cell
        anyRange.Cells(iLoc, 1).Select
    Else

        'if entry was not a date say so!
        MsgBox "Please enter a date!"
    End If
End Sub

22.5 How To Find A Date In A Range
The following code illustrates how to find a date in a range. The range must be one column wide.

```vba
Sub FindDate()
    Dim searchRange As Range
    Dim dDate As Date
    Dim I As Variant
    Dim cell As Range
    dDate = "3/1/2003"
    Set searchRange = Range("A1:A10")
    I = Application.Match(CLng(dDate), searchRange, 0)
    If Not IsError(I) Then
        Set cell = searchRange(I)
        MsgBox cell.Address
    Else
        MsgBox "No match found"
    End If
End Sub
```

Another approach is the following:

```vba
Dim searchResult As Variant
searchResult = Application.WorksheetFunction.Match(
    CLng(DateSerial(2008,1,1)),
    Worksheets("Sheet1").Range("A1:A100"), 1)
If IsError(searchResult) Then
    MsgBox "Date out of range"
Else
    'action to take if found
End If
```


### 22.6 Using A Macro To Insert Current Time

Date and time (only time showing)

```vba
Sub EnterTime()
    ActiveCell.Value = Now
    ActiveCell.NumberFormat = "hh:mm"
End Sub
```

Time only

```vba
Sub EnterTime()
    ActiveCell.Value = Time
    ActiveCell.NumberFormat = "hh:mm"
End Sub
```
22.7 Having Excel Wait For A Few Seconds

The following subroutine will wait for "x" seconds. For example,

`Wait_Awhile 10`

will cause Excel to do nothing for ten seconds and just display the hourglass. Such a routine is useful if you write a message to a sheet and want to pause for the user to read it.

```vba
Sub Wait_Awhile(iSeconds As Integer)
    Dim waitTime
    'calculate the time to wait until
    waitTime = Now() + TimeSerial(0, 0, iSeconds)
    Application.Wait waitTime
End Sub
```

22.8 Application.Wait

`Application.Wait` takes an Excel date-time serial number as its argument. You pass a time-of-day, not a duration, to the method. Each of the following lines of code will pause your code for 10 seconds.

```vba
Application.Wait Now() + TimeValue("0:0:10")
Application.Wait Now() + TimeSerial(0,0,10)
```

22.9 Date Comparisons

The following was posted by John Green to help a user under Excel date comparisons. The question posted was:

```
I am having problems achieving consistent results when using the AutoFilter Method programmatically. This seems to center around when it is acceptable to use an English style date format ("dd/mm/yyyy") and when I must use an American style format ("mm/dd/yyyy").
```

John's response: Excel VBA uses two quite different methods for comparison.

1) The first is when you want equality. In this case, the comparison is made against the formatted appearance of the date on the sheet. If you have formatted a cell as "ddd dd/mm/yy", you must match that format:

```vba
Dim myDate As Date
Dim DateStr As String
myDate = DateSerial(98, 1, 14)
DateStr = ":= = " & Format(myDate, "ddd dd/mm/yy")
ActiveCell.CurrentRegion.AutoFilter Field:=1, Criteria:=DateStr
```
You can remove the leading "=" if you wish, it makes no difference.

2) The second is when you use any of the comparison operators apart from "=" by itself. In this case the comparison is made against the internal date serial number and your date can be in any recognized *AMERICAN* numeric format. The following works on dates formatted as above - "ddd dd/mm/yy".

```vba
Dim myDate As Date
Dim DateStr As String
myDate = DateSerial(98, 1, 14)
DateStr = "<=" & Format(myDate, "mm/dd/yy")
ActiveCell.CurrentRegion.AutoFilter Field:=1, Criteria:=DateStr
```

The following work equally as well

```vba
DateStr = "<=" & Format(myDate, "mm/dd/yyyy")
DateStr = "<=" & Format(myDate, "yyyy-mm-dd")
```

### 22.10 Using Code To Create A Calendar In A Worksheet

The following Microsoft knowledge base article contains sample code for creating a monthly calendar on a worksheet:

[http://support.microsoft.com/support/kb/articles/Q150/7/74.asp](http://support.microsoft.com/support/kb/articles/Q150/7/74.asp)

Q150774 - XL: How to Create a Monthly Calendar

### 22.11 Getting The End Of A Month

To get the end of the month, use a statement like the following:

```
EndOfMonth = DateSerial(Year(anydate), Month(anydate)+1, 0)
```

### 22.12 Inserting The Date On Every Worksheet And Footer

The following code will insert the date in cell A1 of every worksheet in the active workbook:

```vba
Dim ws As Worksheet
For Each ws In ActiveWorkbook.Worksheets
ws.Cells(1, 1).Value = "" & Format(Now(), "mmm d, yyyy")
Next
```

the double quote, quote, and double quote in front of the `Format` statement makes the date a string entry instead of a date entry. This means that the column does not have to be sized to display the date entry.
The following inserts the current date into the right footer of every worksheet and chart sheet:

```vba
Dim oSh As Object
For Each oSh In ActiveWorkbook.Worksheets
    oSh.PageSetup.RightFooter = Format(Now(), "mmm, d, yyyy")
Next
For Each oSh In ActiveWorkbook.Charts
    oSh.PageSetup.RightFooter = Format(Now(), "mmm, d, yyyy")
Next
```

Notice that the above code did not have to activate or select any sheet. You could also replace `ActiveWorkbook` with a reference to any open workbook and run the above code.

22.13 Using Milliseconds when Excel Waits

First declare the following function at the top of a module:

```vba
Declare Sub Sleep Lib "kernel32" Alias "Sleep" _
(ByVal dwMilliseconds As Long)
```

Then use the following syntax to call the Sleep function or a one second delay:

```vba
Sleep (1000)
```

22.14 Writing The Date And Time Out To A Cell

The following statements illustrate how to write the date and/or time out to a cell:

```vba
ActiveCell.Value = Format(Now(),"mm/dd/yyyy")
```

or, if you want to write the date and time out to a cell, and that cell is identified by a range name use:

```vba
Range("CellForDateTime").Value = "'" _
    Format(Now(),"mm/dd/yyyy hh:mm")
```

Please note that the above assumes that the cell is on the active sheet. If it is not, then qualify the `Range` statement with the sheet. Also, by prefixing the date and time in the second example with a quote, it makes the entry a string so that the column does not have to be widened to display the value.

The following illustrates how to write the date to a cell and format the cell:

```vba
With Range("B2")
    .Value = Now()
    .NumberFormat = "mm/dd/yy"
End With
```

22.15 Measuring Time Change
If you use GetTickCount or TimeGetTime from the WIN API you won't have to worry about the midnight rollover. They return the time in milliseconds since the system was started. I believe the latter "rolls over" every 49 days.

Declare Function GetTickCount Lib "kernel32" () As Long
Declare Function TimeGetTime Lib "winmm.dll" () As Long

22.16 Automatically Entering The Date Into A Edit Box

The following illustrates how to enter the date into an edit box:

UserForm1.TextBox1.Text = Format(Now(), "mmm-yy")

22.17 Days Left Counter

Dates are stored as the number of days from a base date, so just subtracting the dates should give you the number of days.

Sub tester2()
    DateVal = #6/25/99#
    MsgBox CLng(Date - DateVal) & " days have elapsed"
End Sub

You can also look at the Excel VBA function DateDiff.

22.18 Select Case Using Dates

Try enclosing your dates inside of pound signs, and use the "To" keyword:

    Case #4/20/99# To #5/21/99#

22.19 Validating Date Entries

Validating dates is one of Visual Basic's weaknesses. For example for the US, the string 15/2/98
refers to month 15, which obviously is incorrect. If your system is set to use a day/month/year
format, then 2/15/98 would be an incorrect entry.

The following code illustrates a way around this problem

Sub ValidateADate()
    Dim strMyDateString As String

    'prompt for a date
    strMyDateString = InputBox("Enter a date in mm/dd/yy format")

    'exit if no date entered
If strMyDateString = "" Then Exit Sub

'convert entry to a date and compare to the system's short format

If Format(CDate(strMyDateString), "Short Date") = strMyDateString Then
    MsgBox "it is a date."
Else
    MsgBox "it is not a date"
End If
End Sub

Other defined formats that you can use are:

Format Name Description

Long Date Display a date according to your system's long date format.

Medium Date Display a date using the medium date format appropriate for the language version of the host application.

Short Date Display a date using your system's short date format.

You may want to use a custom format instead of the above, that way you can insure that the format is the way you request it. If you do this, you can not use m for month, d for date, and y for year as these are language dependent. The following illustrates this:

Sub Validate_With_Custom_Format()
    Dim strMyDateString As String
    Dim mStr As String, dStr As String, yStr As String
    Dim sStr As String
    Dim customFormat As String

    'get user's settings for month, day, year, and date separator
    With Application
        mStr = .International(xlMonthCode)
        dStr = .International(xlDayCode)
        yStr = .International(xlYearCode)
        sStr = .International(xlDateSeparator)
    End With

    'construct a custom format (in US format, this would be m/d/yy)
    customFormat = mStr & sStr & dStr & sStr & yStr & yStr

    'prompt for a date in the custom format
strMyDateString = InputBox("Enter a date in " & _
    customFormat & " format")

'exit if no date entered

If strMyDateString = "" Then Exit Sub

'convert the string to a date in the custom format and compare to the
'string that was entered

    If Format(CDate(strMyDateString), customFormat) = _
    strMyDateString Then
    MsgBox "It is a date."
    Else
    MsgBox "It is not a date"
    End If
End Sub
23. SHORTCUT KEYS

23.1 Individual Disable Shortcut Keys

The following lists individual commands that can disable the different shortcut keys in Excel. Please note nothing prevents a user from double clicking on a workbook containing code that resets the keys (and menus).

Sub DisableExcelShortcutKeys()

'Customize Function Keys

Application.OnKey "{F1}", "" 'Displays On-Line Help
Application.OnKey "+{F1}", "" 'What's This
Application.OnKey "%{F1}", "" 'Insert a chart sheet
Application.OnKey "%+{F1}", "" 'Insert a worksheet

Application.OnKey "{F2}", "" 'Edit active cell
Application.OnKey "+{F2}", "" 'Edit a cell comment
Application.OnKey "%{F2}", "" 'Save As Command
Application.OnKey "%+{F2}", "" 'Save Command

Application.OnKey "{F3}", "" 'Paste number into formula
Application.OnKey "+{F3}", "" 'Paste function into formula
Application.OnKey "^{F3}", "" 'Define a Name
Application.OnKey "^+{F3}", "" 'Create names from addresses

Application.OnKey "{F4}", "" 'Repeat last action
Application.OnKey "+{F4}", "" 'Repeat last find
Application.OnKey "^{F4}", "" 'Close window
Application.OnKey "%{F4}", "" 'Exit

Application.OnKey "{F5}", "" 'GoTo
Application.OnKey "+{F5}", "" 'Find command
Application.OnKey "^{F5}", "" 'Restore window size

Application.OnKey "{F6}", "" 'Move to next pane
Application.OnKey "+{F6}", "" 'Move to previous pane
Application.OnKey "^{F6}", "" 'Move to next Workbook window
Application.OnKey "^+{F6}", "" 'Move to previous Workbook window

Application.OnKey "{F7}", "" 'Spelling command
Application.OnKey "^{F7}", "" 'Move the window

Application.OnKey "{F8}", "" 'Extend a selection
Application.OnKey "+{F8}", "" 'Add to the selection
Application.OnKey "^{F8}", "" 'Resize window
Application.OnKey "%(F8)"", "" 'Display macro dialog
Application.OnKey "(F9)"", "" 'Calculate all workbooks
Application.OnKey "+(F9)"", "" 'Calculate active worksheet
Application.OnKey "^(F9)"", "" 'Minimize the workbook window
Application.OnKey "(F10)"", "" 'Activate menu bar
Application.OnKey "+(F10)"", "" 'Display a shortcut menu
Application.OnKey "^(F10)"", "" 'Restore workbook window
Application.OnKey "(F11)"", "" 'Create a chart
Application.OnKey "+(F11)"", "" 'Insert worksheet
Application.OnKey "^(F11)"", "" 'Insert Excel 4 macro sheet
Application.OnKey "%(F11)"", "" 'Activate VBE
Application.OnKey "(F12)"", "" 'Save As command
Application.OnKey "+(F12)"", "" 'Save command
Application.OnKey "^(F12)"", "" 'Open command
Application.OnKey "+%{F12}"", "" 'Print command

'Customize Navigation Keys
Application.OnKey "{DOWN}"", "" 'Down arrow
Application.OnKey "{END}"", "" 'End
Application.OnKey "{HOME}"", "" 'Home
Application.OnKey "{LEFT}"", "" 'Left arrow
Application.OnKey "{PGDN}"", "" 'Page down
Application.OnKey "{PGUP}"", "" 'Page up
Application.OnKey "{RIGHT}"", "" 'Right arrow
Application.OnKey "{UP}"", "" 'Up arrow
Application.OnKey "^{}.", "" 'Move within selection
Application.OnKey "%^{LEFT}"", "" 'Jump left between selections
Application.OnKey "%^{RIGHT}"", "" 'Jump right between selections

'Customize File functions
Application.OnKey "^{n}"", "" 'New
Application.OnKey "^{o}"", "" 'Open
Application.OnKey "^{p}"", "" 'Print
Application.OnKey "^{s}"", "" 'Save

'Customize Editing functions
Application.OnKey "^{c}"", "" 'Copy
Application.OnKey "^{v}"", "" 'Paste
Application.OnKey "^{x}"", "" 'Cut
Application.OnKey "^{d}", " " 'Fill Down
Application.OnKey "^{r}", " " 'Fill Right
Application.OnKey "^{ENTER}\'', " " 'Fill Selection
Application.OnKey "^{f}", " " 'Find
Application.OnKey "^{g}", " " 'GoTo
Application.OnKey "^{h}", " " 'Replace
Application.OnKey "^{y}", " " 'Repeat last action
Application.OnKey "^{z}", " " 'Undo

'Customize Insert functions
Application.OnKey "^{k}", " " 'Insert hyperlink
Application.OnKey "^{+{+}}", " " 'Insert blank cells

'Customize Format functions
Application.OnKey "^{1}\'', " " 'Format cells
Application.OnKey "^{+{~}}", " " 'Apply general format
Application.OnKey "^{+{\$}}", " " 'Apply currency format
Application.OnKey "^{+{\%}}", " " 'Apply percentage format
Application.OnKey "^{+{^}}", " " 'Apply exponential format
Application.OnKey "^{+{#}}", " " 'Apply date format
Application.OnKey "^{+{8}}", " " 'Apply time format
Application.OnKey "^{+{!}}", " " 'Apply Number format
Application.OnKey "^{+{&}}", " " 'Apply Outline Border
Application.OnKey "^{+{_}}", " " 'Remove Borders
Application.OnKey "^{b}\'', " " 'Toggle Bold format
Application.OnKey "^{i}\'', " " 'Toggle Italic format
Application.OnKey "^{u}\'', " " 'Toggle Underline format
Application.OnKey "^{5}\'', " " 'Toggle Strikethrough
Application.OnKey "^{9}\'', " " 'Hide rows
Application.OnKey "^{0}\'', " " 'Hide columns
Application.OnKey "^{+{()}\'', " " 'Unhide rows
Application.OnKey "^{+{}}\'', " " 'Unhide columns

'Customize Miscellaneous functions
Application.OnKey "^{-}\'', " " 'Delete selection
Application.OnKey "^{(DELETE)\'', " " 'Delete to the end
Application.OnKey "^{+{ENTER}\'', " " 'Complete and move up
Application.OnKey "^{%(ENTER)\'', " " 'New Line in a cell
Application.OnKey "^{(TAB)\'', " " 'Complete and move right
Application.OnKey "^{+(TAB)\'', " " 'Complete and move left
Application.OnKey "^{=}\'', " " 'Insert AutoSum Formula
Application.OnKey "^{(j)\'', " " 'Insert Date
Application.OnKey "^{+{:}}\'', " " 'Insert Time
Application.OnKey "^+{"}"", "" 'Copy value from above
Application.OnKey "^{'}"", "" 'Toggle Display
Application.OnKey "^{`}"", "" 'Copy formula from above
Application.OnKey "^{a}"", "" 'Display formula pane
Application.OnKey "^+{a}"", "" 'Insert argument names

'Customize Transition Navigation Keys

With Application
    .TransitionMenuKey = ""
    .TransitionMenuKeyAction = xlExcelMenus
    .TransitionNavigKeys = False
End With
End Sub

Now if you want to enable these keys, then you would need to exclude the last argument. For example:

Application.OnKey "(F1)"

23.2 Redefining The Plus And Minus Keys

This is an interesting way to increase values in cells - by redefining the plus and minus keys to add or subtract one from the value in a cell. Don't forget to reset the keys when done!

Sub KeysOn()
    Application.OnKey "{+}","IncrementCell"
    Application.OnKey "-","DecrementCell"
End Sub

Sub KeysOff
    Application.OnKey "{+}"
    Application.OnKey "-"
End Sub

Sub IncrementCell()
    If IsNumeric(ActiveCell.Value) Then
        ActiveCell.Value = ActiveCell.Value+1
    End If
End Sub

Sub DecrementCell()
    If IsNumeric(ActiveCell.Value) Then
        ActiveCell.Value = ActiveCell.Value-1
    End If
End Sub

23.3 Disabling Almost All Of TheShortcut Keys

The following code will disable or enable almost all of the Excel special keys such as CTL-F for find, or CTL-F3 for creating range names. You can disable or enable specific keys to tailor the
code to your needs. For more information on specifying keys, highlight the keyword **OnKey** and press F1 for help. If you test this code, make certain you always have a way to run the macro that enables the keys.

```vba
Sub DisableKeys()
    Disable_Or_Enable_Keys "disable"
End Sub

Sub Enable_Keys()
    Disable_Or_Enable_Keys
End Sub

Sub Disable_Or_Enable_Keys(Optional setOption)

    'disables or resets ALL keys
    'if argument is "disable" then keys are disabled
    'if no argument supplied, or it is not "disable" then keys are enabled

    Dim Keys, I As Integer, J As Integer
    Keys = Array("", "+", "^", "%", "+^", "+%", "^%", "^+%")
    On Error Resume Next
    With Application
        For I = 1 To 8
            'handle combination keys
            For J = 1 To 12
                If LCase(setOption) = "disable" Then
                    .OnKey Keys(I) & "{" & "F" & J & "}", 
                Else
                    .OnKey Keys(I) & "{" & "F" & J & "}"
                End If
            Next
            If I > 2 Then
                '"Normal" characters CAN be used!!
                For J = 33 To 148
                    Select Case J
                        Case 123 To 145, 37, 39, 40 To 43, 91 To 94
                        Case Else
                            If LCase(setOption) = "disable" Then
                                .OnKey Keys(I) & Chr$(J), 
                            Else
                                .OnKey Keys(I) & Chr$(J)
                            End If
                    End Select
            End If
        Next
        'handle special keys such as CTL-pgup or CTL-tab
    End With
End Sub
```
If LCase(setOption) = "disable" Then
  .OnKey "^[pgup]", ""
  .OnKey "^[pgdn]", ""
  .OnKey "^[tab]", ""
  .OnKey "^+{tab}", ""
Else
  .OnKey "^[pgup]"
  .OnKey "^[pgdn]"
  .OnKey "^[tab]"
  .OnKey "^+{tab}"
End If
End With
End Sub

23.4 Disabling Shortcut Menu Commands

The names of the row and column short cut menus are "Row" and "Column". The name of the cell pop-up menu is "Cell". To disable them use a statement like

CommandBars("Row").Enabled = False
CommandBars("Column").Enabled = False
CommandBars("Cell").Enabled = False

However, there is a trap. There are three extra pop up menus called Cell, Row and Column that operate in the PageBreak view. To get to them you can use something like:

CommandBars(CommandBars("Row").Index + 3).Enabled = False

The following is a way to disable the Cell popups:

Dim cellIndex As Long

'store the indexi to the cell popup commandbar

cellIndex = Application.CommandBars("cell").Index
Application.CommandBars(cellIndex).Enabled = False
Application.CommandBars(cellIndex + 3).Enabled = False

Be sure to re-enable the commandbars you disable as the disable setting is not reset when Excel closes.

23.5 Making shortcut Keys Sheet Specific

Shortcut keys are always global in scope with Excel. However, if you'd like a macro run by a shortcut key to behave differently based on what sheet is currently active, that's very easy to do. Here is an example.

Let us say this macro is assigned to a short cut key, for example Alt-A. It will run specific code based on the sheet name.
Sub MultiSheetMacro()
    Select Case ActiveSheet.Name
        Case "Sheet1"
            ' Run code here that is specific to Sheet1
        Case "Sheet2"
            ' Run code here that is specific to Sheet2
        Case Else
            MsgBox "Sorry, this macro doesn't apply to this sheet"
    End Select
End Sub

You can add as many sheet-specific cases as you want, and you don't have to put all the code inside the Select Case statement. You can create separate procedures for each sheet and then simply call those procedures from the appropriate Case.

To insure that the sheet name is recognized, put Option Compare Text at the top of your code. Or, convert the sheet name to upper case and compare to upper case names:

Sub MultiSheetMacro()
    Select Case Ucase(ActiveSheet.Name)
        Case "SHEET1"
            ' Run code here that is specific to Sheet1
        Case "SHEET2"
            ' Run code here that is specific to Sheet2
        Case Else
            MsgBox "Sorry, this macro doesn't apply to this sheet"
    End Select
End Sub
24. TOOLBARS

24.1 Using Attached Toolbars

In Excel you can create your own toolbar by selecting View, Toolbars, and using the New toolbar option available on the dialog that appears. And, you can assign buttons to the toolbar by selecting the Customize option. If you choose the macro category of buttons, you can assign your own macros to buttons on the toolbar.

Such a custom toolbar is typically only useful for a particular workbook. You can attach the toolbar to a workbook by doing the following:

♦ Choose View, Toolbars, Customize

♦ Click the Attach button and select the custom toolbar and copy the custom toolbar to the workbook

If you modify an attached toolbar, you need to repeat the above steps to attach the new version. Otherwise, the modifications will be lost.

There are several problems with attached toolbars:

♦ The toolbar doesn't disappear when you close the workbook

♦ The toolbar doesn't appear when you open the toolbar

♦ If you distribute the workbook to others, change the toolbar, and re-distribute it, the users may not see the new toolbar.

To solve these problems, use macros like the following, assuming that your toolbar is named "MyToolBar", and that it has two buttons on it:

Sub Auto_Open()
    SetUpButtons
End Sub

Sub SetUpButtons()
    With CommandBars("MyToolBar")
        .Visible = True
        .Controls(1).OnAction = "FirstMacro"
        .Controls(2).OnAction = "SecondMacro"
    End With
End Sub

Sub Auto_Close()
    On Error Resume Next
    CommandBars("MyToolBar").Delete
End Sub
24.2 Resetting The Macros On A Custom Toolbar

Excel allows you to create a custom toolbar and attach it to a workbook. However, if you give this workbook to someone else, the toolbar buttons will refer back to the original workbook’s path. The result is a nasty error box saying a macro could not be found. The buttons’ OnAction property still points to the original placement of the workbook, when the macros were made. The following code illustrates a neat way to solve this, and will work with any custom toolbar.

Sub Fix_Toolbar

' the above macro name should be included in your Auto_Open macro
'so that it is run each time the workbook is opened.

Dim tToolbar As String
Dim i As Integer

' put your toolbar name in place of Name of the toolbar

tToolbar = "Name of the toolbar"

' rotate through each of the buttons on the toolbar

For i = 1 To CommandBars(tToolbar).Controls.Count

' get the current OnAction, which includes the path

nName = CommandBars(tToolbar).Controls(i).OnAction

' re-assign the macro name excluding the path

    CommandBars(tToolbar).Controls(i).OnAction = _
    Right(nName, Len(nName) - InStr(nName, "!"))

Next i

End Sub

If you make corrections to an attached toolbar, you have to attach it again, or the corrections will not be saved

24.3 Using A Macro To Create A Toolbar

The following macro illustrates the code needed to create a toolbar with several buttons on it, with descriptive tooltips. The face on the toolbar is set by the FaceID property. The macro after this one shows you how to find the Ids for over 2000 button faces.

Sub CreateAToolbar()

    Const tBarName As String = "Tool Bar name"

    ' Delete CommandBar if it exists
On Error Resume Next
CommandBars(tBarName).Delete
On Error GoTo 0

'create CommandBar

CommandBars.Add Name:=tBarName

'define an object variable to refer to the CommandBar

With CommandBars(tBarName)

'add button use 1 to specify a blank custom face

With .Controls.Add(ID:=1)
  .OnAction = "AddInfo"

'this adds the smiley FaceID

  .FaceID = 59
  .Caption = "Add Report Information"
End With

'add next button with a separator bar

With .Controls.Add(ID:=1)
  .OnAction = "RemoveInfo"

'this adds the eraser FaceID

  .FaceID = 47
  .Caption = "Remove Report Information"

'this adds the separator bar

  .BeginGroup = True
End With

'display the toolbar

  .Visible = True
End With
End With
End Sub

24.4 Using FaceIDs to specify a Toolbar Button Face
The **FaceID** property of a control specifies the button face or image. For example:

```
CommandBars("My toolbar").Controls(1).FaceID = 80
```

puts a button face with the letter "A" on the control.

The following macro creates a workbook that lists all of the available button FaceIDs:

```vba
Sub DisplayControlFaces()
  Const tBarName As String = "Temp toolbar"

  'add a new Workbook for the faces
  Workbooks.Add

  'Delete CommandBar if it exists
  On Error Resume Next
  CommandBars(tBarName).Delete

  'create CommandBar
  CommandBars.Add Name:=tBarName

  'define an object variable to refer to the CommandBar
  With CommandBars(tBarName)

    'use an error trap to handle missing FaceIDs in the code below
    On Error GoTo eTrap

    'specify an ID of one for a blank custom button face
    With .Controls.Add(ID:=1)

    'change the button image through all that are available
    For i = 1 To 5500
      .FaceID = i
    Next i

    'copy face and paste to the worksheet
    .CopyFace
    ActiveSheet.Paste

  End With

End Sub
```
'record the face ID

    With ActiveCell.Offset(1, 0)
    .Value = i
    .HorizontalAlignment = xlLeft
    End With

'increase counter and select next destination cell

    J = J + 1
    If J <= 5 Then
        ActiveCell.Offset(0, 1).Select
    Else
        ActiveCell.Offset(3, -5).Select
        J = 0
    End If
    donext:
    Next
    End With

'remove the commandbar as it is not needed

    CommandBars(tBarName).Delete
    Exit Sub
    eTrap:

'venot all FaceIDs exist. This handles any missing ones

    Resume donext
    End Sub

24.5 Putting Custom Button Faces On Toolbar Buttons

The following macro creates a custom toolbar and then adds buttons to it using custom button faces that have been saved on a sheet named "Button Faces". To create custom faces:

- Go to the Customize Dialog Box - Select View, Toolbars, Customize
- Select the button you want to edit. Then right click on it to display the button popup menu. Select the Edit option and modify the button face.
- With the button selected, right click on the button and select Copy Button Face
- Exit the customize dialog and go to a worksheet where you can store the button face. The sheet name used in this example is called "Button Faces" and is part of the workbook containing the macro code.
- On this sheet, just do a normal Edit, Paste. The button face will be pasted into the worksheet as a picture object. It will be very small, which is OK. With
the button selected, select **Format, Selected Object** and remove the frame around the object if there is one.

◆ When you select the button face, you will see the name that Microsoft Excel has given this picture object in the Name box. For example, it may be called "Picture 1". Click in the name box (adjacent to the formula edit box) and rename the picture by typing in a new name and pressing enter. The name can contain spaces. To help you in the future, enter this name in a cell adjacent to the button face.

At this point, you can now use the button face that you have stored in the above worksheet and have a macro paste it on your own button. The following macro illustrates creating a new toolbar with two custom buttons and one regular button. The faces for the custom buttons are named "add button" and "change button", and are located on a sheet named "button faces" in the workbook containing this code.

```vba
Sub ToolBarWithCustomFace()
    Const tBarName As String = "Tool Bar name"

    'Delete CommandBar if it exists
    On Error Resume Next
    CommandBars(tBarName).Delete
    On Error GoTo 0

    'create CommandBar
    CommandBars.Add Name:=tBarName

    'define an object variable to refer to the CommandBar
    With CommandBars(tBarName)

        'add button use 1 to specify a blank custom face
        With .Controls.Add(ID:=1)
            .OnAction = "AddInfo"
            .Caption = "Add Report Information"
        End With

        'this adds a custom face to the button
        ThisWorkbook.Sheets("button faces")_.Shapes("Picture 2").Copy .PasteFace
    End With

    'display the toolbar
End Sub
```
24.6 Hiding And Restoring The Toolbars And Menus

The following code will hide all the toolbars and menus:

```vba
Dim c
For Each c In CommandBars
    c.Enabled = False
Next
```

The above code will not only hide all the toolbars and menus, it will disable all the menus, with one exception. That is the toolbar that appears if one right clicks on a toolbar button - if the user has not upgraded to Excel SR-2.

To display a specific toolbar or menu, you first have to set its `Enabled` property back to `True`. If it is not visible, you also have to set the `Visible` property to `True`. For example:

```vba
With CommandBars("My custom Menu")
    .Enabled = True
    .Visible = True
End With
```

To re-display the menus and the toolbars as the user had them, use the following code:

```vba
Dim c
For Each c In CommandBars
    c.Enabled = True
Next
```

You can also disable specific menus by referring to them by name:

```vba
CommandBars("Worksheet Menu Bar").Enabled = False
```

24.7 How To Prevent Your Custom Toolbar Buttons From Appearing Faded

If you construct special toolbar buttons using the toolbar button editor, the buttons may appear faded on other users' machines. This phenomena occurs when the toolbar buttons are constructed with Windows colors set to "millions of colors" and the user with the faded toolbar has his Windows colors set to a lower setting. The solution is to use a lower color setting for your display.

24.8 Adding Tool Tips To Buttons
When you create a toolbar, the button tool tip is the caption that is assigned to the button. For example:

```vba
With CommandBars("My toolbar")
  .Controls(1).Caption = "Load new data"
End with
```
25. COMMANDBARS AND MENUS

25.1 Using Excel's Built-In Dialogs

Microsoft Excel has over 200 built-in dialogs that you can use in your macros. To display a built-in dialog, you would use a statement like the following:

```
Application.Dialogs(vb constant).Show
```

or

```
response = Application.Dialogs(vb constant).Show
```

For example, the following displays the built-in printer selection dialog:

```
Application.Dialogs(xlDialogPrint).Show
```

or

```
response = Application.Dialogs(xlDialogPrint).Show
```

The first statement just displays the box. The second statement displays the box and assigns the variable response the value True of the OK button was selected, and the value False if the Cancel button was selected. The above not only displays the dialog box, but takes the action that the box is designed to do if OK is selected in the box. Please note that not all the "xlDialog..." constants will display a built-in dialog.

Both statements use a Visual Basic constant to indicate which dialog box to display. The Visual Basic constants that begin with "xlDialog" are the ones to use. Please note that not all the "xlDialog..." constants will display a built-in dialog. To see a list of the constants, do the following:

- display the Object Browser, select "<All Librarys>". In the Classes list, select "xlBuiltInDialog". A list of the constants will appear in the members list box.

For example, if you want to use the dialog box that appears when you save a new file, you could use the following statement:

```
Application.Dialogs(xlDialogOpen).Show
```

To display the Save As dialog and insure that the use saved the file, you could use the following statements:

```
Do
    response = Application.Dialogs(xlDialogSaveAs).Show
Loop Until response = True
```
Additional examples of using the **SaveAs** and **DialogOpen** built in dialogs are found in the file section of this book.

The following is a list of just a few of the Visual Basic constants and the dialog boxes that they display.

<table>
<thead>
<tr>
<th>Constant Dialog Box</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>xlDialogOpen</strong> The open file box</td>
</tr>
<tr>
<td><strong>xlDialogSaveAs</strong> The save as box</td>
</tr>
<tr>
<td><strong>xlDialogSetPrintTitles</strong> The set print titles box (from Excel 4)</td>
</tr>
<tr>
<td><strong>xlDialogChartWizard</strong> The chart wizard</td>
</tr>
<tr>
<td><strong>xlDialogCreateNames</strong> The create names box</td>
</tr>
<tr>
<td><strong>xlDialogFont</strong> The cell font box</td>
</tr>
<tr>
<td><strong>xlDialogGoalSeek</strong> The goal seek box</td>
</tr>
<tr>
<td><strong>xlDialogUnhide</strong> The unhide a file box</td>
</tr>
<tr>
<td><strong>xlDialogZoom</strong> The zoom box</td>
</tr>
</tbody>
</table>

One caution on using the Microsoft Excel dialog boxes: **If you try to display a built-in dialog box on a sheet where it can't be used, it will fail.** For example, the zoom box can't be used on a module sheet. But it works fine if the active sheet is a worksheet or a chart sheet.

### 25.2 CommandBar.Add Yields Err 91 on Workbook_Open

When referring to the **CommandBars** object in an event procedure such as the workbook open code in a workbook object's module, you need to precede it by **Application**:

```vba
Set cbarMine = Application.CommandBars.
    .Add("My Toolbar", msoBarTop, True, True)
```

if you do not, you may get an Err 91 message.

### 25.3 Adding A Menu Item To A Menu

The following are four macros that add and remove a menu item with a separator bar from a menu. The first is named "Auto_Open" and calls a routine named AddMenuItm that adds a new
menu item to a menu on the worksheet menus. The second is named "Auto_Close" and calls a routine named RemoveMenuAddition that removes the menu item when the file is closed. You can call AddMenuItem and RemoveMenuAddition repeatedly to add multiple menu items. Macros with the names Auto_Open and Auto_Close are run automatically by Excel when a file is either opened or closed.

Sub Auto_Open()
'Call the routine that adds the item, specifying the worksheet menu for
'the new menu item, the name to appear on the menu, the macro to be run,
'and whether or not to add a separator bar above the new item
AddMenuItem "Tools", "Convert Files", _
"Main_Procedure_For_Converting_Files", True
End Sub

Sub Auto_Close()
'Calls the routine that removes a menu item from the worksheet menu
'specifying the menu name and the menu item name.
RemoveMenuAddition "Tools", "Convert Files"
End Sub

Sub AddMenuItem(menuName As String, itemName As String, _
macroName As String, bAddSeperator As Boolean)
Dim mItem, newItem
' this removes item if it is on the menu
RemoveMenuAddition menuName, itemName
' add new menu item; setting the temporary property to true
' insures that the menu item disappears when Excel is closed
With CommandBars ("Worksheet Menu Bar").Controls(menuName)
  Set newItem = .Controls.Add(Type:=msoControlButton, _
    temporary:=True)
End With

' set caption, assign macro, add separator bar

With newItem
  .Caption = itemName
  .OnAction = macroName
  .BeginGroup = bAddSeperator
End With
End Sub

Sub RemoveMenuAddition(menuName As String, itemName As String)
Dim mItem, I As Integer
'this removes the menu item added by the above macro when the file is closed
With CommandBars ("Worksheet Menu Bar").Controls(menuName)
  For Each mItem In .Controls
    I = I + 1
    ' check for menu item; delete if found
    If mItem.Caption = itemName Then
      .Controls(I).Delete
    End If
  Next
End Sub
25.4 Adding A Menu and Sub Menus to the Worksheet Menu

The following illustrates how to add a new menu to the worksheet menu, and then how to add menu items and sub menus to the new menu. For simplicity, all menu items that have an OnAction property are set to run the same macro, "Hello_World". In your application, you can use different OnActions.

Sub AddingMenusAndSubMenus()

Dim c

' delete the new menu if it exists - calls subroutine listed below

Remove_New_Menu

' add the new menu to the worksheet menu before the help menu

With Application.CommandBars(1).Controls.
    Add(msoControlPopup, , , 9, True)
    .Caption = "MyMenu"
End With

' add a menu item to the new menu

Set c = .Controls.Add(msoControlButton)
c.Caption = "Item 1"
c.OnAction = "Hello_World"

' add a sub menu to the new menu

With .Controls.Add(msoControlPopup)
    .Caption = "Item 2"
End With

' add a menu item to the sub menu

Set c = .Controls.Add(msoControlButton)
c.Caption = "Sub 1 Item 1"
c.OnAction = "Hello_World"

' add a sub menu to the sub menu

With .Controls.Add(msoControlPopup)
    .Caption = "Sub 1 Item 2"
End With

' add three menu items to the lowest sub menu
Set c = .Controls.Add(msoControlButton)
c.Caption = "Sub 2 Item 1"
c.OnAction = "Hello_World"
Set c = .Controls.Add(msoControlButton)
c.Caption = "Sub 2 Item 2"
c.OnAction = "Hello_World"
Set c = .Controls.Add(msoControlButton)
c.Caption = "Sub 2 Item 3"
c.OnAction = "Hello_World"
End With

'add a menu item to first sub menu

Set c = .Controls.Add(msoControlButton)
c.Caption = "Sub 1 Item 3"
c.OnAction = "Hello_World"
End With

'add a menu item to the menu

Set c = .Controls.Add(msoControlButton)
c.Caption = "Item 3"
c.OnAction = "Hello_World"
End With
End Sub

Sub Hello_World()
    MsgBox "hello world"
End Sub

When the workbook is closed, the following runs and removes the new menu

Sub Auto_Close()
    Remove_New_Menu
End Sub

Sub Remove_New_Menu()
    Dim c

    'delete the new menu if it exists

    For Each c In Application.CommandBars(1).Controls
        If c.Caption = "MyMenu" Then c.Delete
    Next
End Sub

25.5 How To Add A New Menu Bar Like The Worksheet Menu Bar

The following adds a menu bar like the worksheet menu bar, except that the menus are named Menu1 and Menu2.
Sub Create_A_New_Menu_System()
    Dim My_Menu As CommandBar, newControl, newItem, subMenu

    'remove custom menu if it exists
    On Error Resume Next
    CommandBars("New Menu System").Delete
    On Error GoTo 0

    'create new menu and display it
    Set My_Menu = CommandBars.Add(Name:="New Menu System", _
        Position:=msoBarTop, _
        MenuBar:=False)
    My_Menu.Visible = True

    'add a menu to the new CommandBar
    Set newControl = My_Menu.Controls.Add(Type:=msoControlPopup)
    newControl.Caption = "Menu1"

    'add a menu item to the new menu
    With newControl
        Set newItem = .Controls.Add(Type:=msoControlButton)
        Set subMenu = .Controls.Add(Type:=msoControlPopup)
    End With

    With newItem
        .Caption = "This Says Hello"
        .OnAction = "SayHello"
    End With

    'add a sub menu to the new menu and add items to it
    With subMenu
        .Caption = "Additional Choices"
        Set newItem = .Controls.Add(Type:=msoControlButton)
        newItem.Caption = "Check On Fishing"
        newItem.OnAction = "FishingStatus"
        Set newItem = _
            .Controls.Add(Type:=msoControlButton)
        newItem.Caption = "Check On Golfing"
        newItem.OnAction = "GolfingStatus"
    End With
End Sub
'add a menu item that will restore the original menus

Set newItem = _
    newControl.Controls.Add(Type:=msoControlButton)
With newItem
    .Caption = "Remove the new menu system"
    .OnAction = "RemoveCustomMenu"
End With

'This next statement adds a separator bar

    .BeginGroup = True
End With

'add a menu to the new CommandBar

Set newControl = My_Menu.Controls.Add(Type:=msoControlPopup)
newControl.Caption = "Menu2"
Set newItem = newControl.Controls.Add(Type:=msoControlButton)
With newItem
    .Caption = "Say Goodbye"
    .OnAction = "SayGoodbye"
End With
End Sub

Sub SayHello()
    MsgBox "Hello world"
End Sub

Sub SayGoodBye()
    MsgBox "Goodbye!"
End Sub

Sub fishingStatus()
    MsgBox "Fishing is great all the time!!!"
End Sub

Sub golfingStatus()
    MsgBox "Who cares? I'd rather be fishing!"
End Sub

Sub RemoveCustomMenu()
    CommandBars("New Menu System").Delete
End Sub

25.6 Button Like Control On A Menu

The following creates a floating commandbar that has button like menu item in addition to a drop down menu item:

Sub Floating_New_Menu_System()
    Dim newMenu As CommandBar, newControl, newItem, subMenu
'remove custom menu if it exists

On Error Resume Next
CommandBars("New Menu System").Delete
On Error GoTo 0

'create new menu and display it

Set newMenu = CommandBars.Add(Name:="New Menu System", _
    Temporary:=True, _
    MenuBar:=False)
newMenu.Visible = True

'add a menu to the new CommandBar

Set newControl = newMenu.Controls.Add(Type:=msoControlPopup)
newControl.Caption = "Menu1"

'add a menu item to the new menu

With newControl
    Set newItem = .Controls.Add(Type:=msoControlButton)
End With

With newItem
    .Caption = "This Says Hello"
    .OnAction = "SayHello"
End With

'add second control that acts as a button

Set newItem = newMenu.Controls.Add( _
    Type:=msoControlButton, Temporary:=True)
With newItem
    .Caption = "Control Text"
    .Style = msoButtonCaption
    .TooltipText = "Control Tool Tip"
    .OnAction = "SayHello"
End With
End Sub

Sub SayHello()
    MsgBox "Hello"
End Sub

25.7 Hiding The Worksheet Menu

You can hide the Excel worksheet menu by using the following statement
However, if you press the ALT key and then the arrow keys, the menu is re-displayed. There is an easy way to solve the ALT key access to the disabled menus:

Application.CommandBars(1).Enabled = False
MenuBars.Add.Activate

The second line prevents the ALT key from working by creating and activating a blank menubar. Since it has no menus, it is not visible or accessible.

To restore the menus, use the following statements:

Application.CommandBars(1).Enabled = True
MenuBars(xlWorksheet).Activate

### 25.8 Putting A DropDown On A CommandBar

The following creates a commandbar with a dropdown on it and responds with the user's selection.

```vba
Sub CommandBarDemo()
    Dim cBar As CommandBar
    Dim I As Integer

    'delete the bar if it exists
    On Error Resume Next
    CommandBars("Combo Bar").Delete
    On Error GoTo 0

    'create the commandbar and make visible
    Set cBar = CommandBars.Add("Combo Bar", msoBarFloating)
cBar.Visible = True

    'add a dropdown control
    With cBar.Controls.Add(msoControlDropdown)

    'assign a macro to the drop down
    .OnAction = ThisWorkbook.Name & "!DropDownOnAction"

    'make wider:
    .Width = 200

    'add items to the drop down box
```
For I = 1 To 10
    .AddItem "drop down item " & I
Next
End With
End Sub

Sub DropDownOnAction()

' this displays what was selected

With CommandBars.ActionControl
    MsgBox "You selected " & .Text
End With
End Sub

25.9 Creating A Menu That Appears Only When A Particular Workbook Is Active

The following code illustrates how to have a unique menu added to the worksheet menu whenever a particular workbook is active. To do this, place the following code in the workbook's code module:

Private Sub Workbook_Deactivate()

' call routine that removes menu for this workbook

    Remove_Workbook_Menu
End Sub

Private Sub Workbook_Activate()

' call routine that adds the menu for this workbook

    Add_Workbook_Menu_And_Items
End Sub

You can access the workbook's code module by right clicking on the workbook object in the Project Explorer and selecting view code.

In a regular code module, put the following code:

Sub Remove_Workbook_Menu()

' this removes the menu if it is present

    On Error Resume Next
    CommandBars("Worksheet Menu Bar").Controls("OPTIONS").Delete
    On Error GoTo 0
End Sub

Sub Add_Workbook_Menu_And_Items()
Dim newMenu
Dim newMenuItem

'delete the menu if it exists by calling this subroutine
Remove_Workbook_Menu

'add a new menu to the worksheet menu. The menu is temporary and
'will disappear when Excel closes
With CommandBars("Worksheet Menu Bar")
  Set newMenu = .Controls.Add(_
    Type:=msoControlPopup, _
    before:=.Controls("Window").Index, _
    temporary:=True)
End With

'give the new menu a name
newMenu.Caption = "OPTIONS"

'add a menu item to the new menu
Set newMenuItem = newMenu.Controls.Add(Type:=msoControlButton)

'give the new menu a name and assign a macro to it
newMenuItem.Caption = "Menu Item 1"
newMenuItem.OnAction = "MenuItem1OnAction"

'add a second menu item to the new menu
Set newMenuItem = newMenu.Controls.Add(Type:=msoControlButton)

'give the new menu a name and assign a macro to it
newMenuItem.Caption = "menu Item 2"
newMenuItem.OnAction = "MenuItem2OnAction"
End Sub

Lastly, create an Auto_Close macro that calls the Remove_Menu macro when the workbook is
closed and an Auto_Open macro that displays the menu when the workbook is first opened.

Sub Auto_Close()
  Remove_Workbook_Menu
End Sub

Sub Auto_Close()
  Add_Workbook_Menu_And_Items
End Sub
25.10 Adding A Menu And Menu Items To The Worksheet Menu

The following adds a menu at the end of the worksheet menu, and then adds menu items and sub menus to it.

```
Sub AddMenu()
    With Application.CommandBars(1).Controls
        .Add(msoControlPopup)
        .Caption = "NewMenu"
        .Controls.Add(msoControlButton).Caption = "Item 1"
        .Caption = "Item 2"
        .Controls.Add(msoControlButton).Caption = "Sub1Item2"
        .Caption = "Sub2Item2"
        .Controls.Add(msoControlButton).Caption = "Sub1Sub2"
        .Caption = "Sub2Sub2"
        .Controls.Add(msoControlButton).Caption = "Sub3Sub2"
    End With
    .Controls.Add(msoControlButton).Caption = "Sub3Item2"
End Sub
```

25.11 Adding A New Menu To The Worksheet Menu

The following code is another example that adds a new custom menu to the worksheet menu, just before the Help menu, and place two commands on it. The OnAction property specifies the macro each command runs:

```
Public Sub AddCustomMenu()
    Dim barWS As CommandBar
    Dim mnuCustom As CommandBarControl
    Dim HelpIndex As Integer

    Set barWS = CommandBars("Worksheet Menu Bar")
    HelpIndex = barWS.Controls("Help").Index
    Set mnuCustom = barWS.Controls.Add(Type:=msoControlPopup, _
    Before:=HelpIndex)
    With mnuCustom
        .Caption = "&Custom"
        .Controls.Add(Type:=msoControlButton).
        .Caption = "&Show Data Form"
        .OnAction = "ShowDataForm"
    End With
    With .Controls.Add(Type:=msoControlButton)
        .Caption = "&Print Data List"
    End With
End Sub
```
25.12 Disable SaveAs Menu

This following will disable the Save As menu item on the worksheet menu:

```vba
Sub DisableSaveAsMenuItem()
    With CommandBars("Worksheet Menu Bar")
        .Controls("File").Controls("Save As...").Enabled = False
    End With
End Sub
```

Setting the Enabled property back to True turns the Save As menu item back on.

25.13 Resetting The Menus

The following will reset the worksheet menus:

```vba
MenuBars(xlWorksheet).Reset
```

Please note that this will remove all modifications made by any add-in.

25.14 Protecting Commandbars

To protect a commandbar from modification, use a statement like the following:

```vba
CommandBars("Worksheet Menu Bar").Protection = msoBarNoCustomize
```

To remove the protection, use:

```vba
CommandBars("Worksheet Menu Bar").Protection = msoBarNoProtection
```

The value of Protection can be be one of or a sum of the following:

- MsoBarNoProtection
- MsoBarNoCustomize
- msoBarNoResize
- msoBarNoMove
- msoBarNoChangeVisible
- msoBarNoChangeDock
- msoBarNoVerticalDock
- msoBarNoHorizontalDock.
25.15 How To Add A Menu Item Separator Bar

Set the **BeginGroup** property to **True** to add a separator bar before a new menu item or a button on a command bar.

The following illustrates how to add a separator bar above a menu item added to the Tools menu:

```vba
Dim menu_Item

'add a temporary menu item to the Tools menu
'it will disappear when Excel is closed
'but not when the file is closed unless the remove macro below is called.

With CommandBars("Worksheet Menu Bar").Controls("Tools")
    Set menu_Item = _
    .Controls.Add(Type:=msoControlButton, Temporary:=True)
End With

(assign name and macro to the menu item. Also, put a
'separator line above it by setting BeginGroup to True

With menu_Item
    .Caption = "This Says Hello"
    .OnAction = "SayHello"

'this puts the separator bar above the menu item
    .BeginGroup = True
End With
```

25.16 Determining Which Button Was Clicked On A Toolbar

The following illustrates how to determine which button was clicked on a toolbar. Create a custom toolbar called TestBar1, add three custom controls to it, and set their captions to One, Two, and Three. Assign this macro to each button:

```vba
Sub WhichButtonWasPressed()
    With CommandBars("TestBar1")

    'return the caption or name of the button that was clicked
    Select Case CommandBars.ActionControl.Caption

    'match the caption to a case statement
    Case "One"
        MsgBox "You pressed One"
    End Case

    End With
End Sub
```
Case "Two"
    MsgBox "You pressed Two"
Case "Three"
    MsgBox "You pressed Three"
End Select
End With
End Sub

Please note that the above Case statements are case sensitive unless you put Option Compare Text at the top of your module or convert all text to the same case.

Another way to determine which button is clicked is to use the Tag or Parameter property of the control:

MsgBox CommandBars.ActionControl.Parameter

25.17 CommandBars And Control Numbers

The following illustrates modifying a menu using the menu's control number instead of its name. Names are language specific.

Sub Add_Menu_Item()
    Dim ctlMenu As CommandBarControl
    Dim ctlMenu_Item As CommandBarControl

    'remove before adding!

    RemoveMenu_Item

    'the number 30007 is Tools menu

    Set ctlMenu = __
    Application.CommandBars(1).FindControl(, 30007)
    Set ctlMenu_Item = __
    ctlMenu.Controls.Add(Type:=msoControlButton)
    ctlMenu_Item.Caption = "QuickTable"
    ctlMenu_Item.OnAction = ThisWorkbook.Name & "!QuickTable"
End Sub

Sub RemoveMenu_Item()
    Dim ctlMenu As CommandBarControl

    On Error Resume Next
    Set ctlMenu = __
    Application.CommandBars(1).FindControl(, 30007)
    ctlMenu.Controls("QuickTable").Delete
End Sub
25.18 How To Add A Short Cut Menu

Syntax-wise, you’d access shortcut menus exactly the same way you add them to regular menus. The trick is knowing which one to reference, since there are 40 of them. The following article on MS Knowledge base is about 19 pages, and is a good reference on command bars in general. Page 12 contains a list of the built-in shortcut bar names.

"XL97: WE1183 "Customizing Menu Bars, Menus and Menu Items"

http://support.microsoft.com/support/kb/articles/q166/7/55.asp

25.19 TextBoxes On CommandBars

TextBoxes on CommandBars work just like normal textboxes. The easiest way to read the text is in the procedure that you have assigned to the textbox control place the following code:

```vba
Dim szText As String
szText = CommandBars.ActionControl.Text
```

25.20 Listing The Shortcut Menus

The following macro creates a list on the active sheet of all the shortcut menus and the menu items on each.

```vba
Sub List_Short_Cut_Command_Bars_And_Menus()
    Dim cell As Range
    Dim R As Integer
    Dim c As Integer
    Dim ctlBar As CommandBar
    R = 2
    Range("a1").Value = "Index #"
    Range("b1").Value = "Commandbar Name"
    Range("c1").Value = "menu item captions"
    For Each ctlBar In CommandBars
        If ctlBar.Type = msoBarTypePopup Then
            Cells(R, 1) = ctlBar.Index
            Cells(R, 2) = ctlBar.Name
            For c = 1 To ctlBar.Controls.Count
                Cells(R, c + 2) = ctlBar.Controls(c).Caption
            Next
            R = R + 1
        End If
    Next ctlBar
    Cells.EntireColumn.AutoFit
    MsgBox "All done"
End Sub
```

25.21 Menu Code Available On The Internet
If you head over to the Baarns web site, you can pick up their Developer Jumpstart file. In addition to many other code samples, this file contains a table driven CommandBar builder. All you have to do is fill out the table and include it and one extra code module in your project, and it will build whatever commandbars you specified in the table automatically. It builds menubars, toolbars and popup menus all from the same table.

You can find several examples of menu-making code at John Walkenbach's site,

http://www.j-walk.com/ss

Look in the Excel Developer Tips section. Also, check out the downloads section. Several of the files include code that adds a menu item to the Tools menu.

25.22 Internet Articles On How To Change The Menus

For more information on how to change the Excel menus, download and run the following files. When you run the files, they install a word document that you can then open and read.

Customizing Menu Bars, Menus, and Menu Items

How to Prevent Customization of Menus and Toolbars

25.23 Disabling Commandbar Customization

If you are using Excel 2002, it is possible to prevent a user from modifying the commandbars. The statement is::

    CommandBars.DisableCustomize = True

To enable customization use:

    CommandBars.DisableCustomize = False

If your application still needs to run under an earlier version, use the following approach:

    Dim cBars As Object
    Set cBars = Application.CommandBars
    If Val(Application.Version) >= 10 Then
        cBar.DisableCustomize = True
    End If
26. BUTTONS AND OTHER CONTROLS

26.1 Assigning A Macro To A Button

If you draw a button on a worksheet using the Forms toolbar, you can assign a macro to it by:

- First selecting the button by holding down the control key and clicking on the button
- Next, right click on the button and select the option "Assign Macro"
- Select a macro from the list
- Click on a cell in the worksheet to un-select the button

If you need to change the assigned macro or change the text in the future, do so by right clicking on the button or holding down the shift button and clicking on the button.

Another approach is to put a commandbar button on a worksheet:

- Display the Control toolbar by selecting View, Toolbars
- Click on the command button and draw a button on the sheet. This puts you in what is called "Design Mode" so that you can edit buttons and other objects you draw using the Control toolbar
- Double click on the button to display the worksheet code sheet. This also creates the following code:

```
Private Sub CommandButton1_Click()
End Sub
```

Please note the name of the Sub may be different depending on how many buttons you have on your worksheet.

- In the above code add the name of the macro to run when the button is clicked:

```
Private Sub CommandButton1_Click()
    name of macro to run
End Sub
```

- To change the text on the command button, click on the properties button on the Control toolbar and change the entry for the caption property.
Click on the first button on the Control toolbar to exit design mode.

To change the properties of the button in the future, display the Control toolbar and click on the first button to enter design mode. Then select the button and click on the properties button to change the button's property. To change the macro assigned to the button, click on the sheet's tab and select the View Code option. Edit the code on the button's click event macro.

26.2 Working With Command Buttons

You can place a command button on a worksheet by doing the following:

- Select View, Toolbars, and place a check beside the Visual Basic toolbar to display it
- Click on the Control toolbox button to display the control toolbox toolbar
- Click on the command button and draw a button on the sheet

When you double click on the new button while the design button on the Visual Basic toolbar is activated, code like the following is displayed:

```vba
Private Sub CommandButton2_Click()
End Sub
```

You can then place code in this procedure so that whenever the user clicks on the button, the code is run. **However, there are actions you must take for your code to run.**

The following will not work because the command button still has the focus and the spreadsheet does not.

```vba
Private Sub CommandButton2_Click()
    Dim MyRange As Range
    Set MyRange =Worksheets("Sheet1").Range("A1:B2")
    MyRange.Interior.ColorIndex = 35
End Sub
```

The following will work, since it puts the focus back into Excel, away from Visual Basic by activating the current selection.

```vba
Private Sub CommandButton2_Click()
    Dim MyRange As Range
    Set MyRange =Worksheets("Sheet1").Range("A1:B2")
    Selection.Activate
    MyRange.Interior.ColorIndex = 35
End Sub
```

The above approach works with any control placed on a work book. On command buttons, you can also solve the problem by doing the following steps instead of activating the current selection or selecting a range on a worksheet:
1. Click the Design Mode button on the Control Toolbox (Toolbar)... 

2. Right-Click the CommandButton1 and select Properties from the shortcut menu.

3. Locate the 'TakeFocusOnClick' property and change it to False.

26.3 Problems With Buttons And Controls

One of the default settings of buttons is that the TakeFocusOnClick property is set to True. However, if your button is to execute code that affects a worksheet, then you need to set this property to False. If you do not, then you will likely get the following error message:

"Unprotect method of Worksheet class failed"

To eliminate this error with buttons, change the TakeFocusOnClick property to False. Do this by

If the button is on a userform:

◆ selecting the button
◆ pressing F4 to display the properties
◆ change the TakeFocusOnClick property to False

If the button is on a worksheet:

◆ display the control toolbox toolbar
◆ click on the design mode button to enter design mode
◆ select the button
◆ click the properties button
◆ change the TakeFocusOnClick property to False

If the control is not a command button (for example an option button), include ActiveSheet.Activate or ActiveCell.Activate before trying to work with the cells.

26.4 Hiding Controls Placed On Worksheets
The following statements illustrate how to hide controls like commandbar buttons, edit boxes, and combo boxes that are placed on a worksheet:

```vba
ThisWorkbook.Worksheets("Sheet1")._OLEObjects("ComboBox1").Visible=False
```

If you are using the older style dropdown control from the Forms Toolbar, use

```vba
ThisWorkbook.Worksheets("Sheet1")._DropDowns("Drop Down 1").Visible=False
```

### 26.5 How To Remove Buttons From A Sheet

The following subroutines will delete all the buttons on a worksheet. One is for buttons created from the forms toolbar and. The other is for deleting command buttons

To delete buttons created from the forms toolbar:

```vba
Sub ZappthebuttonsExample1()
    Dim I As Integer, N As Integer

    'get the number of buttons on the sheet
    N = ActiveSheet.Buttons.Count

    'step through the buttons, starting with the highest index number
    'and delete the button
    For I = N To 1 Step -1
        ActiveSheet.Buttons(I).Delete
    Next
End Sub
```

To delete commandbar buttons

```vba
Sub ZappthebuttonsExample2()
    Dim I As Integer, N As Integer
    With ActiveSheet

        'get the number of OLEObjects on the sheet
        N = .OLEObjects.Count
        For I = N To 1 Step -1

            'check the type of object. If a command button, delete it
            If LCase(TypeName(.OLEObjects(I).Object)) = _
                "commandbutton" Then _
                .OLEObjects(I).Delete
        Next
    End With
End Sub
```
26.6 Hiding Or Showing Combo Boxes Via Code

If you have a combo edit, dropdown box on a worksheet, you can have your code either hide or display the box as needed. For example:

If the combo box was created using the control toolbar, then something like this would hide the box:

```vba
ThisWorkbook.Worksheets("Sheet1") _.OLEObjects("ComboBox1").Visible=False
```

If you are using the older style drop down control created from the Forms Toolbar, use

```vba
ThisWorkbook.Worksheets("Sheet1") _.DropDowns("Drop Down 1").Visible=False
```

26.7 Creating Combo Boxes With Code

The following example shows how to create a combo box with code and place it on a spreadsheet:

```vba
Sub Create_Combo_Box()
    Dim objOLE As OLEObject
    Dim rng As Range
    Set rng = ActiveCell

    'create the combo box and assign to an object variable at the same time
    'the box will be positioned and sized based on the range assigned to the
    'range variable rng
    Set objOLE = ActiveSheet.OLEObjects.Add( _
        ClassType:="Forms.ComboBox.1", _
        Left:=rng.Left, Top:=rng.Top, _
        Width:=rng.Width * 2, _
        Height:=rng.Height * 2)

    'assign the list file range to fill the box and a link cell for its output
    objOLE.ListFillRange = "A1:A10"
    objOLE.LinkedCell = rng.Offset(2, 0).Address
End Sub
```

26.8 Preventing Typing In A ComboBox
If you set the Style property of a combobox to 2 (or the predefined constant `fmStyleDropDownList`, which is exactly the same thing) then one can not type into the combobox.

### 26.9 How To Have A Worksheet ComboBox Drop Down

To have a ComboBox that is on a worksheet, not in a userform, drop down automatically when a user clicks on it, put the following code in the worksheet's code module:

```vba
Private Sub ComboBox1_GotFocus()
    ComboBox1.DropDown
End Sub
```

### 26.10 Self Modifying List Box Example

The following is the solution given to a user who needs a list box to look at a particular cell and if the contents of that cell changes, he needs the list box to populate with the values in a range based on the cell's value.

```vba
Private Sub Worksheet_Change(ByVal Target As Excel.Range)
    'Check and see if the cell changed was the one ranged named BrandName
    If UCase(Target.Name.Name) <> "BRANDNAME" Then Exit Sub
    With Sheets("Sheet1").Shapes("ListBox1").DrawingObject
      'change the list based on the cell's value
      Select Case UCase(Range("BrandName").Value)
        Case "BRANDA"
          .ListFillRange = "A1:A5"
        Case "BRANDB"
          .ListFillRange = "B1:B5"
      End Select
    End With
End Sub
```

In this example, the name of the list box is "ListBox1" and it is on a sheet called "Sheet1". The lookup cell is range-named "BrandName". The above macro changes the strings to upper case to insure a proper comparison

The code goes in the module for the worksheet which the list box is on. You'll see it in the Visual Basic Project Explorer under Microsoft Excel Objects in the VBAProject for your workbook, e.g. Sheet1.

The Worksheet_Change event is triggered for a Worksheet when the user changes a cell or cells on that Worksheet. Target is the range which has been changed.
27. POP-UP MENUS

27.1 Disabling The Cells Shortcut Menu

You can disable the entire worksheet cells shortcut menu like this:

```vba
ShortcutMenus(xlWorksheetCell).Enabled = False
```

To reset the shortcut menu just use this non-intuitive statement:

```vba
ShortcutMenus(xlWorksheetCell).Enabled = True
```

You can not disable a built-in menu item on the shortcut menu, but you can remove it:

```vba
ShortcutMenus(xlWorksheetCell).MenuItems _
("Clear Contents").Delete
```

27.2 Replacing The Cell Pop-Up Menu

You can replace the cell pop-up on a given sheet with your own pop-up menu. You would need to:

1) Create the `commandbar` you want to use as the popup (we'll assume its name is MyPopup)

2) Add the following `BeforeRightClickEvent` to the worksheet's code module:

```vba
Private Sub Worksheet_BeforeRightClick( _
    ByVal Target As Excel.Range, Cancel As Boolean)
    CommandBars("MyPopup").ShowPopup
    Cancel = True
End Sub
```

27.3 Disabling The Right Click Pop-Up Menu In A Workbook

To disable the cell shortcut menu for a workbook,

1. Go to the Visual Basic Editor (VBE), Alt-F11.
2. Locate the workbook's name in the project Explorer [ex: VBAProject(Book2.xls)] ...
3. (if necessary) Expand the project / (Double-Click VBAProject(Book2.xls)) so that the 'ThisWorkbook' object is visible.
4. Right-click 'ThisWorkbook' and click View Code on the shortcut menu. A code window should open to the right of the project explorer. At the Top of the code window you'll see two Drop-Down boxes.

5. Click the left dropdown (should say 'General') & Click Workbook. A workbook open procedure should appear. Ignore it.

1. Click the right Drop-Down (should say 'Open') & select the 'SheetBeforeRightClick' item. That procedure should show up in the code window. The following will appear

```vba
Private Sub Workbook_SheetBeforeRightClick(ByVal Sh As Object, ByVal Target As Excel.Range, Cancel As Boolean)
End Sub
```

7. Add the following line to that procedure...

```vba
Cancel = True
```

You can delete the workbook open procedure.

8. Close all the VBE windows & save the workbook.

From now on, whenever the user right-clicks on a sheet in that workbook, nothing will happen. This does not affect any other workbooks that might be open. If you want, you can even have the Workbook_SheetBeforeRightClick display a message box or dialog of your choice to the user.

### 27.4 Disabling The Tool List Pop-Up Menu

If you right click above the main menu bar, you are shown a popup menu with menubars you can enable or disable. This popup menu can be disabled if you are using Excel 97 SR1 or higher by the following:

```vba
Sub DisableToolbarMenu()
    CommandBars("Toolbar List").Enabled = False
End Sub
```

Unfortunately, you can not disable it in the pre SR1 release of Excel 97.

After you run this subroutine, the Toolbars command on the View menu is unavailable. Also, you cannot display a list of available toolbars by pointing to, and then right-clicking a toolbar.

The following subroutine enables the Toolbar List shortcut menu:

```vba
Sub EnableToolbarMenu()
    CommandBars("Toolbar List").Enabled = True
End Sub
```
27.5 Replacing The Cell Pop-Up Menu With A Custom Menu

You can replace the cell pop-up with a custom dialog. The cell pop-up is the one that appears when you right click on a cell in a worksheet. The first step is to construct a custom user form. Assuming that the name of the user form is "UserForm1", then put the following code in each worksheet where you want the custom dialog to appear.

```vba
Private Sub Worksheet_BeforeRightClick(ByVal Target As Excel.Range, Cancel As Boolean)
    'displays the custom user form
    UserForm1.Show

    'prevents the normal cell pop-up from appearing
    Cancel = True
End Sub
```

The easiest way to put the code in a worksheet is to double click on the sheet's name in the VBE project explorer.

27.6 How To Customize The Popup Menus

There are a number of pop-up menus that appear when you right click in Excel. For example, different pop-ups appear when you right click on cell or on a sheet tab. If you want to add need menu items or remove menu items from these menus, you can do so with code. The following illustrates how to do this for the cell pop-up menu.

```vba
Sub ModifyCellPopupMenu()
    Dim oMenu

    'set an object variable to refer to the pop-up menu
    Set oMenu = ShortcutMenus(xlWorksheetCell)

    'add a separator bar before adding menu items
    oMenu.MenuItems.Add Caption:="-

    'add a menu item
    oMenu.MenuItems.Add Caption:="Do Something", _
    OnAction:="macroName"
End Sub
```

The following illustrates how to remove a menu item from the cell pop-up menu.
Sub DeletePopupItem()
    Dim I As Integer, oMenu As Object

    'set an object variable to refer to the pop-up menu
    Set oMenu = ShortcutMenus(xlWorksheetCell)

    'work backward through the menus since deleting should be from bottom up
    For I = oMenu.MenuItems.Count To 1 Step -1

    'look for a matching caption. Please note the test is case sensitive
    'unless Option Compare Text is declared at the top or the strings
    'are converted to upper case using the Ucase() function
    If oMenu.MenuItems(I).Caption = "Do Something" Then
        oMenu.MenuItems(I).Delete
        Exit Sub
    End If
    Next
End Sub

The above routines reference a pop-up menu by means of a constant. For example,
ShortcutMenus(xlWorksheetCell) uses the constant xlWorksheetCell to identify the cell pop-
up menu. The following is a list of useful constants, which allow you to modify other pop-up menus

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xlWorksheetCell</td>
<td>Worksheet Cell</td>
</tr>
<tr>
<td>xlWorkbookTab</td>
<td>Workbook Tab</td>
</tr>
<tr>
<td>xlColumnHeader</td>
<td>Column</td>
</tr>
<tr>
<td>xlRowHeader</td>
<td>Row</td>
</tr>
<tr>
<td>xlAxis</td>
<td>Chart Axis</td>
</tr>
<tr>
<td>xlButtonDown</td>
<td>Button</td>
</tr>
<tr>
<td>xlChartSeries</td>
<td>Chart Series</td>
</tr>
<tr>
<td>xlChartTitles</td>
<td>Chart Titles</td>
</tr>
<tr>
<td>xlDesktop</td>
<td>Desktop</td>
</tr>
<tr>
<td>xlDialogSheet</td>
<td>Dialog Sheet</td>
</tr>
</tbody>
</table>

460
27.7 Creating and assigning a custom Pop-up Menu

the following creates a custom pop-up menu with two menu items:

```vba
Sub Create_Custom_PopUp()
    Dim myBar As Object

    'delete the pop-up if it exists
    On Error Resume Next
    CommandBars("custom_popup").Delete
    On Error GoTo 0

    'create a new commandbar and name custom_popup.
    'Make it a popup menu and temporary so it disappears when Excel is closed
    Set myBar = CommandBars.Add(Name:="custom_popup", _
        Position:=msoBarPopup, Temporary:=False)

    'add two menu items to the new commandbar
    With myBar
        .Controls.Add Type:=msoControlButton
        .Controls.Add Type:=msoControlButton
    End With

    'assign captions to the menu items and OnActions
```
With myBar
    .Controls(1).Caption = "Show Example"
    .Controls(2).Caption = "Help"
End With
End Sub

'please note that the following OnAction macros would have to be created

    .Controls(1).OnAction = "ShowExample"
    .Controls(2).OnAction = "DisplayHelp"

You can then put the following code in the **MouseDown** event of an object, such as a worksheet, button, edit box or a combo box. It causes the above pop-up menu to be displayed. Please note it does not go in the click event of the object.

'**test to see if right button pressed** on a worksheet (code in worksheet module)

    Private Sub Worksheet_BeforeRightClick( __
        ByVal Target As Range, Cancel As Boolean)
        CommandBars("custom_popup").ShowPopup
    End Sub

If you have a combo box on a worksheet, you can put the following in the worksheet's code module:

    Private Sub ComboBox1_MouseDown(ByVal Button As Integer,
        ByVal Shift As Integer, ByVal X As Single, ByVal Y As Single)
        'test to see if right button pressed- also msbuttonright constant
        If Button = 2 Then
            'if True, display the popup
            CommandBars("custom_popup").ShowPopup
        End If
    End Sub

27.8 Disabling The Worksheet Tab And Navigation Pop-Up Menus

The following statement disables the workbook navigation pop-up menu that appears if you right click on the navigation tabs at the bottom left of the workbook. Since the names is "workbook tabs" it appears that someone in Microsoft got confused.

    CommandBars("Workbook Tabs").Enabled = False

To disable the pop-up that appears when you right click on a sheet tab, use the following statement
CommandBars("Fly").Enabled = False
28. DEBUGGING AND HANDLING ERRORS

28.1 Debugging Tricks

To place a break point, which kicks you into the Visual Basic debugger when the line is reach, click on the line and then click on the button labeled "Break Point". You can toggle break points by clicking in the margin to the left of the line.

To set permanent breakpoints, insert Stop statements. It control when these statements are executed, set a constant at the top of your module and use an If statement. This allows you to keep break points in place even if you close the file:

```
Public Const bBreak As Boolean = True
```

and then in your code:

```
If bBreak Then Stop
```

'stand alone Stop statement:

```
Stop
```

'stop statement in an If statement

```
If I = 25 Then Stop
```

Some additional tricks:

♦ If you are stepping through your code, you can drag the current executing line (yellow arrow) to continue execution somewhere else entirely, including going back a few lines.

♦ You can also set bookmarks to allow you to jump to a location. If you are debugging a macro and are in debug mode, the current line to execute is highlighted with a yellow arrow. This arrow can be dragged to continue execution somewhere else entirely, either forward or backward.

♦ You can display a variable's value, while in break mode, by placing the mouse pointer over a variable name.

♦ While debugging you can click into the Excel sheets and change to other sheets to see changes that may have occurred on sheets that are not the active sheet. Please note that changing the active workbook or active sheet may result in your code not working on the correct sheet or cells.
To step through your code, you can press the F8 key or click on the step buttons on the toolbar. Pressing F5 or the resume button resumes the macro. Clicking on the reset button (the square button) halts the macro.

28.2 Break On Unhandled Errors

The Break in Class Module setting really means Break On Unhandled Errors in Class Modules. What this does is prevent the Error Handler in the procedure that called the class from handling the error. This makes debugging class module errors quite a bit easier.

However, it doesn't abnegate you from the responsibility of putting error handlers in your class. You have to treat procedures in class module exactly the way you would treat any other procedure. Give it an error handler. Communicate errors which have occurred in the class to the calling procedure via custom properties or return values from methods.

28.3 Error Trapping

The following illustrates how to trap errors in your code that would normally result in an error message appearing if there were no error handling.

```vba
Sub Error_Handling_1()
    Dim V

    'turn on error trapping
    On Error GoTo LabelA
    V = 1/0

    'turn off error trapping
    On Error GoTo 0

    'statements
    Exit Sub

LabelA:

    'statements that handle the error and exit the macro

End Sub
```

To trap errors and resume your procedure at a different location after the error has been trapped, do the following:

```vba
Sub Error_Handling_2()
    Dim V

    'turn on error trapping
```
On Error GoTo LabelA
V = 1/0

'turn off error trapping

On Error GoTo 0
LabelB:
On Error GoTo LabelC

'statements

Exit Sub

LabelA:
V = 0
Resume LabelB
LabelC:

'statements to handle an error

End Sub

If you want to have your code resume on the next line but take some action before doing so, then you can do so by using the Resume Next statement. Please note that the original error handler stays in effect. If a Resume statement is not used, then future errors in the subroutine will not be trapped. The following illustrates using a Resume Next statement

Sub Error_Handling_3()
Dim V
'turn on error trapping
On Error GoTo LabelA
V = 1/0
'turn off error trapping
On Error GoTo 0
LabelB:
'statements
Exit Sub

LabelA:
V = 0
Resume Next
End Sub

The following uses On Error Resume Next to ignore errors. This statement tells Visual Basic to ignore the error and execute the next statement.

Sub Error_Handling_4()
Dim V

'turn on error trapping to resume on next line if an error occurs

On Error Resume Next
V = 1/0
'turn off error handling

On Error GoTo 0
End Sub

Lastly, the following illustrates how NOT TO use error handling:

Sub BadErrorHandlingCode()
    Dim v
    On Error GoTo A
    v = 1/0
A:
    On Error GoTo B
    v = 1/0
B:
End Sub

Since the above does not have a Resume statement, the second error handler will not trap the error.

28.4 Avoiding Excel/VBA Crashes

One secret to preventing crashes is to fully declare every variable, object, etc. and to fully qualify all object references. The default Variant data type has some (hidden?) characteristics which cannot be coerced to other data types effectively in every instance. And, trying to do so can cause crashes.

28.5 Modifying Code And Repeating Steps While Debugging

You can do a fair amount of editing and modification of your code while stepping through your code. Because we have had a few instances of Excel crashing while modifying the code when stepping throughout the statements, we strongly recommend that you save your code before you step through and modify the code at the same time.

Another feature of is the ability to step back up or jump over sections of code by dragging the yellow locator arrow to a new line. This arrow is found in the left hand border area and highlights the next line of code to be executed. This allows you to step through lines of code until you are satisfied with the results or understand how to modify them to meet your needs.

28.6 Error Handling Different In Excel 97/2000 For Functions

In certain situations, error handling in Excel 8 VBA is different than Excel 5 and 7. In Excel 5 and Excel 7, if a function performed a calculation involving an Excel error value then it would halt execution of the function only, and return that error value.
Under the same conditions in Excel 97/2000, VBA is halted (not just the function, but calling programs as well!) and [#VALUE!] is returned regardless of the actual error encountered.

The net result is that while you could ignore error handling in previous versions and be confident that VBA would do the right thing, you must explicitly trap and handle all errors in Excel 8 or your application may not run at all!

28.7 What To Do If You Get Strange Problems With Perfectly Good Code

Sooner or later you will find that perfectly good code will stop working or cause Excel to crash, either with an error box or the dreaded “This application has performed an illegal operation”. Or the code may start running extremely slow.

One cause of this is excessive macro runs which result in errors which in turn cause you to use the debugger. We're talking a lot of runs and crashes in one session: probably well in excess of 100, although it can happen with fewer runs. Typically what happens is that an error box pops up and a perfectly good line of code is highlighted, typically one that has worked many a time. After you've verified that a crash should not have happened, try the following solution: Exit Excel, turn your computer off, and restart your machine and Excel. If the error does not repeat itself, then the problem was caused by excessive runs and crashes.

Another problem can be caused by the size of your modules. The maximum file size allowed for a VBA module is about 64k. Anything larger than that won't cause immediate failure, but you'll start getting all kinds of strange problems that won't go away until you break up the large modules into several smaller ones. You can find out how big your modules are by exporting them to text files and then looking at the text file size in Explorer.

Other problems can happen because Excel does not fully remove old code when you delete lines in a module. Thus you need to “clean” your code periodically in Excel projects or there is a good chance that your file will eventually become corrupted. Cleaning involves exporting all the modules and userforms to text files, deleting the old ones, then re-importing everything back into your project. In doing this you will often see a reduction in file size in the range of 30% to 40% if you've never done it before. Do this before your workbook becomes corrupt.

Rob Bovey's code cleaning utility is available at http://www.appspoor.com/Utilities/CodeCleaner.htm, will automate the cleaning process that I described above. There are separate versions for different releases of Excel, so be sure to get the right ones. And read the instructions carefully before using them.

28.8 Observing Excel While Debugging In Visual Basic

The trick to observing your changes in Excel while debugging is dependent on which version of Excel you are running. You will need to size the VB editor window to allow you to see parts of the worksheet. You can also leave the debug window active and checkout other sheets and
workbooks other than the active ones. Be sure to return to the original active sheet before continuing your debugging.

28.9 Detecting Error Values In Cells

The function IsError(any value) will return True if the value is an error value such as division by zero, and False if not. For example,

\[
\text{If IsError(ActiveCell.Value) Then} \\
\quad \text{MsgBox "The cell at " & ActiveCell.Address & "} \\
\quad \quad \text{" has an error value.} \\
\text{End If}
\]

28.10 Out Of Memory Error Solutions

Often, "out of memory" problems are caused by a lack of video memory. Try any or all of the following:

1) Make sure the Zoom setting is 100%

2) Reduce the number of fonts

3) Reduce the number of formats (colors, borders, patterns, etc.)

4) Reduce the number of graphs, images, controls, maps, OLE objects,

etc.

5) Delete (not just clear) all unused rows and columns. You should select the rows and columns by clicking the row or column headers (the "1" and "A"), not by selecting the cells. The choose Delete from the Edit menu. Pressing the Delete key does not delete the Rows. It does a "ClearContents".

Rob Bovey has developed a free Code Cleaner application which exports all your code modules, userforms and so forth, deletes them and then imports them back in. Apparently excel VBA accumulates overhead and excess mass overtime which can cause out of memory errors. An indication that you need to run this utility typically occurs when perfectly good code stops working. The code cleaner is available at http://www.appsfpro.com/Utilities/CodeCleaner.htm

Another cause of memory problems is the page setup feature of Excel. Frequent page setup code will cause problems. This is even more of a problem if you are using a HP printer.

28.11 Excel Crashes When Using A Range

If you have the problem with Excel crashing when you are setting a cell's value by using of the Range function or a Range variable. then you might want to take a look at the following Microsoft Knowledge Base KB article.
Microsoft recommends that you upgrade to the latest version of Excel 97. They also recommend that when setting a value, that you use the *Value* property when doing so:

```
Range("A1").Value = 9
```

### 28.12 Stack Overflow / Out Of Memory Problems

If you display a dialog or userform while another dialog is still being shown, then you run the risk of a stack overflow or an out of memory error. For example, if you assign a button on a dialog to a macro which displays another dialog, this situation can occur. Stack overflow can also occur if you try to print or print preview a worksheet with a dialog displayed.

The following illustrates one way to solve this stack overflow problem:

Declare a public variable at the top of a module (but not in a userform module)

```
Public buttonNumber As Integer
```

Assign a macro to each of the buttons on your dialog, and have the macros set the buttonNumber variable to indicate which button was chosen.

You would assign code like the following to the click events of the buttons in the userform's code module (double click a button to access the code module)

```
Private Sub CommandButton1_Click()
    UserForm1.Hide
    buttonNumber = 1
End Sub

Private Sub CommandButton2_Click()
    UserForm1.Hide
    buttonNumber = 2
End Sub
```

Then, you should use code like the following:

```
Sub ExcelExample()
    UserForm1.Show
    If buttonNumber = 1 Then

      'code to show dialog or to print

    ElseIf buttonNumber = 2 Then

      'code to show a different dialog or to print
```
End If
End Sub

If the user needs to do multiple prints from the dialog, then set it up with a loop.

Sub ExcelExampleWithLoop()
    Do
        UserForm1.Show
        If buttonNumber = 1 Then
            'code to show dialog or to print
        ElseIf buttonNumber = 2 Then
            'code to show a different dialog or to print
        End If
    Loop
End Sub

28.13 Keeping An Error Handling In Effect After An Error Occurs

An error handler must end with a Resume statement for that error handler to remain in effect:

Sub MySub()
    Dim N As Integer, I As Integer
    On Error GoTo ErrHandler
    Do
        'count the loops so that the Loop Until will halt when 10 loops are done
        I = I + 1
        'create an intentional error
        N = 1 / 0
    DoNext:
        Loop Until I = 10
    Exit Sub

    ErrHandler:
        'Error handling code
        'return to the loop, resetting the error handler
        Resume DoNext
End Sub
If you replace the Resume DoNext with GoTo DoNext, the above will crash the second time through, as the error handler has not been reset.

**28.14 Excel Crashes When A UserForm Is Displayed**

Some users have experienced problems with Excel crashing with the dreaded “this program has performed an illegal operation” when a user form is displayed. Typically, this is a form that has gone through a re-sizing. There are two possible cures (neither guaranteed)

Delete the Form and re-create it

OR

1) Save (export) the form

2) Delete the form

3) Retrieve the form

**28.15 Error Handling And Getting the Error Line**

If you number your lines, you can get the line of code with the error variable "erl". For example:

```vba
Sub Example()
  a=5
  On Error GoTo ErrorHdl
  b=6
  MsgBox 5/0 '<==error here
  Exit Sub
ErrorHdl:
  MsgBox erl '<==this will show a 4.
End Sub
```

Note that the #s are NOT labels (no ":"), but line #s. You can number just the lines which MIGHT cause an error.

**28.16 ErrObject**

The easiest way to access the ErrObject is through the global Err() method. It was set up this way for backwards computability with previous versions of VBA. The Err() method returns an ErrObject, so all you have to do to use it is to use statements like:

```vba
Err.Description
Err.LastDLEError
```
29. DIALOGSHEETS

29.1 How To Create And Display Dialogsheets

In Excel 97/2000, dialogsheets have been replaced with userforms. However, you can still create dialogsheets in Excel 97/2000. The main advantage of a dialog sheet is using a editbox to return a range from any worksheet in any workbook.

From a worksheet, right click on the worksheet tab and select Insert and then MS Excel 5 Dialog.

To display a dialog created on a dialogsheet, use statements like the following:

```
Dim bResponse As Boolean
bResponse = ThisWorkbook.DialogSheets("Dialog1").Show
If bResponse = False Then Exit Sub
```

or

```
If Not ThisWorkbook.DialogSheets("Dialog1").Show Then _
    Exit Sub
```

Both examples use ThisWorkbook to qualify the dialogsheet. If you do not qualify the dialogsheet, and the active workbook is not the workbook containing the dialogsheet, your code will crash.

29.2 Selecting A Range Using An Excel 5/7 Dialog Sheet

The first step in this process is to create a new dialog sheet. If you are using Excel 5/7 you can do this by selecting Insert, Macro, Dialog. A new dialog sheet will appear with an OK and Cancel button. To create a dialog sheet In Excel 97/2000, right click on a worksheet tab in Excel and select Insert and then pick the Excel 5.0 Dialog option. Name the dialog sheet "Range Dlg" so that it works with the code below.

When the dialog sheet appears, the Forms toolbar should also appear. If it does not, then select View, Toolbars and activate the Forms toolbar.

On the Forms toolbar, click on the edit box button and draw an edit box on the dialog box. Next, with the edit box selected, click on the control properties button and change the edit validation option to reference. This allows you to pick a range on a worksheet.

To set the tab order so that the edit box is the active control when the dialog is displayed, right click in a blank area of the dialog sheet. This will display a pop-up menu with an option named "Tab Order". Select this option and move the edit box control to the top of the tab order.

Now put the following code on a module.
Sub Get_A_Range()
    Dim rangeSelected As Range
    With ThisWorkbook.DialogSheets("Range Dlg")

        'clear the edit box

        .EditBoxes(1).Text = ""

        'display the dialog and exit if cancel selected

        If Not .Show Then Exit Sub

        'Exit if no range supplied

        If .EditBoxes(1).Text = "" Then
            MsgBox "No range selected"
            Exit Sub
        End If

        'assign the range selected to a range variable

        Set rangeSelected = Range(.EditBoxes(1).Text)
    End With

    'code that uses the selected range

End Sub

The above code will display the dialog sheet, exit if cancel is selected or assign the selected range to a variable if one is selected. Since there is only one edit box on this particular dialog, the use of EditBoxes(1) is the easy way to refer to the edit box. It could also be referred to by its name and identified using an object variable. The following illustrates this approach along with several other techniques:

Sub Get_A_Range()
    Dim rangeSelected As Range
    Dim dlg As DialogSheet
    Dim eBox As EditBox
    Set dlg = ThisWorkbook.DialogSheets("Range Dlg")
    Set eBox = dlg.EditBoxes("edit box 1")

        'clear the edit box

        eBox.Text = ""

        'display the dialog and exit if cancel selected

        If Not dlg.Show Then Exit Sub
        If .EditBoxes(1).Text = "" Then
            MsgBox "No range selected"
            Exit Sub
        End If

End Sub
'assign the range selected to a range variable

    Set rangeSelected = Range(eBox.Text)

'code that uses the selected range

End Sub

29.3 Changing The Name Of Your Dialogsheet Objects

You can put text labels, edit boxes, list boxes, and many other objects on a dialog sheet. These objects are assigned names like "edit box 1", edit box 2", "list box 1", and so forth. If you want to assign these objects names that are more descriptive and make it easier for you to identify them in your code do the following:

♠ Click on the object you wish to rename to select it

♠ Then click on the Name Box (upper left corner of the screen) and type the new name. PRESS ENTER. If you don't press Enter, the name won't stick.

You can now refer to the object by its new name in your code. For example, if you renamed "edit box 1" to "Last Name", then the following code (located in the same workbook as the dialog sheet) would refer to this box, and assign its value to a variable named "lastName"

    lastName = ThisWorkbook.DialogSheets("My Dialog") _
        .EditBoxes("Last Name").Text

By using ThisWorkbook to qualify the dialog sheet, you insure that Excel knows to look in the workbook containing the code for the dialog sheet, as opposed to looking into the active workbook, which may not be the same workbook. This is very important when you are creating add-ins.

29.4 Setting The Tab Order In A DialogSheet

The tab order of a dialog sheet is the order in which one moves from object to object when the dialog or userform is displayed. Also, in Excel 5/7 it is the order in which controls or object of the same type (such as text boxes, buttons, etc.) are referenced by their index numbers. If you change the tab order, the names of the objects no longer indicate the index or tab order.

To change the tab order:

♠ Make sure no controls are selected.

♠ Right-click in the dialog, but not on a control.

♠ From the shortcut menu, choose Tab Order.
Select the name of a control you want to reposition in the tab order.

Choose Move Up or Move Down until the control name is in the appropriate position in the tab order.

29.5 Displaying Dialogsheets

To display a dialogsheet, use the Show method. If a cancel button is selected, then the Show method returns False. If an OK button or a button whose dismiss property is set on is selected, the Show method returns True. To check the setting of a button's properties, select the button and then click on the control properties button of the Visual Basic toolbar.

The following illustrate statements that display dialogs:

If the dialog is in the active workbook, this will work

**DialogSheets**("Dialog1").**Show**

If the active workbook is not the one containing the dialog sheet, then use a statement like this:

**ThisWorkbook**.**DialogSheets**("Dialog1").**Show**

Use of the **ThisWorkbook** qualifier is highly recommended

To determine if the Show method returns True or False, you need to use statements like the following:

```vba
Dim bResponse As Boolean
bResponse = **ThisWorkbook**.**DialogSheets**("Dialog1").**Show**
If bResponse Then
'actions to take if cancel button not selected
End If
```

The following is a simpler construction that stops the code if a cancel button is selected:

```vba
If Not **ThisWorkbook**.**DialogSheets**("Dialog1").**Show** Then End
```

or

```vba
If Not **ThisWorkbook**.**DialogSheets**("Dialog1").**Show** Then Exit Sub
```

The difference in the above two examples is the use of **End** versus **Exit Sub**. The **End** statement halts all macro activity and resets any global variables. The **Exit Sub** statement stops the current macro, but allows any calling routine to continue.

If the macro is a subroutine in another routine, then you should set a Public variable or a shared variable so that the calling routine can determine what occurred in the sub-routine.
Public variable approach:

'at the top of a module declare the following variable

Public bResponse As Boolean

Sub Main_Routine()

'call routine that displays the dialog

Display_Dialog
If Not bResponse Then Exit Sub

'code for whatever....

End Sub

Sub Display_Dialog()
  If Not ThisWorkbook.DialogSheets("Dialog1").Show Then
    bResponse = False
    Exit Sub
  End If
  bResponse = True

'code for whatever.....

End Sub

Shared variable approach

Sub Main_Routine()
  Dim bResponse As Boolean

'call routine that displays the dialog

Display_Dialog bResponse
If Not bResponse Then Exit Sub

'code for whatever....

End Sub

Sub Display_Dialog(bResponse As Boolean)
  If Not ThisWorkbook.DialogSheets("Dialog1").Show Then
    bResponse = False
    Exit Sub
  End If
  bResponse = True

'code for whatever.....

End Sub
In the second example, if a variable is used to pass a value to a called routine, then the called routine can change the value stored in the variable. This allows the called routine to pass information back to the calling routine without using Public variables or module level variables. In the above, the same name was used for the variable in both routines. This was a matter of convenience. Different names could have been used.
30. CONTROLLING USER INTERRUPTIONS

30.1 Capturing When Esc Or Ctrl-Break Are Pressed

If you use the following statement:

```
Application.EnableCancelKey = xlDisabled
```

The Esc key is completely disabled, and your macros will ignore the Esc key.

If you use:

```
Application.EnableCancelKey = xlInterrupt;
```

This allows the Esc key to stop a running macro (the default state)

If you use

```
Application.EnableCancelKey = xlErrorHandler;
```

The interrupt is sent to the running procedure as an error, trappable by an error handling statement set with an `On Error GoTo` statement. The trappable error code number is 18.

If the `Application` property `EnableCancelKey` is set to `xlErrorHandler`, then control is transferred to the current error handling routine that is set by an `On Error GoTo` statement. The following illustrates this:

```vba
Sub CaptureKeyBoardInterruptExample()
    Dim X

    'Set the error handler
    On Error GoTo ReActToEvent

    'set the EnableCancelKey property
    Application.EnableCancelKey = xlErrorHandler

    'run a loop that won't stop until either esc or ctrl-break is pressed.
    While 1 = 1
        X = X
    Wend

    Exit Sub
```

'control comes to this routine when either esc or ctrl-break is pressed

ReActToEvent:
    MsgBox "Break Key Hit"

'turn back on the interrupt property

    Application.EnableCancelKey = xlInterrupt
End Sub

You can set the EnableCancelKey property to xlErrorHandler and it stays in effect for all subroutines, with control being transferred to whatever label that On Error GoTo is set to for error handling.

Setting the EnableCancelKey property back to xlInterrupt allows the Esc key or the Ctrl-Break key combination to interrupt your code and display a debug dialog box.

If you want your code to handle the user pressing the ESC key while the code is running, then put the following code in your routines

    Application.EnableCancelKey = xlErrorHandler
    On Error GoTo HandleError

'your code

Exit Sub
.handleError:

'code to run if ESC pressed

End Sub

Several important points about the above

♦ HandleError can be any label name you want (other than words that Visual Basic recognizes).

♦ The last error handler set is the one used to handle the ESC key - if it is in the routine running when ESC is pressed, or if it is in a higher routine that calls the running routine

♦ Setting On Error GoTo 0 in a called routine does not clear the error handler that handles the ESC key - instead, it goes to the last set error handler from a higher routine

♦ Setting On Error GoTo 0 in the same routine that sets EnableCancelKey will cause the ESC key not to be handled.

Because of the various combinations you can run into with error trapping, you should test your handling of the ESC key to insure that it does handle the different situations.
Lastly, if your error handler ends with a *Resume* statement, the code will continue and not stop. You should use an *Exit Sub* or an *End* statement to stop activity.

### 30.2 Keeping Your Code From Being Stopped By The Esc Or Ctrl-Break Keys

If you set the *Application* property *EnableCancelKey* to *xlDisabled* this prevents the Esc key or the Ctrl-Break key combination from interrupting your code. The following illustrates such a statement.

```
Application.EnableCancelKey = xlDisabled
```

You should use with caution as an endless loop can not be stopped in this situation without shutting down Excel. If you have the Visual Basic editor open, you may be able stop the looping by clicking on the square reset button.

Setting the *EnableCancelKey* property back to *xlInterrupt* allows the Esc key or the Ctrl-Break key combination to interrupt your code and display a debug dialog box.

### 30.3 Determining Which Key Was Pressed

The following illustrates how to capture which key was pressed by a user. The *Sub* routine *Run_Until_Esc_Pressed* will run until you press the ESC key.

```
Type KeyboardBytes
    kbb(0 To 255) As Byte
End Type

Declare Function GetKeyboardState Lib "User32.DLL" _
    (kbArray As KeyboardBytes) As Long

Sub Run_Until_Esc_Pressed ()
    Dim kbArray As KeyboardBytes
    Do
        DoEvents
        GetKeyboardState kbArray
        If kbArray.kbb(27) And 128 Then ESCPressed
    End If

    'Wait for Esc

    Loop Until kbArray.kbb(17) And 128
End Sub

Sub ESCPressed()
    MsgBox "You pressed ESC"
End Sub
```
30.4 Traping the Key Pressed Event

In userforms, there is a keydown event that one can monitor for comboboxes, listboxes, refedit boxes and textboxes, and the userform itself. To have the VB editor create the initial event code, first double click on the object. Then in the upper right dropdown which shows the various events, select the keydown event. If you do this for a textbox, the following code appears:

```vbnet
Private Sub TextBox1_KeyDown(ByVal KeyCode As MSForms.ReturnInteger, ByVal Shift As Integer)
End Sub
```

In order to prevent the keystroke from being sent or used, you need to reset the KeyCode to zero:

```vbnet
Private Sub TextBox1_KeyDown(ByVal KeyCode As MSForms.ReturnInteger, ByVal Shift As Integer)
    If KeyCode = 88 Then
        KeyCode = 0
        MsgBox "You typed an x. That is not an allowed entry"
    End If
End Sub
```

By setting KeyCode to zero, the X is not entered into the textbox.
31. EVENT HANDLING

31.1 Auto_Open And Workbook_Open Macros

If you put a macro named Auto_Open() in a regular module, it will run whenever the workbook is manually opened (unless the user holds down the shift key while opening the file). If you put a macro named Workbook_Open() in the workbook code module that macro which will run upon opening the workbook. If both are present, both will run. The Workbook_Open macro will run first. You can access the workbook code module by displaying the Visual Basic project explorer, and double clicking on the ThisWorkbook object.

If code from another workbook opens a file containing Auto_Open() or Workbook_Open() macros, the Auto_Open macro() will not run, but the Workbook_Open() macro will run. To have the Auto_Open procedure run when the workbook is opened by Visual Basic code, the code opening the workbook can run the Auto_Open() procedure using this statement:

```vba
ActiveWorkbook.RunAutoMacros xlAutoOpen
```

Another approach is:

```vba
Dim myWB As Workbook

' this opens the file and sets an object variable to the opened file
Set myWB = Workbooks.Open(FileName:="Whatever.xls")

' this runs the Auto_Open macro if there is one.
' Note that the above object variable is used in this statement
myWB.RunAutoMacros xlAutoOpen
```

If the workbook does not have an Auto_Open macro, then this statement is ignored.

The following illustrate Auto_open and Auto_Close macros

```vba
Sub Auto_Open()

' code to run when the file is opened

End Sub
```

31.2 Preventing An Auto_Open or Workbook_Open Macro From Running
To prevent an Auto_Open() macro from running when you open a workbook, hold down the shift key, or open the file using a macro. This also works if you have a Workbook_Open() macro in the workbook code sheet.

### 31.3 Having A Dialog Appear When A Workbook Is First Opened

The following will display UserForm1 when the workbook is opened:

```vba
Sub Auto_Open()
    UserForm1.Show
End Sub
```

### 31.4 Running A Macro Whenever A Workbook Is Closed

To run a macro or a set of Visual Basic commands whenever a workbook is closed, use a procedure named Auto_Close:

```vba
Sub Auto_Close()
    'code to run or procedure to run when workbook closes
End Sub
```

Please note that if the workbook is closed by another workbook, the Auto_Close procedure will not run. It is triggered only by manually closing the workbook. To have the Auto_Close() procedure run in this situation, the code closing the workbook must run the Auto_Close() procedure using an `Application.Run` statement:

```vba
Application.Run "'My workbook.xls'!Auto_Close"
```

If you have a macro named Workbook_BeforeClose() in the ThisWorkbook code module it also runs when the workbook is closed:

```vba
Private Sub Workbook_BeforeClose(Cancel As Boolean)
    'code to run or procedure to run when workbook closes
End Sub
```

### 31.5 Order Of Close Events

The order of events on closing in Excel is
Workbook_BeforeClose
Auto_Close
Workbook_BeforeSave

When you use ThisWorkbook.Save, the Workbook_BeforeSave event will be called, but it will be before the Auto_Close procedure runs, rather than after.

31.6 Intercepting The Excel and Workbook Close Events

In the workbook's Auto_Close procedure you can check the ThisWorkbook.Saved property. If it's True you can confirm the closing and halt it if the user chooses to cancel. If it's False then you should ask the user to save changes or cancel.

If the user says yes to save changes then you should save and let Excel close. Excel will not again ask the user to save.

If the user says no then you should set ThisWorkbook.Saved to True so Excel will not prompt the user.

If he cancels you should run this macro which halts the closing of both the workbook and Excel:

The following statement will halt the closing of Excel:

   Application.ExecuteExcel4Macro "HALT(True)"

31.7 Disabling Events From Running

If you set the Excel EnableEvents property to False with a statement like the following, this will prevent Excel event macros from running. To allow event macros to run, you will need to set the EnableEvents properly back to True.

   Application.EnableEvents = False

   'code that might trigger the event

   Application.EnableEvents = True

31.8 Running A Macro Every Minute

If you want a macro to run every minute, then you need to set an OnTime event. The following runs a procedure, and then the last statement sets an OnTime event that runs the code again in a minute:

   Sub MyCode

   'code to run
The value 1/1440 is equal to exactly one minute as there are 1440 minutes in a day.

31.9 OnTime method - how to handle fractions of seconds

There are 86,400 seconds in a day. This is how to specify a half a second.

Application.OnTime Now + 1/86400 * 0.5, "OtherSub"

31.10 How To Make A Macro Run Every Two Minutes

In general, the way to get code to run every 2 minutes (or any set interval) is:

'at the top of a module put the following declaration

Dim dNext As Date

Sub StartDoingIt()
    DoItAgain
End Sub

Sub DoItAgain()

'calculate when code should run again

dNext = Now + TimeValue("00:02:00")

'set on time event to trigger running

Application.OnTime dNext, "DoItAgain"

'whatever you want to do

DoIt
End Sub

Sub DoIt
    MsgBox "Hello there, time to work"
End Sub

Sub StopDoingIt()
    On Error Resume Next

'this turns off the on time event

Application.OnTime dNext, "DoItAgain", schedule:=False
End Sub
'use this to turn off the Application on time event otherwise workbook will reopen

Sub Auto_Close()
    StopDoingIt
End Sub

You can also pass arguments in the OnTime statement:

'if arg1 and arg2 are values 1 and 2

Application.OnTime = "'mysub 1, 2'"
or

'if arg1 and arg2 are literal strings

Application.OnTime = "'mysub ""arg1"", ""arg2"'''

31.11 How To Cancel An OnTime Macro

If you have set Application.OnTime to run a macro at a certain time, you can cancel it - if you have stored the time use to set OnTime. It is best to store the time value in a global variable that you can later use. However, if you are going to edit your code, or use an End statement to stop code, this will reset the global variable. In this case, you should first round the variable to say 4 decimals, set the OnTime macro, and store that value in a worksheet cell for later user.

'This stores the time in a variable and then sets OnTime to run in 1 minute

Timetorun=Now + TimeValue("00:01:00")
Application.OnTime Timetorun, "OnTimeMacro"

'this cancels the above OnTime event

Application.OnTime earliesttime:=Timetorun, _
procedure:="OnTimeMacro", schedule:=False

31.12 Detecting When A Cell Is Changed

You can code the Worksheet_Change event to determine if a cell has been changed. It has the following appearance and is located in the worksheet's code module:

Private Sub Worksheet_Change(ByVal Target As Range)
In that sub, you can operate on the changed range (Target) as you see fit, including checking to see if it falls in a certain area of the workbook, what its value is compared to some other values in the workbook, etc.

In OnEntry, you can use Application.Caller to return the cell which triggered the event.
31.13 Macro Execution Linked To Cell Entry

You can use the Worksheet_Change event procedure to check a cell's entry and take action based on the cell’s value or address:

```vbs
Private Sub Worksheet_Change(ByVal Target As Excel.Range)
    If Target.Address = "$B$12" Then
        If Target.Value = 125 Then
            ' your code
        End If
    End If
End Sub
```

31.14 How To Run A Macro When The User Changes The Selection

The SelectionChange event of a worksheet captures any change in selection. For example, the following code in a worksheet's code module will display a message of the range selected.

```vbs
Private Sub Worksheet_SelectionChange( ByVal Target As Excel.Range)
    MsgBox "You selected " & Target.Address
End Sub
```

31.15 How To Run A Macro When A Sheet Is Activated

If you want a unique macro for a specific sheet (rather than one macro that launches whenever any sheet is activated), select the sheet tab and right click on it. Select view code. In the left box, select worksheet, in the right box, select Activate. Put your code in the skeleton macro created:

```vbs
Private Sub Worksheet_SelectionChange( ByVal Target As Excel.Range)
    'code goes here
End Sub
```

If you want the same code to run whenever any sheet is activated, you can go into the VBE, select ThisWorkbook in the project window, right click, select view code, left dropdown => workbook, right dropdown => SheetActivate

```vbs
Private Sub Workbook_SheetActivate(ByVal Sh As Object)
    'code goes here
End Sub
```

You can also use the following approach:
Sub Auto_Open
    ThisWorkbook.Worksheets("Sheet1").OnSheetActivate = _
        "SheetActivateMacro"
    ThisWorkbook.Worksheets("Sheet1").OnSheetDeactivate = _
        "SheetDeactivateMacro"
    'reference for additional sheets as required
End Sub

Sub SheetActivateMacro
    'code goes here
End Sub

Sub SheetDeactivateMacro
    'code goes here
End Sub

31.16 Excel Events That Are Triggered When A Cell Changes

Try this two events in two different sheets to see the difference. Please note this code goes in the worksheet code module. Right click on the worksheet name and select view code to access the module.

Private Sub Worksheet_Change(ByVal Target As Excel.Range)
    If target.Address="$A$1" Then _
        MsgBox ("The range A1 is selected")
End Sub

Private Sub Worksheet_SelectionChange( _
    ByVal Target As Excel.Range)
    If target.Address="$A$1" Then _
        MsgBox ("The range A1 is updated/calculated")
End Sub

31.17 Using The Worksheet Change Event

The Worksheet_Change event allows you to take an action based on a user's entry. For example if you needed to move user input in cell A1 into new cells to retain each input, the Worksheet_Change subroutine could do this for you after the user types in an entry. For example: Five separate inputs to A1 might be .5, .76, .8, .9. Code in the Worksheet_Change subroutine would automatically move the entries to cells C1, C2, C3, C4, C5.
To create a Worksheet_Change subroutine for a worksheet, right click on the sheet's tab and select view code. Visual Basic automatically inserts the following into a blank sheet:

```vbscript
Private Sub Worksheet_SelectionChange(ByVal Target As Excel.Range)
End Sub
```

However, this is not the one you want. Instead, in the right hand drop down, select Change. That will add the following code:

```vbscript
Private Sub Worksheet_Change(ByVal Target As Excel.Range)
End Sub
```

Modify the code to the following:

```vbscript
Private Sub Worksheet_Change(ByVal Target As Excel.Range)

'disable event handlers so that this routine doesn't cause itself
' to be called, and called, and called....
Application.EnableEvents = False

'check to see if the cell being modified is cell A1
If Target.Address(False, False) = "A1" Then

'check first two cells for an empty cell
If IsEmpty(Cells(1, 3)) Then
    Cells(1, 3).Value = Target.Value
ElseIf IsEmpty(Cells(2, 3)) Then
    Cells(2, 3).Value = Target.Value
End If
Else
    Cells(1, 3).End(xlDown).Offset(1, 0).Value = _
    Target.Value
End If
Target.ClearContents

'turn event monitoring back on
Application.EnableEvents = True
End Sub
```

31.18 Validating User Entries Using OnEntry

An OnEntry macro runs each time you press enter. To set an OnEntry macro that works on all worksheets, use the following statement:
Application.OnEntry = "MyMacroName"

To set an OnEntry macro that works only on a specific worksheet, use a statement like the following

Worksheets("Sheet1").OnEntry = "MyMacroName"

Whenever the enter key is hit, Excel runs the MyMacroName macro, which then validates the entry.

To turn it off, use the following statement:

Application.OnEntry = ""

Note that an OnEntry works until it is reset, Excel is closed, or the macro can not be found. In the last situation, you get error messages each time you press enter.

An OnEntry macro can be used to validate a user's entry, as illustrated by the following code:

Sub Auto_Open()

'if you put this statement in the Auto_Open of the workbook,
'OnEntry will automatically be active when the workbook is opened

Sheets("My Data").OnEntry = "CheckIt"
Sheets("System Data").OnEntry = "CheckIt"
End Sub

Sub CheckIt()

' set On Error in case the cell modified is not in the range to be checked

OnError Resume Next
Dim x
Set x = Intersect(Range(Application.Caller.Address), _
    Range("A1:E3"))

' substitute the validation range for A1:E3 in the above statement

If TypeName(x) = "Range" Then
    MsgBox "do your validation here"
End If
End Sub

You can specify that the OnEntry applies to for an entire workbook by specifying the workbook instead:

ThisWorkbook.OnEntry = "CheckIt"

31.19 Auto Capitalizing
The following code, put in a worksheet’s code module, will automatically capitalize any entry on that worksheet:

```vba
Private Sub Worksheet_Change(ByVal Target As Excel.Range)

 'turn off event handling while macro runs To avoid looping
 Application.EnableEvents = False

 'check and see if cell has formula. Only change contents if no formula
 'and the entry is not numeric
 If (Not ActiveCell.HasFormula) And _
 (Not IsNumeric(ActiveCell.Value)) Then _
 ActiveCell.Value = UCase(ActiveCell.Value)
 Application.EnableEvents = True
End Sub
```

### 31.20 Using OnEntry To Force Entries To Be Uppercase

By using an `OnEntry` macro, you can force all text entries to be upper case. An OnEntry macro can be set for either a specific sheet or for all sheets.

'This subroutine turns on the OnEntry macro, which runs each time an entry is made. Please note that this OnEntry macro is sheet specific, to a sheet named "Sheet1" in the ActiveWorkbook

```vba
Sub Set_Upper_Case_Entry_On()
    Worksheets("Sheet1").OnEntry = "ChangeToUpperCase"
End Sub
```

'this is the macro that runs each time an entry is made

```vba
Sub ChangeToUpperCase()
    If Not ActiveCell.HasFormula Then _
        ActiveCell.Value = UCase(ActiveCell.Value)
End Sub
```

'this macro turns off the OnEntry macro

```vba
Sub Set_Upper_Case_Entry_Off()
    Worksheets("Sheet1").OnEntry = ""
End Sub
```

To have the above macro, ChangeToUpperCase, run in any worksheet, change the first macro to set the OnEntry for the entire application:
Sub Set_Upper_Case_Entry_On()
    Application.OnEntry = "ChangeToUpperCase"
End Sub

To turn off the OnEntry macros, use statements like the following:

Worksheets("Sheet1").OnEntry = ""

or

Application.OnEntry = ""

31.21 Running A Macro When The User Double Clicks

To do this, you must go into the Visual Basic editor, and double-click the worksheet (under the Microsoft Excel Objects folder on the left) that contains the cell you wish to trigger the macro. Then change the (left dropdown to Worksheet and change the right dropdown to BeforeDoubleClick. Then you can just do something similar to the code below to check the range that's passed to the subroutine (this one is set to run if A1 is double-clicked):

Private Sub Worksheet_BeforeDoubleClick( _
    ByVal target As Excel.Range, _
    Cancel As Boolean)
    Dim addr As String
    addr = target.Address
    If addr = "$A$1" Then

        'put your code here

    End If
End Sub

31.22 Preventing A User From Closing A File

The following code in the workbook module of a workbook will prevent the user from closing the file:

Private Sub Workbook_BeforeClose(Cancel As Boolean)
    Cancel = True
End Sub

The only way to close the workbook is to use the following statement in your code, which prevents the above event (and other automatic events) from running, and thus preventing the workbook from closing.

Application.EnableEvents = False

The following code put in the workbook’s module will prevent the user from saving the file unless the name found via Tools, Options, General is listed as one allowed to save the
workbook. Get to the workbook’s code module by selecting the workbook option in the Visual Basic Project Explorer and selecting View, Code.

Private Sub Workbook_BeforeSave(ByVal SaveAsUI As Boolean, _
    Cancel As Boolean)

'Check name and allow save if UserName equals "Name"

If Application.UserName < "Name" Then

'***Cancels Save***'

    Cancel = True
End If
End Sub

Change the file name or the drive will still not allow one to save the file unless authorized. You should password protect your code so that the user can not change it.

31.23 Preventing A User From Closing Any File

1. In the Visual Basic Editor (VBE), add a class module to your code workbook (Insert-->Class Module). Note the name of the class module (which is probably called Class1).

2. In the code window that appears, type: Public WithEvents xlAppTrap as Excel.Application

3. At the top of the code window, you’ll see two drop down boxes. Click the one on the left (should say General) and select: xlAppTrap

4. A private sub should appear on the module, probably called Private Sub xlAppTrap_NewWorkbook.

5. Click the Drop-down on the right and select WorkbookBeforeClose. That procedure should now appear in the code window. You can delete the first one (xlAppTrap_NewWorkbook)

6. To prevent them from closing any workbook as well as Excel, simply enter one line of code.

    Cancel=True

Finally, you need to ‘activate’ the class by defining and initializing a variable

7. At the top of the module where you keep your global variables (or any regular module), type the following

    Public clsAppTrap As New Class1

Where Class1 is the name of the class module you inserted in Step 1

8. In your Auto_Open or Workbook_Open procedure, add the following line of code
Set clsAppTrap.AppTrap = Excel.Application

9. Once this code is run, the user will not be able to close any workbook, or exit the application, using neither the Excel or control menus. You'll need to add the following line of code to your Auto_Close (or any routine that attempts to close the app or workbooks):

    Set clsAppTrap.AppTrap = Nothing

That line will disable the trap. Any routine that closes workbook(s) without closing your App will need to use the code in step 9 before the close statement, then use the code in step 8 after the statement so that the trap is still active.

31.24 Using Application.Caller And OnEntry Macros

Application.Caller when used with an OnEntry macro refers to the cell which is modified by an entry. For example, if you set an OnEntry macro, by doing something like the following:

Sub TurnOnEntryOn()
    Application.OnEntry = "AppCallerExample"
End Sub

Then the following illustrate some of what Application.Caller would return if a cell is changed

Sub AppCallerExample()

    'this returns the value in the cell
    MsgBox Application.Caller.Value

    'this sets a range variable equal to the modified cell
    Dim cell As Range
    Set cell = Application.Caller

    'this returns the address of the modified cell
    MsgBox Application.Caller.Address

    'this returns the formula in the cell, including the equal sign
    MsgBox Application.Caller.Formula
End Sub

To de-activate the Application.OnEntry command, use the following statement in your code:

    Application.OnEntry = ""

31.25 Stopping Event Looping
The Worksheet_Change event (and all other events) will be fired by actions taken by your code as well as by users. For Excel object-specific events, the easiest thing to do is the following:

```vba
Private Sub Worksheet_Change(ByVal Target As Excel.Range)
  Application.EnableEvents = False
  'Your code goes here.
  Application.EnableEvents = True
End Sub
```

Note: this only works for Excel object events. For UserForms you have to code the same sort of thing manually using module-level flag variables.

### 31.26 Capturing When The User Changes The Selected Cell

You can use the worksheet `SelectionChange` event to capture when the user changes the selected cell. To create this code, do the following:

- Right click on the worksheet tab and select View Code
- Excel will automatically create the following code in the worksheet's module:

```vba
Private Sub Worksheet_SelectionChange(ByVal Target As Excel.Range)
End Sub
```

- Add your code to the above. The following are illustrations of such code:

This example displays in a message box the range the user selects

```vba
Private Sub Worksheet_SelectionChange(ByVal Target As Excel.Range)
  MsgBox "You selected " & Target.Address(False, False)
End Sub
```

This example restricts the user's selection to B5:B10. If the user selects any cell but these cells, the nearest cell in this range is selected. Also, the user is not allowed to select a range.

```vba
Private Sub Worksheet_SelectionChange(ByVal Target As Excel.Range)
  Dim R As Integer, C As Integer
  Dim bChangeCell As Boolean
  'disable the event handlers so that action of this code does not
  'trigger the code a second time
End Sub
```
Application.EnableEvents = False

'get the ActiveCell row and column and determine if the user selected 'a range outside of the allowed range, B5:B10
'set bChangeCell to true if the selected cell is outside this range

R = ActiveCell.Row
C = ActiveCell.Column
If R < 5 Then
    R = 5
    bChangeCell = True
ElseIf R > 10 Then
    R = 10
    bChangeCell = True
End If
If C <> 2 Then
    C = 2
    bChangeCell = True
End If

'select cell in allowed range if necessary and select only the active cell if more than one cell selected

If bChangeCell Then Cells(R, C).Select
If Selection.Cells.Count > 1 Then ActiveCell.Select

're-enable the event handlers

Application.EnableEvents = True
End Sub

31.27 Determining When A Worksheet Is Selected Or A Workbook Activated

The Workbook_SheetActivate event will detect when a worksheet is selected. You can then include in it code you want executed. This event and its code goes in the workbook's module. For example,

Private Sub Workbook_SheetActivate(ByVal Sh As Object)
    MsgBox "You selected sheet " & Sh.Name
End Sub

displays the name of the sheet selected. Please note it does not detect when you go to another workbook and return. To do that, you need to use the Workbook_Activate event:

Private Sub Workbook_Activate()
    MsgBox "You are now in " & ActiveWorkbook.Name
End Sub

31.28 Canceling a Close Event

497
You can cancel the close event in the Workbook Before_Close event procedure by setting Cancel = True.
32. HTML

32.1 Opening A Hyperlink

```vba
Declare Function ShellExecute Lib "shell32.dll" _
    Alias "ShellExecuteA" _
    (ByVal Hwnd As Long, ByVal lpOperation As String, _
       ByVal lpFile As String, ByVal lpParameters As String, _
       ByVal lpDirectory As String, ByVal nShowCmd As Long) As Long

Sub LaunchLink()
    ShellExecute 0, "Open", "http://www.add-ins.com", ",", ",", 1
End Sub
```

And another way:

```vba
ActiveWorkbook.FollowHyperlink "http://www.add-ins.com", , True
```

32.2 Opening A HTML Page From Excel

The following will open a web page from Excel, without needing to know which browser is setup as the default browser to run HTML pages:

```vba
Dim x As Double
On Error Resume Next
x = Shell("explorer.exe http://www.microsoft.com")
If Err.Number <> 0 Then
    MsgBox "Houston, we have a problem."
End If
```

Another suggested approach is:

```vba
ActiveWorkbook.FollowHyperlink _
    Address:="http://home.netscape.com", _
    NewWindow:=True
```

32.3 Save As HTML

Excel 97/2000 SR-1 and greater have the capability to programmatically save a portion of a page to HTML. This page will give you details:

http://support.microsoft.com/support/kb/articles/Q168/5/61.asp

XL97: How to Programmatically Save a Worksheet as HTML

The following is an example of saving a sheet as a web page.
Sub jdbCreateWebPage(curr As String)
Dim ObjToConvert(1) As Variant
Dim Result As Integer
k = Array(Sheets("Chart").Range("Chart_title"), _
Sheets("Chart").ChartObjects(1))

'Populate the ObjToConvert array with the ranges and chart that you want to export.
Set ObjToConvert(0) = Sheets("Chart").Range("Chart_title")
Set ObjToConvert(1) = Sheets("Chart").ChartObjects(1)
MyName = ActiveSheet.Range("A2")

'Load the Internet Assistant Wizard add-in.
AddIns("Internet Assistant Wizard").Installed = True

'Create the HTML page.
web_fileName = CurDir & "\" & web_FilesDir & "\" & _
curr & ".htm"
Result = _
  htmlconvert(Rangeandcharttoconvert:=ObjToConvert, _
  useexistingfile:=True, _
  usefrontpageforexistingfile:=False, _
  addtofrontpageweb:=False, codepage:=1252, _
  htmlfilepath:=web_fileName, _
  ExistingFilePath:=web_template, _
  headerfullpage:="Test Page", _
  linebeforetablefullpage:=False, _
  Namefullpage:=MyName)

' If the conversion is successful, the code htmlconvert_success is returned.
If Result = htmlconvert_success Then
  MsgBox "Web Page Created Successfully"
Else
  MsgBox "Error Creating Web Page"
End If
End Sub

32.4 Deleting HyperLinks

The following example shows how to delete all the hyperlinks in a worksheet:

For Each hLink In ActiveSheet.Hyperlinks
  hLink .Parent.Clear
Next

To delete all the hyperlinks in a selection use:

Selection.Hyperlinks.Delete
32.5 Inserting a Hyperlink to a Chart Sheet

on the sheet containing the hyperlinks, put the following in the

sheet module:

```vba
Private Sub Worksheet_SelectionChange(ByVal Target As Excel.Range)
    ChangeToChartSheet
End Sub
```

In a regular module, put the following:

```vba
Sub ChangeToChartSheet()
    Dim cSheet
    If Not IsEmpty(ActiveCell) Then
        With ActiveCell
            If .Font.Underline = xlUnderlineStyleSingle Then
                If .Font.ColorIndex = 5 Then
                    On Error Resume Next
                    Set cSheet = Sheets(ActiveCell.Value)
                    On Error GoTo 0
                    If IsEmpty(cSheet) Then
                        MsgBox "Sheet " & ActiveCell.Value & " not found"
                        Exit Sub
                    End If
                    If TypeName(cSheet) = "Chart" Then
                        If cSheet.Visible = xlSheetVisible Then
                            cSheet.Select
                        Else
                            MsgBox "Chart sheet " & ActiveCell.Value & " is hidden"
                        End If
                    End If
                End If
            End If
        End With
    End If
End Sub
```

The above code will change to a chart sheet if the cell is formatted to look

like a hyperlink.

32.6 How To Invoke A Hyperlink
If just one hyperlink on the worksheet, then use

`Worksheets(1).Hyperlinks(1).Follow`

If the hyperlink is attached to a specific cell, you can use...

`Worksheets(1).Range("B4").Hyperlinks(1).Follow`

If the hyperlink is attached to a shape, use...

`Worksheets(1).Shapes("AutoShape 1").HyperLink ._ Follow NewWindow:=True`

### 32.7 Getting A Cell's Hyperlink

The following code returns the hyperlink of the active cell:

```vba
On Error Resume Next
HyperlinkAddress = ActiveCell.Hyperlinks(1).Address
If hyperlinkaddress = 0 then hyperlinkaddress = ""
```

### 32.8 Getting Stock Prices From A Web HTTP Query

The following code pulls the Dupont company stock price from the Internet and pastes into a new page:

```vba
Sub Get_Stock_Info()
    Dim sPage As String
    Dim destSheet As Worksheet
    sPage = _
    "URL; http://finance.yahoo.com/q?s=dd
    Set destSheet = Sheets.Add
    With destSheet.QueryTables.Add(Connection:=sPage, _
    Destination:=destSheet.Range("A1"))
    .BackgroundQuery = True
    .TablesOnlyFromHTML = False
    .Refresh BackgroundQuery:=False
    .SaveData = True
    End With
End Sub
```

Andrew Baker has a [very detailed example](#) of using VBA to query a web site. This example can be found at:
33. WORKING WITH OTHER APPLICATIONS

33.1 Using Excel To Send E-Mails

The following code uses Excel and Outlook to send e-mails. The address is stored in one cell and the body of the message in another cell. You can alter the following macro so that you can include a file attachment too. Furthermore, you can alter the macro so that the e-mail is automatically sent (without a preview).

'Use early binding to create a new Outlook Application Object

Public olApp As New Outlook.Application
Public nsMAPI As Outlook.Namespace
Public exp As Outlook.Explorer

Sub NewMailMessage()
    Dim itmMail As Outlook.MailItem
    Dim x As Variant
    Dim y As Variant

    'Return a reference to the MAPI layer
    Set nsMAPI = olApp.GetNamespace("MAPI")

    'Set the current cell selection equal to the e-mail address
    x = ActiveCell.Value

    'Set the cell underwork!b8 equal to the message body
    Application.ScreenUpdating = False
    WorkSheets("underwork").Activate
    y = Range("b8").Value

    'Create a New mail message item
    Set itmMail = olApp.CreateItem(olMailItem)
    With itmMail

    'Add the subject of the mail message
    .Subject = "Research Database"

    'Create some body text
    .Body = y & vbCrLf

End Sub
'Add a recipient and test to make sure that the address is valid using the Resolve method

With .Recipients.Add(x)
    .Type = olTo
    If Not .Resolve Then
        MsgBox "Unable to resolve address.", vbInformation
        Exit Sub
    End If
End With

'Remove quotes to attach a file as a link with an icon
'With .Attachments.Add_

'("D:\ofc - 06\acme.mdb", olByReference)

'DisplayName = "Training info"

'End With

'Display the item

.Display

'Send the mail message

.Send
End With

'Release memory

Set itmMail = Nothing
Set nsMAPI = Nothing
Set olApp = Nothing
End Sub

33.2 Sending E-Mail From Outlook Express

If you use Outlook Express, you can use SendMail to mail the active workbook. You should save the workbook before mailing, as the last saved copy is what is mailed. With SendMail, you can only send the active workbook, and specify a subject and recipients (separated by commas). You can not enter any text in the body of the e-mail.

Sub MailActiveWorkbook()
    ActiveWorkbook.SendMail_
        .Recipients:="john@yahoo.com",_
        .Subject:="Test Mail"
End Sub
33.3 Sending E-Mail With Outlook

This example shows how to send just a mail message, no text via Microsoft Outlook. You need a reference to the Outlook object library via VB editor, Tools, References.

```vba
Sub Send_Outlook_E_Mail()
    Dim oOutlook As New Outlook.Application
    Dim oNameSpace As Outlook.NameSpace
    Dim OMailitem As Outlook.MailItem
    Set oNameSpace = oOutlook.GetNamespace("MAPI")
    oNameSpace.Logon
    Set OMailitem = oOutlook.CreateItem(olMailItem)
    With OMailitem
        .Subject = "Subject Text"
        .Recipients.Add "mailaddress"
        .Body = "Your body text here"
        .Body = "more text here"
        .Body = "more text here"
        .Send
    End With
    oNameSpace.Logoff
    Set oNameSpace = Nothing
End Sub
```

To send the active workbook:

```vba
Sub SendActiveBook()
    ActiveWorkbook.HasRoutingSlip = False
    With ActiveWorkbook.RoutingSlip
        .Recipients = "somebody@domain.com"
        .Subject = "Distribution: test.xls"
        .Message = "Hello"
        .Delivery = xlAllAtOnce
    End With
    ActiveWorkbook.Route
End Sub
```

33.4 Sending E-Mail From Excel

Intuitive Data Solutions has an ActiveX Code Component that lets you incorporate universal e-mail send and receive capabilities in your macros. It supports Outlook, Lotus Notes, Exchange, and any other mail systems based on MAPI, VIM, SMTP/POP3 (Internet mail), MHS (Novell), Banyan VINES, or Active Messaging. For example, this is all the Excel VBA code you would need to send a message with a file attachment:

```vba
Dim idsMail As Object
Set idsMail = CreateObject("IDSMailInterface.Server")
idsMail.ObjectKey = "ABC123"
idsMail.NewMessage
idsMail.AddRecipientTo "Jim Smith"
idsMail.AddRecipientCc "Mary Brown, Doug Williams"
idsMail.Subject = "Meeting Agenda"
```
idsMail.Message = "Here is the agenda for the weekly meeting."
idsMail.AddAttachment "C:\MEETINGS\AGENDA.DOC"
idsMail.Send

For more info on IDSMail, go to

http://www.intuitive-data.com

### 33.5 How To Send An E-Mail On A SMTP Mail System

If you have Microsoft Messaging installed (which can be configured to work with SMTP servers), you can use code similar to the following to mail the active workbook to someone.

```vba
ActiveWorkbook.SendMail "user@domain.com", _
"Message Subject", Null
```

Check the `SendMail` function in the online help for details by placing the cursor on `SendMail` and pressing F1.

You do not need to have a Microsoft server (Mail, Exchange) for this to work; as it uses MAPI settings to communicate with whatever server you configure.

### 33.6 Using Outlook To Send Mail

The following example, posted by Bill Manville, shows how to write a macro that sends an Outlook mail message. You must first create a Tools / Reference from a module in your workbook to the Outlook 9x Object library.

```vba
Sub TellTheBossItsDone()
    Dim oOutl As New Outlook.Application
    Dim oNS As Outlook.NameSpace
    Dim oMail As Outlook.MailItem
    Set oNS = oOutl.GetNamespace("MAPI")
oNS.Logon

    ' user will be prompted for name and password if parameters not given here
    Set oMail = oOutl.CreateItem(olMailItem)
    With oMail
        .Subject = "Monthly update completed"
        .Recipients.Add "TheBoss"
        .Body = "The figures for last month are now available"
        .Send
    End With
oNS.Logoff
    Set oNS = Nothing
End Sub
```

### 33.7 E-Mailing A File With Outlook
Assuming:

1- that the name of your workbook will be MyFile.xls

2- you use Outlook for your email client

To do the e-mail part use this:

'Use early binding to create a new Outlook Application Object

Public olApp As New Outlook.Application
Public nsMAPI As Outlook.Namespace
Public exp As Outlook.Explorer

Sub NewMailMessage()
    Dim itmMail As Outlook.MailItem
    Dim x As Variant
    Dim y As Variant

    'Return a reference to the MAPI layer
    Set nsMAPI = olApp.GetNamespace("MAPI")

    'Create a New mail message item
    Set itmMail = olApp.CreateItem(olMailItem)

    With itmMail

        'Add the subject of the mail message
        .Subject = "Type message subject here"

        'Create some body text
        .Body = "type message body here" & vbCrLf

        'Add a recipient and test to make sure that the
        'address is valid using the Resolve method
        With .Recipients.Add("type email@address.here")
            .Type = olTo
            If Not .Resolve Then
                MsgBox "Unable To resolve address."
                Exit Sub
            End If
        End With

        'Attach a file as a link with an icon
With .Attachments.Add _
("C:\My Documents\jimmp.xls", olByReference) _
  .DisplayName = "MyFile.xls"
End With

'Display the item
  .Display

'Send the mail message
'.Send

End With

'Release memory

  Set itmMail = Nothing
  Set nsMAPI = Nothing
  Set olApp = Nothing
End Sub

33.8 Launching Another Windows Program Or Application

The following example shows how to launch another Windows program or application. This example launches the Windows calculator. Please note it does not check to see if it already running. If it is, you will get another version running.

Sub LaunchExample()
  Dim RetVal As Long

    'Launch calculator and give it the focus
    'the shell command returns a value, thus the need to capture
    'it in a variable

    RetVal = Shell("C:\windows\calc.exe", vbNormalFocus)
End Sub

33.9 Running A Shortcut From A Macro

Use the DOS Start command to run a shortcut from Excel VBA. For example:

Shell("start c:\windows\desktop\myshortcut.lnk")

33.10 Open Window Explorer
Sub OpenExplorer()
  x = Shell("explorer", 1)
End Sub

33.11 Getting Excel To Pause While A Shell Process Is Running

Here are some sources of information on how to do this.

http://support.microsoft.com/support/kb/articles/q214/2/48.asp

XL2000: How to Force Macro Code to Wait for Outside Procedure

http://support.microsoft.com/support/kb/articles/Q129/7/96.ASP

HOWTO: 32-Bit App Can Determine When a Shelled Process Ends

http://support.microsoft.com/support/kb/articles/q147/3/92.asp

XL: How to Force Macro Code to Wait for Outside Procedure

The API. Listed below is a Sub that will stall Excel INDEFINITELY until the Program Shelled has terminated.

Private Const INFINITE = &HFFFFFFFF

'Allows an Infinite timeout

Private Const SYNCHRONIZE = &H100000
Private Declare Function CloseHandle Lib "kernel32" (ByVal hObject As Long) As Long
Private Declare Function OpenProcess Lib "kernel32" (ByVal dwDesiredAccess As Long, _
  ByVal bInheritHandle As Long, _
  ByVal dwProcessId As Long) As Long
Private Declare Function IsWindow& Lib "User32" (ByVal hwnd As Long) As Long
Private Declare Function WaitForSingleObject _
  Lib "kernel32" (ByVal hHandle As Long, _
  ByVal dwMilliseconds As Long) As Long

Sub HardShell(FilePathName As String)
  Dim ProcessId As Long
  ProcessId = Shell(FilePathName, vbMaximizedFocus)
  Call PauseUntilTerminate(ProcessId)
End Sub

Private Sub PauseUntilTerminate(ProcessId As Long)
  Dim phnd&
  phnd = OpenProcess(SYNCHRONIZE, 0, ProcessId)
  If phnd < 0 Then
Application.StatusBar = "Waiting for termination"
Call WaitForSingleObject(phnd, INFINITE)
Call CloseHandle(phnd)
End If
Application.StatusBar = False
End Sub

The following is another way to pause your code until a shelled application completes it processing. In this case Notepad. If you shell to another application, your Excel macros will normally continue to run.

Private Declare Function WaitForSingleObject Lib "kernel32" _
(ByVal hHandle As Long, ByVal dwMilliseconds As Long) As Long

Private Declare Function CloseHandle Lib "kernel32" _
(ByVal hObject As Long) As Long

Private Declare Function OpenProcess Lib "kernel32" _
(ByVal dwDesiredAccess As Long, ByVal bInheritHandle As Long, _
ByVal dwProcessId As Long) As Long

Private Const INFINITE = -1&
Private Const SYNCHRONIZE = &H100000

' This code goes in the command button that is clicked by the user to ' start the shell process
Sub Shell_And_Wait()
  Dim iTask As Long, ret As Long, pHandle As Long
  iTask = Shell("notepad.exe", vbNormalFocus)
  pHandle = OpenProcess(SYNCHRONIZE, False, iTask)
  ret = WaitForSingleObject(pHandle, INFINITE)
  ret = CloseHandle(pHandle)
  MsgBox "Process Finished!"
End Sub

33.12 Activating A Running Application

The AppActivate statement will activate an active application if the full string of the title in the application window is used as its argument.

Sub ActivateSomething()
  Dim A
  On Error GoTo AppActivateHandler
  AppActivate "Calculator"
  Exit Sub
AppActivateHandler:
  A = Shell("Calc.Exe", 1)
End Sub

33.13 Determining If Another Application Is Running
The problem with \texttt{Application.Activate} is that it requires you know the exact Window name of the open application. Many applications change their Windows name based on the file that is open. Notepad is a good example. The following procedures, developed by Rob Bovey, test based on a partial Window Name.

\texttt{put these statements at the top of the module}

\begin{verbatim}
Declare Function FindWindowA Lib "user32" _
   (ByVal lpClassName As String, _
   ByVal lpWindowName As Any) As Long
Declare Function GetWindowText Lib "user32" _
   Alias "GetWindowTextA" (ByVal hWnd As Long, _
   ByVal lpString As String, _
   ByVal cch As Long) As Long
Declare Function IsIconic Lib "user32" _
   (ByVal hWnd As Long) As Long

Sub ActivateNotePad_Excel5_Version()
   Dim sTtl As String * 128, hWnd As Long, a As Long
   hWnd = FindWindowA("NOTEPAD", 0&)
   If hWnd = 0 Then

      'if not open, open it

      Shell "NOTEPAD.EXE", 1
   Else

      'activate the notepad window

      a = GetWindowText(hWnd, sTtl, 128)
      If IsIconic(hWnd) <> 0 Then

         'do this if minimized

         SendKeys "+~"
      End If

      'activate the window

      AppActivate Left(sTtl, a)
   End If
End Sub
\end{verbatim}

\textbf{33.14 Starting Word From Excel}

The following will open Word if it is not open and display the document you specify:

You will need to first set a reference to Word first via the VB Editor – select Tools, References and check the Word application.
Dim WordObj As Word.Application

On Error Resume Next
Err.Number = 0

Set WordObj = GetObject(, "Word.Application.8")
If Err.Number = 429 Then
    Set WordObj = CreateObject("Word.Application.8")
    Err.Number = 0
End If

WordObj.Visible = True
WordObj.Documents.Open fileName:="c:\whatever.doc"

'rest of your code

WordObj.Quit
Set WordObj = Nothing

The following is another approach:

Sub OpenWord()
    Dim wdApp As Word.Application
    Dim wdDoc As Word.Document
    
    Set wdApp = New Word.Application
    Set wdDoc = wdApp.Documents.Open(_
            "C:\whatever.doc", , True, False)
    wdApp.Visible = True
End Sub

33.15 Opening A MS Word Document From Excel

The following will open up another instance of Word and open the document C:\Test.Doc

CreateObject("word.Application").Documents._
    .Open("C:\Test.doc").Application.Visible = True

or

Sub OpenAWordFile()
    Dim wordApp As Object
    Dim fNameAndPath As String
    fNameAndPath = "c:\test.doc"
    Set wordApp = CreateObject("Word.Application")
    wordApp.Documents.Open (fNameAndPath)
    wordApp.Visible = True
End Sub
33.16 Running Word Macros From Excel

The following is an example of how to do this, plus a number of articles available from Microsoft's knowledge base.

http://support.microsoft.com/support/kb/articles/q177/7/60.asp

VBA: How to Run Macros in Other Office Programs (OFF 97)

The following Sub procedure assumes that the document WordDoc.Doc contains a macro called "WordMacro."

```vba
Sub msWordExample()
    Dim msWord as Object
    Set msWord = CreateObject("Word.Application")

    ' Note that the project name and module name are required to
    ' path the macro correctly.

    msWord.Run "Project.Module1.WordMacro"
End Sub
```

Other articles of interest on the internet:

http://support.microsoft.com/support/kb/articles/q128/4/05.asp

XL: How to Run a WordBasic Macro from an MS Excel Macro

http://support.microsoft.com/support/kb/articles/q165/5/18.asp

Calling Macros Using OLE from MS Visual Basic for Applications

(this has word running an excel macro, but the idea should the same)

http://support.microsoft.com/support/kb/articles/q149/8/30.asp

XL: Macro to Link a Range of Cells in Word

(Example of Excel controlling word, but not running a word macro)

http://support.microsoft.com/support/kb/articles/q165/9/26.asp

OFF97: Can't Dimension Word as Application from Other Program

(Examples of Working with Word 97 using OLE)

http://support.microsoft.com/support/kb/articles/q135/0/82.asp
Invalid Page Fault Running VB Macro in Hidden Word Session

(Word 7, when Word is Closed causes a problem)

http://support.microsoft.com/support/kb/articles/q167/2/23.asp

Microsoft Office 97 Automation Help File Available on MSL

-------- DDE, XLM and Older Versions ---------

http://support.microsoft.com/support/kb/articles/q93/6/57.asp

WD: Running Word for Windows as a DDE Server

http://support.microsoft.com/support/kb/articles/q94/6/24.asp

Excel Macro to Determine If Word for Windows Is Loaded

http://support.microsoft.com/support/kb/articles/q68/5/10.asp

XLM: Opening and Closing Word for Windows Using an Excel Macro

http://support.microsoft.com/support/kb/articles/q141/7/72.asp

XL: Visual Basic Examples Using DDE

33.17 Opening A PowerPoint Presentation

The following code will open a new instance of PowerPoint and display a presentation. When one stops the presentation (by pressing ESC, then PowerPoint stays open:

```vba
Sub Open_And_Run_PPT1()
    Dim PPTObject As Object
    Dim pptPresentation As Object
    Set PPTObject = CreateObject("PowerPoint.Application")
    PPTObject.Visible = True
    Set pptPresentation = _
    PPTObject.Presentations.Open _
    ("C:\temp\presentation example.ppt")
    PPTObject.ActivePresentation.SlideShowSettings.Run
End Sub
```

If you want PowerPoint to close after the presentation, then use code like the following:

```vba
Sub Open_And_Run_PPT_and_Close()
    Dim PPTObject As Object
    Dim pptPresentation As Object
    Set PPTObject = CreateObject("PowerPoint.Application")
    PPTObject.Visible = True
    Set pptPresentation = _
```
```vbnet
PPTObject.Presentations.Open _
("C:\temp\presentation example.ppt")
PPTObject.ActivePresentation.SlideShowSettings.Run
While PPTObject.SlideShowWindows.Count = 1
  DoEvents
  Wend
PPTObject.ActivePresentation.Close
PPTObject.Quit
End Sub

33.18 Displaying A DOS Window

The following will display a DOS window:

Sub GiveMeDos()
  Dim taskID
  taskID = Shell("Command.com", 1)
End Sub

33.19 Getting Data From Access

One of the most popular ways to do this is to use ADO or DAO to get the data and then to use Excel's CopyFromRecordset method to plop the data down. In Excel 97 and prior, the CopyFromRecordset method only supports DAO recordsets. However, Excel 2000 now supports ADO recordsets.

The following illustrate two methods of getting data into Excel from Access:

Sub GetDataWithDAO()
  Dim db As DAO.Database
  Dim rst As DAO.Recordset
  Set db = OpenDatabase("C:\My Documents\SalesDb.mdb")
  Set rst = db.OpenRecordset("SalesData")
  Range("A1").CopyFromRecordSet rst
End Sub

Sub GetDataWithADO()
  Dim cnt As New ADODB.Connection
  Dim rst As New ADODB.Recordset
  cnt.Open "Provider=Microsoft.Jet.OLEDB.4.0;" & _
    "Data Source=C:\My Documents\SalesDb.mdb;"
  rst.Open "Select * From SalesData", cnt
  Range("A1").CopyFromRecordSet rst
End Sub

If you need to get data from Access while running a Visual Basic macro, also check the following Microsoft Internet pages for information of using DAO to get data from Access.

http://support.microsoft.com/support/excel/dao.asp

This page has a number of articles and a help file you can download.
Microsoft knowledge base article Q149254 also gives an example of how to return DAO query results directly to a list box in Excel.

The following is a posting on the Excel Visual Basic user group that gives some additional examples

Question asked: "Looking for some advice relating to connecting to an Access database within excel"

**Response:**

If it's not a secured database, then you can simply call

```vba
set rs = DBEngine.Opendatabase(DBPath, Options)
```

- check the help file for details. You will, of course, have to select one of the DAO libraries in the VBA references dialog first - 3.5 is the recommended one, assuming you're on Office 97.

If it is secured, you need to mess about with user id's and passwords.

Find the security file

```vba
DBEngine.SystemDB = "f:\Data\Workgroup.mdw"
```

Open a workspace

```vba
Set wsp = DBEngine.CreateWorkspace( _
   "SomeWorkspaceName", txtUserID, txtPassword, dbUseJet)
DBEngine.Workspaces.Append wsp
```

And then open the database in this secure workspace:

```vba
Set db = wsp.OpenDatabase(strDBName, Options, etc)
```

And read the data

```vba
Set rs = db.OpenRecordset strSQL, etc, etc
```

You will need really good error trapping, otherwise you end up with un-initialized or dropped variables all over the place.

**33.20 How To Exchange Data Between Access And Excel**

Take a look at the articles at

33.21 SQL Query Strings

SQL query strings greater than 255 characters should be broken into an array and each portion of the query should be less than or equal to 127 characters. You will need to use a variant array. See the below articles for sample code that does this.

http://support.microsoft.com/support/kb/articles/q114/9/92.asp

XL: SQLExecQuery and SQLRequest Limited to 255 Characters

http://support.microsoft.com/support/kb/articles/q124/2/18.asp

XL: Macro Examples Using XLODBC Functions

[ See Sample 8 ]

33.22 Excel GetObject To Open Word

Make sure you've got references enabled to Word. From the VBA Editor, go to the Tools menu, References, and check "Microsoft Word 8.0 Object Library". After you do this, the following statement will work and not return an error:

Dim mywd As Object
Set mywd=GetObject (, "Word.Application")

33.23 Using Barcodes in Excel

There are a number of ways to get bar code data into Excel depending on the type of bar code reader that you have. Most bar code readers are available with one of two output options. The first option is called "Keyboard Wedge" output where you unplug your keyboard, plug the bar code reader into the keyboard port on your PC and then plug your keyboard into the bar code reader. This arrangement makes the bar code reader appear as it were simply a second keyboard. Your original keyboard continues to work as normal however when you read a bar code, the data encoded in the bar code appears to any application running on your PC as if it were typed in. The keyboard wedge interface is extremely simple however it has a few drawbacks. If you swipe a bar code, the cursor has to be in the correct input field in the correct application otherwise you end up reading bar code data into whatever application has the focus. This can cause all sorts of potential problems as you can imagine. The keyboard output also is limited in that you cannot modify the data in any way before sending it into the program that is to receive the data. For example, if you needed to parse a bar code message up into pieces or remove some of a bar code message or add in a date or time stamp you would not be able to with a normal keyboard wedge reader.

The other possible output option is to get a bar code reader with an RS232 or "Serial" interface. With these types of bar code readers, you connect the reader to an available serial port on the back of your PC. You would then need a program called a "Software Wedge" to take the data from the bar code reader and feed it to the application where you want the data to go. The disadvantage to this approach is that it is a little more complex however you gain much more
control over how and where your data ends up when you read a bar code. With a Software Wedge, you can control exactly where the data goes in the target application and you can also perform all sorts of modifications on the data before it is sent to the application. You can even cause a macro to run in Excel when you read a bar code so that you can trap in code whenever a bar code is scanned.

Tal Technologies sells a product called WinWedge which is a Software Wedge for Windows. Visit:

http://www.taltech.com

for more information. This web site is also an extremely good place to obtain information about bar coding in general.
34. NEAT THINGS TO KNOW

34.1 Using SendKeys In Your Macros

Unlike Lotus 1-2-3 macros, 99.999% of Excel macros can be written without using SendKeys. There are however some situations where you may have to use SendKeys to send keystrokes to Excel's menus. For help on creating SendKey statements, type the word SendKeys in a module, highlight it, and press the F1 key.

To insure that your keystrokes are run by Excel, use a DoEvents statement. It is just the word DoEvents by itself on a line. This yields control to Excel so that the keystrokes and their affect can occur.

For example, the following forces a full re-calculation of all cells, and then allows the remaining code to run. A full re-calculation is equivalent of pressing ALT-CTL-F9, for which there is no equivalent Visual Basic statement.

```
SendKeys "%^{F9}"
DoEvents
```

34.2 Hiding The Active Menu And Using Full Screen

The command

```
Application.CommandBars.ActiveMenuBar.Enabled = False
```

will hide the active menu. This command, coupled with setting the screen to full screen and hiding the full screen toolbar will give you a non-Excel look:

```
Application.DisplayFullScreen = True
Application.CommandBars("Full Screen").Enabled = False
```

34.3 Hiding Screen Update Activity - Stop Screen Flashing

If your code changes the active cell, workbook, or worksheet, then the screen will flash or flicker as it changes location. It will also flash if you use commands such as PasteSpecial which automatically change the active cell. To eliminate this flashing, put the following statement in your procedures:

```
Application.ScreenUpdating = False
```

If you need to display the active sheet to the user, such as when you display an input box for selection of a cell or range, and you have earlier set the above property to False, then you need to set this property back to True to allow the user to select a cell:
Application.ScreenUpdating = True

Screen updating is automatically turned back on when your code is done.

### 34.4 Stopping Alert Messages / Display Alert Warning

Microsoft Excel will frequently display alert boxes to warn you about something you are doing or to advise you that something you wanted to happen did not happen. For example, if you issue the `SaveAs` command to save a file and the existing file exists, a warning or alert box will appear. If you issue the `Find` command and no match is found, then an alert box will also appear. To eliminate these alert boxes put the following line in your code:

```vba
Application.DisplayAlerts = False
```

Visual Basic is inconsistent in resetting the `Application.DisplayAlerts` property back to **True**. Sometimes it resets it and sometimes it doesn't. The best approach is to always set it back to **True** Before your code is done.

How do you know if it hasn't been reset? Usually you find this out when you've modified a file and then expect Excel to prompt you to save the file when you close Excel and it doesn't.

### 34.5 Speeding Up Your Procedures And Controlling Calculation

The easiest way to speed up your code's execution is to turn calculation off. You can do so with the following statement:

```vba
Application.Calculation = xlManual
```

To turn calculation back on, use the following statement:

```vba
Application.Calculation = xlAutomatic
```

Please note you must have a workbook open in order to run the above two commands. You can check to see if a workbook is open by:

```vba
If Not ActiveWorkbook Is Nothing Then
    Application.Calculation = xlManual
End If
```

Another way to turn off calculation if no workbook is active (this assumes that you are running an add-in) is to do the following:

```vba
ThisWorkbook.Activate
Application.Calculation = xlManual
```

At any point in your code, you can use the following statement to cause Excel to re-calculate:
One of the problems with the above approach is that the user may have already set calculation to manual and the last statement will turn it on, which is a state that the user may not want. The following is one way to solve this problem:

```vba
Dim calcSetting As Integer

' store the calculation setting
calcSetting = Application.Calculation

'turn off calculation and run your code
Application.Calculation = xlManual

' <your code here>

're-calculate Excel
Calculate

're-store calculation back to the user's original setting
Application.Calculation = calcSetting
```

To have Excel perform a calculation when you need cell values updated, put the following statement in your code as needed:

```vba
Calculate
```

Another way to speed up your procedures is to refer to cells directly, instead of changing to the cells in question and then performing an operation of some kind.

**Slow:**
```vba
Range("A5").Select
ActiveCell.Value = 10
```

**Fast:**
```vba
Range("A5").Value = 10
```

The following is another example of how to change code to make it more efficient:

**Slow:**
```vba
WorkBooks("New Data").Activate
Sheets("YTD").Select
Range("C9").Select
ActiveCell.Copy
Workbooks("Actual.Xls").Activate
Sheets("Results").Activate
Range("C5").Select
ActiveSheet.Paste
```
Fast: **WorkBooks**("New Data").**Sheets**("YTD").**Range**("C9").**Copy**  
**Workbooks**("Actual.Xls").**Sheets**("Results").**Range**("C5")

This second approach specifies the cell to be copied and pastes it to the destination cell all in one step, without changing workbooks, sheets, or cell ranges.

### 34.6 Speeding Up Your Procedures - More Suggestions

Refreshing the screen typically slows down your macros. If you use the following statement:

```vba
Application.ScreenUpdating = False
```

Then the screen will not refresh until this is set to **True** or until your macro is done.

If you have many pagebreaks in your worksheets, then inserting and deleting rows or columns may be slow. Use the following line of code to hide the pagebreaks. This keeps Excel from updating their displayed location, thus speeding up your macros:

```vba
ActiveSheet.DisplayPageBreaks = False
```

### 34.7 Macros Run Really Slow

If you are using Excel and Outlook, your macros may run 5-10X longer than they should. The journalizing feature of Outlook logs many actions taking time from Excel. Eventually, the journal files become quite large, and performance is slowed. To prevent this, you should turn journalizing off for Microsoft Excel. To do this in Outlook select Tools / Options / Journal / and uncheck Microsoft Excel.

Other items you can do to speed up your code are:

Use the statement

```vba
Application.ScreenUpdating = False.
```

This stops screen updating, which takes a fair amount of time.

Turn calculation off with the statement

```vba
Application.Calculation = xlManual
```

### 34.8 Determining How Long Your Code Took To Run

You can use code like the following to find out how long your code took to run:

```vba
Dim t As Date
't set a variable equal to the starting time
```
t = Now()

'your code here

calculate and display the time the code took

MsgBox Format(Now() - t, "hh:mm:ss")

34.9 A Solution To Excel Running Slow

A number of users have reported problems with Excel running very slow and sluggish. They have found out that if they delete the files in their Windows Temp directory, that this will speed up Excel. One way to delete these files is with the following VB code:

```vba
Sub DeleteWindowsTempFiles()
    Dim winTempDirectory As String
    winTempDirectory = Environ("Temp")
    Dim fName As String
    If MsgBox(
        "Delete all files in your Windows temp directory", vbOKCancel) = vbCancel Then Exit Sub
    fName = Dir(winTempDirectory & "\*.*")
    On Error Resume Next
    While fName <> ""
        Kill fName
        fName = Dir()
    Wend
End Sub
```

One user found that the slowness with Excel occurred whenever he added a control to a userform.

34.10 How To Hide Excel Itself

The statement

```vba
    Application.Visible = False
```

will hide Excel. The following statement will make it visible:

```vba
    Application.Visible = True
```

34.11 Opening Without A Blank Workbook And No Splash Screen

If you start Excel with a "/e", it will open with a blank workbook. If you use "/n", it will open without a splash screen:
34.12 Closing Excel Via Visual Basic with Application.Quit

To close down Excel from one of your procedures, include the following statement in your code:

```vba
Application.Quit
```

Two cautions - if you have add-ins loaded that must execute an Auto_Close set of commands, such as recording information in an INI file, they will try to execute when Excel is closed by the Application.Quit statement, and very likely crash Excel. The second caution is to run all your code before the Application.Quit is encountered. You do not want to rely on Visual Basic to run statements past the Application.Quit statement.. In Excel 97/2000, code after the Application.Quit statement is not executed. In Excel 5/7, it is executed.

If you have troubles with Excel crashing when you use an Application.Quit statement, remove all your add-ins to see if this is the cause, and then add them back one at a time to determine which one is the cause.

34.13 Playing WAV Files

There are several different ways to play WAV files via Visual Basic code. The following is the simplest way to play a WAV file

Place the following at the top of one of your modules

```vba
Declare Function PlaySound Lib "winmm.dll" Alias "sndPlaySoundA" (ByVal wavFile As String, ByVal lNum As Long) As Long
```

in your code, place a statement like the following to play a WAV file:

```vba
Call PlaySound("C:\Windows\MySound.wav", 0)
```

The following is a more elaborate example that allows you control of when your code begins and when the sound ends:

```vba
Private Declare Function PlaySound Lib "winmm.dll" Alias "PlaySoundA" (ByVal lpszName As String, ByVal hModule As Long, ByVal dwFlags As Long) As Long
Const SND_SYNC = &H0
Const SND_ASYNC = &H1
Const SND_FILENAME = &H20000
Sub PlayWAV1()
    Dim WAVFile
    WAVFile = "C:\Windows\Media\logoff"
```
'Finish sound before executing further code (SND_SYNC)

Call PlaySound(WAVFile, 0&, SND_SYNC Or SND_FILENAME)

'Use SND_ASYNC for the code to continue through the sound

End Sub

34.14 Running Macros That Are Located In A Different Workbook

To run a macro that is located in another workbook, use Application.Run. The name of the workbook and the macro should be enclosed in double quotes. If the file name contains spaces, the file name must be enclosed in single quotes. Also, an exclamation point is needed between the workbook name and the macro name.

`Application.Run "'My Workbook.xls'!SomeRoutine"

This is something you may need to do if you open up a workbook or add-in via code and need to run the Auto_Open procedure found in file you are opening. For example:

`Workbooks.Open "MyAddInfile.XLA"
Application.Run "MyAddInfile.XLA!Auto_Open"

An Auto_Open procedure is a macro that Excel automatically runs if the file is opened manually or an add-in is opened by Excel from the add-in list.

If you need to run a function that is in another workbook, then you can use a statement like the following:

`RetVal = Application.Run("MyAddin.xla!MyFunction")`

If the function has arguments, then you would use the following approach:

`RetVal = Application.Run("MyAddin.xla!NeatFunction", 5)`

The same approach would work with procedures that take arguments.

Please note if you need to run subroutines that are located in the same workbook as the main routine, then all you need to do is to refer to them by their name:

`Sub Main_Procedure()

'run first procedure

Procedure1Name

'run second procedure`
34.15 Writing Text To A Shape Or Text Box

If you record the code to write text out to a text box, you will code like the following:

```vba
ActiveSheet.Shapes(“Text Box 1”).Select
Selection.Characters.Text = ”Any message"
```

However, if you run the code, you will get an error message. The correct syntax is:

```vba
With ActiveSheet.Shapes(“Text Box 1”) .TextFrame.Characters.Text = ”Any message” End With
```

34.16 How To Prevent A Macro From Showing In The Macro List

One way to prevent a macro from showing up in the macro list when you do Tools, Macros is to put a optional dummy argument to the macro:

```vba
Sub MyMacro(Optional dummyVar)
   ‘your code
End Sub
```

Another way is to put the word `Private` before the `Sub` declaration:

```vba
Private Sub MyMacro()
   ‘your code
End Sub
```

You can also put

```vba
Option Private Module
```

at the top of your module if all the macros on that module won't be called from outside the project.

34.17 Using SendKeys To Force A Recalculation
If you want to force Excel to recalculate all formulas in your open workbooks, then you can do so by pressing Alt-Ctrl-F9. If you want to do this in your code, then you would use the following two statements:

```
SendKeys "^%{F9}"
DoEvents
```

'more Visual Basic code

The `DoEvents` statements returns control to Excel so that it can process the `SendKeys` statement. Otherwise, the code in the procedure will run and then the `SendKey` statement is executed, not what you want!

Although the `SendKeys` has an optional parameter, call a `Wait` parameter to do the same thing as the `DoEvents`, this parameter may not always work.

The above statements are useful, because in Excel 97 there is a bug that can happen if you open a workbook which was originally written in an earlier version of Excel and which has User-Defined Functions. The VBA code in these workbooks will sometimes JUST STOP after it encounters an `Application.Calculate` or `Calculate` command. If you encounter this problem, replace these statements with the `SendKeys` and `DoEvents` statements.

### 34.18 Bypassing The Warning About Macros

This is a frequent question on user groups. It is not possible to bypass this warning. However, the user can select the option not to display the warning. If there was a way have code bypass this warning, then it would be useful to virus builders.

### 34.19 Hiding The Cell Pointer

One way to hide the cell pointer (the frame around the active cell) is to put Excel into what is called data entry mode. This is done with either of the following two statements:

```
Application.DataEntryMode = xlOn
```

or

```
Application.DataEntryMode = xlStrict
```

The first statement above allows one to exit the data entry mode by pressing the ESC key. The second requires a Visual Basic statement to turn off data entry mode. The statement that turns off data entry mode is:

```
Application.DataEntryMode = xlOff
```

The disadvantage is that data entry mode turns off the scroll bars.
If you can set the EnableSelection property of a worksheet to xlNotEnabled and the active cell indicator disappears when you protect the sheet.

```vba
With ActiveSheet
  .EnableSelection = xlNoSelection
  .Protect Contents:=True, UserInterfaceOnly:=True
End With
```

### 34.20 Turning The Caps Lock Key Off Or On

The following will automatically set the caps lock key on when you open a workbook containing this code.

```vba
Declare Function GetKeyboardState Lib "user32" _
(pKeyState As Byte) As Long
Declare Function SetKeyboardState Lib "user32" _
(lppbKeyState As Byte) As Long

Sub Auto_Open()
  'This routine runs automatically when the workbook is manually opened.
  SetCapLock
End Sub

Sub SetCapLock()
  Dim Res As Long
  Dim KBState(0 To 255) As Byte
  Res = GetKeyboardState(KBState(0))
  KBState(&H14) = 1
  ' 1 to turn on, 0 to turn off
  Res = SetKeyboardState(KBState(0))
End Sub
```

### 34.21 Modifying The Windows Registry

You can use VB code to modify the registry. For example, store an invoice number, modify it, or delete the entry. The following illustrates this:

```vba
Sub Put_Setting_In_Registry()
  Dim InvoiceNo As Long
  InvoiceNo = 133
  'this puts the value in variable InvoiceNo into the registry and
  'identifies it as "CurrentNo"
  SaveSetting "XLInvoices", "Invoices", "CurrentNo", InvoiceNo
End Sub
```
Sub Get_Value_From_Register()
    Dim InvoiceNo As Long

    'the following returns the invoice number stored in the registry, 
    'or 1000 if no number stored

    InvoiceNo = _
    GetSetting("XLIInvoices", "Invoices", "CurrentNo", 1000)
    MsgBox InvoiceNo
End Sub

Sub Delete_Registry_Entry()

    'this removes the registry entry

    DeleteSetting "XLIInvoices", "Invoices"
End Sub

34.22 Getting Values from the Registry

The following example illustrates how to get values from the Windows 95 registry with 
Windows API calls. For example:

'place these statements and functions at the top of your module

Const MAX_STRING As Long = 128
Public Const REG_BINARY = 3&
Public Const REG_DWORD = 4&

Declare Function RegOpenKeyA Lib "ADVAPI32.DLL" _
(ByVal hkey As Long, _
    ByVal sKey As String, _
    ByRef plKeyReturn As Long) As Long

Declare Function RegQueryValueExA Lib "ADVAPI32.DLL" _
(ByVal hkey As Long, _
    ByVal sValueName As String, _
    ByVal dwReserved As Long, _
    ByVal lValueType As Long, _
    ByVal sValue As String, _
    ByVal lResultLen As Long) As Long

Declare Function RegCloseKey Lib "ADVAPI32.DLL" _
(ByVal hkey As Long) As Long

Public Const HKEY_CURRENT_USER = &H80000001

'This shows how to get the value of an entry

Sub ShowExcel97Setting()
    MsgBox GetRegistryValue(HKEY_CURRENT_USER, _
        "software\microsoft\office\8.0\excel\microsoft excel", _
        "DefaultPath")
Function GetRegistryValue(KEY As Long, SubKey As String, _
ValueName As String) As String
Dim Buffer As String * MAX_STRING, ReturnCode As Long
Dim KeyHdlAddr As Long, ValueType As Long, ValueLen As Long
Dim TempBuffer As String, Counter As Integer
ValueLen = MAX_STRING
ReturnCode = RegOpenKeyA(KEY, SubKey, KeyHdlAddr)
If ReturnCode = 0 Then
ReturnCode = RegQueryValueExA(KeyHdlAddr, ValueName, _
0&, ValueType, Buffer, ValueLen)
RegCloseKey KeyHdlAddr

'If successful ValueType contains data type
' of value and ValueLen its length

If ReturnCode = 0 Then
Select Case ValueType
Case REG_BINARY
For Counter = 1 To ValueLen
TempBuffer = TempBuffer & _
Stretch(Hex(Asc(Mid(Buffer, Counter, 1)))) & " "
Next
GetRegistryValue = TempBuffer
Case REG_DWORD
TempBuffer = "0x"
For Counter = 4 To 1 Step -1
TempBuffer = TempBuffer & _
Stretch(Hex(Asc(Mid(Buffer, Counter, 1))))
Next
GetRegistryValue = TempBuffer
Case Else
GetRegistryValue = Buffer
End Select
Exit Function
End If
End If

'If unsuccessful "error" is returned

GetRegistryValue = "Error"
End Function

Function Stretch(ByteStr As String) As String
If Len(ByteStr) = 1 Then ByteStr = "0" & ByteStr
Stretch = ByteStr
End Function

34.23 Removing An Outline

If a user has turned on Excel's outline feature, you can turn it off with the following statement:
34.24 Turning Num Lock Off Or On

The following will set the Num Lock key to off. To set it to on instead, change KeyState(&H90) = 0 to KeyState(&H90) = 1

```
Declare Function SetKeyboardState Lib "User32" _
(kbArray As Byte) As Long

Declare Function GetKeyboardState Lib "User32" _
(lpKeyState As Byte) As Long

Sub SetNumLock()
   Dim KeyState(0 To 255) As Byte
   GetKeyboardState KeyState(0)
   KeyState(&H90) = 0
   SetKeyboardState KeyState(0)
End Sub
```

34.25 Turning Scroll Lock Off Or On

The following allows you to control the Scroll lock setting. As it is written, it turns off the scroll lock.

```
Declare Function SetKeyboardState Lib "User32" _
(kbArray As Byte) As Long

Declare Function GetKeyboardState Lib "User32" _
(lpKeyState As Byte) As Long

Sub SetScrollLock()
   Dim KeyState(0 To 255) As Byte
   GetKeyboardState KeyState(0)
   KeyState(&H91) = 0
   SetKeyboardState KeyState(0)
End Sub
```

If you want to turn scroll lock on change

```
KeyState(&H91) = 1
```

to

```
KeyState(&H91) = 0
```
34.26 Disabling The Delete Key

You can use code like the following to disable and enable the delete key:

```vba
Sub DisableDeleteKey()
    Application.OnKey "{Del}", ""
End Sub

Sub EnableDeleteKey()
    Application.OnKey "{Del}"
End Sub
```

34.27 Writing To The Serial Port

Just a few clues on how to write to a serial port: You can use the MSComm ActiveX control to control the serial ports. It comes with Visual Basic if you have it, otherwise you could try searching the net for a similar control.

Another way is to write the data to an ASCII file and launch your own EXE to send it to the serial port. Search for the keyword ASCII in this help file on how to create an ASCII file. You can launch an EXE by using the `Shell` command. Search for the topic "Launching Another Windows Program Or Application From Excel" for information on how to do that.

There is also a company that sells a serial communications program called the Software Wedge that you may find to be a good tool for adding serial communications capabilities to your Excel application. The Software Wedge is an executable program that can pass serial data back and forth to other programs using either DDE (Dynamic Data Exchange) or by converting incoming serial data to keystrokes (i.e. it stuffs the keyboard buffer with the incoming serial data). The program is extremely easy to use and is designed to have you up and running sending and receiving serial data directly from within your application in just a few minutes. Please visit http://www.taltech.com for more information.

34.28 COM PORTs

There are several ways to communicate with the com port. All use some sort of ActiveX control.

If you have Visual Basic, Professional edition or better, you can use MSCOMM32.OCX. It is used in VBA (Excel VB) as in regular Visual Basic.

If you do not have access to MSCOMM32, here are some alternates:

Sax Comm is available at: http://www.saxsoft.com/comm

CommX is available from http://www.greenleafsoftware.com
34.29 Capturing Win 95 Network Login User Name

This function will return the Windows user name:

```vba
Declare Function GetUserName Lib "advapi32.dll" _
    Alias "GetUserNameA" _
    (ByVal lpBuffer As String, nSize As Long) As Long

Sub Test()
    MsgBox CurrentUserName
End Sub

Function CurrentUserName() As String
    Dim UserName As String * 100
    GetUserName UserName, 100
    CurrentUserName = _
        Left(UserName, InStr(1, UserName, Chr(0)) - 1)
End Function
```

The following is another approach, that uses a function to return the login name:

```vba
Declare Function GetUserName Lib "ADVAPI32.DLL" _
    Alias "GetUserNameA" _
    (ByVal lpBuffer As String, nSize As Long) As Long

Function UserName() As String
    Dim S As String
    Dim N As Long
    Dim Res As Long
    S = String$(200, 0)
    N = 199
    Res = GetUserName(S, N)
    UserName = Left(S, N - 1)
End Function
```

34.30 Convert To PDF File Via VBA

The simpliest way to create a PDF file via a macro is to first install a PDF print driver. Then, your macro would do the following:

Store the current printer

- Change the printer to the PDF printer
- Print the document
- Change the printer back to the original printer

The following illustrates this task:
Sub CreatePDFFile()
    Dim curPrinter As String
    curPrinter = Application.ActivePrinter
    Application.ActivePrinter = "$PDF-XChange 2.5 on Ne01:"
    ActiveWindow.SelectedSheets.PrintOut
    Application.ActivePrinter = curPrinter
End Sub

To printout multiple pages to one PDF file is not so simple and very dependent on the PDF print
driver. We use PDF-Xchange because it gives the option to append the output to an existing PDF
file (but not automatically). Details on PDF-Xchange can be be found at docu-track.com

Another alternative is found at http://www.rcis.co.za/dale/info/pdfguide.htm.

34.31 How To Display HTML Help Files

At the top of a module, put the following code:

Private Declare Function HtmlHelpTopic Lib "hhctrl.ocx" _
    Alias "HtmlHelpA" _
    (ByVal hwnd As Long, ByVal lpHelpFile As String, _,
    ByVal wCommand As Long, ByVal dwData As String) As Long

Sub ShowHelp()
    'Example of how to use
    ShowHtmlHelp "C:\MYHelpFile.CHM"
End Sub

Sub ShowHtmlHelp(ByVal strHelpFile As String, _
    Optional ByVal strHelpPage As String)
    Const HH_DISPLAY_TOPIC As long = &H0
    On Error Resume Next
    'open the help page in a modeless window - sHelpPage is an optional
    'parameter that allows you to specify a particular HTML page
    'within the chm file. If omitted, the default (index, usually)
    'page is shown.
    HtmlHelpTopic 0&, strHelpFile, HH_DISPLAY_TOPIC, strHelpPage
End Sub

34.32 Turn Off Asterisk As Wildcard

The asterisk is a wildcard character to Excel. To use it as a literal in your code for text matches
or cell searches, prefix it with a tilde: ~*. This also works for other wildcard characters.

34.33 Preventing VBA Help from Resizing the Editor
If you use the help in the visual basic editor, it will resize the editor, which is very irritating. You can fix this by making or changing a registry entry. Run RegEdit, and go to the following key:

HKEY_CURRENT_USER\Software\Microsoft\Office\9.0\Common\HelpViewer

If there is a value named "IsFloating", change its value to 1. If there is not a value with that name, add a DWORD value named "IsFloating", and give it a value of 1.

Restart Excel and VBA, and the editor will not resize when you use help.

34.34 VBA Screen Capture Routines

The following will automate screen captures in Excel by using the win32 api function keybd_event as follows:

```
Declare Sub keybd_event Lib "user32" (ByVal bVk As Byte, ByVal bScan As Byte, ByVal dwFlags As Long, ByVal dwExtraInfo As Long)
Public Const VK_SNAPSHOT = &H2C

Sub PrintScreen()
    keybd_event VK_SNAPSHOT, 0, 0, 0
End Sub
```

Where the second parameter:

bScan = 0 for a snapshot of the full screen

bScan = 1 for a snapshot of the active window.
35. INTERESTING MACRO EXAMPLES

35.1 An Example Of A Rounding Macro

The following macro will prompt the user for the number of digits to round a range of cells. It will then either round the numbers or modify the formulas so that they round.

Sub Round_Numbers()
    Dim digits As Variant
    Dim cell As Range

    'turn off screen updating to prevent flashing screens
    Application.ScreenUpdating = False

    'prompt for a number. The Type:=1 requires a number
    digits = Application.InputBox( _
        prompt:="How many digits do you want to round to?", _
        Title:="Number of digits to round to", _
        Type:=1, _
        default:=3)

    'check to see if cancel was selected
    If TypeName(digits) = "Boolean" Then Exit Sub

    'convert the inputbox output from a text string to a numeric value
    digits = Val(digits)

    'loop through just the selected cells in the used range
    For Each cell In Intersect(Selection, ActiveSheet.UsedRange)

        'see if the cell has a numeric entry
        If Application.IsNumber(cell.Value) Then
            If cell.HasFormula Then

                'if a formula, modify formula to use Round()
                cell.Formula = "=Round(" & _
                    Mid(cell.Formula, 2) _
                    & ", " & digits & ")"

            Else

            End If

        End If

    Next cell
End Sub
'if a number, then just round the value

    cell.Value = Application.Round(cell.Value, digits)
    End If
End If
Next cell
Application.ScreenUpdating = True
End Sub

35.2 Finding Entries That Are Not In A List

The following example compares two selections and highlights the entries in the second selection that are not in the first selection.

Sub FindOddBalls()
    Dim MasterList As Range
    Dim ItemsToMatch As Range
    Dim cell As Range
    Dim FoundRow As Long
    On Error Resume Next

    'get the ranges from the user.
    'The above on error handles cancel being selected
    Set MasterList = Application.InputBox( _
        prompt:="Select the range to look in for a match", _
        Type:=8)

    'if no range supplied, exit macro
    If MasterList Is Nothing Then End

    'get second range

    Set ItemsToMatch = Application.InputBox( _
        prompt:="Select the cells to be checked for a match", _
        Type:=8)

    'if no range supplied, exit macro
    If ItemsToMatch Is Nothing Then End

    'restrict the ranges to the used range on the sheet in case entire rows or columns selected above

    Set MasterList = Intersect(MasterList, _
        ActiveSheet.UsedRange)
    Set ItemsToMatch = Intersect(ItemsToMatch, _
        ActiveSheet.UsedRange)

    'rotate through each cell and see if it is in the first range
For Each cell In ItemsToMatch

' check only non-blank cells

If Application.Trim(cell) <> "" Then

' reset Err for each loop

Err = 0

' use the match function to see if there is a match. Using a value of Zero for the last argument means that an exact match is required

FoundRow = Application.Match(cell.Value, MasterList, 0)

' If a match is found Err stays zero; color the cell in that case

If Err <> 0 Then
    With cell.Interior
        .ColorIndex = 6
        .Pattern = xlSolid
    End With
End If

Next cell

End Sub

35.3 Deleting Leading Tick Marks From A Selection

The following example removes leading tick marks such as those shown below

' 1/2/99

' 02/05/99

from a selection.

Sub DontQuoteMe()
    Dim rng As Range
    For Each rng In Selection
        rng_Formula = rng_Formula
    Next rng
End Sub

If you want to confine the conversions to just date cells, use:

Sub DontQuoteMe()
    Dim rng As Range
    For Each rng In Selection
        If IsDate(rng_Formula) Then

538
35.4 How To Convert Formulas To Absolute References

The following code will convert all formulas on all sheets in the ActiveWorkbook to absolute references:

```vba
Sub ConvertFormulas()
    Dim cell As Range
    Dim Fmla As String
    Dim WS As Worksheet
    Dim cellsToChange As Range

    'turn off alert messages
    Application.DisplayAlerts = False

    'rotate through all the worksheets in the active workbook
    For Each WS In Worksheets
        'initialize cellsToChange each time through
        Set cellsToChange = Nothing

        'only check sheets with entries otherwise SpecialCells
        'will display a message that no cells were found
        If Application.CountA(WS.Cells) > 0 Then

            'set an error trap in case no formula cells
            On Error GoTo errorTrap
            Set cellsToChange = WS.Cells.SpecialCells(xlFormulas)

            'turn off error trap
            On Error GoTo 0

            'rotate through all cells with formulas
            For Each cell In cellsToChange

                'change cell's formula to absolute
                cell.Formula = Application.ConvertFormula(_
                                                   cell.Formula, xlA1, xlA1, xlAbsolute)
        End If
    Next WS
End Sub
```
Next cell
End If
nextSheet:
Next WS
Application.ScreenUpdating = True
Exit Sub
errorTrap:
Resume nextSheet
End Sub

35.5 A Database Modification Example

In this example, the user has a set of data in columns with the first row of the data being labels. The data begins in cell A1. The challenge is to insert a column between column A and B, and add a formula that rations column C values by column D values. Also a title needs to be placed at the top of column B.

you can use the CurrentRegion property to define the range.

Set rng1 = Range("A1").CurrentRegion

To insert a column between columns A and B, you would use the following statement:

Columns(2).Insert

This next statement would put a title at the top of column 2:

Range("B1").Value = "Ratio of C to D"

Now to get the column 2 range that coincide with the data, you would do the following first

Set rng2 = Intersect(rng1,Range("B1").EntireColumn)

But this will include the header row, so you want to resize it to only include row 2 through the last row

Set rng2 = rng2.Offset(1,0).Resize(rng2.Rows.Count-1)

Now to get the ratio of column 3 to column 4 the formula in A2 would be =C2/D2. This can be done in one step and the formula will be adjusted automatically if we use relative arguments

Rng2.Formula = "+=C2/D2"

All together:

Sub ModifyDataBase()
    Dim rng1 As Range, rng2 As Range

    'assign current region to a variable
Set rng1 = Range("A1").CurrentRegion

'insert a column
Columns(2).Insert

'put a title above the column
Range("B1").Value = "Ratio of C to D"

'define a second range of just the cells in the column 2 where formulas are needed

Set rng2 = Intersect(rng1, Range("B1").EntireColumn)

'modify the range variable to exclude the first cell
Set rng2 = rng2.Offset(1, 0).Resize(rng2.Rows.Count - 1)

'insert a formula in the above cells
rng2.Formula = "=C2/D2"
End Sub

35.6 Generating Unique Sequential Numbers For Invoices

Although not a Visual Basic solution, the following can be used to generate a unique number, with each value generated larger than the previous one.

MsgBox Int(Now()*100000)

35.7 Generating Random Numbers

The following procedure shows how to generate a set of not repeating random numbers:

Sub GenerateRandomNumbers()

'declare an array to store a true value if the number has already been generated. Since this is a Boolean array, its values are initially False and can be set to True

Dim UsedList() As Boolean

'declare an array to store the random numbers as they are generated

Dim RndNumList() As Integer
'declare counter variables

Dim Num As Integer
Dim R As Integer
Dim I As Integer

'set a variable equal to the number of numbers to be generated

Dim numberOfNumbersToCreate As Integer
numberOfNumbersToCreate = 20

'specify the max value for the random numbers. The min value is 1

Dim MaxValue As Integer
MaxValue = 100

'redeclare the UsedList array for the maximum number of Boolean variables needed, determined by the maximum value

ReDim UsedList(1 To MaxValue)

'redeclare the array size to for the number of variables to be generated

ReDim RndNumList(1 To numberOfNumbersToCreate)

'make certain that the max value is greater than or equal to the number of random numbers needed

If MaxValue < numberOfNumbersToCreate Then
    MsgBox "The maximum value must be greater than or equal to the number of random numbers needed."
    Exit Sub
End If

'run this For..Next loop to generate the random variables

For Num = 1 To numberOfNumbersToCreate
    Do

    'generate a random number between one and max values

    R = Int(Rnd() * MaxValue) + 1

    'see if the number has been selected. If the value of 'usedListed() is false, then it has not been selected

    If Not UsedList(R) Then

    'if not used, keep it and flag as used
RndNumList(Num) = R
UsedList(R) = True
Exit Do
End If
Loop
Next Num

'write list out to the active sheet
For I = 1 To numberOfNumbersToCreate
    Cells(I, 1).Value = RndNumList(I)
Next
End Sub

35.8 Another Random Number Example

The following example generates a set of random numbers and writes them out to column A of the active sheet.

Sub GenerateUniqueRandomNumbers()
    Dim randomNums As New Collection
    Dim I As Integer
    Dim K As Integer
    Dim minValue As Integer
    Dim maxValue As Integer
    Dim numbersNeeded As Integer

    'set min and max values
    minValue = 1
    maxValue = 100

    'set number needed
    numbersNeeded = 100

    'check to insure there are enough numbers between the
    'min and max values to generate the number required
    If maxValue - minValue + 1 < numbersNeeded Then
        MsgBox "Please check your input!"
        Exit Sub
    End If

    'loop until numbers created
    Do
        I = Rnd() * maxValue
        If I > minValue - 1 And I <= maxValue Then
            'use on error to prevent error message
            'if random number is already in the collection
            On Error Resume Next
            randomNums.Add I
            randomNums(0).Value = I
        End If
    Loop
End Sub
randomNums.Add I, CStr(I)
On Error GoTo 0
' exit when number needed reached
If randomNums.Count = numbersNeeded Then Exit
Do
   End If
Loop

' write results to column A of the active sheet
For K = 1 To numbersNeeded
   Cells(K, 1).Value = randomNums(K)
Next
End Sub

35.9 Deleting Rows Based On Entries In The Row

The following will delete all rows that have the value 17 in column A.

Sub DeleteCertainRows()
   Dim TestRange As Range
   Dim lastRow As Long, firstRow As Long
   Dim i As Long

   ' restrict the range to be tested to just cells in column A that are
   ' in the active sheet's used range
   Set TestRange = Intersect(Range("A:A"), _
      ActiveSheet.UsedRange)

   ' get last row number in the range

   ' get the first row number in the range
   firstRow = TestRange.Cells(1).Row

   ' cycle through the rows from last to first - this is done in case
   ' a row is delete
   For i = lastRow To firstRow Step -1

   ' before testing the value of the cell, make certain it is a
   ' numeric cell. Otherwise an error will occur
   If IsNumeric(Cells(i, 1)) Then
      If Cells(i, 1).Value = 17 Then

' if the value in column A (column 1) is 17, delete the row
If the test you want to run is a string test instead of a numeric test, then use the following statements instead, assume you wanted to deleted all rows that have "ABC" in a cell in column A:

```vba
If TypeName(Cells(i, 1).Value) = "String" Then

'convert the value to upper case and then test
If Ucase(Cells(i, 1).Value) = "ABC" Then

'if the value in column A (column 1) is ABC, delete the row
Rows(i).Delete
End If
End If
End If
```

### 35.10 Counting Unique Values In A Range

Function `CountUniqueValues(InputRange As Range) As Long`  
Dim `cl As Range, UniqueValues As New Collection`  
Application.Volatile  
On Error Resume Next  
For Each `cl In InputRange`  
    UniqueValues.Add cl.Value, CStr(cl.Value)
Next cl  
On Error GoTo 0  
CountUniqueValues = UniqueValues.Count
End Function

`CountUniqueValues(A1:A100)` will count the unique values in the range A1:A100.

### 35.11 Counting Entries In A Filtered Column

This macro assumes that you have data in a column that has been filtered and you wish to return the count of the visible entries.

Sub `CountVisibleEntries()`  
'Set constants  
Const First_Data_Row As Integer = 2  
Const Data_Column As String = "A"

'Declare variables  
Dim lastCell As Range  
Dim anyR As Range
```
'get last visible cell in box column
'assumes no entries below last data row
'also assumes Excel 97-2003, not Excel 2007 or higher
Set lastCell = Cells(65536, Data_Column).End(xlUp)

'set range to cells in box column
Set anyR = Range(Cells(First_Data_Row, Data_Column), lastCell)

'correct range to only visible cells - assumes sheet not protected
Set anyR = anyR.SpecialCells(xlCellTypeVisible)

'display the answer - assumes no blank cells
MsgBox anyR.Cells.Count & " boxes"

End Sub

35.12 Last Row Number and Last Column Number

Before Excel 2007, it was easy to get the last row number or the last column number: the answers were 65536 and 256 respectively. With Excel 2007, the answer, if an Excel 2007 workbook are 1048576 and 1024 respectively. Things get interesting if one opens an Excel 97-2003 workbook in Excel 2007. One can not just assume the answer based on the Excel version.

The following two functions return the last row number and last column number. Just refer to them in your code as needed. Each requires that you supply the worksheet target as an argument. The row number is declared as a Long as an Integer declaration is too small.

Sub Example_Of_How_To_Use()
    Dim lastCol As Integer
    Dim lastRow As Long

    lastCol = LastColumnNumber(ActiveSheet)
    lastRow = LastRowNumber(ActiveSheet)

    MsgBox "Last Column Number " & lastCol
    MsgBox "Last Row Number " & lastRow
End Sub

Function LastColumnNumber(wS As Worksheet) As Long
Dim C As Long
Dim cell As Range
If Val(Application.Version) < 12 Then
    LastColumnNumber = 256
Else
    C = 16384
    On Error Resume Next
    Set cell = wS.Cells(1, C)
    If cell Is Nothing Then
        LastColumnNumber = 256
    Else
        LastColumnNumber = C
    End If
End If
End Function

Function LastRowNumber(wS As Worksheet) As Long
    Dim R As Long
    Dim cell As Range
    If Val(Application.Version) < 12 Then
        LastRowNumber = 65536
    Else
        R = 1048576
        On Error Resume Next
        Set cell = wS.Cells(R, 1)
        If cell Is Nothing Then
            LastRowNumber = 65536
        Else
            LastRowNumber = R
        End If
    End If
End Function