Contents

Who Are These Guys, Anyway?

Well, since a picture's worth a thousand words, here's a rare photograph of the co-conspirators, er, co-founders of PRIME Consulting Group, also pen pals in the production of The Underground Guide to Excel 5.0 for Windows and *The Underground Guide to Microsoft Office, OLE, and* VBA. This appears to be one of the secret entrances to The Underground. Judging by the ambiance, deep in the heart of a teeming industrial sector somewhere in the Western United States ...

Introduction

PRIME 5 is a collection of add-on utilities for Microsoft Excel 5 for Windows, and is distributed as shareware. The only difference between the unregistered and the registered versions is that registered users receive a diskette version that contains no nag screens. Thanks in advance for registering!

PRIME 5 was developed by PRIME Consulting Group, Inc., the firm whose principals Timothy-James Lee and Lee Hudspeth co-authored The Underground Guide to Excel 5.0 for Windows (Addison-Wesley) and *The Underground Guide to Microsoft Office, OLE, and* VBA (Addison-Wesley). In fact, *The Underground Guide to Excel* was the genesis of PRIME 5. PRIME Consulting Group, Inc. (PCG) co-authored the Microsoft Education Services (MES, formerly Microsoft University) WordBasic development course. Jim and Lee served as Technical Editors, along with the indefatigable Scott Krueger, on Leonhard and Chen's *Hacker's Guide to Word for Windows*, first edition (Addison-Wesley). PCG developed PRIME 4, a Word 2 / PackRat 4 / FAXit integration application. PCG developed and self-published *Hacker's Guide to PackRat 4*.

PRIME Consulting Group, Inc. is a member of the Microsoft Solution Provider program, and is both a certified Microsoft Consulting Partner and Microsoft Training Partner. The firm's staff are certified to teach the following MES courses:

- *Introduction to Programming for Microsoft Windows Using Microsoft Visual Basic 3.0*
- *Programming in Microsoft Visual Basic 3.0*
- *Application Development Using Microsoft Word*

In addition, PRIME Consulting Group offers its own courses.

- *Putting OLE 2, VBA, and DDE to Work: Integrating Windows Applications*
- *The Underground Guide to Excel 5, Live!*
- *The Underground Guide to Word 6, Live!*

For more information about PRIME Consulting Group's consulting, training, and development services, including a complete courseware catalog and schedule, please call 310-318-5212.

Registration

PRIME 5 is shareware. You give it a whirl and if you like it, we'd very much appreciate it if you'd buy it within 30 days of first starting to tinker with it. The benefits of registering include:

- No nag screens.
- The most recent versions of all PRIME 5's components.
- Free telephone support via a toll call.
- Additional free support on CompuServe.

Thanks in advance for registering!

For single-user license agreement details, see License Agreement.

For site licensees: if your organization uses more than one copy of PRIME 5 at a time, you should license as many copies of PRIME 5 as you have copies of Microsoft Excel 5 for Windows. It's that simple.

PRIME 5 is $29.95US plus $4.50US shipping and handling, $9.50US outside North America. Site licenses (more than ten users) are available at considerable savings.

You can register right now by calling 800-317-5525, or 314-965-5630. We take Mastercard, Visa, American Express, purchase orders, and try hard to ship within 24 hours. To register by mail, send a check (in U.S. dollars, please) to:

PRIME Consulting Group, Inc.
c/o Advanced Support Group, Inc.
11900 Grant Place
Des Peres, Missouri USA 63131

License Agreement

This License Agreement is entered into between PRIME Consulting Group, Inc. as Licensor and recipient of enclosed diskette and related materials as Licensee.

This is a legal agreement between Licensor and Licensee. If Licensee does not agree to the terms of this Agreement, Licensee shall promptly return the disk(s) and accompanying material to Licensor.

Licensor hereby grants to Licensee the right to install the PRIME 5 components − including but not limited to DLLs, macros, workbooks, add-ins, and XLLs ("PRIME 5") − on any one (1) personal computer. If Licensor installs PRIME 5 on a network server, Licensor agrees that one and only one user shall have access to that specific installed copy of PRIME 5. Use of PRIME 5 by more than one user is a violation of the terms of this agreement.

Use of PRIME 5 covered by this license by Licensee constitutes acceptance by Licensee of the terms and conditions of this agreement and Licensee agrees to be bound by the terms of same.

For the purpose of this agreement both parties hereby agree that the word "use" means loading files containing PRIME 5 features (PRIME5.XLA or any other related workbook, add-in, DLL, or XLL supplied with PRIME 5) into RAM as well as installation on a hard disk or other storage device. In no event shall the term "use" include the right to modify, update, publicize, publish, republish, copyright, sell or transfer either this license, or the rights granted by this license, or the technology thereunder in relation thereto.

This license is limited to Licensee, which may access PRIME 5 from a hard disk, or any other method Licensee chooses to utilize, so long as Licensee otherwise complies with the terms of this License Agreement.

Licensee agrees and acknowledges that PRIME 5 was specifically written for Microsoft Excel for Windows version 5.0 through 5.0c and Microsoft Windows 3.1, and that PRIME 5 may or may not work with any other version of these applications and operating systems/environments, either earlier or later.

LICENSOR DISCLAIMS, AND LICENSEE AGREES AND ACKNOWLEDGES THE DISCLAIMER OF ALL WARRANTIES, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. IN NO EVENT SHALL LICENSOR, OR ANY PERSON, FIRM OR ENTITY RELATED OR AFFILIATED THEREWITH, BE LIABLE FOR ANY DAMAGES WHATSOEVER INCLUDING, WITHOUT LIMITATION, DAMAGES FOR LOSS OF BUSINESS PURPOSE, BUSINESS INTERRUPTION, BUSINESS INFORMATION, OR ANY OTHER PECUNIARY LOSS WHATSOEVER, ARISING OUT OF THE USE OF, OR THE INABILITY TO USE, THE SUBJECT OF THIS LICENSE.

Licensee acknowledges that it has no rights whatsoever to copyright, in any country, anything in relation to PRIME 5, and promises and represents that it shall not attempt to do so, the same being exclusively vested in Licensor.

In the event Licensee breaches any term of this License Agreement it shall pay to Licensor, in addition to damages, all costs of suit and/or appeal to enforce the terms of this License Agreement, including, but not limited to, reasonable attorney's fees.

Both parties agree that this Agreement represents the complete and final agreement and understanding between the parties and all other prior and/or contemporaneous understandings or agreements are merged herein and shall have no further force and/or effect, and, that this Agreement is an "Integrated" agreement, and shall be governed exclusively by the laws of the United States of America and the State of California.

Acknowledgments

PRIME 5 was conceived and developed by PRIME Consulting Group, Inc.

We'd like to thank Woody Leonhard for his boundless support and encouragement. Ad infinitum. Amen.

Kudos and back-slapping are certainly due to Bill Edmett and Jonathan Sachs for exquisite programming, design, and technical writing contributions to PRIME 5. Bill implemented Cell Protection Viewer. Jonathan implemented Zoomer, and many of his procedures from PRIME 6 for Word's API ported quite nicely to Excel VBA, yes indeedy.

Thanks to Brian Gray of Aries Entertainment for a superb job on the setup routine. (Aries Multimedia is located in Fresno, California, 209-222-9171, CompuServe 73750,253.)

PRIME 5 wouldn't be what it is without the input we received from our beta testers (any PRIME 5 errors, goofs, or general misbehaviors are of course credited solely to PRIME Consulting Group), listed here in alphabetical order ...

Dick Apcar, Mark Bennett, Brad Brunell, Roy Busdiecker, Paul Cameron, Vince Chen, Peter Deegan, Stephen Edgington, Michael Gordon, Kurt Haeusermann, Arthur Hixon, Michael J. Holden, Stephen James, Joel Kanter, Doug Klippert (Doug, please accept our sincerest apologies for accidentally omitting your name from the PRIME 6 for Word help file's beta acknowledgments section), John Kois, Ray Lassiter, Woody Leonhard, Jeffrey Leyser, Justin Maas, Scot Maga, Tom Manuel, Ken Mocabee, Dick Moffat, Erik Mogensen, Ellen Nagler, Rob Perelli-Minetti, Jim Powell, Adam Rodman, Kurt Shadle, Terry Sherb, Paul Solyn, Dan Strange, Stephen Stuart, Karl Swensson, Pete Thompson, and Bill Walton. (If we left you out or misspelled your name, sorry, let us know and we'll send you some home-baked fudge, OK?)

Thanks to Tailor-Made Marketing and Advertising Programs for marketing and public relations consulting services par excellence. (Tailor-Made is located in Hermosa Beach, California, 310-318-7099).

We used WexTech Systems' amazing Doc-To-Help to produce this *User's Guide* and help file for PRIME 5.

Related Topics:

Buy PRIME 6 for Word Now (Puh-leez)!

And Don't Forget to Buy WOPR 6 for Word Too!

Buy PRIME 6 for Word Now (Puh-leez)!

From PRIME Consulting Group, Inc., the perfect compliment to Word 6 includes utilities like *Fast Spell Checker* for quicker spell checking with all misspelled words in a document available in a single list. *FileNew* for complete template management. *AutoText Lister* for viewing local and global AutoText contents and statistics side-by-side. *Window Manager* for one-stop window management. *Document Variable Manager* for adding, viewing, modifying, deleting or cataloging those pesky document variables. *Bookmark Manager*, the definitive utility for dealing with bookmarks. *Zoomer, Macro Manager, Toolbar Lister, ResetChar, ProgMan Icon Generator, 36 custom WordBasic subroutines and functions,* a list of Word and WordBasic bugs and workarounds, et cetera.

List price is $39.95 (plus $4.50 shipping and handling in the USA and Canada; $9.50, all other countries). *Just mention that you're a registered owner of PRIME 5 for Excel and get $5.00 off the price of* PRIME 6 for Word*!*

Call Advanced Support Group at 800-788-0787 or 314-965-5630.

And Don't Forget to Buy WOPR 6 for Word Too!

Call Advanced Support Group at 800-659-4696 or 314-965-5630.

A synopsis of WOPR 6's features:

- Enveloper
- File Manager
- Stellar Spellar
- Toolbar Editor
- File New
- 2x4
- TSS Address Book Lite
- Clip Art Scrapbook
- WinBreak Lite
- and a dozen or so LittleWOPRs

Minimum Requirements

To successfully run PRIME 5 you need the following.

- Microsoft Excel for Windows 5.0 or later (to find out about or order maintenance releases of Excel call Microsoft at 800-426-9400; see also Custom Menu Item and Toolbar Button Deficiencies in Excel 5.0 and 5.0a)
- PRIME 5.0 for Excel is compatible with Excel 5.0, 5.0a, and 5.0c (all 16-bit versions) running under all currently available Windows operating systems:
  a) Windows 3.1

  b) Windows for Workgroups 3.1 and 3.11

  c) Windows NT 3.5

  d) Windows 95
- A separate version of PRIME 5 is available specifically for 32-bit Excel 5. You can order "PRIME 5.0 for Excel NT/95" directly by calling Advanced Support Group at 800-317-5525 (314-965-5630 outside the U.S.).

Related Topics:

Custom Menu Item and Toolbar Button Deficiencies in Excel 5.0 and 5.0a

Custom Menu Item and Toolbar Button Deficiencies in Excel 5.0 and 5.0a

Excel 5.0 and Excel 5.0a (the "a" suffix indicates a maintenance release) are missing several important visual features related to custom menu items and custom toolbar buttons. These custom objects should inherit (a) the status bar text of their assigned macros, (b) the help context ID of their assigned macros, and (c) the custom help file's name. (When you click on a custom toolbar button with Excel's Help button − and that button has been assigned to a macro − if the macro has a "help context ID" then you'll immediately see the appropriate help file topic for that custom button. The same effect occurs when you select a custom menu item then press F1.) But they do not inherit these important properties in Excel 5.0 and 5.0a. Instead, these objects display inappropriate values on the status bar and, when clicked with the Help button, go to Excel's help file instead of the correct custom help file.

These very useful features only work properly in Excel 5.0c. As of this writing "c" is the very latest maintenance release. We urge you to upgrade to 5.0c by calling Microsoft at 800-426-9400. It's a free upgrade.

How to Install PRIME 5

PRIME 5 is installed via an automated setup routine (SETUP.EXE). For instructions on how to uninstall PRIME 5, see How to Uninstall PRIME 5.

1. Exit Excel if it's already running.
2. Go to Program Manager, select File then choose Run.
3. Type "a:\setup.exe" (without the quotes) and press Enter or click OK. (If you're installing from your b: diskette drive, type "b:\setup.exe".)
4. Follow the instructions on the installation screen.
5. When installation is complete, it leaves you in Notepad to review the latest PRIME 5 README.TXT file. Exit Notepad when you've reviewed any relevant late-breaking news.
6. Start Excel.

   You'll see PRIME 5's toolbar displayed. If for some reason you don't see it, select View on the main menu bar, select Toolbars, scroll down the Toolbars list until you see "PRIME 5.0," select the PRIME 5.0 check box, then click OK.

Each PRIME 5 utility has its own menu item hanging off various standard menu bar commands (this applies to the built-in Worksheet, Chart, and Visual Basic Module menu bars only), as well as its own button on the PRIME 5.0 toolbar.

Custom Toolbars

PRIME 5 comes with its own custom toolbar called "PRIME 5.0." This toolbar is automatically created on the way in and destroyed on the way out. However, PRIME 5 does store this toolbar's last position so when it is recreated – the next time you start Excel – it will reappear where you left it. Any buttons you add to or delete from this toolbar will remain added or deleted *for the life of the current Excel session only*; these button additions/deletions will be discarded at the end of the current Excel session.

When PRIME 5 starts for the first (and only the first) time, it creates three custom toolbars.

- Underground Guide to Excel Auditing
- Underground Guide to Excel Formatting
- Underground Guide to Excel Standard

The configuration of these toolbars matches those recommended in our book The Underground Guide to Excel 5.0 for Windows (Lee Hudspeth and Timothy-James Lee, Addison-Wesley, ISBN 0-201-40651-9). These toolbars and their related features are described in detail in the following sections. For more information on the macros assigned to these custom toolbar buttons, see The Macros Behind the Buttons.

Related Topics:

Swapping Excel's Built-in Toolbars for the Underground Guide Versions

"Underground Guide to Excel Auditing" Custom Toolbar

"Underground Guide to Excel Formatting" Custom Toolbar

"Underground Guide to Excel Standard" Custom Toolbar

The Macros Behind the Buttons

Swapping Excel's Built-in Toolbars for the Underground Guide Versions

We may be biased, but we're going to suggest that you hide Excel's built-in Standard, Formatting, and Auditing toolbars and activate ours (the Underground Guide to Excel versions). Here's how.

1. Select View / Toolbars / clear the Standard check box only if it is checked / click OK. This hides Excel's Standard toolbar. (Note: you can combine steps 1-3 into one step by not clicking the OK button until you've hidden the last toolbar in the series.)
2. Select View / Toolbars / clear the Formatting check box only if it is checked / click OK.
3. Select View / Toolbars / clear the Auditing check box only if it is checked / click OK.
4. Typically, upon first installing PRIME 5, you'll see the TUGExcel Standard, Formatting, and Auditing toolbars floating in the upper left portion of your display space. If you see them right now then you don't have to make them visible and you can skip to the next step. Otherwise, Select View / Toolbars / select the Underground Guide to Excel Standard check box, repeat this step for the Underground Guide to Excel Formatting toolbar and the Underground Guide to Excel Auditing toolbar, and click OK when you're done.
5. You can now position any or all of the three visible Underground Guide to Excel toolbars as you wish.

"Underground Guide to Excel Auditing" Custom Toolbar

The following table shows what command or feature is called by each button on the Underground Guide to Excel Auditing custom toolbar. *Note: the button numbers shown here ignore gaps.*

Button Number	Command/Feature Called
1	built-in Trace Precedents
2	built-in Remove Precedent Arrows
3	built-in Trace Dependents
4	built-in Remove Dependent Arrows
5	built-in Remove All Arrows
6	built-in Trace Error
7	built-in Attach Note
8	built-in Show Info Window
9	PRIME Formula Flipper
10	PRIME Notation Style Flipper
11	PRIME Edit Go To Special

"Underground Guide to Excel Formatting" Custom Toolbar

The following table shows what command or feature is called by each button on the Underground Guide to Excel Formatting custom toolbar. *Note: the button numbers shown here ignore gaps.*

Button Number	Command/Feature Called
1	built-in Style
2	built-in Font
3	built-in Font Size
4	built-in Bold
5	built-in Italic
6	built-in Underline
7	built-in Align Left
8	built-in Center
9	built-in Align Right
10	built-in Center Across Columns
11	built-in Borders
12	built-in Drop Shadow
13	built-in Light Shading
14	built-in Toggle Grid
15	built-in Full Screen

"Underground Guide to Excel Standard" Custom Toolbar

The following table shows what command or feature is called by each button on the Underground Guide to Excel Standard custom toolbar. *Note: the button numbers shown here ignore gaps.*

Button Number	Command/Feature Called
1	PRIME New Workbook
2	built-in Open

3	built-in Save
4	PRIME File Close
5	PRIME Exit Excel
6	built-in Print
7	built-in Print Preview
8	built-in Spelling
9	built-in Set Print Area
10	built-in Cut
11	built-in Copy
12	built-in Paste
13	built-in Format Painter
14	built-in Undo
15	built-in Repeat
16	built-in Paste Names
17	built-in AutoSum
18	built-in ChartWizard
19	built-in Text Box
20	built-in Shape
21	built-in Zoom Control
22	built-in TipWizard
23	built-in Help

The Macros Behind the Buttons

Since The Underground Guide to Excel 5.0 for Windows was published we've fine-tuned the macros behind the TUGExcel toolbar buttons, primarily just incorporating calls to various PRIME 5 API procedures. Here's an updated source code listing.

(Portions of the following sections reprinted by permission of Addison-Wesley from *The Underground Guide to Excel 5.0 for Windows*).

Related Topics:

New Workbook (TUGExcel Standard)

File Close (TUGExcel Standard)

Exit Excel (TUGExcel Standard)

Formula Flipper (TUGExcel Auditing)

Notation Style Flipper (TUGExcel Auditing)

Edit Go To Special (TUGExcel Auditing)

New Workbook (TUGExcel Standard)

The first three buttons on Excel's ship-with Standard toolbar deal with files, creating books, opening existing books, and saving files to disk. Open and Save work fine, but we had best deal with the way the first button, New, behaves. This button gives you a new book, bang! "What's wrong with this," you ask? Well, you should be using templates, and you should be able to click on this button and choose what template to use when creating a new book instead of being forced to accept the default book template.

Here's our replacement for Excel's ship-with "new file/workbook" button.

```
' Copyright © 1994-95 PRIME Consulting Group, Inc.
Sub sFileNew()
    ExecuteExcel4Macro "NEW?()"
End Sub
```

File Close (TUGExcel Standard)

Did you notice anything odd about the first three buttons of Excel's ship-with Standard toolbar? What's missing from this group? Well, if you're like us, you will probably close every file you open at some point, so how about a FileClose button? Microsoft left it out of Excel. They put one in WinWord, but they left it off the toolbar; you have to put it on yourself. In Excel they left it out, period, as in there ain't no such animal.

Here's the code behind our button for closing files/workbooks.

```
' Copyright © 1994-95 PRIME Consulting Group, Inc.
Sub sFileClose()
    If fblnNullMenuState Then
        Beep
        Exit Sub
    End If
    On Error Resume Next
    ActiveWorkbook.Close
End Sub
```

Exit Excel (TUGExcel Standard)

This is a completely new addition, and if it had been a snake, well, we'd be snake-bit. Thanks to Bill Walton for requesting such a useful addition to the TUGExcel Standard toolbar.

```
' Copyright © 1994-95 PRIME Consulting Group, Inc.
Sub sFileExit()
    Application.Quit
End Sub
```

Formula Flipper (TUGExcel Auditing)

You can flip a sheet between formulas and results using the CTRL + ` (back apostrophe) key combination. But since we think it is such a useful tool, we've added it to the Auditing toolbar as a custom button.

```
' Copyright © 1994-95 PRIME Consulting Group, Inc.
Sub sFormulaFlipper()
    If fblnNullMenuState Then
        Beep
        Exit Sub
    End If
    intRC = fintActiveSheetType
    If intRC = 1 Or intRC = 4 Or intRC = 5 Then
        ' DisplayFormulas applies only to all worksheet sheet types
        On Error GoTo EndMacro
        ActiveWindow.DisplayFormulas = Not ActiveWindow.DisplayFormulas
    Else
        Beep
    End If
    Exit Sub
EndMacro:
End Sub
```

Notation Style Flipper (TUGExcel Auditing)

Good ol' R1C1 notation is a great tool in the spreadsheet auditor's kit. In fact, it is so useful that – you guessed it! – we've added a button to the Auditing toolbar to toggle worksheets between A1 and R1C1.

```
' Copyright © 1994-95 PRIME Consulting Group, Inc.
Sub sNotationStyleFlipper()
    If fblnNullMenuState Then
        Beep
        Exit Sub
    End If
    On Error GoTo EndMacro
    ' since ReferenceStyle applies to the Application object, don't
    '   need to trap for any particular sheet type as the active sheet
    If Application.ReferenceStyle = xlA1 Then
        Application.ReferenceStyle = xlR1C1
    Else
        Application.ReferenceStyle = xlA1
    End If
    Exit Sub
EndMacro:
End Sub
```

Edit Go To Special (TUGExcel Auditing)

Last, but certainly not least, in our arsenal of trouble trappers are the features provided in the Edit Go To Special dialog box. This dialog box presents an impressive array of tools for finding problems in spreadsheet models.

Go To Special is a way to select a number of cells within a worksheet simultaneously, based on specific criteria. Set the criteria by selecting one of several radio buttons (sure, sure, "option" buttons if you prefer). After you have the cells selected, you can use the Tab key to jump from selected cell to selected cell through the worksheet.

Ordinarily, getting to the Edit Go To Special dialog box is a little convoluted. You have to pull down the Edit menu, click Go To, and then, from the Go To dialog box, click the Special button. Whew, finally the Go To Special dialog box is displayed. Here's the code behind the custom button that makes this a single-click joy.

```
' Copyright © 1994-95 PRIME Consulting Group, Inc.
Sub sEditGotoSpecial()
    If fblnNullMenuState Then
        Beep
        Exit Sub
    End If
    intRC = fintActiveSheetType
    If intRC = 1 Or intRC = 3 Or intRC = 4 Or intRC = 5 Then
        ' native EditGoTo Special OK only for all worksheet sheet
        '   types and dialog sheets
        On Error GoTo EndMacro
        Application.Dialogs(xlDialogSelectSpecial).Show
    Else
        Beep
    End If
    Exit Sub
EndMacro:
    If Err = 1004 Then
        MsgBox "No cells found.", vbExclamation
    End If
End Sub
```

File Inventory

A file inventory follows. Each file's required storage path is shown in parentheses.

- CSLIDER.DLL (Windows System) – custom control used by ZOOMXL.XLL
- CTL3D.DLL (Windows System) – substitutes 3D controls for Windows' standard "flat" controls
- CUSTOM.MPP (master diskette) – sample Project source file for use with OLE Automation Routine (for more information see sOLEAutoRoutine)
- CUSTOM.XLS (master diskette) – sample Excel source file for use with OLE Automation Routine (for more information see sOLEAutoRoutine)
- DDEROUTI.XLS (master diskette) – sample Excel source file for use with various DDE procedures (for more information see Dynamic Data Exchange)
- DUMMY1.DOC (master diskette) – sample Word source file for use with OLE Automation Routine (for more information see sOLEAutoRoutine)
- OAROUTIN.XLS (master diskette) – source code for OLE Automation Routine (for more information see sOLEAutoRoutine)
- P5UNINST.XLS (master diskette) – open this Excel workbook to uninstall PRIME 5 (for more information see Uninstall PRIME 5 Automatically Using P5UNINST.XLS)
- PRIME5.XLA (Excel's Library Path) – add-in where all the utilities live
- P5.HLP (Windows) – help file
- README.TXT (master diskette) – general last-minute comments and stuff
- ZOOMXL.XLL (Excel's Library Path) – Zoomer XLL

Technical Support

For technical support call 314-965-5630, fax 314-966-1833, or use CompuServe (type GO WOPR).

How to Uninstall PRIME 5

You can uninstall PRIME 5 automatically or manually. Follow the steps in the subsequent sections as appropriate.

Related Topics:

Uninstall PRIME 5 Automatically Using P5UNINST.XLS

Uninstall PRIME 5 Manually

Uninstall PRIME 5 Automatically Using P5UNINST.XLS

To remove PRIME 5 from your system using our automated uninstall application, follow these steps.

1. Start Excel if it isn't already open.
2. Open P5UNINST.XLS. (Note: if you want to edit this project for any reason, simply select Tools / Protection / Unprotect… for the workbook and both sheets. The passwords are empty strings.)
3. Uninstall PRIME 5 for Excel as per the instructions therein.
4. Exit Excel. *This step is important due to the way Excel's own Add-in Manager updates EXCEL5.INI.*

Uninstall PRIME 5 Manually

To manually remove PRIME 5 from your system, follow these steps.

1. Start Excel if it isn't running.
2. Select Tools / Add-ins / clear the "PRIME 5 for Excel" check box / OK.
3. Select Tools / Add-ins / clear the "PRIME Zoomer for Excel 5" check box / OK.
4. Close Excel if it's still running.
5. Remove PRIME5.XLA from Excel's Library Path.
6. Remove P5.HLP from Windows.
7. Remove ZOOMXL.XLL from Excel's Library Path.

To remove PRIME 5 from memory while Excel is still running, select Tools / Add-ins / clear the "PRIME 5 for Excel" check box / OK. Repeat this procedure for the "PRIME Zoomer for Excel 5" check box.

You Can Call PRIME 5 Custom VBA Functions and Subroutines

This section contains Application Programming Interface (API) specifications for the VBA functions and subroutines stored in PRIME 5's public procedures.

Note that for simplicity we have omitted from each procedure's section title the required "As type" argument components . The true, correct argument typing information is displayed in the body text immediately following each section title.

Additional notes:

1. For a listing of PRIME 5's custom VBA procedures by category, see Custom VBA Procedure Categories.
2. For a summary of PRIME 5's custom VBA procedures, see Custom VBA Procedure Summary.
3. Each example source code block comes with its own **Option Explicit** statement in order to reinforce the importance of using VBA's strong data typing capabilities.
4. Each example source code block comes with its own module-level declarations. Many of these variable and constant declarations appear redundant because they would ordinarily be declared globally. But for your ease of use and testing we're supplying each test procedure with turnkey declarations.
5. To copy example source code directly from the help file, select Edit / Copy / select the code statements of interest / click the Copy button. Now the code is on the clipboard and you can paste it directly into your VBA module.
6. See How to Establish a Direct Reference to PRIME5.XLA for important information about how to use PRIME 5's public procedures from within your own procedures. PRIME 5's function procedures are stored in a module called modFunctions, and PRIME 5's subroutines are stored in a module called modSubroutines.
7. Many of the OLE Automation procedures in PRIME 5's API are discussed in even more detail (including source code listings) in The Underground Guide to Microsoft Office, OLE, and VBA. Also, a stripped-down version of Name Assistant is used as an exercise in building a custom dialog-box-based application in the book's Chapter 6 (including source code listings).
8. Function procedures are all stored in PRIME5.XLA's module called modFunctions.
9. Subroutine procedures are all stored in PRIME5.XLA's module called modSubroutines.
10. All procedure listed in this section are declared as Public.

Related Topics:

How to Establish a Direct Reference to PRIME5.XLA

Coding and Naming Conventions

fblnDDEFileStatus(strSourceFileName, strShortTopic, intSysChanNum)

fblnIsValidToolbarName(strToolbarName)

fblnIsWorkbookOpen(strFullName)

fblnNullMenuState

fblnObjectExists(objCollection, vntIdentifier)

fblnOnlyOneTrueElement(vntArrayOfBooleans)

fintActiveSheetType

fintAppLoad(strModuleName, strShellName)

fintFirstTrueElement(vntArrayOfBooleans)

fintOAAppLoad(objOAServer, strOAServerName, strModuleName)

fintOAFileStatus(objOAServer, strShellName, strSourceFileName)

fintSheetType(shtObject)

fintSplitVertical(intPercent)

fintUsageCount(hMod)

fintWindowCountHidden

fintWorkbookCountHidden(strHiddenWorkbookNames())

flngBinSearch(strList(), lngCount, strVal)

flngBinSearchInit(strList(), lngCount, strVal)

flngDeleteFromArray(strArray(), lngSize, lngDeletePt)

fobjSelectedObject(objCollection, strListControl(), objListControl)

fstrActiveWorkbookName(intPathName)

fstrFileNameFromPath(strSourceFileName)

fstrGetCellContents(vntRange)

fstrGetHostTitleBar

fstrGetRangeAddress(strRangeName, intNotation)

fstrGetRangeFromRefersTo(strRefersTo)

fstrGetSheetFromRefersTo(strRefersTo)

fstrPathFromParts(strPath, strFile)

fstrReplaceMarker(strMsg, strReplacement)

fvntGetCellResults(vntRange, blnFormatting)

fvntMax(x, y)

fvntMin(x, y)

fvntOAFileNames(objCollection)

fvntOAWindowFileNames(objCollection)

fvntWorkbookNames

sActiveWorkbookNames(strNames())

sActiveWorkbookSheetNames(strArray(), intOrder)

sGetToolbarButtonInfo(strTheToolbarName, strButtonInfo())

sInsertInArray(strArray(), lngSize, lngInsertPt, strElement)

sMsgStrArray(strArray(), intElements, strName)

sNotImplemented(strDboxTitle, strPrompt)

sOAGetOpenFiles(objOAServer, strShellName, strFileNames())

sOAShutDownApp(objOAServer, strShellName, intAppStatus)

sOLEAutoRoutine

sShellSort(strArray())

sShutDownApp(intSysChanNum, strShellName intAppStatus)

sWaitCursor(blnState)

How to Establish a Direct Reference to PRIME5.XLA

To use PRIME 5's public procedures from within your own procedures, you must first establish a reference to them. This is the way Excel is designed. If an XLA (add-in) is installed but there's no reference to it, you can run all the add-in's utilities but can't refer directly to its procedures. If an add-in is installed *and there is a reference to it*, you can run its utilities and refer directly to its public procedures. Here's how to establish a reference to the PRIME 5 add-in.

1. Select Tools / References.
2. Locate PRIME5.XLA in the Available References list (towards the bottom of the list).
3. Select PRIME5.XLA.
4. Click OK.

Note that once you establish a reference, you don't have to do so again in the current or any future Excel session (unless you manually clear the reference, of course).

Here's an example of a reference to a PRIME 5 public function procedure.

```
strPrompt = fstrReplaceMarker("The answer is <<>>.", "LtCmdrData")
```

If you have your own fstrReplaceMarker() procedure in the active workbook and want to specifically refer to ours instead, then use the following syntax.

```
strPrompt = [PRIME5.XLA].modFunctions.fstrReplaceMarker("The answer is
<<>>.", "LtCmdrData")
```

PRIME 5's function procedures are stored in a module called modFunctions, and PRIME 5's subroutines are stored in a module called modSubroutines.

Coding and Naming Conventions

This discussion of conventions involves

- variable names
- procedure names
- control names

(Portions of this section and sub-sections reprinted by permission of Addison-Wesley from *The Underground Guide to Microsoft Office, OLE, and* VBA).

Related Topics:

Variable Naming Conventions

Procedure Naming Conventions

Control Naming Conventions

OLE Automation Object Variable Naming Conventions

Variable Naming Conventions

To maintain order and improve readability as projects reach a certain critical "confusion mass," we strongly recommend you follow the naming and other coding conventions outlined in the *Microsoft Office Developer's Kit* (ODK). The first step to take on the path to consistent use of conventions is in naming variables and functions. The conventions shown in the following table are straight out of the ODK.

Data Type	**Prefix**	**Example**
Boolean	bln	blnFound
Currency	cur	curRevenue
Date (Time)	dat	datStart
Double	dbl	dblTolerance
Error	err	errOrderNum
Integer	int	intQuantity
Long	lng	lngDistance
Object	obj	objCurrent
Single	sng	sngAverage
String	str	strFName
User-defined type	udt	udtEmployee
Variant	vnt	vntCheckSum

By convention, user-defined constant names are in all capitals, like this.

```
Const MSG_HELLO = "Hello World"
```

(Historical note – A small part of these conventions can be found in the *Microsoft Visual Basic 3.0 Programmer's Reference*. For what it's worth, as best as our moles can determine these conventions started out somewhere deep in the labyrinths of Microsoft Consulting Services, Microsoft's mysto consulting division. They then found their way into Microsoft Education Services' Visual Basic 3 courseware and a Knowledge Base article, and now have come home to roost in the ODK for all the world to see.)

Procedure Naming Conventions

Note that you should use these data type prefixes for *both* variables and functions. Remember that a function returns a value, and that the return value must be *typed* (the default is Variant). No data type prefix is required for a subroutine because it doesn't itself return a value.

We prefer an additional convention whereby the single-letter "f" prefix is placed at the beginning of a function's name, and the single-letter "s" prefix for a subroutine's name. Thus you might end up with a function named "fblnDDEFileStatus()" where "f" is for function, "bln" is for a Boolean return type, and "DDEFileStatus" is the descriptive portion of the function name.

Variant functions deserve special mention. You can get away with dropping the `As [type]` component of the opening Function statement, in which case VBA internally declares it to be a Variant function. However, we suggest you not get lazy! In the interest of clarity if you want to use the Variant type you should explicitly include As Variant and be sure to name the function with the "f" and "vnt" prefixes like this.

```
Function fvntIAmVariant() As Variant
    fvntIAmVariant = Array("Underground Guide to Excel", _
        "Underground Guide to Office, OLE 2, & VBA", _
        "Underground Guide to Unix", _
        "Underground Guide to Word")
End Function
```

Control Naming Conventions

The ODK lists control naming conventions on pages 251-2 of Appendix B.

OLE Automation Object Variable Naming Conventions

In summary, here are the components of our multi-application OLE Automation object variable naming convention. For a more extensive discussion see *The Underground Guide to Microsoft Office, OLE, and* VBA.

1. A one-letter abbreviation designating the OLEAuto server application – "e" for Excel, "p" for Project, "v" for Visual Basic, and "w" for Word (all in lower-case).
2. The characters "obj" to designate this is a generic object type variable (all in lower-case).
3. The three-consonant abbreviation of the OLEAuto server's object type (normal capitalization), for example, "Dlg" for a DialogSheet object.
4. A discretionary name (normal capitalization), for example, "ConfRoom101" to refer to the resource known around the water cooler as Conference Room 101.
5. The exception to the three-consonant abbreviation rule is our use of "App" for the ubiquitous Application object, instead of the rather awkward "Ppl" mandated by strictest adherence to the letter of the law. ("Ppl" represents the first three consonants of the word "Application" with the first letter capitalized for distinction.)

fbInDDEFileStatus(strSourceFileName, strShortTopic, intSysChanNum)

```
Function fbInDDEFileStatus(strSourceFileName As String, strShortTopic As
String, intSysChanNum As Integer) As Boolean
```

Determine if a particular file is currently loaded in the server application. This function assumes you've already started a valid conversation on *intSysChanNum*.

Inputs:

- *strSourceFileName* - the fully-qualified topic's filename
- *strShortTopic* - special case for Excel 5 where sheets are represented like "[book_name]sheet_name", for example, "[CUSTOM.XLS]CUSTOM"
- *intSysChanNum* - channel number of initial System topic

Returns: Boolean.

Example source code:

For an example of how to use this procedure, see DDEROUTI.XLS from your master diskette.

See also

fintAppLoad(strModuleName, strShellName)

sShutDownApp(intSysChanNum, strShellName intAppStatus)

fblnIsValidToolbarName(strToolbarName)

```
Function fblnIsValidToolbarName(ByVal strToolbarName As String) As
Boolean
```

See if a given string is a valid toolbar name. Ported from PRIME 6.0's WordBasic global procedure library (renamed from fIsValidToolbarName).

Inputs:

- *strToolbarName* - the toolbar's name

Returns: Boolean.

Example source code:

```
' Copyright © 1994-95 PRIME Consulting Group, Inc.
Option Explicit
Dim strQuote As String
Sub Test_fblnIsValidToolbarName()
    Dim strTemp As String
    strQuote = Chr(34)
    strTemp = "TUG Is Cool"
    strPrompt = "PRIME fblnIsValidToolbarName(" & _
        strQuote & strTemp & strQuote & _
        ") returns " & fblnIsValidToolbarName(strTemp)
    MsgBox strPrompt
End Sub
```

See also

sGetToolbarButtonInfo(strTheToolbarName, strButtonInfo())

fblnIsWorkbookOpen(strFullName)

```
Function fblnIsWorkbookOpen(strFullName As String) As Boolean
```

Indicates whether a specific fully-qualified workbook is open right now. The caller is responsible for passing a valid fully-qualified name. NOTE: Providing only an 8.3 filename in *strFullName* will return False as this function is looking at the FullName property.

Inputs:

- *strFullName* - the fully-qualified name of the workbook

Returns: Boolean.

Example source code:

```
' Copyright © 1994-95 PRIME Consulting Group, Inc.
Option Explicit
Sub Test_fblnIsWorkbookOpen()
    Dim strTestXLSName As String
    strTestXLSName = "C:\FOOBAR\FOOBAR.XLS"
    MsgBox fblnIsWorkbookopen(strTestXLSName)
End Sub
```

See also

fintWorkbookCountHidden(strHiddenWorkbookNames())

fstrActiveWorkbookName(intPathName)

fstrFileNameFromPath(strSourceFileName)

fstrPathFromParts(strPath, strFile)

fblnNullMenuState

```
Function fblnNullMenuState() As Boolean
```

Indicates whether Excel is in "null menu" state (only File and Help in the active menu bar).

Inputs: None.

Returns: Boolean.

Example source code (see also *The Underground Guide to Microsoft Office, OLE, and* VBA):

```
' Copyright © 1994-95 PRIME Consulting Group, Inc.
Option Explicit
Sub sAnybodyHome()
    If fblnNullMenuState Then
        MsgBox "Excel is in null menu mode."
    Else
        MsgBox "Excel is not in null menu mode."
    End If
End Sub
```

fblnObjectExists(objCollection, vntIdentifier)

```
Function fblnObjectExists(objCollection As Object, vntIdentifier As
Variant) As Boolean
```

Determines if an object in a particular collection with a given identifier exists right now. The built-in function IsObject() is designed to tell you if a valid object is an OLE programmable object, and it causes a run-time error if the object in question does not currently exist. For example, the statements

```
IsObject(MenuBars(xlWorksheet))
```

and

```
IsObject(MenuBars("Worksheet"))
```

both return True, but

```
IsObject(MenuBars("ACMEAddIn"))
```

causes a run-time error 1004. Since it's often helpful to know if an object exists now or not, without causing an error, the custom function fblnObjectExists() allows you to pass an identifier along with the object's parent collection and see if it exists.

Inputs:

- *objCollection* - the object's parent collection or a pointer to it
- *vntIdentifier* - the numeric or string identifier of the object inside its parent collection, e.g., xlWorksheet (-4167) or "Worksheet" for the Worksheet MenuBar object

Returns: Boolean.

Example source code:

```
' Copyright © 1994-95 PRIME Consulting Group, Inc.
Option Explicit
Sub Test_fblnObjectExists()
    Dim mnb As MenuBar
    ' xlWorksheet = -4167 = "Worksheet"
    If Not fblnObjectExists(MenuBars, xlWorksheet) Then
        ' exit gracefully if the primary menu bar doesn't exist
        MsgBox "Uh oh!"
        Exit Sub
    Else
        Set mnb = MenuBars(xlWorksheet)
        MsgBox "Got it!"
    End If
End Sub
```

fblnOnlyOneTrueElement(vntArrayOfBooleans)

```
Function fblnOnlyOneTrueElement(vntArrayOfBooleans As Variant) As
Boolean
```

Looks at an array of Booleans to see if only one element is True. Useful when dealing with multi-select list boxes. If the function returns True then you can deal with the list box as though it were a single-selection list box. For more information about getting the state of multi-select list controls see the VBA help file topic on the Selected property.

Inputs:

- *vntArrayOfBooleans* - an array of Booleans in a Variant

Returns: Boolean.

See also

fintFirstTrueElement(vntArrayOfBooleans)

fintActiveSheetType

```
Function fintActiveSheetType() As Integer
```

Positively identify the active sheet's type using a proprietary return code scheme. There's no special significance to these return values other than they appear in order of most frequent occurrence in the "real world."

Inputs: None.

Returns: Integer (see below).

- 1 - Worksheet sheet
- 1 - embedded chart as window (NB: it's *embedded* chart)
- 2 - "standard" Chart sheet
- 2 - empty Chart sheet
- 3 - DialogSheet sheet
- 4 - Excel 4 macro sheet
- 5 - Excel 4 macro sheet (intl)
- 6 - Module sheet
- 7 - Info window
- 0 - any error condition

Example source code (see also *The Underground Guide to Microsoft Office, OLE, and* VBA):

```
' Copyright © 1994-95 PRIME Consulting Group, Inc.
Option Explicit
Sub sWhatPlanetYouFrom()
    Dim sht As Object
    Dim strTemp As String
    For Each sht In Sheets
        sht.Select
        strTemp = "The sheet " & sht.Name
        strTemp = strTemp & " has a type of" & _
            Str(fintActiveSheetType)
        MsgBox strTemp
    Next sht
End Sub
```

See also

fintSheetType(shtObject)

sActiveWorkbookSheetNames(strArray(), intOrder)

fintAppLoad(strModuleName, strShellName)

```
Function fintAppLoad(strModuleName As String, strShellName As String) As
Integer
```

Loads an application if it isn't already running, leaves it alone if it is already running, and returns an integer indicating final status either way. *Does not use OLE Automation!* Therefore, this function is general-purpose and not subject to the numerous application-specific problems observed with Office 4.x and OLE Automation.

When using Shell, if the executable file isn't in a subdirectory that's explicitly in your Path environment variable, you must use a fully-qualified executable filename.

Inputs:

- *strModuleName* - the target app's module name
- *strShellName* - the target app's shell name

Returns: Integer (see below).

- 0 - error
- 1 - was already loaded
- 2 - was *not* already loaded

Example source code:

```
' Copyright © 1994-95 PRIME Consulting Group, Inc.
Option Explicit
Const WORD6_MODULE_NAME = "WINWORD.EXE"
Const WORD6_SHELL_NAME = "WINWORD"
Dim strQuote As String
Sub Test_fintAppload()
    strModuleName = WORD6_MODULE_NAME
    strQuote = Chr(34)
    strShellName = WORD6_SHELL_NAME
    intRC = fintAppLoad(strModuleName, strShellName)
    strPrompt = "PRIME fintAppLoad(" & _
        strQuote & strModuleName & strQuote & _
        ", " & strQuote & strShellName & strQuote & _
        ") returns " & LTrim(Str(intRC))
    MsgBox strPrompt
End Sub
```

For another example of how to use this procedure, see DDEROUTI.XLS from your master diskette.

See also

fblnDDEFileStatus(strSourceFileName, strShortTopic, intSysChanNum)

sShutDownApp(intSysChanNum, strShellName intAppStatus)

fintFirstTrueElement(vntArrayOfBooleans)

```
Function fintFirstTrueElement(vntArrayOfBooleans As Variant) As Integer
```

Looks at an array of Booleans and returns index of the first True one. If none are True, returns 0.

Inputs:

- *vntArrayOfBooleans* - an array of Booleans in a Variant

Returns: Integer.

See also

fblnOnlyOneTrueElement(vntArrayOfBooleans)

fintOAAppLoad(objOAServer, strOAServerName, strModuleName)

```
Function fintOAAppLoad(objOAServer As Object, strOAServerName As String,
strModuleName As String) As Integer
```

Uses OLE Automation to load an application if it isn't already running, connect to it if it is already running, and returns an integer indicating final status either way.

Inputs:

- *objOAServer* - pointer to the server
- *strOAServerName* - the server's top-level class name, for example, "Excel.Application" for Excel 5
- *strModuleName* - the server's module name

Returns: Integer (see below).

- 0 - error
- 1 - was already loaded
- 2 - was *not* already loaded

Example source code (see also *The Underground Guide to Microsoft Office, OLE, and* VBA):

For an example of how to use this procedure, see OAROUTIN.XLS from your master diskette and sOLEAutoRoutine.

See also

fintOAFileStatus(objOAServer, strShellName, strSourceFileName)

fvntOAFileNames(objCollection)

fvntOAWindowFileNames(objCollection)

sOAGetOpenFiles(objOAServer, strShellName, strFileNames())

sOAShutDownApp(objOAServer, strShellName, intAppStatus)

sOLEAutoRoutine

fintOAFileStatus(objOAServer, strShellName, strSourceFileName)

```
Function fintOAFileStatus(objOAServer As Object, strShellName As String,
strSourceFileName As String) As Integer
```

Uses OLE Automation to determine if a file is already open. This function assumes you've already Set a valid object variable.

Inputs:

- *objOAServer* - pointer to the server
- *strShellName* - server's shell name
- *strSourceFileName* - source filename (should be fully qualified)

Returns: Integer (see below).

- 0 = bogus filename
- 1 = already open
- 2 = we opened it successfully
- 3 = filename is OK but the file open method failed

Example source code (see also *The Underground Guide to Microsoft Office, OLE, and* VBA):

For an example of how to use this procedure, see OAROUTIN.XLS from your master diskette and sOLEAutoRoutine.

See also

fintOAAppLoad(objOAServer, strOAServerName, strModuleName)

fvntOAFileNames(objCollection)

fvntOAWindowFileNames(objCollection)

sOAGetOpenFiles(objOAServer, strShellName, strFileNames())

sOAShutDownApp(objOAServer, strShellName, intAppStatus)

sOLEAutoRoutine

fintSheetType(shtObject)

```
Function fintSheetType(shtObject As Object) As Integer
```

This is a generalized form of fintActiveSheet() and works on any sheet, not just the active sheet.

Positively identify the passed sheet's type using a proprietary return code scheme. There's no special significance to these return values other than they appear in order of most frequent occurrence in the "real world."

Inputs:

- *shtObject* - alias for the sheet object of interest

Returns: Integer (see below).

- 1 - Worksheet sheet
- 1 - embedded chart as window (NB: it's *embedded* chart)
- 2 - "standard" Chart sheet
- 2 - empty Chart sheet
- 3 - DialogSheet sheet
- 4 - Excel 4 macro sheet
- 5 - Excel 4 macro sheet (intl)
- 6 - Module sheet
- 7 - Info window
- 0 - any error condition

Example source code:

```
' Copyright © 1994-95 PRIME Consulting Group, Inc.
Option Explicit
Sub Test_fintSheetType()
    ' set up Sheet1 as a worksheet, Chart1 as a chart sheet,
    '   and Module1 as a module sheet
    MsgBox fintSheetType(Sheets("Sheet1"))
    MsgBox fintSheetType(Sheets("Chart1"))
    MsgBox fintSheetType(Sheets("Module1"))
End Sub
```

See also

fintActiveSheetType

sActiveWorkbookSheetNames(strArray(), intOrder)

fintSplitVertical(intPercent)

```
Function fintSplitVertical(intPercent As Integer) As Integer
```

Vertically splits the active window on a percentage basis. Note that "Vertical" refers to the pane orientation, so the SplitVertical property indicates where a split bar is *horizontally* located along the vertical pane axis. Returns -1 if an error; if no error, returns the SplitVertical value.

Inputs:

- *intPercent* - percent of height at which to split

Returns: Integer.

Example source code:

```
' Copyright © 1994-95 PRIME Consulting Group, Inc.
Option Explicit
Sub Test_fintSplitVertical()
    ' set up and activate Sheet1 as a worksheet in a maximized window
    MsgBox fintSplitVertical(50)
End Sub
```

fintUsageCount(hMod)

```
Function fintUsageCount(ByVal hMod As Integer)
```

Get an application's true running instance count (includes hidden instances). Written by Vincent Chen.

Inputs:

- *hMod* - Module handle of the target application

Returns: Integer.

Example source code:

```
' Copyright © 1994-95 PRIME Consulting Group, Inc.
Option Explicit
Sub Test_fintUsageCount()
    Dim hMod As Integer, n As Integer
    hMod = GetModuleHandle("WINWORD.EXE")
    n = fintUsageCount(hMod)
    MsgBox n & ", " & GetModuleUsage(hMod)
End Sub
```

fintWindowCountHidden

```
Function fintWindowCountHidden() As Integer
```

Count the number of hidden windows in Excel.

Inputs: None.

Returns: Integer.

Example source code:

```
' Copyright © 1994-95 PRIME Consulting Group, Inc.
Option Explicit
Sub Test_fintWindowCountHidden()
    Dim intVisibleWindows As Integer
    intVisibleWindows = Windows.Count - fintWindowCountHidden()
    MsgBox "There are exactly " & intVisibleWindows & _
        " visible windows right now."
End Sub
```

See also

fintWorkbookCountHidden(strHiddenWorkbookNames())

fintWorkbookCountHidden(strHiddenWorkbookNames())

```
Function fintWorkbookCountHidden(strHiddenWorkbookNames() As String) As
Integer
```

Count the number of open hidden workbooks (not necessarily the same as the number of hidden windows).

Inputs:

- *strHiddenWorkbookNames()* - returns packed with any hidden workbook names

Returns: Integer.

Example source code (see also *The Underground Guide to Microsoft Office, OLE, and* VBA):

```
' Copyright © 1994-95 PRIME Consulting Group, Inc.
Option Explicit
Dim strPrompt As String
Sub sPeekABoo()
    Dim i As Integer
    Dim intCountHidden As Integer
    Dim strArray() As String
    Dim strPrompt As String
    intCountHidden = fintWorkbookCountHidden(strArray())
    strPrompt = "fintWorkbookCountHidden() returns " & _
        LTrim(Str(intCountHidden))
    MsgBox strPrompt
    For i = 0 To UBound(strArray)
        MsgBox strArray(i)
    Next i
End Sub
```

See also

fblnIsWorkbookOpen(strFullName)

fintWindowCountHidden

flngBinSearch(strList(), lngCount, strVal)

```
Function flngBinSearch(strList() As String, lngCount As Long, strVal As
String) As Long
```

This function does a binary search and returns the 0-based index of the matching *strList()* element if a match is found (the search is case-sensitive), and returns a negative number if no match is found. In this case, to find the index where *strVal* should be inserted, subtract the return value from -1.

Inputs:

- *strList()* is an array containing a string list; the function assumes the list is sorted alphabetically in ascending order
- *lngCount* is the number of entries in the list (not necessarily the number of elements in the array)
- *strVal* is the string to search for

Returns: Long.

Example source code:

```
' Copyright © 1994-95 PRIME Consulting Group, Inc.
Option Explicit
Sub Test_flngBinSearch()
    Dim i As Integer
    Dim intCount As Integer
    Dim intUBound As Integer
    Dim lngRC As Long
    Dim strList() As String
    Dim strValue As String
    intCount = 10
    intUBound = intCount - 1
    strValue = "Index item 4"
    ReDim strList(intUBound)
    For i = 0 To intUBound
        strList(i) = "Index item " + LTrim(Str(i))
    Next i
    sMsgStrArray strList(), intCount, "strList()"
    lngRC = flngBinSearch(strList(), CLng(intCount), strValue)
    MsgBox lngRC
End Sub
```

See also

flngBinSearchInit(strList(), lngCount, strVal)

flngDeleteFromArray(strArray(), lngSize, lngDeletePt)

sInsertInArray(strArray(), lngSize, lngInsertPt, strElement)

sMsgStrArray(strArray(), intElements, strName)

sShellSort(strArray())

flngBinSearchInit(strList(), lngCount, strVal)

```
Function flngBinSearchInit(strList() As String, lngCount As Long, strVal
As String) As Long
```

Same as flngBinSearch(), but searches for an initial match, e.g., a search for "xyz:" finds a match on "xyz:abc".

Example source code:

```
' Copyright © 1994-95 PRIME Consulting Group, Inc.
Option Explicit
Sub Test_flngBinSearchInit()
    Dim intCount As Integer
    Dim intUBound As Integer
    Dim lngRC As Long
    Dim strList() As String
    Dim strValue As String
    intCount = 3
    intUBound = intCount - 1
    strValue = "xyz:"
    ReDim strList(intUBound)
    strList(0) = "xyz:abc"
    strList(1) = "xyz:bcd"
    strList(2) = "yza:abc"
    sMsgStrArray strList(), intCount, "strList()"
    lngRC = flngBinSearchInit(strList(), CLng(intCount), strValue)
    MsgBox lngRC
End Sub
```

See also

flngBinSearch(strList(), lngCount, strVal)

flngDeleteFromArray(strArray(), lngSize, lngDeletePt)

sInsertInArray(strArray(), lngSize, lngInsertPt, strElement)

sMsgStrArray(strArray(), intElements, strName)

sShellSort(strArray())

flngDeleteFromArray(strArray(), lngSize, lngDeletePt)

```
Function flngDeleteFromArray(strArray() As String, lngSize As Long,
lngDeletePt As Long) As Long
```

Delete an entry from a string array.

Inputs:

- *strArray()* - the array to delete from (ASSUMED TO BE 0-BASED)
- *lngSize* - the absolute number of elements in the array (one greater than the upper boundary dimension since *strArray()* is assumed to be 0-based)
- *lngDeletePt* - the 0-based index of the element to delete

Returns: Long. If the highest element was deleted, the function returns *lngDeletePt* - 1, else it returns *lngDeletePt*. If the only remaining element was deleted, the function makes strArray(0) null and returns -1.

Even though VBA has built-in mechanisms for calculating an array's upper and lower boundary and hence its size (for a 0-based array, size equals UBound() +1), WordBasic has no UBound() function. So we left the *lngSize* argument in the VBA version of the function for compatibility across multiple dialects of Basic.

Example source code:

```
' Copyright © 1994-95 PRIME Consulting Group, Inc.
Option Explicit
Sub Test_flngDeleteFromArray()
    Dim i As Integer
    Dim intSize As Integer
    Dim lngDeletePt As Long
    Dim lngRC As Long
    Dim strArray() As String
    intSize = 5
    ReDim strArray(intSize - 1)
    lngDeletePt = 3
    For i = 0 To intSize - 1
        strArray(i) = "Index " + LTrim(Str(i))
    Next i
    sMsgStrArray strArray(), intSize, "strArray()"
    lngRC = flngDeleteFromArray(strArray(), CLng(intSize), lngDeletePt)
    MsgBox lngRC
    sMsgStrArray strArray(), intSize, "strArray()"
End Sub
```

See also

flngBinSearch(strList(), lngCount, strVal)

flngBinSearchInit(strList(), lngCount, strVal)

sInsertInArray(strArray(), lngSize, lngInsertPt, strElement)

sMsgStrArray(strArray(), intElements, strName)

sShellSort(strArray())

fobjSelectedObject(objCollection, strListControl(), objListControl)

```
Function fobjSelectedObject(objCollection As Object, strListControl() As
String, objListControl As Object) As Object
```

Return an object (not just its identifier) that is represented by a specific, selected item (the upper-most one) in a dialog box's multi-select list control.

Inputs:

- *objCollection* - the object's parent collection or a pointer to it
- *strListControl()* - the string array that is supplying the contents of the list control (ASSUMED TO BE 0-BASED)
- *objListControl* - the list control object or a pointer to it

Returns: Object.

See also

fintFirstTrueElement(vntArrayOfBooleans)

fstrActiveWorkbookName(intPathName)

```
Function fstrActiveWorkbookName(intPathName As Integer) As String
```

Provide the active workbook's 8.3 or full name depending on input.

Inputs:

- *intPathName* - equal to 0 indicates return 8.3 filename; not equal to 0 indicates return full filename (complete with path)

Returns: String.

Example source code (see also *The Underground Guide to Microsoft Office, OLE, and* VBA):

```
' Copyright © 1994-95 PRIME Consulting Group, Inc.
Option Explicit
Sub sHeyYouWithTheFocus()
    MsgBox fstrActiveWorkbookName(0)
    MsgBox fstrActiveWorkbookName(1)
End Sub
```

See also

fblnIsWorkbookOpen(strFullName)

fvntWorkbookNames

fstrFileNameFromPath(strSourceFileName)

```
Function fstrFileNameFromPath(strSourceFileName As String) As String
```

Strips the 8.3 filename from a fully qualified filename. Returns a zero-length string if no "\" is found.

Inputs:

- *strSourceFileName* - the fully qualified filename

Returns: String.

Example source code (see also *The Underground Guide to Microsoft Office, OLE, and* VBA):

```
' Copyright © 1994-95 PRIME Consulting Group, Inc.
Option Explicit
Sub Test_fstrFileNameFromPath()
    MsgBox fstrFileNameFromPath("C:\OLEDDE2\CUSTOM.XLS")
End Sub
```

See also

fblnIsWorkbookOpen(strFullName)

fstrPathFromParts(strPath, strFile)

fstrGetCellContents(vntRange)

```
Function fstrGetCellContents(vntRange As Variant) As String
```

Provide a cell's contents in string (viewable) format.

Inputs:

- *vntRange* - Either a pointer to a cell object or just a plain A1-style cell address. Note that in the case of something like Sheets("Sheet1").Range("H3"), the Range method's Syntax 1 dictates that the address must be in A1 notation. If *vntRange* is just a plain A1-style cell address, the function assumes caller passes the address in the format sheet_name!A1_address, i.e., the sheet's name and the bang are required.

Returns: String.

Example source code (see also *The Underground Guide to Microsoft Office, OLE, and* VBA):

```
' Copyright © 1994-95 PRIME Consulting Group, Inc.
Option Explicit
Sub sGenieInABottle2()
    Dim objRange As Range
    Dim strA1Address As String
    strA1Address = "Sheet1!H1"
    ' strA1Address = "RangeName1"   ' this works fine!
    Set objRange = Sheets("Sheet1").Range("H1")
    MsgBox "Passing objRange returns " & _
        fstrGetCellContents(objRange)
    MsgBox "Passing an A1 notation address returns " & _
        fstrGetCellContents(strA1Address)
End Sub
```

See also

fvntGetCellResults(vntRange, blnFormatting)

fstrGetHostTitleBar

```
Function fstrGetHostTitleBar() As String
```

Get Excel's current title bar text.

Inputs: None.

Returns: String.

Example source code (see also *The Underground Guide to Microsoft Office, OLE, and* VBA):

```
' Copyright © 1994-95 PRIME Consulting Group, Inc.
Option Explicit
Dim strModuleName As String
Dim strShellName As String
Dim strTitleBar As String
Sub sThisProcedureIsMinimalist()
    Dim intAppStatus As Integer
    strModuleName = "WINWORD.EXE"
    strShellName = "WINWORD"
    strTitleBar = fstrGetHostTitleBar()
    intAppStatus = fintAppLoad(strModuleName, strShellName)
    ' interact with OLE Automation or DDE server,
   '   then bring the focus back to Excel if needed...
    AppActivate strTitleBar
    ' shut down the server
    ' and so on...
End Sub
```

fstrGetRangeAddress(strRangeName, intNotation)

```
Function fstrGetRangeAddress(strRangeName As String, intNotation As
Integer) As String
```

Determine a range name's address in A1 or R1C1 notation. Return an empty string if the range name is invalid.

Inputs:

- *strRangeName* - the range name
- *intNotation* - the notation desired (0 for A1, <> 0 for R1C1)

Returns: String.

Example source code (see also *The Underground Guide to Microsoft Office, OLE, and* VBA):

```
' Copyright © 1994-95 PRIME Consulting Group, Inc.
Option Explicit
Sub sGiveMeARange()
    Dim i As Integer
    Dim objName As Object
    Dim strTemp As String
    For i = 0 To 1
        With ActiveWorkbook
            ' walk through the active workbook's Names collection
            For Each objName In .Names
                strTemp = objName.Name
                If i = 0 Then
                    ' A1 notation request
                    strTemp = strTemp & " A1 address is "
                Else
                    ' R1C1 notation request
                    strTemp = strTemp & " R1C1 address is "
                End If
                strTemp = strTemp & _
                    fstrGetRangeAddress(objName.Name, i)
                MsgBox strTemp
            Next objName
        End With
        ' pass a bogus range name
        strTemp = "Smoky" & " A1 address is " & _
            fstrGetRangeAddress("Smoky", i)
        MsgBox strTemp
    Next i
End Sub
```

fstrGetRangeFromRefersTo(strRefersTo)

```
Function fstrGetRangeFromRefersTo(strRefersTo As String) As String
```

Parses the range from a Name object's RefersTo property string. Returns an empty string if RefersTo has no bang character; absence of a bang character typically indicates a named formula. If there is a bang in RefersTo, the function does *not* do any validation of the resulting range string. An exception to the "there's no bang so it's a named formula" rule would be a named formula like

```
=VALUE("Hello!")
```

In a cell this formula would result in a #VALUE! error but could nonetheless exist as a named formula.

Inputs:

- *strRefersTo* - the Name object's RefersTo property string

Returns: String.

Example source code:

Example source code for calling fstrGetRangeFromRefersTo() is provided with the example source code in the section fstrGetSheetFromRefersTo(strRefersTo).

See also

fstrGetSheetFromRefersTo(strRefersTo)

fstrGetSheetFromRefersTo(strRefersTo)

```
Function fstrGetSheetFromRefersTo(strRefersTo As String) As String
```

Parses the sheet name from a Name object's RefersTo property string. Returns an empty string if RefersTo has no bang character; absence of a bang character typically indicates a named formula. Returns an empty string if the parsed sheet name proves to be invalid for the active workbook. An exception to the "there's no bang so it's a named formula" rule would be a named formula like

```
=VALUE("Hello!")
```

In a cell this formula would result in a #VALUE! error but could nonetheless exist as a named formula. In this case, the string

```
VALUE("Hello
```

would be tested and proven to be an invalid sheet name so the function would return an empty string.

Inputs:

- *strRefersTo* - the Name object's RefersTo property string

Returns: String.

Example source code:

```
' Copyright © 1994-95 PRIME Consulting Group, Inc.
Option Explicit
Sub Test_fstrGetSheetFromRefersTo()
    ' manually create a range name "RangeName1" that refers to
    '   A1:B2 on Sheet1
    Const MSG_3D_OR_NAMED_FORMULA = "Must be a 3-D range or " & _
        "a named formula."
    Dim strRange As String
    Dim strRefersTo As String
    Dim strSheet As String
    strRefersTo = ActiveWorkbook.Names("RangeName1").RefersTo
    strSheet = fstrGetSheetFromRefersTo(strRefersTo)
    If strSheet = "" Then
        MsgBox MSG_3D_OR_NAMED_FORMULA
        Exit Sub
    Else
        Sheets(strSheet).Activate
    End If
    strRange = fstrGetRangeFromRefersTo(strRefersTo)
    If strRange = "" Then
        MsgBox MSG_3D_OR_NAMED_FORMULA
        Exit Sub
    Else
        Range(strRange).Select
    End If
End Sub
```

See also

fstrGetRangeFromRefersTo(strRefersTo)

fstrPathFromParts(strPath, strFile)

```
Function fstrPathFromParts(strPath As String, strFile As String) As
String
```

This function assembles a complete path name from its parts, consisting of the path, a backslash if necessary, and the filename.

Inputs:

- *strPath* - the pathname, which may be null
- *strFile* - the filename

This function does not validate *strPath* or *strFile* in any way. That is the caller's responsibility.

Returns: String.

See also

fblnIsWorkbookOpen(strFullName)

fstrFileNameFromPath(strSourceFileName)

fstrReplaceMarker(strMsg, strReplacement)

```
Function fstrReplaceMarker(strMsg As String, strReplacement As String)
As String
```

Provide a simple way to replace a marker in global constant message strings with context-relevant, current information. If the marker isn't found, function returns an empty string.

Inputs:

- *strMsg* - message string (*must* contain the "<<>>" marker)
- *strReplacement* - string to replace the marker with

Returns: String.

Example source code (see also *The Underground Guide to Microsoft Office, OLE, and* VBA):

```
' Copyright © 1994-95 PRIME Consulting Group, Inc.
Option Explicit
Const MSG_NAME_MGR_DELETE = "You are about to delete name <<>>. " & _
    "Are you sure?"
Dim strPrompt As String
Sub Test_fstrReplaceMarker()
    MsgBox MSG_NAME_MGR_DELETE
    strPrompt = fstrReplaceMarker(MSG_NAME_MGR_DELETE, _
        "LtCmdrData")
    MsgBox strPrompt
End Sub
```

fvntGetCellResults(vntRange, blnFormatting)

```
Function fvntGetCellResults(vntRange As Variant, blnFormatting As
Boolean) As Variant
```

Provide a cell's results.

Caller is responsible for handling the case where fvntGetCellResults is returned as an error type (VarType() returns vbError or 10).

Inputs:

- *vntRange* - Either a pointer to a cell object or just a plain A1-style cell address. Note that in the case of something like Sheets("Sheet1").Range("H3"), the Range method's Syntax 1 dictates that the address must be in A1 notation. If *vntRange* is just a plain A1-style cell address, the function assumes caller passes the address in the format sheet_name!A1_address, i.e., the sheet's name and the bang are required.
- *blnFormatting* - If True then return formatting characters, e.g., "$7,000.55 " (string type) for a constant of 7000.545 with the format code of $#,##0.00_);($#,##0.00); else don't return formatting characters.

Returns: Variant.

Example source code (see also *The Underground Guide to Microsoft Office, OLE, and* VBA):

```
' Copyright © 1994-95 PRIME Consulting Group, Inc.
Option Explicit
Sub sACellularMeisterstuck()
    Dim blnFormatting As Boolean
    Dim intAnswer As Integer
    Dim objRange As Range
    Dim strA1Address As String
    Dim vntResults As Variant
    strA1Address = "Sheet1!H14"
    Set objRange = Sheets("Sheet1").Range("H14")
    intAnswer = MsgBox("Do you want to see formatting characters " & _
        "(if any)?", vbYesNo + vbQuestion)
    If intAnswer = vbYes Then
        blnFormatting = True
    Else
        blnFormatting = False
    End If
    vntResults = fvntGetCellResults(objRange, blnFormatting)
    If IsError(vntResults) Then
        MsgBox "The cell contains an error value."
    ElseIf IsEmpty(vntResults) Then
        MsgBox "The cell is empty."
    Else MsgBox "Passing objRange returns " & _
            fvntGetCellResults(objRange, blnFormatting)
        MsgBox "Passing an A1 notation address returns " & _
```

```
                  fvntGetCellResults(strA1Address, blnFormatting)
          End If
      End Sub
```

See also

fstrGetCellContents(vntRange)

fvntMax(x, y)

```
Function fvntMax(x As Variant, y As Variant) As Variant
```

Returns the maximum of two values.

Inputs:

- *x* - a value
- *y* - a value

Returns: Variant.

Example source code:

```
' Copyright © 1994-95 PRIME Consulting Group, Inc.
Option Explicit
Sub Test_fvntMax()
    Dim x As Long
    Dim y As Long
    Dim strTemp As String
    x = -1
    y = 42
    strTemp = LTrim(Str(fvntMax(x, y)))
    MsgBox "PRIME fvntMax(-1, 42) returns " & strTemp
    x = 42
    y = -1
    strTemp = LTrim(Str(fvntMax(x, y)))
    MsgBox "PRIME fvntMax(42, -1) returns " & strTemp
End Sub
```

See also

fvntMin(x, y)

fvntMin(x, y)

```
Function fvntMin(x As Variant, y As Variant) As Variant
```

Returns the minimum of two values.

Inputs:

- *x* - a value
- *y* - a value

Returns: Variant.

Example source code:

```
' Copyright © 1994-95 PRIME Consulting Group, Inc.
Option Explicit
Sub Test_fvntMin()
    Dim x As Long
    Dim y As Long
    Dim strTemp As String
    x = -1
    y = 42
    strTemp = LTrim(Str(fvntMin(x, y)))
    MsgBox "PRIME fvntMin(-1, 42) returns " & strTemp
    x = 42
    y = -1
    strTemp = LTrim(Str(fvntMin(x, y)))
    MsgBox "PRIME fvntMin(42, -1) returns " & strTemp
End Sub
```

See also

fvntMax(x, y)

fvntOAFileNames(objCollection)

```
Function fvntOAFileNames(objCollection As Object) As Variant
```

Uses OLE Automation to provide an array of the full names (path plus 8.3 name) of all open (hidden and unhidden) files in the server.

Inputs:

- *objCollection* - points to server's collection of "file" objects (in Excel, it's the Workbooks collection; in Project it's the Projects collection)

Returns: Variant array.

Example source code (see also *The Underground Guide to Microsoft Office, OLE, and* VBA):

For an example of how to use this procedure, see OAROUTIN.XLS from your master diskette and sOLEAutoRoutine.

See also

fintOAAppLoad(objOAServer, strOAServerName, strModuleName)

fintOAFileStatus(objOAServer, strShellName, strSourceFileName)

fvntOAWindowFileNames(objCollection)

sOAGetOpenFiles(objOAServer, strShellName, strFileNames())

sOAShutDownApp(objOAServer, strShellName, intAppStatus)

sOLEAutoRoutine

fvntOAWindowFileNames(objCollection)

```
Function fvntOAWindowFileNames(objCollection As Object) As Variant
```

Uses OLE Automation to provide an array of the full names (path plus 8.3 name) of all files in each of the server's child windows.

Inputs:

- *objCollection* - points to server's Windows collection

Returns: Variant array.

Example source code (see also *The Underground Guide to Microsoft Office, OLE, and* VBA):

For an example of how to use this procedure, see OAROUTIN.XLS from your master diskette and sOLEAutoRoutine.

See also

fintOAAppLoad(objOAServer, strOAServerName, strModuleName)

fintOAFileStatus(objOAServer, strShellName, strSourceFileName)

fvntOAFileNames(objCollection)

sOAGetOpenFiles(objOAServer, strShellName, strFileNames())

sOAShutDownApp(objOAServer, strShellName, intAppStatus)

sOLEAutoRoutine

fvntWorkbookNames

```
Function fvntWorkbookNames() As Variant
```

Provide an array of the full names (path plus 8.3 name) of all open (hidden and unhidden) workbooks.

Inputs: None.

Returns: Variant array.

Example source code (see also *The Underground Guide to Microsoft Office, OLE, and* VBA):

```
' Copyright © 1994-95 PRIME Consulting Group, Inc.
Option Explicit
Sub sWhatsYourName()
    Dim i As Integer
    Dim intUBound As Integer
    Dim strTemp() As String
    Dim vntTemp As Variant
    vntTemp = fvntWorkbookNames
    intUBound = UBound(vntTemp)
    For i = 0 To intUBound
        MsgBox vntTemp(i)
    Next i
End Sub
```

See also

fstrActiveWorkbookName(intPathName)

sActiveWorkbookNames(strNames())

```
Sub sActiveWorkbookNames(strNames() As String)
```

Load an array argument with the active workbook's names. Caller is responsible for making sure there *is* an active workbook.

Inputs:

- *strNames()* - string array to be filled with the names

Example source code (see also *The Underground Guide to Microsoft Office, OLE, and* VBA):

```
' Copyright © 1994-95 PRIME Consulting Group, Inc.
Option Explicit
Sub sShowMeAllNames()
    Dim i As Integer
    Dim strNames() As String
    sActiveWorkbookNames strNames()
    For i = 0 To UBound(strNames)
        MsgBox "Name" & Str(i + 1) & " is " & strNames(i)
    Next i
End Sub
```

sActiveWorkbookSheetNames(strArray(), intOrder)

```
Sub sActiveWorkbookSheetNames(strArray() As String, intOrder As Integer)
```

Returns a list of all the active workbook's sheet names, either in positional (collection) order, or sorted alphabetically.

Inputs:

- *strArray()* - array to pack with the names
- *intOrder* - if equal to 0 then leave in positional order; if not equal to 0 then sort alphabetically

Example source code (see also *The Underground Guide to Microsoft Office, OLE, and* VBA):

```
' Copyright © 1994-95 PRIME Consulting Group, Inc.
Option Explicit
Sub sMyKingdom()
    Dim i As Integer
    Dim strArray() As String
    sActiveWorkbookSheetNames strArray(), 1
    For i = 0 To UBound(strArray)
        MsgBox strArray(i)
    Next i
End Sub
```

See also

fintActiveSheetType

sGetToolbarButtonInfo(strTheToolbarName, strButtonInfo())

```
Sub sGetToolbarButtonInfo(ByVal strTheToolbarName As String,
strButtonInfo() As String)
```

Return a toolbar's contents (structure described below).

Inputs:

- *strTheToolbarName* - the toolbar's name
- *strButtonInfo()* - array used to return contents to caller

Returns an array in a redimensioned strButtonInfo(); each item is comprised of strTheToolbarName's button information as three concatenated items: (1) button number (1-based), (2) " - ", and (3) category item name; validates strTheToolbarName and redimensions strButtonInfo() to 0 if bogus strTheToolbarName.

Example source code:

```
' Copyright © 1994-95 PRIME Consulting Group, Inc.
Option Explicit
Dim strPrompt As String
Dim strQuote As String
Sub Test_sGetToolbarButtonInfo()
    Dim strButtonInfo() As String
    Dim strToolbarName As String
    strQuote = Chr(34)
    strToolbarName = "Standard"
    sGetToolbarButtonInfo strToolbarName, strButtonInfo()
    strPrompt = "PRIME sGetToolbarButtonInfo() for " & _
        "the toolbar " & _
        strQuote & strToolbarName & strQuote & _
        " returns strButtonInfo(0) as " & _
        strQuote & strButtonInfo(0) & strQuote
    MsgBox strPrompt
End Sub
```

See also

fblnIsValidToolbarName(strToolbarName)

sInsertInArray(strArray(), lngSize, lngInsertPt, strElement)

```
Sub sInsertInArray(strArray() As String, lngSize As Long, lngInsertPt As
Long, strElement As String)
```

This subroutine inserts an entry in a string array.

Inputs:

- *strArray()* is the target array
- *lngSize* is the number of elements in the array (one greater than the upper boundary dimension)
- *lngInsertPt* is the 0-based index of the element before which the entry is to be inserted
- *strElement* is the string to be inserted

Even though VBA has built-in mechanisms for calculating an array's upper and lower boundary and hence its size (for a 0-based array, size equals UBound() +1), WordBasic has no UBound() function. So we left the *lngSize* argument in the VBA version of the function for compatibility across multiple dialects of Basic.

Example source code:

```
' Copyright © 1994-95 PRIME Consulting Group, Inc.
Option Explicit
Sub Test_sInsertInArray()
    Dim intCount As Integer
    Dim intUBound As Integer
    Dim strList() As String
    Dim strValue As String
    intCount = 3
    intUBound = intCount - 1
    strValue = "I want in!"
    ReDim strList(intUBound)
    strList(0) = "xyz:abc"
    strList(1) = "xyz:bcd"
    strList(2) = "yza:abc"
    sMsgStrArray strList(), intCount, "strList()"
    sInsertInArray strList(), CLng(intCount), 1, strValue
    sMsgStrArray strList(), UBound(strList) + 1, "strList()"
End Sub
```

See also

flngBinSearch(strList(), lngCount, strVal)

flngBinSearchInit(strList(), lngCount, strVal)

flngDeleteFromArray(strArray(), lngSize, lngDeletePt)

sMsgStrArray(strArray(), intElements, strName)

sShellSort(strArray())

sMsgStrArray(strArray(), intElements, strName)

```
Sub sMsgStrArray(strArray() As String, intElements As Integer, strName
As String)
```

Development utility to display the first *n* elements of an array.

Inputs:

- *strArray()* - the subject array
- *intElements* - the absolute number of elements to display
- *strName* - the name of the array (or any text the caller provides) displayed in the resulting message box's title bar

Example source code:

```
' Copyright © 1994-95 PRIME Consulting Group, Inc.
Option Explicit
Dim strQuote As String
Sub sShowMenuBarCaptions()
    Dim i As Integer
    Dim mnbActive As MenuBar
    Dim mnuObject As Menu
    Dim strArray() As String
    strQuote = Chr(34)
    Set mnbActive = Application.ActiveMenuBar
    MsgBox "The active menu bar's caption is " & _
        strQuote & mnbActive.Caption & strQuote
    ' examine the MenuBars collection
    MsgBox "The number of objects in the Menubars collection is " & _
        LTrim(Str(MenuBars.Count))
    ReDim strArray(MenuBars.Count - 1)
    For i = 0 To MenuBars.Count - 1
        strArray(i) = MenuBars(i + 1).Caption
    Next i
    sMsgStrArray strArray(), UBound(strArray) + 1, _
        "strArray (Menubar captions)"
    ' examine the Menus collection
    MsgBox "The number of Menu objects in the active menu bar is " & _
        LTrim(Str(mnbActive.Menus.Count))
    ReDim strArray(mnbActive.Menus.Count - 1)
    i = 0
    For Each mnuObject In mnbActive.Menus
        strArray(i) = mnuObject.Caption
        i = i + 1
    Next mnuObject
    sMsgStrArray strArray(), UBound(strArray) + 1, _
        "strArray (Menu captions)"
        End Sub
```

See also

fIngBinSearch(strList(), lngCount, strVal)

fIngBinSearchInit(strList(), lngCount, strVal)

fIngDeleteFromArray(strArray(), lngSize, lngDeletePt)

sInsertInArray(strArray(), lngSize, lngInsertPt, strElement)

sShellSort(strArray())

sNotImplemented(strDboxTitle, strPrompt)

```
Sub sNotImplemented(ByVal strDboxTitle As String, strPrompt As String)
```

Display a "not implemented" message for work-in-process features.

Inputs:

- *strDboxTitle* - message box title
- *strPrompt* - local portion of the message text

Example source code:

```
' Copyright © 1994-95 PRIME Consulting Group, Inc.
Option Explicit
Sub Test_sNotImplemented()
    sNotImplemented "See Ya in the Underground!", _
        "Tachyon Detector."
End Sub
```

sOAGetOpenFiles(objOAServer, strShellName, strFileNames())

```
Sub sOAGetOpenFiles(objOAServer As Object, strShellName As String,
strFileNames() As String)
```

Uses OLE Automation to stuff an array with the fully qualified path-and-filenames of only non-macro child windows.

Inputs:

- *objOAServer* - a pointer to the OLE Automation server's Application (top-level) object
- *strShellName* - used as a unique hook to each server
- *strFileNames()* - string array to be loaded with fully qualified non-macro child windows, i.e., documents only

Example source code (see also *The Underground Guide to Microsoft Office, OLE, and* VBA):

For an example of how to use this procedure, see OAROUTIN.XLS from your master diskette and sOLEAutoRoutine.

See also

fintOAAppLoad(objOAServer, strOAServerName, strModuleName)

fintOAFileStatus(objOAServer, strShellName, strSourceFileName)

fvntOAFileNames(objCollection)

fvntOAWindowFileNames(objCollection)

sOAShutDownApp(objOAServer, strShellName, intAppStatus)

sOLEAutoRoutine

sOAShutDownApp(objOAServer, strShellName, intAppStatus)

```
Sub sOAShutDownApp(objOAServer As Object, strShellName As String,
intAppStatus As Integer)
```

Use OLE Automation methods to shut down an OLEAuto-invoked application.

Inputs:

- *objOAServer* - a pointer to the OLE Automation server's Application (top-level) object
- *strShellName* - the server application's shell name
- *intAppStatus* - the return code for a call to fintAppLoad()

Example source code (see also *The Underground Guide to Microsoft Office, OLE, and* VBA)

For an example of how to use this procedure, see OAROUTIN.XLS from your master diskette and sOLEAutoRoutine.

See also

fintOAAppLoad(objOAServer, strOAServerName, strModuleName)

fintOAFileStatus(objOAServer, strShellName, strSourceFileName)

fvntOAFileNames(objCollection)

fvntOAWindowFileNames(objCollection)

sOAGetOpenFiles(objOAServer, strShellName, strFileNames())

sOLEAutoRoutine

sOLEAutoRoutine

This procedure effectively demonstrates how PRIME 5's various OLE Automation procedures can work together to produce a turnkey, customized OLE Automation controller application. The source code for this routine is provided in the workbook OAROUTIN.XLS (see your master diskette and also *The Underground Guide to Microsoft Office, OLE, and* VBA).

The sOLEAutoRoutine procedure is simply one example of how you might stitch together your own OLEAuto quilt. *It orchestrates all the general-purpose functions and subroutines you'll need to put an OLEAuto session together*. This is why you'll see an entire section of sOLEAutoRoutine commented like this, "Put your OLEAuto commands below (examples follow)." It (and the accompanying procedures) is written to work from within any OLEAuto-compatible client, which as of the date of this writing are Excel, Project, and Visual Basic 3. The specific code you see throughout this and following sections was taken directly from an Excel project. To port the code to run from inside Project, you'll have to make a relatively small number of changes. We discuss this port to Project later in this section. Porting to Visual Basic 3 is rendered a bit more tedious by the fact that VB3 doesn't support VBA named arguments, so for that port to work you have to convert all the VBA named arguments to positional arguments. Tedious? Yes, but nonetheless all the code herein is ready to be ported to any VBA dialect as well as VB3. *Three utilities for the price of one!*

Here are the steps for setting up sOLEAutoRoutine.

1. Make sure you've established a reference to PRIME5.XLA.
2. Throughout sOLEAutoRoutine, comment out all initializations that refer to Excel in the "Case where Excel is the server" section, if they aren't already. We assume here that Excel is the client in your first test.
3. Throughout sOLEAutoRoutine in Excel, pick either Project or Word as the target server, un-remark all initializations that refer to the target application and comment out all initializations that refer to the other application.
4. Calibrate your chosen server's target file. For example, make sure the filename variable for the target file points to a valid directory and filename, and so on. Of course, you may want to test these out-of-bounds, error-producing conditions as well.
5. Run sOLEAutoRoutine.

Here are the steps for porting this utility from Excel VBA to Project VBA.

1. Use the clipboard to copy the contents of Excel's modules over to Project.
2. Throughout modOLEAutoRoutine in Project, comment out all Application.StatusBar references because Project doesn't support the StatusBar property. (Tip: use a module-wide Edit / Replace.)
3. Throughout sOLEAutoRoutine in Project, comment out all initializations that refer to Project in the "Case where Project is the server" section.
4. Throughout sOLEAutoRoutine in Project, pick either Excel or Word as the target server, un-remark all initializations that refer to the target application and comment out all initializations that refer to the other application.
5. Run sOLEAutoRoutine.

See also

fintOAAppLoad(objOAServer, strOAServerName, strModuleName)

fintOAFileStatus(objOAServer, strShellName, strSourceFileName)

fvntOAFileNames(objCollection)

fvntOAWindowFileNames(objCollection)

sOAGetOpenFiles(objOAServer, strShellName, strFileNames())

sOAShutDownApp(objOAServer, strShellName, intAppStatus)

sShellSort(strArray())

```
sShellSort(strArray() As String)
```

Sort a string array. Ported w/ permission from WOPR 2's ShellSort, © 1991-92 Pinecliffe International. CRITICAL: Option Compare Text must be set in sShellSort's owner module. This sub will not behave properly if the default Option Compare Binary is used. A fast, practical sort routine based on Donald L. Shell's technique (CACM 2, July 1959, pp 30-32), as adapted by Don Knuth ("The Art of Computer Programming, Vol 3, Sorting and Searching", Addison-Wesley, 1975, pp 84-87); see Knuth for a detailed description.

Inputs:

- *strArray()* - target string array (ASSUMED TO BE 0-BASED)

Example source code (see also *The Underground Guide to Microsoft Office, OLE, and* VBA):

```
' Copyright © 1994-95 PRIME Consulting Group, Inc.
Option Explicit
Sub Test_sShellSort()
    Dim strArray() As String
    Dim strTemp As String
    GoTo Init1
Init1:
    ' ----- Ordinary, simple test
    ReDim strArray(2)
    strArray(0) = "C"
    strArray(1) = "B"
    strArray(2) = "a"
    GoTo StartTest
Init2:
    ' ----- Single-element, 0-based test with element *not* empty
    ReDim strArray(0)
    strArray(0) = "Sheet3"
    GoTo StartTest
Init3:
    ' ----- Single-element, 0-based test with element empty
    ReDim strArray(0)
    GoTo StartTest
    ' ----- End of test initializations
StartTest:
    sMsgStrArray strArray(), UBound(strArray) + 1, _
        "strArray() unsorted"
    sShellSort strArray()
    sMsgStrArray strArray(), UBound(strArray) + 1, _
        "strArray() sorted by sShellSort()"
End Sub
```

See also

fIngBinSearch(strList(), lngCount, strVal)

fIngBinSearchInit(strList(), lngCount, strVal)

fIngDeleteFromArray(strArray(), lngSize, lngDeletePt)

sInsertInArray(strArray(), lngSize, lngInsertPt, strElement)

sMsgStrArray(strArray(), intElements, strName)

sShutDownApp(intSysChanNum, strShellName intAppStatus)

```
Sub sShutDownApp(intSysChanNum As Integer, strShellName As String,
intAppStatus As Integer)
```

Use DDEExecute commands to shut down a DDE-invoked application. Assumes that *intSysChanNum* is valid.

Inputs:

- *intSysChanNum* - channel number of initial System topic
- *strShellName* - shell name of source application
- *intAppStatus* - machine state from fintAppLoad()

Example source code:

```
' Copyright © 1994-95 PRIME Consulting Group, Inc.
Option Explicit
Const WORD6_MODULE_NAME = "WINWORD.EXE"
Const WORD6_SHELL_NAME = "WINWORD"
Sub Test_sShutDownApp()
    Dim intAppStatus As Integer
    Dim intSysChanNum As Integer
    strModuleName = WORD6_MODULE_NAME
    strShellName = WORD6_SHELL_NAME
    intAppStatus = fintAppLoad(strModuleName, strShellName)
    intSysChanNum = DDEInitiate(strShellName, "SYSTEM")
    sShutDownApp intSysChanNum, strShellName, intAppStatus
End Sub
```

For another example of how to use this procedure, see DDEROUTI.XLS from your master diskette.

See also

fblnDDEFileStatus(strSourceFileName, strShortTopic, intSysChanNum)

fintAppLoad(strModuleName, strShellName)

sWaitCursor(blnState)

```
Sub sWaitCursor(blnState As Boolean)
```

WordBasic has a built-in WaitCursor command but oddly, Excel VBA does not. So we added one. Our custom procedure changes the mouse cursor to an hourglass or back to a standard arrow.

Inputs:

- *blnState* - a Boolean when True turns on the hourglass cursor; when False turns on the standard arrow cursor

Custom VBA Procedure Categories

The following sections provide a categorized listing of all PRIME 5 for Excel custom VBA procedures.

For an overview of PRIME 5's custom VBA procedures, see You Can Call PRIME 5 Custom VBA Functions and Subroutines.

For a summary of PRIME 5's custom VBA procedures, see Custom VBA Procedure Summary.

Related Topics:

Arrays

Custom Dialogs

Dynamic Data Exchange

Excel Environment

Files

Messaging

Numerical Analysis

Objects (General)

OLE Automation

Ranges

Sheets

Windows (Excel Windows)

Windows Environment

Workbooks

Arrays

The following procedures relate to arrays.

fIngBinSearch(strList(), lngCount, strVal)

fIngBinSearchInit(strList(), lngCount, strVal)

fIngDeleteFromArray(strArray(), lngSize, lngDeletePt)

sInsertInArray(strArray(), lngSize, lngInsertPt, strElement)

sMsgStrArray(strArray(), intElements, strName)

sShellSort(strArray())

Custom Dialogs

The following procedures relate to custom dialogs.

fblnOnlyOneTrueElement(vntArrayOfBooleans)

fintFirstTrueElement(vntArrayOfBooleans)

fobjSelectedObject(objCollection, strListControl(), objListControl)

Dynamic Data Exchange

The following procedures relate to Dynamic Data Exchange.

fblnDDEFileStatus(strSourceFileName, strShortTopic, intSysChanNum)

fintAppLoad(strModuleName, strShellName)

sShutDownApp(intSysChanNum, strShellName intAppStatus)

Excel Environment

The following procedures relate to the Excel environment.

fbInNullMenuState

fstrGetHostTitleBar

Files

The following procedures relate to files.

fstrFileNameFromPath(strSourceFileName)

fstrPathFromParts(strPath, strFile)

Messaging

The following procedures relate to messaging.

fstrReplaceMarker(strMsg, strReplacement)

sNotImplemented(strDboxTitle, strPrompt)

sWaitCursor(blnState)

Numerical Analysis

The following procedures relate to numerical analysis.

fvntMax(x, y)

fvntMin(x, y)

Objects (General)

The following procedures relate to general objects.

fblnIsValidToolbarName(strToolbarName)

fblnObjectExists(objCollection, vntIdentifier)

sActiveWorkbookNames(strNames())

sGetToolbarButtonInfo(strTheToolbarName, strButtonInfo())

OLE Automation

The following procedures relate to OLE Automation.

fintOAAppLoad(objOAServer, strOAServerName, strModuleName)

fintOAFileStatus(objOAServer, strShellName, strSourceFileName)

fvntOAFileNames(objCollection)

fvntOAWindowFileNames(objCollection)

sOAGetOpenFiles(objOAServer, strShellName, strFileNames())

sOAShutDownApp(objOAServer, strShellName, intAppStatus)

Ranges

The following procedures relate to ranges.

fstrGetCellContents(vntRange)

fstrGetRangeAddress(strRangeName, intNotation)

fstrGetRangeFromRefersTo(strRefersTo)

fstrGetSheetFromRefersTo(strRefersTo)

fvntGetCellResults(vntRange, blnFormatting)

Sheets

The following procedures relate to sheets.

fintActiveSheetType

fintSheetType(shtObject)

sActiveWorkbookSheetNames(strArray(), intOrder)

Windows (Excel Windows)

The following procedures relate to Excel window objects, not "Windows" as in the Windows operating environment.

fintSplitVertical(intPercent)

fintWindowCountHidden

Windows Environment

The following procedures relate to the Windows environment.

fintUsageCount(hMod)

Workbooks

The following procedures relate to workbooks.

fblnIsWorkbookOpen(strFullName)

fintWorkbookCountHidden(strHiddenWorkbookNames())

fstrActiveWorkbookName(intPathName)

fvntWorkbookNames

sActiveWorkbookNames(strNames())

sActiveWorkbookSheetNames(strArray(), intOrder)

Custom VBA Procedure Summary

This section provides a summary of all PRIME 5 for Excel custom VBA procedures. The summary table is sorted in alphabetical order by procedure name.

For an overview of PRIME 5's custom VBA procedures, see You Can Call PRIME 5 Custom VBA Functions and Subroutines.

For a listing of PRIME 5's custom VBA procedures by category, see Custom VBA Procedure Categories.

Procedure Name	**Description**
fblnDDEFileStatus	Determine if a particular file is currently loaded in the DDE server application (see fblnDDEFileStatus(strSourceFileName, strShortTopic, intSysChanNum))
fblnIsValidToolbarName	See if a given string is a valid toolbar name (see fblnIsValidToolbarName(strToolbarName))
fblnIsWorkbookOpen	Indicates whether a specific fully-qualified workbook is open right now (see fblnIsWorkbookOpen(strFullName))
fblnNullMenuState	Indicates whether Excel is in null menu state (see fblnNullMenuState)
fblnObjectExists	Determines if an object in a particular collection with a given identifier exists right now (see fblnObjectExists(objCollection, vntIdentifier))
fblnOnlyOneTrueElement	Looks at an array of Booleans to see if only one element is True (see fblnOnlyOneTrueElement(vntArrayOfBooleans))
fintActiveSheetType	Positively identify the active sheet's type using a proprietary return code scheme (see fintActiveSheetType)
fintAppLoad	Loads an application if it isn't already running, leaves it alone if it is already running, and returns an integer indicating final status (uses Shell, not OLE Automation) (see fintAppLoad(strModuleName, strShellName))
fintFirstTrueElement	Looks at an array of Booleans and returns index of the first True one (see fintFirstTrueElement(vntArrayOfBooleans))
fintOAAppLoad	Uses OLE Automation to load an application if it isn't already running, connect to it if it is already running, and returns an integer indicating final status (see fintOAAppLoad(objOAServer, strOAServerName, strModuleName))
fintOAFileStatus	Uses OLE Automation to determine if a file is already open (see fintOAFileStatus(objOAServer, strShellName, strSourceFileName))
fintSheetType	A generalized form of fintActiveSheet() that works on any sheet, not just the active sheet (see fintSheetType(shtObject))
fintSplitVertical	Vertically splits the active window on a

	percentage basis (see fintSplitVertical(intPercent))
fintUsageCount	Get an application's true running instance count (includes hidden instances) (see fintUsageCount(hMod))
fintWindowCountHidden	Count the number of hidden windows in Excel (see fintWindowCountHidden)
fintWorkbookCountHidden	Count the number of open hidden workbooks (see fintWorkbookCountHidden(strHiddenWorkbookNames()))
flngBinSearch	Does a binary search of a string array's elements and returns the 0-based index of the matching element if a match is found (see flngBinSearch(strList(), lngCount, strVal))
flngBinSearchInit	Same as flngBinSearch but searches for an initial match (see flngBinSearchInit(strList(), lngCount, strVal))
flngDeleteFromArray	Delete an entry from a string array (see flngDeleteFromArray(strArray(), lngSize, lngDeletePt))
fobjSelectedObject	Return an object that is represented by a specific, selected item (the upper-most one) in a dialog box's multi-select list control (see fobjSelectedObject(objCollection, strListControl(), objListControl))
fstrActiveWorkbookName	Provide the active workbook's 8.3 or full name (see fstrActiveWorkbookName(intPathName))
fstrFileNameFromPath	Strips the 8.3 filename from a fully qualified filename (see fstrFileNameFromPath(strSourceFileName))
fstrGetCellContents	Provide a cell's contents in string (viewable) format (see fstrGetCellContents(vntRange))
fstrGetHostTitleBar	Get Excel's current title bar text (see fstrGetHostTitleBar)
fstrGetRangeAddress	Determine a range name's address in A1 or R1C1 notation (see fstrGetRangeAddress(strRangeName, intNotation))
fstrGetRangeFromRefersTo	Parses the range from a Name object's RefersTo property string (see fstrGetRangeFromRefersTo(strRefersTo))
fstrGetSheetFromRefersTo	Parses the sheet name from a Name object's RefersTo property string (see fstrGetSheetFromRefersTo(strRefersTo))
fstrPathFromParts	Assembles a complete path name from its parts, consisting of the path, a backslash if necessary, and the filename (see fstrPathFromParts(strPath, strFile))
fstrReplaceMarker	Provide a simple way to replace a marker in global constant message strings with context-relevant, current information (see fstrReplaceMarker(strMsg, strReplacement))
fvntGetCellResults	Provide a cell's results (see fvntGetCellResults(vntRange, blnFormatting))

fvntMax	Returns the maximum of two values (see fvntMax(x, y))
fvntMin	Returns the minimum of two values (see fvntMin(x, y))
fvntOAFileNames	Uses OLE Automation to provide an array of the full names (path plus 8.3 name) of all open (hidden and unhidden) files in the server (see fvntOAFileNames(objCollection))
fvntOAWindowFileNames	Uses OLE Automation to provide an array of the full names (path plus 8.3 name) of all files in each of the server's child windows (see fvntOAWindowFileNames(objCollection))
fvntWorkbookNames	Provide an array of the full names (path plus 8.3 name) of all open (hidden and unhidden) workbooks (see fvntWorkbookNames)
sActiveWorkbookNames	Load an array argument with the active workbook's names (see sActiveWorkbookNames(strNames()))
sActiveWorkbookSheetNames	Returns a list of all the active workbook's sheet names, either in positional (collection) order, or sorted alphabetically (see sActiveWorkbookSheetNames(strArray(), intOrder))
sGetToolbarButtonInfo	Return a toolbar's contents (see sGetToolbarButtonInfo(strTheToolbarName, strButtonInfo()))
sInsertInArray	Inserts an entry in a string array (see sInsertInArray(strArray(), lngSize, lngInsertPt, strElement))
sMsgStrArray	Development utility to display the first *n* elements of an array (see sMsgStrArray(strArray(), intElements, strName))
sNotImplemented	Display a "not implemented" message for work-in-process features (see sNotImplemented(strDboxTitle, strPrompt))
sOAGetOpenFiles	Uses OLE Automation to stuff an array with the fully qualified path-and-filenames of only non-macro child windows (see sOAGetOpenFiles(objOAServer, strShellName, strFileNames()))
sOAShutDownApp	Use OLE Automation methods to shut down an OLEAuto-invoked application (see sOAShutDownApp(objOAServer, strShellName, intAppStatus))
sShellSort	Sort a string array (see sShellSort(strArray()))
sShutDownApp	Use DDEExecute commands to shut down a DDE-invoked application (see sShutDownApp(intSysChanNum, strShellName intAppStatus))
sWaitCursor	Change the mouse cursor to an hourglass or back to a standard arrow (see sWaitCursor(blnState))

Cell Protection Viewer

PRIME Cell Protection Viewer provides an at-a-glance visual display of all unlocked or locked (or both) cells in the current worksheet. The utility can examine either the current selection or the used area. You can also set your own preference settings for the desired outcome of both cell protection states.

If the current worksheet is protected with a password you will be prompted with the Unprotect Sheet dialog. If you enter the correct password, the utility will continue and then prompt you to protect the worksheet again if you so desire. If you do not enter the correct password then a message box will explain that you cannot continue.

Specify the range to scan by selecting one of the **Current Selection** or **Used Area** option buttons in the **Range** frame. The used area − also called the "used range" − of a worksheet is defined as the range bounded by the upper left corner cell (A1 or R1C1) through the current lower right corner cell (the one you get to when you press Ctrl + End).

Specify the cells affected by selecting one of the **Unlocked**, **Locked**, or **Both** option buttons in the **Cells Affected** frame. This determines which protection state will be tested for when the sheet is scanned and so determines which cells will be formatted.

Click the **Setup** button to choose your own color preferences for both unlocked and locked cells. The utility's default color for unlocked cells is yellow. The utility's default color for locked cells is gray. The **Setup** push button is a toggle. When you click **Setup** the dialog box gets taller momentarily and the **Setup** button label changes to **Save**. When you click **Save** the current settings are saved in PRIME5.INI, the dialog is reduced to its original size, and the button label changes back to **Setup**.

When the dialog is in its taller Setup mode, three new controls appear. A **Color** drop-down list box allows you to choose a color format for cells affected (locked or unlocked). The **Defaults** button returns the format applied to unlocked and locked cells to the utility's default colors, yellow and gray respectively. This only affects what the utility suggests as defaults from this point forward, it does not ever apply these colors until you explicitly click **OK** in the main dialog box. To clear all pattern formatting from a category of cells, select the "(none)" item at the top of the **Color** drop-down list.

If you want to save your preferences, click the **Save** button, otherwise click **Cancel**.

Click the **OK** button to invoke Cell Protection Viewer and change the color of all unlocked or locked (or both) cells in the chosen range.

Click **Cancel** to dismiss the utility.

Click **Help** to get context-sensitive help on this utility.

Create Program Manager Icon

From within Excel, create a Program Manager Item for the current workbook in the Program Group of your choice. You can also add a new Program Group, all from the same dialog box.

When invoked, this utility first checks to see if the current workbook has been saved. If it has not been saved, then it prompts you via the Save As dialog to save the file. If you save the file, the utility's main dialog box appears, but if you cancel out of Save As then a message box explains you must save a file at least once before creating a Program Manager icon for it.

The **Choose Existing Group** list box displays all Program Groups sorted alphabetically.

The **New Item Description (40 chars max)** text box displays the current workbook name in its DOS 8.3 filename format. You can overtype this if you like. Program Item descriptions are limited to a maximum of 40 characters.

The **Items Currently in Chosen Group** list box displays all Program Items in the current Group, sorted alphabetically.

The **Add Group** button invokes a new dialog box panel with an **Add a New Group** text box and the prior **New Item Description (40 chars max)** text box. Once you enter a new Group and click **OK**, the new Group is added to Program Manager and the current document is added to the new Group. The new Item's description comes from the **New Item Description (40 chars max)** text box.

Click **Help** to get context-sensitive help on this utility.

Name Assistant

PRIME Name Assistant provides a list of all names in the current workbook, *including hidden names*, and displays statistics for those names. Name Assistant also allows you to go to a range name's location, paste a list of all names and their statistics into a worksheet, toggle names between visible and hidden, and delete names.

If you're familiar with what a name in Excel is, keep reading. Otherwise, you might want to study the section A Primer on Excel Names, then come back to this topic. Here is a list of Name Assistant's advantages over Excel's native name-handling features.

1. PRIME Name Assistant shows you absolutely all names.

   Excel's Name Box only displays the first 100 names, so if you've got a workbook with more than 100 names you won't see all of them in the Name Box. (Unlike the Name Box, Excel's Edit Go To dialog box and Excel's various Insert Name dialog boxes do not truncate after the 100th name.)

2. PRIME Name Assistant displays all names, both visible and hidden.

   Excel, by design, doesn't display hidden names. You can look for them in the Name Box, the Edit Go To dialog, and the various Insert Name dialogs but none of them reveal hidden names. Even the Insert Name / Paste Name dialog's Paste List feature doesn't reveal hidden names. This isn't a big surprise since a hidden name is, well, hidden. There's nothing wrong with Excel's design per se, but if you have a legitimate need to see hidden names then PRIME Name Assistant is for you.Excel, by design, doesn't display hidden names. You can look for them in the Name Box, the Edit Go To dialog, and the various Insert Name dialogs but none of them reveal hidden names. Even the Insert Name / Paste Name dialog's Paste List feature doesn't reveal hidden names. This isn't a big surprise since a hidden name is, well, hidden. There's nothing wrong with Excel's design per se, but if you have a legitimate need to see hidden names then PRIME Name Assistant is for you.

3. PRIME Name Assistant's list of names is multi-selectable.

   None of Excel's native name-handling features are multi-selectable, only single-selectable. This means, for example, using Excel's Define Name dialog you can only delete one name at a time. With PRIME Name Assistant you can delete several or even all names from the same list. (Deleting all names our way is a quick three-click process.)

4. PRIME Name Assistant allows you to toggle a name between hidden and visible.

   Excel's native name-handling feature set doesn't allow this.

5. PRIME Name Assistant displays the full name of every name, which makes it possible to distinguish between (and go to) duplicate book-level and sheet-level names.

   Excel's native name-handling feature set makes it almost impossible to dig your way out of the dark, deep hole that awaits you if you should ever use the same name in both a book-level and a sheet-level context. We advise against having names in the same book like "Snout" and "Sheet1!Snout" but if you should do this, PRIME Name Assistant reveals (and goes to) these names regardless of which sheet is active.

The **Names** list box shows all visible and hidden names in the current workbook. *Name Assistant's list box is multi-select!* For more information about names in Excel, see A Primer on Excel Names.

The **Statistics** frame displays the reference behind the current name and its visibility status.

Click the **Go To** button to activate the range referred to by the current name. You can also go to a name by double-clicking it. An appropriate, graceful error message appears if the selected name is a range that can't be activated on screen; for example, if the name points to a 3-D range or if the name represents a named formula. (This feature acts only on the upper-most name in a multiple selection.) *Due to a bug in Excel, while a custom dialog box is displayed, Excel will not display the current selection or the active cell.* The name's sheet will be activated, but you won't be able to see the range in the usual way without canceling the dialog box. Our workaround is to align the active window's upper-most row and left-most column in such a way that these row and column headings "reveal" the active cell's address. See the next paragraph for details.

The **Align window with range's upper left corner** check box provides a workaround for a bug in Excel. The bug is this: while a custom dialog box is displayed, Excel will not display the current selection or the active cell. It will allow you to change sheets, and make selections, but you won't be able to actually see the active sheet's selection or the active cell on your screen until you close the dialog box. Even though you could look at the **Statistics** frame's text to see what the target name's range is, we also give you this alignment check box option. If **Align window with range's upper left corner** is checked (the default) when you **Go To** a name, Name Assistant forces the window's upper-most row and left-most column to match the name's upper-left corner cell row and column. This way the row and column headers reveal the active cell's address while the active cell is handily tucked into the window's upper left corner. If this check box is cleared when you **Go To** a name, this alignment doesn't occur. Name Assistant remembers this check box's setting based on the setting in effect when you last closed the dialog box (the setting is stored for you in PRIME5.INI). It's a "set once and forget it" preference that you can still override at any time.

Click the **Paste Names** button to paste, at the location of the active cell, three columns of information about each name: (1) the name itself, (2) the name's visibility status, and (3) the range or formula the name refers to. This feature is only available when the current sheet is a worksheet, Excel 4 macro sheet, or Excel 4 international macro sheet. *This feature warns you prior to the paste that if there is data in the destination area of the current worksheet, the data will be overwritten and that the action can't be undone, at which point you can proceed or cancel the operation.* (This feature always includes all names regardless of the current selections in the **Names** list box.)

Click the **Toggle Visible** button to toggle the visibility status between visible and hidden. (This feature acts on all the names in a multiple selection.)

Click the **Delete** button to delete names. (This feature acts on all the names in a multiple selection.)

Click the **Cancel** button to exit the utility.

Click **Help** to get context-sensitive help on this utility.

Related Topics:

A Primer on Excel Names

A Primer on Excel Names

What exactly is a **name** in Excel? You can name a cell, a group of cells (contiguous or noncontiguous), a row, a column, a group of rows and/or columns, or the entire bloody worksheet. *A name is usually referred to as a range name even if it references a single cell.*

You can use alphanumeric characters in your names, so long as you do not try to use a name that Excel might mistake for a cell reference. For example, you cannot name a cell A1, A1, $A1, or R1C5, no matter what notation style you are currently using. You're limited to 255 characters. Spaces are verboten. You can't start a name with a number, period, or question mark. You can't partake of most nonalphanumeric characters, although you can use:

- the underscore character _
- the backslash character \
- the period . (but not as the first character)
- the question mark ? (but not as the first character)

You should use proper case for your range names. For example, you could use a name like MyCoolCell. This has several benefits. It makes the name easier to read, does not require you to type separator characters like those just mentioned, and if you type the name in lowercase in a formula and Excel recognizes it, Excel switches it to proper case.

The Name Box is a drop-down list box that displays the first 100 range names in the current sheet and/or book. This list is sorted alphabetically. You have to watch the 100-name limitation because once you get used to always using the Name Box, you may think a name is missing in action when in reality you exceeded the list limit.

A book-level name is a defined range that means the same range on the same sheet no matter which sheet in the book you refer to it from. For example, you select cell A1 on Sheet1 and name it "Lobster." Say you add several sheets to this book. You are working in Sheet3 and want to reference the total in cell A1 on Sheet1. No problem, just a quick =Lobster and you've got it. Pull down the Name Box (or pop up the Go To box), select Lobster, and there you are. Sheet1 cell A1. Every time. From any sheet in the book.

A sheet-level name is local to the sheet that owns it. How do you create a sheet-level name? You define a sheet-level name the same way you do a book-level name, only you preface the name with the sheet name. It works like this: if you select cell B2 on Sheet2 and you want to make it local (sheet-level) to that sheet, you might name it Sheet2!Snout instead of just Snout. The name of the sheet precedes the range name and is separated from the name by an exclamation point. The name is Snout, but it is local (sheet-level) to Sheet2. If you go to Sheet1 and pull down the Name Box, Snout does not appear. It doesn't exist on Sheet1 nor is it a book-level name available across the entire book. You can refer to it if you include the sheet name in the reference. A formula on Sheet 1 like =Sheet2!Snout returns the contents of B2 from Sheet2. When dealing with sheet-level names, be aware that they take precedence over identical book-level names.

A 3-D name is a name applied to a 3-D reference, which in turn is a reference to a range of sheets in a workbook. You can create a 3-D name only through the Define Name dialog box. You type in a name and, in the Refers to text box, you type in the 3-D range using the syntax =Sheet1:Sheet3!A1, which is the same syntax as a 3-D reference that you might use in a formula. If you create this 3-D name and call it

"Sales," you could enter a formula like =SUM(Sales) anywhere in a worksheet in this book and get the sum of cells A1 in sheets 1, 2, and 3. 3-D names only appear in the Define Name dialog box. You won't find them in the Go To dialog box or the Name Box.

A named formula is a name that refers to either a constant or a formula. You can create a variable, that is, a range name, and set the name equal to a constant or a formula. Say you want to use a percentage in different calculations scattered throughout your sheet. The usual way is to stick the percentage in a cell and either name it and use the name or use the cell reference wherever you need to reference that percentage. But what if you don't want the percentage on the worksheet? No problem, pop up the Define Name dialog box, enter a name like OverHead, and type =3% in the Refers to text box. The 3% is not entered anywhere on the sheet. You can define a name to refer to a formula that does not exist on any sheet. To have a name refer to a formula (not simply a constant), create a name and in the Refers to text box type in the formula.

A hidden name is a name that is not visible. It doesn't appear in Excel's Name Box, Edit Go To dialog, or any of the various Insert Name dialogs. Names you create via Excel's user interface are visible by default, except for named formulas which are hidden by default.

(Portions of this section reprinted by permission of Addison-Wesley from The Underground Guide to Excel 5.0 for Windows).

Toolbar Assistant

PRIME Toolbar Assistant provides a complete list of all active toolbars by name (this includes all hidden and visible toolbars), along with their button contents. Button contents include slot numbers and assignments, ToolTip text, the fully qualified macro assignment, whether the button is built-in or custom, and the button id number. Toolbar Assistant allows you to add or change the ToolTip text and the macro assignment for any button on the fly. You can catalog this comprehensive list into a table on a worksheet.

The **Toolbar Names** drop-down list box shows all toolbar names listed in the order in which they appear in the native Toolbars dialog (not alphabetical). Selecting a toolbar other than Standard updates the **Button Contents** list box, which contains a list of each button slot and its current assignment (a blank represents a space or gap on the toolbar, which *does* occupy a slot). Button separator slots (slots occupied by a space instead of a button image) are represented as "<space>" in the utility's dialog box and the catalog.

The **ToolTip** text box displays the ToolTip text for the current toolbar button (the one selected in the **Button Contents** list box).

To update a toolbar button's ToolTip, type new text in the **ToolTip** text box then click the **Update ToolTip** button. The change takes effect immediately. Gap buttons can't have a ToolTip, and if you try it anyway the utility produces a message box to this effect. If you set a toolbar button's ToolTip to an empty string, Excel automatically sets the ToolTip to read "Custom" and there's nothing you can do about it.

If a toolbar button is assigned to a custom (user-defined) macro, the **Macro** text box displays the fully-qualified name of that macro. If a toolbar button is a built-in button, the **Macro** text box is empty.

To update a toolbar button's assigned macro, type a new name in the **Macro** text box and then click the **Update Macro** button. The utility doesn't validate the macro name you enter here.

The **Read-only** frame contains two text boxes that display read-only values for the current toolbar button. The **Built-in** text box displays "Built-in" if a toolbar button is built-in (including gap buttons) and "Custom" if it is a custom button. The **ID** text box displays the toolbar button identification number if a toolbar button is built-in. The **ID** text box is empty if a toolbar button is custom. Note that in this context the terms built-in and custom refer to the button object itself, not its face (the image on the button). Meaning, a built-in toolbar button is one that ships with Excel and a custom toolbar button is one that does not ship with Excel. These terms describe the button object itself, not the button face.

The **Log All** button generates a six-column table at the active cell location that you can then print or save. Toolbar Assistant's log table column headings are:

- Toolbar Name
- Button Slot Number
- Button Name (ToolTip Text)
- Button Macro Assignment (individual cell contents is #N/A if none)
- Button Built-in vs. Custom Status
- Button ID Number

Note that in Excel VBA, toolbar name strings are case insensitive.

Click **Help** to get context-sensitive help on this utility.

TUGExcel Tools

Popularized in The Underground Guide to Excel 5.0 for Windows, these six handy utilities fire up Excel's most frequently used features with a single click. This is what Making It Easier(TM) is all about!

Related Topics:

Edit Go To Special

Exit Excel

File Close

Formula Flipper

New Workbook

Notation Style Flipper

Edit Go To Special

Ordinarily, getting to the Edit Go To Special dialog box is a little convoluted. You have to pull down the Edit menu, click Go To, and then, from the Go To dialog box, click the Special button. Whew, finally the Go To Special dialog box is displayed. But with the TUGExcel Edit Go To Special utility, it's just one click away.

Click on the Edit Go To Special button on the TUGExcel Auditing toolbar. For more information see Edit Go To Special (TUGExcel Auditing).

Exit Excel

Once you've used this little gem, you won't be able to live without it! TUGExcel Exit Excel dismisses Excel with one simple click. If you have unsaved or changed files in your workspace, Excel prompts you to save them as usual.

Click on the Exit Excel button on the TUGExcel Standard toolbar. For more information see Exit Excel (TUGExcel Standard).

File Close

TUGExcel File Close improves your productivity by 50%! (Well, at least for every File Close process.) TUGExcel File Close is a single-click tool that closes the active workbook and all its open windows. If the workbook is unsaved or has changed, Excel prompts you to save it as usual.

Click on the File Close button on the TUGExcel Standard toolbar. For more information see File Close (TUGExcel Standard).

Formula Flipper

The TUGExcel Formula Flipper utility proves invaluable every single time you audit a workbook. One click toggles your display between result view and formula view.

Click on the Formula Flipper button on the TUGExcel Auditing toolbar. For more information see Formula Flipper (TUGExcel Auditing).

New Workbook

The TUGExcel New Workbook utility gives you single-click access to all your Excel workbook templates, unlike Excel's native feature that forces you to accept the default workbook template.

Click on the New Workbook button on the TUGExcel Standard toolbar. For more information see New Workbook (TUGExcel Standard).

Notation Style Flipper

The TUGExcel Formula Flipper utility proves invaluable every single time you audit a workbook. One click instantly toggles your worksheet display between A1 and R1C1 notation style.

Click on the Notation Style Flipper button on the TUGExcel Auditing toolbar. For more information see Notation Style Flipper (TUGExcel Auditing).

Window Assistant

PRIME Window Assistant allows you to select for closing one, several, or all visible Excel windows. You can also create a new window, split a window, arrange windows, and activate a specific window.

The **Windows** list box shows all visible windows in the current Excel session. *Window Assistant's list box is multi-select!* As you navigate the **Windows** list either with the mouse or the arrow keyboard keys, the label below the list box displays the name of the active sheet for that particular window.

To close one window, simply select the window from the list and click **Close Selected Window(s)**. You can also close a window by double-clicking it.

To close more than one selected window, make the multiple selection and click **Close Selected Window(s)**. When closing multiple windows that share the same workbook (for example, TUG.XLS:3 and TUG.XLS:2 and TUG.XLS:1), when Excel finally gets down to the last such window for that workbook it determines if the workbook has been changed and prompts you to save it accordingly. You can save it, not save it, or cancel the operation entirely as you would normally.

To close all windows, click **Close All Windows**. Window Assistant closes all listed windows, then ends automatically. This leaves Excel in null menu mode, with no visible workbooks open and only File and Help in the menu bar. As with **Close Selected Windows(s)**, Excel always prompts you to save, not save, or cancel for any dirty workbook(s).

New Window opens a new window with the same contents as the current window in the **Windows** list and then activates that new window. This is equivalent to selecting Window then choosing New Window in Excel's native menu. (This feature acts only on the upper-most window in a multiple selection.)

Split a Window splits the current window in the Windows list into two equally sized horizontal panes, and then activates that window. This is equivalent to selecting Window then choosing Split in Excel's native menu. Clicking on **Split a Window** a second time has no effect. (This feature acts only on the upper-most window in a multiple selection.) Only worksheets, macro sheets, and dialog sheets can be split by VBA. If the current window in the list is not a worksheet, a macro sheet, or a dialog sheet, the utility gives you an error message to this effect.

Arrange All Windows invokes Excel's native Arrange Windows dialog box and you can interact with it as you ordinarily would.

Activate and Wait activates the current window (or the upper-most window in a multiple selection) and leaves the dialog running.

Activate and Quit activates the current window (or the upper-most window in a multiple selection) then quits the utility.

Click **Help** to get context-sensitive help on this utility.

Zoomer

PRIME Zoomer makes it easy to control your worksheet's zoom setting. You can control the zoom setting with a slider (like the controls on a stereo) or select it from customized lists of recent and fixed settings.

There are three ways to use the slider:

- Drag the thumb bar to the setting you want. Watch the **Zoom** control to see how the zoom will be set. When the zoom value is right, click OK.
- Click the slider to the left or right of the thumb bar to move the zoom setting down or up 1% at a time. Click OK.
- Double-click at any point on the slider to move the thumb bar to that point. Click OK.

There are other ways to change the zoom setting:

- Type a new value into the **Zoom** control. Click OK.
- Click an entry in the list of **Previous** settings or **Fixed** settings. Click OK.
- Double-click a **Previous** or **Fixed** entry. (You need not click OK.)
- Click **Fit Selection**. (You need not click OK.)
- Click **Fit** Used Area. (You need not click OK.)

Helpful hints:

- The **Customize** button opens the **Zoomer Customize** dialog box. This box controls aspects of PRIME Zoomer's operation such as the number of previous zoom settings in the **Previous** list and the contents of the **Fixed** list.
- The lists of **Previous** and **Fixed** settings are enabled only when there actually are previous settings and fixed settings to select.
- The slider scale shows the range of settings that all of the PRIME Zoomer controls accept. If you wish, you can change this scale to be narrower than the native View Zoom command's range of zoom settings (10% to 400%). Narrowing the range of the slider scale makes the slider more precise. Click the **Customize** button to change the slider's scale.

The **Previous** list box contains up to five of the most recent zoom values you have set with PRIME Zoomer. To change the number of values in the **Previous** list box, click the **Customize** button.

The **Fixed** list box contains up to five fixed zoom values that you use frequently. To add, delete, or change the values in the **Fixed** list box, click the **Customize** button.

The **Fit Selection** button sets the zoom to the largest value that lets only the selected range appear in the active window.

The **Fit Used Area** button sets the zoom to the largest value that lets only the used area appear in the active window.

Click **Help** to get context-sensitive help on this utility.

Related Topics:

Zoomer Customization

Zoomer Customization Error Messages

Zoomer Customization

The **PRIME Zoomer Customize** dialog box lets you customize PRIME Zoomer's behavior.

Slider Range Minimum and **Maximum** set the range of the slider. These values also set limits on the values that may appear in the **Previous** and **Fixed Settings** lists, and the values you may enter in the **Zoom** control.

Previous Settings controls the number of previous settings that the **Previous** list will retain. It may be set to any value from 0 (an empty list) to 5.

Fixed Settings lets you specify up to five values to appear in the **Fixed** list. To make less than five values appear in the list, set one or more to 0%. The 0% values need not be the last ones. Non-zero values will appear in the **Fixed** list in the same order as in this dialog box.

If you enter an invalid value in any field, PRIME Zoomer will beep and display a message on the status line when you try to leave the field. For more detailed explanations of these errors, see Zoomer Customization Error Messages.

Zoomer Customization Error Messages

Minimum must be at least 10% & no greater than Maximum.

The value of **Slider Range Minimum** is invalid. It must be a number no smaller than 10 (10%) and no greater than the value of **Slider Range Maximum**.

Maximum must be no less than Minimum & no greater than 400%.

The value of **Slider Range Maximum** is invalid. It must be a number no smaller than the value of **Slider Range Minimum** and no greater than 400 (400%).

The number of previous settings must be 0 to 5.

The value of **Previous Settings** is invalid. It must be a number in the range 0 to 5. A value of 0 makes PRIME Zoomer's **Previous** list empty; you will not be able to select it.

A Fixed Setting must be at least as great as Minimum and no greater than Maximum, or zero.

One of the **Fixed Settings** is invalid. Each **Fixed Setting** must be a number no smaller than the value of **Slider Range Minimum** and no greater than the value of **Slider Range Maximum**.

The **Fixed Settings** may also be zero. A zero setting is inactive; it will not appear in PRIME Zoomer's **Fixed** list.

Glossary of Terms

3-D Name
Bang Character
Book-level Name
Hidden Name
Name Box
Named Formula
ODK
PRIME 6 for Word
Sheet-level Name
The Underground Guide to Excel 5.0 for Windows
The Underground Guide to Microsoft Office, OLE, and VBA
Used Area
VBA
WOPR

3-D Name

A name applied to a 3-D reference, which in turn is a reference to a range of sheets in a workbook.

Bang Character

The term "bang" is slang for the exclamation mark character.

Book-level Name

A defined range that means the same range on the same sheet no matter which sheet in the book you refer to it from.

Hidden Name

A hidden name is a name that is not visible. It doesn't appear in Excel's Name Box, Edit Go To dialog, or any of the various Insert Name dialogs. Names you create via Excel's user interface are visible by default, except for named formulas which are hidden by default.

Name Box

A drop-down list box, located immediately to the left of the formula bar, that displays the first 100 range names in the current sheet and/or book. This list is sorted alphabetically.

Named Formula

A name that refers to either a constant or a formula

ODK

The Microsoft Office Developer's Kit is comprised of a book for designing and programming integrated solutions, the Object Model Reference Chart, and a CD-ROM. You can get an ODK of your very own by calling 800-426-9400 extension 11771 and the nice Developer Services folks will sell you a copy for $100 or so. Or, if you have Visual Basic 3 Professional it comes with! If you have an older version of VB3-Pro that did not come with the ODK, call the 800 number above and you can get it for $50. And if you subscribe to MSDN, the ODK is on the MSDN CD-ROM. You don't get the hard copy components (the book and the Reference Chart) but their components are on the CD.

PRIME 6 for Word

A shareware add-in for Word 6 from PRIME Consulting Group. PRIME 6 for Word addresses many of Word's, er, uhm, shortcomings, and takes many of Word's good features and makes 'em even better! According to Woody Leonhard, "PRIME 6 is fantastic! A great set of Word enhancements. Don't run Word without it."

PRIME 6 includes these utilities: *Fast Spell Checker* for quicker spell checking with all misspelled words in a document available in a single list; *FileNew* for complete template management; *AutoText Lister* for viewing local and global AutoText contents and statistics side-by-side; *Window Manager* for one-stop window management; *Document Variable Manager* for adding, viewing, modifying, deleting or cataloging those pesky document variables; *Bookmark Manager*, the definitive utility for dealing with bookmarks; plus *Zoomer, Macro Manager, Toolbar Lister, ResetChar, ProgMan Icon Generator, 36 custom WordBasic subroutines and functions,* and a list of Word and WordBasic bugs and workarounds.

To place an order call Advanced Support Group at 800-788-0787 or 314-965-5630.

Sheet-level Name

A sheet-level name is local to the sheet that owns it.

The Underground Guide to Excel 5.0 for Windows

The Underground Guide to Excel 5.0 for Windows by Lee Hudspeth and Timothy-James Lee, published by Addison-Wesley, ISBN 0-201-40651-9, $19.95 U.S., $25.95 Canada.

To get a copy, enter your local bookstore and tell the clerk, "Yo, I hear TUGExcel is on the New York Times' Best Seller list this week. Again. And I'd like to buy, oh, 20 or 30 copies, thank you." Note: If the bookstore does not have any copies, grab the store manager by the lapels and, with a feverish look in your eyes, scream, "I've gotta get UNDERGROUND!" Be sure to have bail money handy if this approach is taken. If all else fails, you can always order the book directly from Addison-Wesley by telephone (via USA Direct Sales)! The number to call is: 1-800-822-6339.

The Underground Guide to Microsoft Office, OLE, and VBA

Used Area

Also called the "used range." This range of a worksheet is defined as the range bounded by the upper left corner cell (A1 or R1C1) through the current lower right corner cell (the one you get to when you press Ctrl + End).

VBA

Visual Basic for Applications, also called Visual Basic, Applications Edition.

WOPR

Woody's Office POWER Pack, a collection of Word 6 add-ins from Pinecliffe International. To place an order for WOPR 6, call 800-659-4696 (or 314-965-5630 outside the US). For additional information call 314-965-5630.