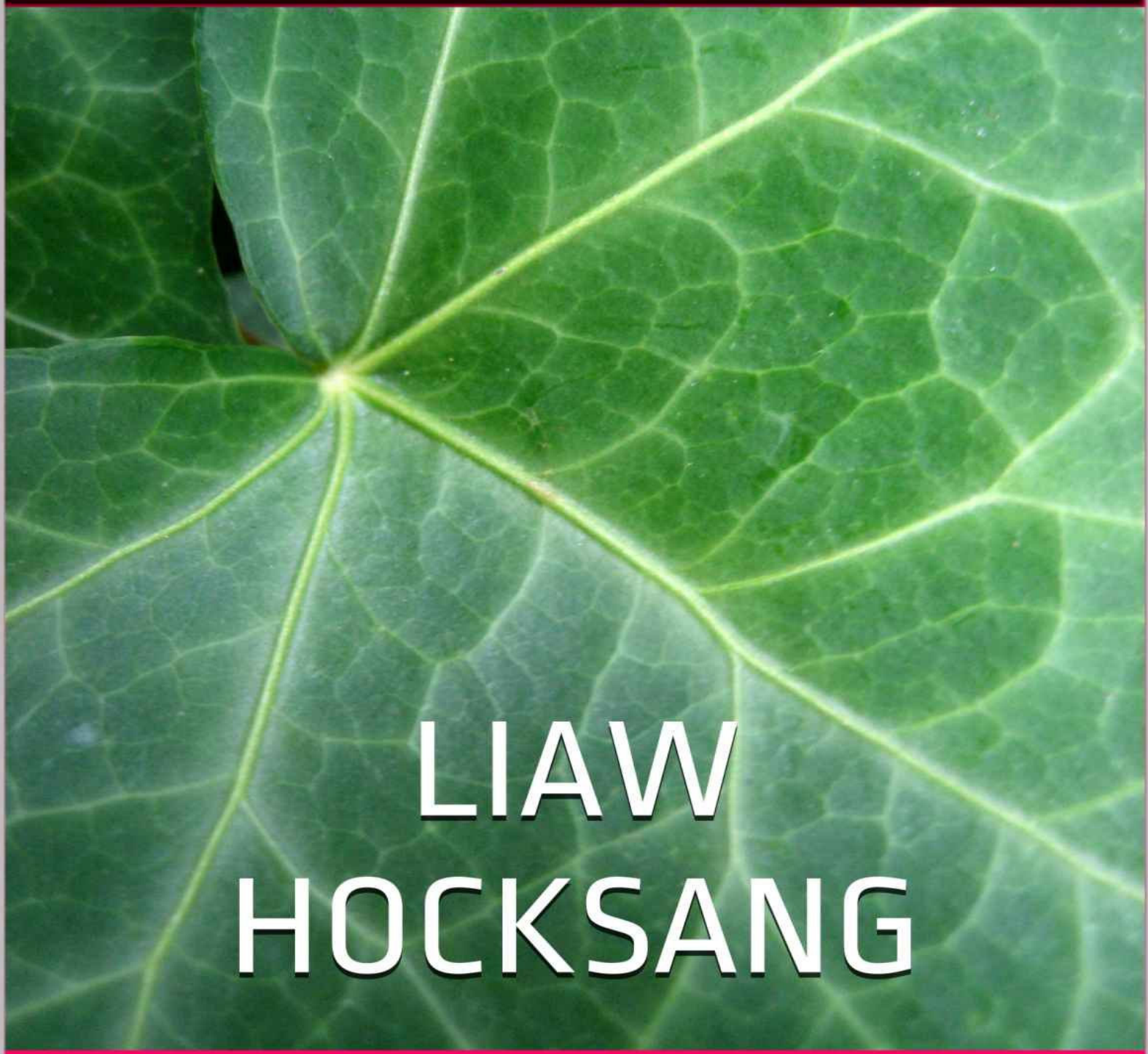
Dissect and Learn Excel® VBA in 24 Hours

changing workbook appearance



LIAW HOCKSANG

Dissect and Learn Excel® VBA in 24 Hours – Changing workbook appearance

For permission requests, please write to: liawhocksang@gmail.com

Trademarks and Copyrighted Content

Every effort has been made to appropriately capitalize all terms that are known to be trademarks or service marks mentioned in this book. The author cannot attest to the accuracy of this information. Use of a term in this book should not be regarded as affecting the validity of any trademark or service mark. Should there be any violations in this respect, the author apologies and shall stop the selling of the book **Dissect and Learn Excel® VBA in 24 Hours - Changing workbook appearance** within the control of the author.

Microsoft, Excel, Word, Outlook, Internet Explorer, Visual Basic, and Windows are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries. All other trademarks and trade names are the property of their respective owners.

Warning and Disclaimer

Every effort has been made to make this book as accurate as possible. However, no warranty or fitness is implied. The author and the publisher shall have neither responsibility nor liability to any person or entity with respect to any losses or damages arising from the information contained in this book.

Thank you

Thank you for purchasing this book. This book is for your personal use only. It may not be resold or given away to other people. If you are reading this book without purchasing it, please destroy it and you may make a purchase at Amazon.com. Thank you for your support and respecting the hard work of the author.

Table of Contents

Introduction

No matter how complicated a program is, it is made of many smaller and tiny fundamental working parts of programming code. Each of them accomplishes a specific task. Some may just consist of only one or a few lines of code. Knowing the functions of these fundamental working parts, you can then easily write an unlimited number of working programs. And knowing them, you can easily understand the programs written by others and adopt into your programs the ideas and the VBA code that are presented in those programs.

Dissect and Learn Excel VBA in 24 Hours is a series of quick references to VBA code for intermediate users who are looking for ideas and samples of VBA code to accomplish certain tasks when they are in the process of writing a program.

In this series, you will see thousands of tiny working parts of VBA code that are used to accomplish many simple and yet meaningful tasks. To add a new workbook, to auto-fill a range, to sort a table of data, to generate a table of contents of all chart sheets and worksheets, to loop through and manipulate a folder of Excel files, to add a control to the Ribbon, to send an email, and to login to an account in the Internet are some examples of these tiny working parts.

This series is for readers who have at least a basic understanding of Excel VBA programming. In order to follow the discussions in the series, a reader must know, for examples, what Sub procedures and Function procedures are, what Visual Basic Editor (VBE) is, how to add a VBA module to a workbook, how to turn on the AutoList Members option in VBE, how to use the macro recorder in Excel in order to find out the new methods and properties of objects that you are not familiar with, how to use the Object Browser window to check the complete list of members for a particular object, how to write some simple Sub procedures, in which VBA module you should store your VBA code, and how to use the debugging tools in VBE.

If you are new to Excel VBA, please teach yourself Excel VBA before exploring the contents in the series. You may refer to another book written by me entitled *Learn Excel® VBA in 24 Hours - A quick reference for beginners*, which was written for those who are new to Excel VBA.

I hope this series of books will serve as quick references in facilitating you to write an unlimited number of working VBA programs. Let Excel VBA work for you.

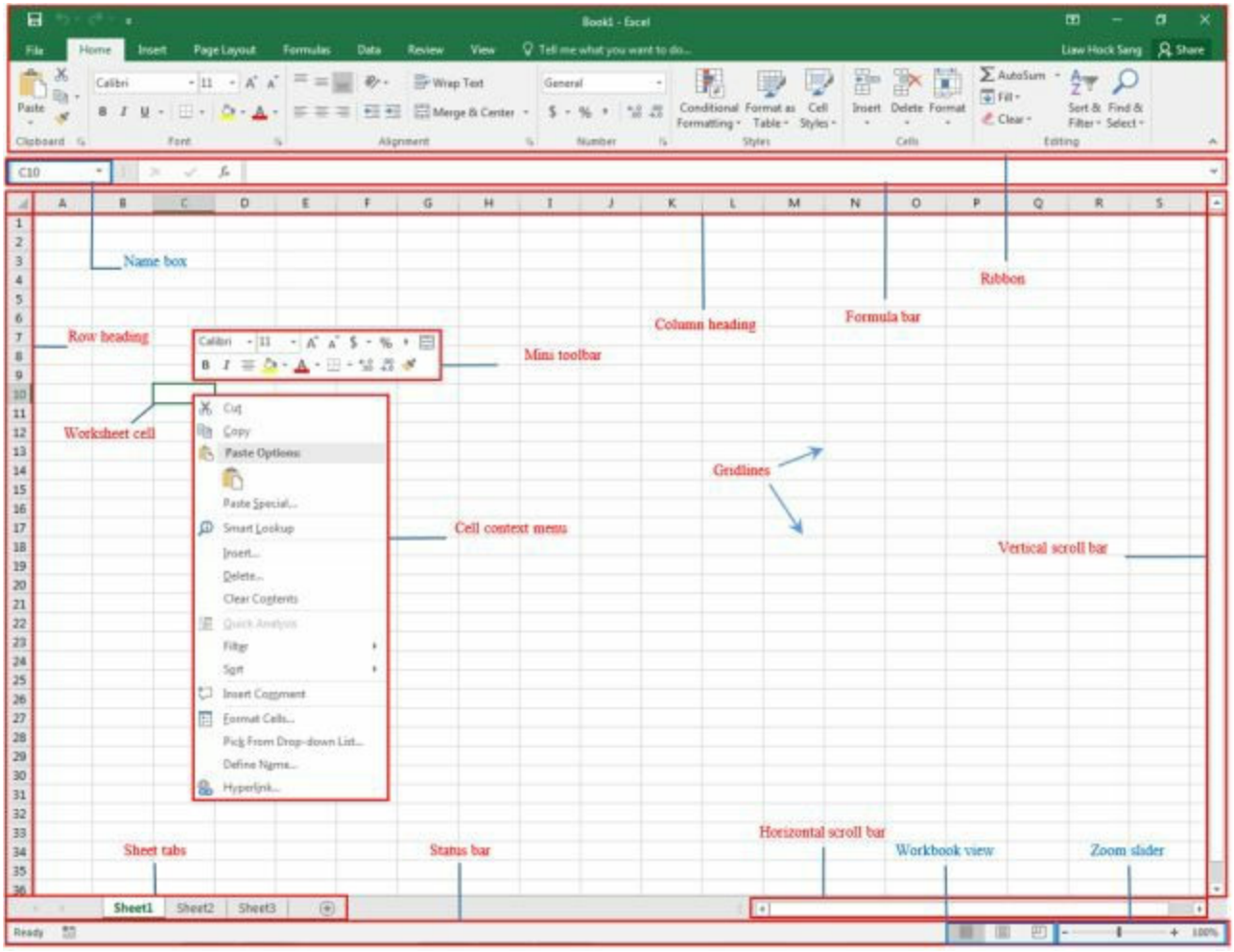
Book 1: Changing workbook appearance

Book 1: Changing workbook appearance focuses on changing the appearances of the worksheet cells, the worksheets of a workbook, the row and column headings, the sheet tabs, the layout view of a worksheet, the status bar, the formula bar, the Ribbon, and the Cell context menu. **Figure 1** shows the components that will undergo certain changes in their appearances. In this book, almost half of it covers on how to customize the Ribbon.

The main purpose of changing the appearance is to let the users to have a better working experience when they are using your program. The changes may affect from just a worksheet cell, a range of cells, and a worksheet to the entire active workbook and every opened workbook. To see how the code affects the appearance, simply copy and paste the code into the Immediate window, a VBA module, or Custom UI Editor, and run the VBA code (or step through the VBA code by using the debugger in VBE) or reopen the workbook file (that stores the code) in Excel.

In discussing the VBA code, the book does not list all the possible built-in constants, methods and properties of the discussed objects. Rather it lists samples of commonly used constants, methods and properties with some short explanations (and with hints on how to explore further) to illustrate the ideas in accomplishing certain tasks. You can then easily find others by exploring the Object Browser window and the VBA Help system. Otherwise, this book would be too thick and dull in repeating the same ideas.

Book 1 only focuses on changing the appearances in Excel 2007-2016 for Windows. All the VBA code and XML code in this book were tested in Excel 2007, 2010, and 2016. And the code should work in Excel 2013 too. The screenshots in this book were taken from Excel 2016. The book does not discuss the workings on cells, sheets, workbooks, and files. Only in the coming books in this series, you will encounter VBA code in, for examples, finding the last row and column of a named range, auto-filling a range, conditional formatting a range of cells, saving and retrieving the settings of your programs in the Windows Registry, copying between worksheets and workbooks, and automating the Outlook and Internet Explorer applications. This book ends with a sample program that adopts most of the ideas that are discussed in the book. You may download the program (for Excel 2007 and 2010-2016) to see how it changes the appearance of a workbook.



Cell context **Figure 1**: The discussed components in this book that will undergo certain changes in their appearances.

Cell formatting

This topic discusses on how to apply some formatting to a worksheet cell and a range of worksheet cells.

Font

```
'To get the content in a cell bold
'(namely, cell A1 in the sample code below)
'and to set its font name, size, and color
With Range("A1").Font
   .Bold = True
  .Name = "Arial"
  .Size = 13
  .Color = vbGreen
```

```
End With
```

To check other color constants, highlight vbGreen in VBE and press F1 to access related information in the VBA Help system.

Alignment

```
'To align the content in a cell
Range("A1").HorizontalAlignment = xlLeft
Range("A1").VerticalAlignment = xlTop

'To left-indent the content in a cell to level 2
Range("A1").IndentLevel = 2
```

Fill with color

```
'To fill the interior of a range (namely, range A1:C3)
'with a particular color
With Range("A1:C3").Interior
  .Color = vbBlack     'xlNone for no color
   .TintAndShade = 0.5  '-1 (darkest) to 1 (lightest)
  .ColorIndex = 3     'An integer between 0 and 56. xlNone for no color
  '3 for red; 4 for green; 5 for blue
End With
```

Border line style, color, and weight

```
'To set the line style for the four borders
'(left, right, top, and bottom) of a range to dash
Range("A1:C3").Borders.LineStyle = xlDash

'To set the line style for the top border of
'a range of cells to double line
Range("A1:C3").Borders(3).LineStyle = xlDouble

'To clear the line style for the four borders
Range("A1:C3").Borders.LineStyle = xlLineStyleNone
```

To check other line styles, highlight LineStyle, press F1, and click XlLineStyle for its enumeration.

```
'To set the line style, color, and weight for a certain border
'(other than the four borders) of a range
With Range("A1:C3").Borders(xlInsideHorizontal)
  .LineStyle = xlDash
   .ColorIndex = 4
   .Weight = 3
End With
```

To check other possible borders (other than the left, right, top, and bottom borders) of a range, press F2 in VBE to display Object Browser. Enter XlBordersIndex in the search box to find the other eight borders.

Number format

```
'To display the numbers in a range of cells as fractions
Range("A1:C3").NumberFormat = "# ?/?"

'To display the full day name for a date in a cell
Range("A1").NumberFormat = "dddd"


'To clear the number format in a cell
Range("A1").NumberFormat = ""
```

To learn more about custom number formats, please refer the following extract from my earlier book entitled *Learn Microsoft® Excel® 2010-2016 for Windows® in 24 Hours*:

One way to learn custom number formats is to study the existing built-in number formats. Enter a number into any cell in a worksheet, press Ctrl+1 to display the Format Cells dialog box, select the Number tab, choose one of the format categories in the Category list box, and lastly select the Custom category to study the number format string in the Type field.

Hide contents

```
'To hide the contents in a range of selected cells
```

```
Selection.NumberFormat = ";;;"
```

To explore further about other methods and properties in formatting a range of worksheet cells, you can use the macro recorder. Record your Excel actions when you are formatting the range. Study the recorded VBA code.

Gridlines

Changing the appearance of the gridlines is only for worksheets, macro sheets, and dialog sheets since chart sheets have no gridlines.

Hide and unhide gridlines

```
'To hide and unhide gridlines in the active sheet
'in the active window

  'Hide the gridlines
  ActiveWindow.DisplayGridlines = False

  'Unhide the gridlines
  ActiveWindow.DisplayGridlines = True
```

A workbook can have more than one window by choosing View | Window | New Window. The window of the workbook that you are working on is the *active window*. Scrolling the active sheet in the active window does not cause scrolling in the other inactive windows. The active sheet can be a worksheet, a chart sheet, a macro sheet, or a dialog sheet.

Color

```
'To change the gridline color in the active sheet
'in the active window
ActiveWindow.GridlineColorIndex = 3
'3 for red; 4 for green; 5 for blue

'To restore gridlines to its default color
ActiveWindow.GridlineColorIndex = xlColorIndexAutomatic
```

Sheet tabs

Hide and unhide the sheet tabs

```
'To hide and unhide the sheet tabs (also known as the workbook tabs)
'in the active window
ActiveWindow.DisplayWorkbookTabs = False  'Hide
ActiveWindow.DisplayWorkbookTabs = True   'Unhide
```

```
'To hide and unhide a particular sheet (in all windows)
Sheets(3).Visible = xlSheetHidden   'Hide 3rd sheet
Sheets(3).Visible = xlSheetVisible  'Unhide 3rd sheet
```

You can reference a particular sheet by

- its tab name (the name that appears on the sheet tab);
- its index number (the position of its sheet tab in the workbook). Even if the sheet is hidden, its index number does not change. With no hidden sheets, the leftmost sheet tab is the first sheet in the workbook and with an index number of 1; or
- its code name (the name that appears in the Properties window, the one that is not in the parentheses). In VBE, press F4 to display the Properties window.

The following code shows the three different ways of referencing a sheet before hiding the sheet:

```
'To hide a sheet named Sheet3
Sheets("Sheet3").Visible = xlSheetHidden

'To hide the third sheet in the workbook
Sheets(3).Visible = xlSheetHidden

'To hide a sheet with the code name of Sheet3
Sheet3.Visible = xlSheetHidden
```

The advantage of using code name is that the written VBA code can still reference the right sheet in the workbook even if the user moves and renames the sheet or added other

sheets. However, one drawback is that the code name cannot reference a sheet that is in a different workbook to the one that the code resides.

Color a sheet tab

```
'To change the color of a sheet tab (in all windows) in a workbook
Sheets(1).Tab.Color = vbGreen
```

To check other color constants, type vbGreen in the Immediate window or in the code window and press F1.

```
'To restore the color of a sheet tab to no color
Sheets(1).Tab.Color = False
```

Status bar

In Excel 2013 and 2016, each window of a workbook has its own status bar. These versions of Excel are single-document-interface applications. Nevertheless, the status bar at the bottom of every Excel window (of every opened workbook) displays the same message. Any changes to the message on the status bar in one of the windows are updated on the status bars in all other windows.

In Excel 2007 and 2010, there is only one status bar in a single application-level window (the Excel window), holding all the windows of the workbooks that are open. These versions of Excel are multiple-document-interface applications.

Hide and unhide the status bar

```
'To hide the status bar (in all opened workbooks)
Application.DisplayStatusBar = False
```

```
'To unhide the status bar
Application.DisplayStatusBar = True
```

Write to and read from the status bar

```
'To write to the status bar
Application.StatusBar = "Working ..."

'To read from the status bar
Debug.Print Application.StatusBar
```

Reset the status bar

```
'To reset the status bar to its normal state
Application.StatusBar = ""
```

Workbook views

Depending on what you are working on, you may choose to display a *worksheet* in normal view, page break preview, or page layout view. However, none of these viewing options is applicable to *chart*, *macro*, and *dialog* sheets.

Normal view

```
'To display the active sheet in the active window in normal view
ActiveWindow.View = xlNormalView
```

Page break preview

```
'To display the active sheet in the active window in page break preview
ActiveWindow.View = xlPageBreakPreview
```

Page layout view

```
'To display the active sheet in the active window in page layout view
ActiveWindow.View = xlPageLayoutView
```

If the active sheet is not a worksheet, the execution of any one of the code statements above causes a runtime error because a chart sheet, a macro sheet, or a dialog sheet does not have these viewing options.

Window views

Maximize, minimize, restore, and display the active window in full-screen mode

```
'To maximize, minimize, and restore the active window
With ActiveWindow
   .WindowState = xlMaximized
   .WindowState = xlMinimized
   .WindowState = xlNormal
End With
```

```
'To display the active window (in Excel 2013 and 2016) or
'the Excel window (in Excel 2007 and 2010) in full-screen mode
Application.DisplayFullScreen = True
```

```
'To turn off the full-screen mode
Application.DisplayFullScreen = False
```

Position and size the active window

```
'To set the position and the size of the active window
With ActiveWindow
   .WindowState = xlNormal
   .Top = 0
   .Left = 0
   .Width = 400
   .Height = 300
End With
```

Center the active window

```
'To center the active window

  'Get the maximum width and height
  Dim mWidth As Integer, mHeight As Integer
  With ActiveWindow
    .WindowState = xlMaximized
    mWidth = .Width
    mHeight = .Height
  End With

  'Center the window
  With ActiveWindow
    .WindowState = xlNormal
    .Top = (mHeight - .Height) / 2
    .Left = (mWidth - .Width) / 2
  End With
```

Split the active window into panes

Splitting the active window into panes is applicable to worksheets, macro sheets, and dialog sheets, but not to chart sheets.

```
'To split the active sheet in the active window
'into upper and lower panes at row 5
With ActiveWindow
  .SplitRow = 5
  .SplitColumn = 0
End With
```

```
'To split the active sheet in the active window
'into left and right panes at column 4
With ActiveWindow
  .SplitRow = 0
  .SplitColumn = 4
End With
```

```
'To split the active sheet in the active window
'into 4 panes at row 5 and column 4
'i.e.: at cell D5 if the active sheet is a worksheet or a macro sheet
With ActiveWindow
   .SplitRow = 5
   .SplitColumn = 4
End With
```

```
'To clear the split panes
ActiveWindow.Split = False
```

Freeze the split panes of the active window

Freezing the split panes of the active window is applicable only to worksheets (in normal view and in page break preview), macro sheets, and dialog sheets, but not to chart sheets.

```
'To freeze the active sheet in the active window at row 2
'Rows 1 and 2 are then frozen.
With ActiveWindow
   .SplitRow = 2
   .SplitColumn = 0
   .FreezePanes = True
End With
'Rows 1 and 2 remain visible as you scroll throughout the sheet.
```

```
'To freeze the active sheet in the active window at column 1
'Column 1 is then frozen
With ActiveWindow
   .SplitRow = 0
   .SplitColumn = 1
   .FreezePanes = True
End With
```

```
'To freeze the active sheet in the active window
'at row 2 and column 1
With ActiveWindow
   .SplitRow = 2
   .SplitColumn = 1
   .FreezePanes = True
End With
```

When the split panes of the active window are frozen, the frozen column(s) and row(s) remain visible as you scroll throughout the sheet. The frozen column(s) and row(s) are the frozen area.

```
'To unfreeze the panes
With ActiveWindow
   .FreezePanes = False
   'If to remove the split panes too, the following line of code
   'is needed because the Split property of the ActiveWindow object
   'can possibly be True initially
   .Split = False
End With
```

Zoom slider

Zoom in and out

```
'To zoom in and out the display size of the active sheet
'in the active window

   'Zoom in to 120%
   ActiveWindow.Zoom = 120

   'Zoom out to 80%
   ActiveWindow.Zoom = 80
```

Scroll bars

Hide and unhide the scroll bars

```
'To hide and unhide the scroll bars (and sheet tabs) in all opened workbooks
Application.DisplayScrollBars = False 'Hide
Application.DisplayScrollBars = True  'Unhide
```

```
'To hide and unhide the horizontal scroll bar(s) in the active window
With ActiveWindow
   .DisplayHorizontalScrollBar = False 'Hide
   .DisplayHorizontalScrollBar = True  'Unhide
End With
```

```
'To hide and unhide the vertical scroll bar(s) in the active window
With ActiveWindow
   .DisplayVerticalScrollBar = False  'Hide
   .DisplayVerticalScrollBar = True   'Unhide
End With
```

Scroll a row and scroll a column

```
'To set a particular row to be the top row in the pane or window
   'Set row 5 to be the top row
   ActiveWindow.ScrollRow = 5
```

```
'To set a particular column to be the leftmost column in the pane or window
   'Set column 8 to be the leftmost column
   ActiveWindow.ScrollColumn = 8
```

If the active window is not split into panes, the effect of scrolling of a row or a column is obvious and clear.

If the active window is split into panes and not frozen, the ScrollRow and ScrollColumn properties refer to the first pane, which is the upper-left pane of the window (if it is split into 4 panes), or is the left or the upper pane of the window (if it is split into 2 panes).

If to specify which particular pane to scroll, you may use the following statement, for example, to set column 10 to be the leftmost column in the second pane:

```
ActiveWindow.Panes(2).ScrollColumn = 10
```

If the split panes are frozen, the ScrollRow and ScrollColumn properties exclude the frozen area and the properties affect only the unfrozen area.

Set the scroll area

```
'To set the scroll area for a worksheet or a macro sheet
ActiveSheet.ScrollArea = "C2:H22"  'Limit to the range C2:H22
```

However, a user can still access any cell beyond the scroll area by entering the address of the cell in the Name box.

```
'To lift the scrolling constraint
ActiveSheet.ScrollArea = ""
```

Rows in a worksheet and a macro sheet

The discussion in this topic is only applicable to worksheets and macro sheets, and not to chart sheets and dialog sheets.

Hide and unhide the row heading

The row and column headings cannot be hidden separately. Both headings are either hidden or displayed together.

```
'To hide and unhide the row and column headings of the active
'worksheet or macro sheet in the active window

  'Hide the headings
  ActiveWindow.DisplayHeadings = False

  'Unhide the headings
```

```
ActiveWindow.DisplayHeadings = True
```

Hide and unhide rows

You may use either one of the following statements to hide all the rows in the active sheet of a workbook:

```
'To hide all the rows in the active sheet of a workbook
ActiveSheet.Rows.Hidden = True
ActiveSheet.Rows.EntireRow.Hidden = True
ActiveSheet.cells.Rows.Hidden = True
```

To hide the rows in other sheets, simply replace ActiveSheet, for example, with Sheets(1) for the first sheet or with Worksheets(1) for the first worksheet. After hiding all the rows, the row heading is very narrow and it is almost hidden, but the column heading is still there in the sheet.

To unhide all the rows, you may use either one of the following statements:

```
'To unhide all the rows in the active sheet of a workbook
ActiveSheet.Rows.Hidden = False
ActiveSheet.Rows.EntireRow.Hidden = False
ActiveSheet.cells.Rows.Hidden = False
```

```
'To hide certain rows

  'Hide only row 2
  ActiveSheet.Rows(2).Hidden = True

  'Hide rows 3 and 4
  ActiveSheet.Rows("3:4").Hidden = True

  'Hide non-contiguous rows
  ActiveSheet.Range("6:6, 8:10, 12:13").EntireRow.Hidden = True
```

The comma in the statement above is a union operator to combine a list of rows.

Height

```
'To set the height of all the rows in the active sheet
ActiveSheet.Rows.RowHeight = 15

'To set the height of row 2
ActiveSheet.Rows(2).RowHeight = 20

'To set the height of rows 4 to 6
ActiveSheet.Rows("4:6").RowHeight = 10

'To set the height of non-contiguous rows
ActiveSheet.Range("8:8, 10:12").RowHeight = 10
```

The height of a hidden row is zero. Setting the height of a hidden row to a nonzero value will unhide the hidden row.

Columns in a worksheet and a macro sheet

Hide and unhide column heading

As mentioned earlier, the row and column headings cannot be hidden separately. Both headings are either hidden or displayed together.

```
'To hide and unhide the row and column headings of the active
'worksheet or macro sheet in the active window

  'Hide the headings
  ActiveWindow.DisplayHeadings = False  'Hide

  'Unhide the headings
  ActiveWindow.DisplayHeadings = True  'Unhide
```

Hide and unhide columns

You may use either one of the following statements to hide all the columns in the active

sheet of a workbook.

```
'To hide all the columns in the active sheet of a workbook
ActiveSheet.Columns.Hidden = True
ActiveSheet.Columns.EntireColumn.Hidden = True
ActiveSheet.cells.Columns.Hidden = True
```

To hide the columns in other sheets, simply replace ActiveSheet, for example, with Worksheets(1) for the first worksheet or with Worksheets(2) for the second worksheet. After hiding all the columns, the column heading is very narrow and it is almost hidden, but the row heading is still there in the sheet.

To unhide all the columns, you may use either one of the following statements:

```
'To unhide all the columns in the active sheet of a workbook
ActiveSheet.Columns.Hidden = False
ActiveSheet.Columns.EntireColumn.Hidden = False
ActiveSheet.cells.Columns.Hidden = False
```

```
'To hide certain columns

  'Hide only column B
  ActiveSheet.Columns(2).Hidden = True

  'Hide columns D and E
  ActiveSheet.Columns("D:E").Hidden = True

  'Hide non-contiguous columns
  ActiveSheet.Range("G:G, J:K, M:N").EntireColumn.Hidden = True
```

The comma in the statement above is a union operator to combine a list of columns.

Width

```
'To set the width of all the columns in the first worksheet
Worksheets(1).Columns.ColumnWidth = 8
```

```
'To set the width of column B
Worksheets(1).Columns(2).ColumnWidth = 3
```

```
'To set the width of columns D to F
Worksheets(1).Columns("D:F").ColumnWidth = 2
```

```
'To set the width of non-contiguous columns
Worksheets(1).Range("G:G, J:K").ColumnWidth = 4
```

The width of a hidden column is zero. Setting the width of a hidden column to a nonzero value will unhide the hidden column.

Formula bar

Hide and unhide the formula bar

```
'To hide and unhide the formula bar
Application.DisplayFormulaBar = False 'Hide
Application.DisplayFormulaBar = True  'Unhide
```

Like the status bar in Excel 2013 and 2016, each window of a workbook has its own formula bar. Nevertheless, hiding the formula bar in a window of a workbook hides the formula bars in every window of the workbook (and every window in other opened workbooks). When the formula bar is hidden, the Name box is hidden too.

In Excel 2007 and 2010, there is only one formula bar in a single application-level window (the Excel window), holding all the windows of the workbooks that are open.

Height

```
'To change the height of the formula bar in the active window
Application.FormulaBarHeight = 3  '1 is the default line of height
```

Names in the Name box

A name in the Name box is a defined name for a range of cells. It can be created by a user or automatically created by Excel (such as Print_Area and Table1).

Hide names

```
'To hide all the names in the active workbook
Dim nm As Name
For Each nm In Names
   If nm.Visible = True Then nm.Visible = False
Next nm
'However, the name of a table created automatically
'by Excel cannot be hidden
```

Even though the names are hidden, you can still go to the range of a particular name by entering the name in the Name box.

```
'To hide a particular name (namely, a range named MyRange1)
'in the active workbook
Names("MyRange1").Visible = False  'True to unhide
```

Ribbon

You can rename and hide built-in tabs and built-in groups, and change their order on the Ribbon. However, you cannot rename and hide built-in controls, change the icons (the Mso images) of the built-in controls, and change the order of the built-in controls in a built-in group on the Ribbon. Examples of built-in controls are Cut, Copy, and Format Painter in the Clipboard built-in group on the Home built-in tab.

To add built-in and custom controls to a group, you must add them to a custom group on a built-in tab or on a new, custom tab.

Hide and unhide the Ribbon

When the Ribbon is hidden, the Ribbon tabs and the controls on the tabs are all hidden.

```
'To hide and unhide the Ribbon
Application.ExecuteExcel4Macro "Show.ToolBar(""Ribbon"",False)"  'hide
Application.ExecuteExcel4Macro "Show.ToolBar(""Ribbon"",True)"  'unhide
```

Despite the Ribbon is hidden, its height is still the one before it is hidden. Hence, checking the height of the Ribbon cannot tell whether it is hidden. The following code is the proper way to check whether the Ribbon is hidden:

```
'To check whether the Ribbon is hidden
If Application.CommandBars("Ribbon").Visible Then
   Debug.Print "Visible"
Else
   Debug.Print "Hidden"
End If
```

Minimize the Ribbon

When the Ribbon is minimized, you can see only the names of the Ribbon tabs. Clicking the name of a tab will temporarily display the controls on the tab. To toggle the display of controls on the tabs without using VBA code is to double-click the name of the currently selected tab or to press Ctrl+F1.

There is no single VBA command to minimize the Ribbon. The following code statement only toggles the display of controls on the tabs:

```
'To toggle the display of controls on the Ribbon
Application.CommandBars.ExecuteMso "MinimizeRibbon"
```

Note: The MinimizeRibbon command does not exist in Excel 2007. The above code statement is only applicable to Excel 2010 and later.

If the Ribbon is initially minimized, an accidental execution of the above statement will restore the display of controls on the tabs. To ensure that you only execute the statement when the Ribbon is not initially minimized, you can first check the height of the Ribbon, which is a value greater than 100 when it is not minimized, or alternatively you may check the state of the Ribbon by using the GetPressedMso method. The following code shows the two possible ways to minimize the Ribbon:

```
'To minimize the Ribbon
'1st way
With Application
```

```
If .CommandBars("Ribbon").Height > 100 Then _
   .CommandBars.ExecuteMso "MinimizeRibbon"
End With

'2nd way
With Application.CommandBars
   If Not .GetPressedMso("MinimizeRibbon") Then _
      .ExecuteMso "MinimizeRibbon"
End With
```

If the Ribbon is hidden, the toggling code statement above will not unhide the Ribbon. Hence, the following code gets the Ribbon unhidden (if it is initially hidden) before getting it minimized:

```
'To minimize the Ribbon

'Unhide the Ribbon if it is hidden
With Application
If Not .CommandBars("Ribbon").Visible Then _
   .ExecuteExcel4Macro "Show.ToolBar(""Ribbon"",True)"
End With

'Minimize the Ribbon
With Application.CommandBars
   If Not .GetPressedMso("MinimizeRibbon") Then _
      .ExecuteMso "MinimizeRibbon"
End With
```

If the Ribbon is auto hidden, the execution of the VBA code statement to toggle the display of controls on the Ribbon is not allowed and causes a runtime error. Hence, the following code does the check on the height of the Ribbon, which is a value less than 25 when it is auto hidden, before minimizing the Ribbon:

```
'To minimize the Ribbon

With Application
   'Check whether the Ribbon is auto hidden
   If .CommandBars("Ribbon").Height < 25 Then
      MsgBox "Cannot minimize the Ribbon when it is auto hidden."
   Else
      'Unhide the Ribbon if it is hidden
      If Not .CommandBars("Ribbon").Visible Then _
```

```
        .ExecuteExcel4Macro "Show.ToolBar(""Ribbon"",True)"

    'Minimize the Ribbon
    With .CommandBars
      If Not .GetPressedMso("MinimizeRibbon") Then _
        .ExecuteMso "MinimizeRibbon"
    End With
  End If
End With
```

Execute commands not in and in the Ribbon

In the previous topic: *Minimize the Ribbon*, the MinimizeRibbon command is executed by the following method:

```
Application.CommandBars.ExecuteMso(idMso)
```

The method takes only one argument: idMso, which is the name of the command (also known as the identifier for the control). MinimizeRibbon is a command that is not in the Ribbon.

The following shows another example of executing a command not in the Ribbon:

```
'To launch the Calculator application
Application.CommandBars.ExecuteMso "Calculator"
```

The next is an example of executing a command in the Ribbon, which is graphically represented by a control on the Ribbon:

```
'To split the active window into panes and to remove the split panes,
'which can be accomplished either by choosing View | Window | Split
'or by executing the following ExecuteMso method
Application.CommandBars.ExecuteMso "WindowSplitToggle"
```

Excel has more than 10,000 idMsos, which are the names of commands in and not in the Ribbon. An idMso can be either the name of a command, the name of a built-in tab, or the name of other built-in element. A later topic discusses further on what they are and how to identify them. Later in this book, when I mention the name of a built-in control, it

also means the name of its underlying command.

Activate a Ribbon tab in two different ways

Here I discuss two different ways to activate a particular Ribbon tab. One is to use the SendKeys method that simulates keystrokes as if you were pressing the keys manually in the active window, and the other is to use XML and VBA code.

The SendKeys method

For example, the following VBA code statement simulates the keystrokes of ALT, A, and ALT:

```
Application.SendKeys "%A%"
```

Inserting the above statement into the Workbook_Open event handler (in the ThisWorkbook module of a workbook) activates the Data tab when the workbook is opened. The following is the event handler:

```
Private Sub Workbook_Open()
  Application.SendKeys "%A%"
End Sub
```

Note: In certain version of Excel, the SendKeys method in the Workbook_Open event handler is executed too early before the Excel window is properly loaded. To work around this problem, you may use the OnTime event to send the keystrokes after a second or two Workbook_Open is executed. The following are the edited Workbook_Open event handler and a procedure in a standard VBA module that sends the keystrokes:

```
Private Sub Workbook_Open()
'The SendKeystrokes procedure is only executed after a second
  Application.OnTime Now + TimeValue("0:0:1"), _
    "SendKeystrokes"
End Sub
```

```
Sub SendKeystrokes()
  Application.SendKeys "%A%"
End Sub
```

The table below shows the key codes for those keys on the keyboard that do not display any text on the screen when you press the keys.

Key	Code
CTRL	^ (caret)
ALT	% (percent sign)
SHIFT	+ (plus sign)
BACKSPACE	{BACKSPACE} or {BS}
BREAK	{BREAK}
CAPS LOCK	{CAPSLOCK}
CLEAR	{CLEAR}
DELETE or DEL	{DELETE} or {DEL}
DOWN ARROW	{DOWN}
END	{END}
ENTER (numeric keypad)	{ENTER}
ENTER	~ (tilde)
ESC	{ESCAPE} or {ESC}
HELP	{HELP}
HOME	{HOME}
INS	{INSERT}
LEFT ARROW	{LEFT}
NUM LOCK	{NUMLOCK}
PAGE DOWN	{PGDN}
PAGE UP	{PGUP}
RETURN	{RETURN}
RIGHT ARROW	{RIGHT}
SCROLL LOCK	{SCROLLLOCK}
TAB	{TAB}
UP ARROW	{UP}
F1 through F15	{F1} through {F15}

To avoid any unexpected result, always use the SendKeys method as a last resort and make sure the active window is the right window that you want to send the keystrokes.

XML and VBA code (in Excel 2010 and later)

Another way to activate a Ribbon tab is to use XML and VBA code. Execute the following steps:

1. Google the text "Updated version of the Custom UI Editor", download, and install the editor, if you do not have the editor installed.
2. Create a new workbook and save it as a macro-enabled workbook.
3. Close the workbook and open it in Custom UI Editor.
4. In Custom UI Editor, click Insert and choose Office 2007 Custom UI Part.

Choosing this option will make the workbook compatible with Excel 2007 and later.

5. Copy and paste the following XML code:

```
<customUI
  xmlns="http://schemas.microsoft.com/office/2006/01/customui"
  onLoad="Initialize">
</customUI>
```

onLoad is a callback attribute, which has the name of a VBA procedure assigned to it, in this case, a procedure named Initialize. This procedure is called when the workbook is opened in Excel,

You may take a few minutes to visit w3schools.com and learn some basic ideas about XML. You only need to know what XML syntax, elements, attributes, and namespace are in order to follow the discussions on XML code in this book.

6. Click the Validate button on the toolbar to check for errors.

Note: Some errors (such as misspelling the value of an attribute) are not detected by the editor during the validation process. By default, no error message is displayed when you open the file in Excel and the file will not work as intended. Hence, to be informed of any error in the XML code, in Excel 2010-2016, choose File | Options | Advanced (or in Excel 2007, choose the Microsoft Office button | Excel Options | Advanced) to display the Advanced tab in the Excel Options dialog box. Scroll down to the General section and tick the check box labelled Show add-in user interface errors.

7. Click the Generate Callbacks button on the toolbar to generate the Initialize callback procedure, which is with the following signature:

```
'Callback for customUI.onLoad
Sub Initialize(ribbon As IRibbonUI)
End Sub
```

Select and copy the callback (by pressing Ctrl+A and Ctrl+C) to the Windows clipboard. Later, you will paste it into a standard VBA module in the

workbook.

8. Save and close the file.

9. Open the file in Excel.

Click OK to the error message since the Initialize procedure is not ready to be executed yet.

10. Press Alt+11 to activate VBE.

11. Insert a standard VBA module and paste the callback that you copied in Step 7 (by pressing Ctrl+V). Modify the pasted callback and declare the myRibbon object variable, as shown below:

```
Public myRibbon As IRibbonUI

'Callback for customUI.onLoad
Sub Initialize(ribbon As IRibbonUI)
  Set myRibbon = ribbon
End Sub
```

When the workbook is opened in Excel, the Initialize procedure is executed and gets the object reference of the Ribbon assigned to the myRibbon object variable. With the reference to the Ribbon, you can then activate a Ribbon tab (and invalidate Ribbon tabs and controls on the Ribbon) via the myRibbon object. The idea of invalidation will be discussed later in the subtopic Add a label control to the Ribbon and in many other topics and subtopics.

12. Save, close, and reopen the workbook in Excel.

To activate a particular built-in Ribbon tab, such as the Data tab, use the following code:

```
'To activate the Data built-in tab of the Ribbon
myRibbon.ActivateTabMso "TabData"
```

Enter the above code statement in the Immediate window to verify how the Data tab can be activated.

If to activate the Data tab when the workbook is opened, insert the code statement into the Initialize procedure, as shown below:

```
Sub Initialize(ribbon As IRibbonUI)
  Set myRibbon = ribbon

  'To activate the Data built-in tab of the Ribbon
  myRibbon.ActivateTabMso "TabData"
End Sub
```

If to activate a custom Ribbon tab, such as a custom tab with an id of MyCustomTab, use

the following code:

```
'To activate a custom tab with the MyCustomTab id
myRibbon.ActivateTab "MyCustomTab"
```

Note: The ActivateTabMso and ActivateTab methods only exist in Excel 2010 and later, but not in Excel 2007.

Adding a custom tab to the Ribbon is in fact an easy task and will be discussed later in the subtopic Add various types of controls to the Ribbon. The TabData parameter in the ActivateTabMso method above is an idMso. The next topic discusses what idMsos are and how to identify them.

Identify the names (idMsos) of built-in controls, groups, tabs, tab sets, and context menus

An idMso is either the name of a particular built-in control (such as the Bold control in the Font group on the Home tab), the name of a particular built-in group of controls (such as the Font group on the Home tab), the name of a particular built-in Ribbon tab (such as the Home tab and the Insert tab), the name of a particular context menu (such as the Cell context menu that appears when right-clicking a worksheet cell), or the name of a particular tab set (such as the Chart Tools contextual tab set that appears when a chart object is selected). The idMsos are not only important in the previously discussed ExecuteMso, GetPressedMso, and ActivateTabMso methods, but also in customizing the Ribbon. Customizing the Ribbon by using XML code is the focus in almost all of the remaining topics in this book.

To identify the name (idMso) of a built-in control, execute the following steps:

1. In Excel 2010-2016, choose File | Options to display the Excel Options dialog box.

   In Excel 2007, choose the Microsoft Office button | Excel Options.
2. Select the Customize Ribbon tab.
3. Locate the control (that is associated with its underlying command) in one of the two list boxes.
4. Hover the mouse pointer over the control until a pop-up screen tip of that control appears.

   The idMso of the control is one in the parentheses.

To identify the names (idMsos) of built-in groups, tabs, tab sets, and context menus – and including all built-in controls, you may download from *Microsoft Download Center*

the files *Office 2016 Help Files: Office Fluent User Interface Control Identifiers* for Office 2016, *Office 2013 Help Files: Office Fluent User Interface Control Identifiers* for Office 2013, Office 2010 Help Files: Office Fluent User Interface Control Identifiers for Office 2010, and 2007 Office System Document: Lists of Control IDs for Office 2007 for Office 2007. The downloaded files are not only for Excel, but also for other applications in Microsoft Office. Only look for those files that are related to the Excel application.

Many idMsos in the later versions of Excel are new to the earlier versions. For examples, the PrintPreviewAndPrint and CopySplitButton idMsos are in Excel 2010-2016, but not in Excel 2007. If you customize the Ribbon using XML code with an idMso of a control that is not recognized by Excel, the control will not be shown on the Ribbon. Hence, if to develop programs for various versions of Excel, always use the earliest idMso file.

Identify the names (imageMsos) of predefined images for controls

Microsoft Office provides thousands of predefined images. Each of these images is identified by an imageMso value, which is the name of the predefined image. To view the predefined images, you may download from Microsoft Download Center the imageMso table and display their images on control buttons by using the following method:

```
Application.CommandBars.GetImageMso(idMso, Width, Height)
```

This method takes three arguments: idMso is the imageMso value; Width and Height (integers between 16 and 128) are to specify the size of the displayed image.

The following Sub procedure creates a control button and displays an Mso image on the button:

```
Sub CreateAButtonWithMsoImage()
'To create a control button with an Mso Image displayed on it

  Dim myCtrl As OLEObject, imageMso As String

  'Specify the imageMso value
  imageMso = "_0"

  'Create a control button
```

```
  Set myCtrl = ActiveSheet.OLEObjects.Add(ClassType:= _
               "Forms.CommandButton.1", Left:=10, _
               Top:=10, Width:=60, Height:=60)

  'Display the Mso Image on the button
  With myCtrl.Object
    .Picture = Application.CommandBars. _
               GetImageMso(imageMso, 32, 32)
    .Caption = imageMso
  End With
End Sub
```

In facilitating you to browse through thousands of these predefined images and select a suitable image for your control, please download the workbook named imageMso browser.

In the next two topics, you will see how to add built-in and custom controls to the Ribbon. If the controls are built-in controls, Mso images are automatically coupled with the controls. Nevertheless, you can still use different images for the built-in controls, if you like. If the controls are custom controls, you can specify suitable Mso images for the controls by stating the imageMso values. Without providing the name (imageMso) of a predefined image for the control, the control on the Ribbon simply appears without an image.

Add built-in controls to the Ribbon

This topic focuses on adding various types of built-in controls to the Ribbon, and the next topic focuses on custom controls.

Add a normal button and a toggle button

To add a normal button and a toggle button (say, the Spelling and Strikethrough built-in controls, respectively) to a Ribbon tab (say, the Home tab), execute the following steps:

1. Google the text "Updated version of the Custom UI Editor", download, and install the editor, if you do not have the editor.
2. Create a new workbook and save it as an Excel workbook or a macro-enabled workbook.
3. Close the workbook and open it in Custom UI Editor.
4. In Custom UI Editor, click Insert and choose Office 2007 Custom UI Part.

Choosing this option will make the workbook compatible with Excel 2007 and later.

If you want to load version-specific XML code, you may include both Office 2007 Custom UI Part and Office 2010 Custom UI Part choices. The namespace of the xmlns attribute for Office 2010-2016 is as below:

```
xmlns=http://schemas.microsoft.com/office/2009/07/customui
```

5. Choose Insert | Sample XML | Custom Tab and do some modification, or copy and paste the following XML code in order to add a group labelled Fav with two built-in controls after the Clipboard group on the Home tab to the Ribbon:

```
<customUI
 xmlns="http://schemas.microsoft.com/office/2006/01/customui">
 <ribbon>
  <tabs>
   <tab idMso="TabHome">
    <group id="Group1" label="Fav"
          insertAfterMso="GroupClipboard">
     <button idMso="Spelling"/>
     <toggleButton idMso="Strikethrough"/>
    </group>
   </tab>
  </tabs>
 </ribbon>
</customUI>
```

You may take a few minutes to visit w3schools.com and learn some basic ideas about XML. Knowing what XML syntax, elements, attributes, and namespace are is sufficient to follow the discussions on XML code in this book.

The tab element:

> The value of the idMso attribute is the name of a built-in tab. In this case, TabHome is the idMso of the Home tab. If to insert the buttons to other built-in tab, replace the TabHome idMso with the idMso of other tab. Use the downloaded file stated in the topic *Identify the names (idMsos) of built-in controls, groups, tabs, tab sets, and context menus* to identify the idMso of the built-in tab.

The group element:

Attribute in the element	**Its value …**
id	generally is a unique identifier for an

	element in the XML code.
label	specifies the displayed text of the group on the Ribbon.
insertAfterMso	specifies the location where the group should be placed on the Home tab.

The button element:

The value of the idMso attribute specifies the name of a built-in control, in this case, the Spelling control.

The toggle button element:

The Strikethrough control is a toggle button. Use the downloaded file stated in the topic *Identify the names (idMsos) of built-in controls, groups, tabs, tab sets, and context menus* to determine the correct type of a built-in control whether it is a button, a toggle button, a split button, a combo box, a menu, a gallery, a check box, a label, a general control, or other type.

You will see other elements and attributes when I discuss various built-in controls in the next subtopic and custom controls in the next topic. If you are curious about the available elements and attributes, you may see the technical article Customizing the Office (2007) Ribbon User Interface for Developers (Part 2 of 3).

Note: XML code is case-sensitive. For example, idMso is not the same as IdMso.

6. Click the Validate button on the toolbar to check for errors.
7. Save and close the file.
8. Open the file in Excel.

The following figure shows how the two built-in controls appear in the Fav group on the Home tab of the Ribbon:

It is important to note that the Ribbon customizations are document-specific. Only the workbook that contains the XML code will display the customized Ribbon. Whenever

the workbook is closed, the customizations on the Ribbon are automatically removed.

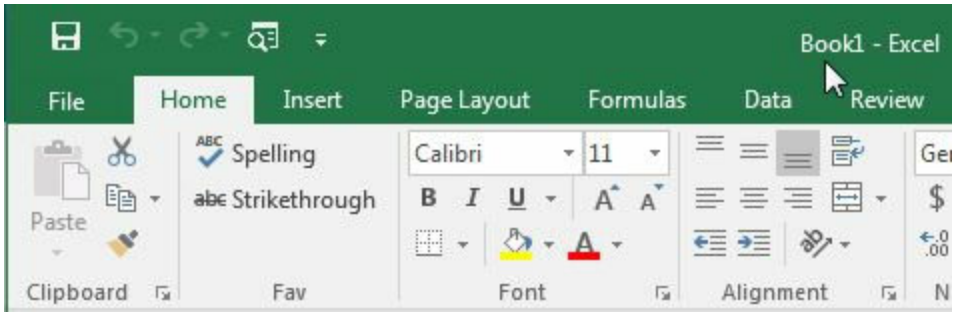
To display the customized Ribbon in other workbooks, save the workbook as an add-in file (Excel Add-in (*.xlam)) and execute the following steps to get the add-in file loaded whenever Excel starts:

1. In Excel 2010-2016, choose File | Options | Add-Ins to display the Add-Ins tab in the Excel Options dialog box.

   In Excel 2007, choose the Microsoft Office button | Excel Options | Add-Ins.
2. Select Excel Add-ins from the Manage drop-down control and click Go.
3. If your add-in is not in the list of available add-ins, click the Browse button to locate the file in the folder that you have saved it.
4. Place a tick next to the check box of your add-in in the list of available add-ins.
5. Click OK to install the add-in.

   If to uninstall the add-in, simply repeat the steps above with the check box of your add-in in the list unticked.

The following figure shows the appearance of the two built-in controls in a new workbook:



Add various types of controls

In this subtopic, you will see how, for example, to add 8 buttons with 3 of them (without labels) arranged in a horizontal position, 2 toggle buttons, 1 split button, 4 dialog box launchers, 2 combo boxes, 2 menus, 2 galleries, 1 label control, 1 edit box, 2 check boxes, 1 general control, and 1 group of built-in controls to a custom tab.

Repeat the steps above in the subtopic Add a normal button and a toggle button with the XML code in Step 5 replaced with the following:

```
<customUI xmlns="http://schemas.microsoft.com/office/2006/01/customui">
 <ribbon>
  <tabs>
```

```
<tab id="MyTab" label="Built-in Controls"
     insertBeforeMso="TabData">
  <group id="Group1" label="button | toggleButton | splitButton">
    <!-- 8 buttons,  2 toggle buttons, & -->
    <!-- 1 split button in  Group1 -->
    <button idMso="Copy"/>
    <button idMso="Paste"/>
    <button idMso="Calculator" label="Calculator"/>
    <buttonGroup id="ButtonGrp1">
      <!-- With the buttonGroup element, buttons are -->
      <!-- grouped and arranged horizontally -->
      <button idMso="Copy" showLabel="false"/>  <!-- no label -->
      <button idMso="Paste" showLabel="false"/>
      <button idMso="Calculator" showLabel="false"/>
    </buttonGroup>
    <button idMso="TableInsertExcel"/>
    <button idMso="TableConvertToRange"/>
    <separator id="separator1"/>  <!-- a vertical separator -->
    <!-- labels for the controls below are hidden, by default -->
    <toggleButton idMso="Bold"/>
    <toggleButton idMso="Italic"/>
    <separator id="separator2"/>
    <splitButton idMso="PasteMenu" size="large"/>
  </group>

  <group id="Group2" label="Dialog box launcher">
    <!-- 4 dialog box launchers in Group2  -->
    <!-- The 4th one is placed next to the group's label  -->
    <button idMso="ShowClipboard"/>
    <button idMso="FormatCellsFontDialog"/>
    <button idMso="PageSetupPageDialog"/>

    <dialogBoxLauncher>
      <button idMso="FormatCellsNumberDialog"/>
    </dialogBoxLauncher>
  </group>

  <group id="Group3" label="comboBox | menu | gallery">
    <!-- 2 combo boxes, 2 menus, & 2 galleries in Group3 -->
    <box id="box1" boxStyle="horizontal">
      <!-- With the box element, controls are then -->
      <!-- arranged horizontally in a box-->
      <comboBox idMso="Font"/>
      <comboBox idMso="FontSize"/>
    </box>
    <box id="box2" boxStyle="horizontal">
      <menu idMso="OrientationMenu"/>
      <menu idMso="HideAndUnhideMenu"/>
```

```
      </box>
      <box id="box3" boxStyle="horizontal">
        <gallery idMso="Undo"/>
        <gallery idMso="CellFillColorPicker"/>
      </box>
    </group>


    <group id="Group4"
           label="labelControl, editBox | checkBox | control">
      <!-- 1 label control, 1 edit box, 2 check boxes, -->
      <!-- & 1 general control in Group4 -->
      <box id="box4" boxStyle="horizontal">
        <!-- a built-in label -->
        <labelControl idMso="TableNameLabel"/>
        <editBox idMso="TableName" showLabel="true"/>
      </box>
      <box id="box5" boxStyle="horizontal">
        <checkBox idMso="GridlinesExcel"/>
        <checkBox idMso="ViewHeadings"/>
      </box>
        <control idMso="FormatPainter" showLabel="false"/>
    </group>

    <group idMso="GroupNumber"/>  <!-- a built-in group -->
  </tab>
 </tabs>
</ribbon>
</customUI>
```

The buttonGroup element is used to arrange a group of buttons horizontally. If to arrange horizontally a group of combo boxes, menus, galleries, check boxes, labels, or general controls, the box element is used instead.

The following figure shows how the various types of controls appear in groups on the Built-in Controls tab of the Ribbon:

Add general controls

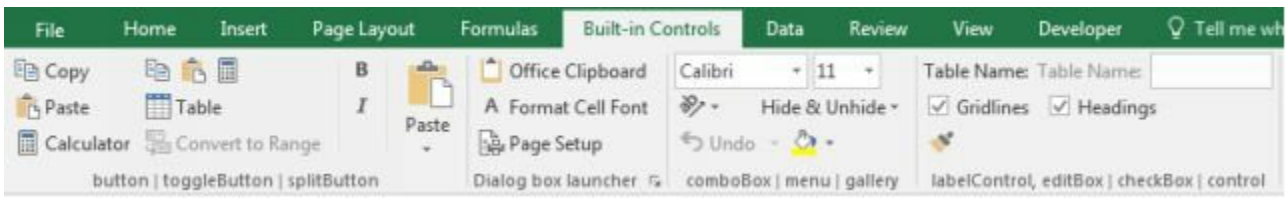
When adding a built-in control to the Ribbon, instead of specifying its type, you can also

use the control element. It can be used for all built-in controls regardless of their actual types. The following XML code illustrates the idea:

```
<customUI xmlns="http://schemas.microsoft.com/office/2006/01/customui">
 <ribbon>
  <tabs>
   <tab id="MyTab" label="Built-in Controls" insertBeforeMso="TabData">
    <group id="Group1" label="button | toggleButton | splitButton">
     <!-- 8 buttons,  2 toggle buttons, & -->
     <!-- 1 split button in Group1 -->
     <control idMso="Copy"/>
     <control idMso="Paste"/>
     <control idMso="Calculator" label="Calculator"/>
     <buttonGroup id="ButtonGrp1">
      <!-- With the buttonGroup element, buttons are -->
      <!-- grouped and arranged horizontally -->
      <control idMso="Copy" showLabel="false"/>  <!-- hide label -->
      <control idMso="Paste" showLabel="false"/>
      <control idMso="Calculator" showLabel="false"/>
     </buttonGroup>
     <control idMso="TableInsertExcel"/>
     <control idMso="TableConvertToRange"/>
     <separator id="separator1"/>  <!-- a vertical separator -->
     <!-- labels for the controls below are hidden, by default -->
     <control idMso="Bold"/>
     <control idMso="Italic"/>
     <separator id="separator2"/>
     <control idMso="PasteMenu" size="large"/>
    </group>


    <group id="Group2" label="Dialog box launcher">
     <!-- 4 dialog box launchers in Group2  -->
     <!-- The 4th one is placed next to the group's label  -->
     <control idMso="ShowClipboard"/>
     <control idMso="FormatCellsFontDialog"/>
     <control idMso="PageSetupPageDialog"/>

     <dialogBoxLauncher>
      <!-- Within the dialogBoxLauncher element, cannot -->
      <!-- simply replace "button" with "control" -->
      <button idMso="FormatCellsNumberDialog"/>
     </dialogBoxLauncher>
    </group>


    <group id="Group3" label="comboBox | menu | gallery">
     <!-- 2 combo boxes, 2 menus, & 2 galleries in Group3 -->
     <box id="box1" boxStyle="horizontal">
```

```
        <!-- With the box element, controls are then -->
        <!-- arranged horizontally in a box-->
        <control idMso="Font"/>
        <control idMso="FontSize"/>
      </box>
      <box id="box2" boxStyle="horizontal">
        <control idMso="OrientationMenu"/>
        <control idMso="HideAndUnhideMenu"/>
      </box>
      <box id="box3" boxStyle="horizontal">
        <control idMso="Undo"/>
        <control idMso="CellFillColorPicker"/>
      </box>
    </group>


    <group id="Group4"
           label="labelControl, editBox | checkBox | control">
      <!-- 1 label control, 1 edit box, 2 check boxes, -->
      <!-- & 1 general control in Group4 -->
      <box id="box4" boxStyle="horizontal">
        <!-- a built-in label -->
        <control idMso="TableNameLabel"/>
        <control idMso="TableName" showLabel="true"/>
      </box>
      <box id="box5" boxStyle="horizontal">
        <control idMso="GridlinesExcel"/>
        <control idMso="ViewHeadings"/>
      </box>
        <control idMso="FormatPainter" showLabel="false"/>
    </group>

  </tab>
 </tabs>
</ribbon>
</customUI>
```

The following figure shows the appearance of the various types of built-in controls that use the control elements in the XML code:



Add custom controls to the Ribbon

The topic now focuses on adding various types of custom controls to the Ribbon, which are associated with the underlying custom commands. The custom commands here mean the VBA Sub procedures (also known as macros) that are written by you as a programmer.

Add buttons

For example, to add two buttons (that will call VBA procedures when they are clicked) to a built-in Ribbon tab (say, the Insert tab), execute the following steps:

1. Create a new workbook and save it as a macro-enabled workbook.
2. Close the workbook and open it in Custom UI Editor.
3. In Custom UI Editor, click Insert and choose Office 2007 Custom UI Part.
4. Choose Insert | Sample XML | Custom Tab and do some modification, or copy and paste the following XML code in order to add a group labelled Attn Sh with two buttons (labelled Insert 0 and Insert 1 and with the Mso images of 0 and 1, respectively) to the Insert tab:

```
<customUI
 xmlns="http://schemas.microsoft.com/office/2006/01/customui">
 <ribbon>
  <tabs>
   <tab idMso="TabInsert">
    <group id="Group1" label="Attn Sh">
     <button id="BtnInsert0"
             label = "Insert 0"
             imageMso = "_0"
             onAction = "Insert0"/>
     <button id="BtnInsert1"
             label = "Insert 1"
             imageMso="_1"
             onAction = "Insert1"/>
    </group>
   </tab>
  </tabs>
 </ribbon>
</customUI>
```

The tab element:

The value of the idMso attribute is the name of a built-in tab. In this case, TabInsert is the idMso of the Insert tab. If to insert the buttons to other built-in tab, other than the Insert tab, replace the TabInsert idMso with the

idMso of other tab.

The group element:

The value of the label attribute in the group element specifies the displayed text of the group on the Ribbon.

The button element:

The imageMso attribute specifies the predefined image for the button. If to use your own designed image, simply replace the imageMso attribute with the image attribute. See Add a gallery on how to insert an image (also known as icon).

onAction is a callback attribute. The value of the attribute is the name of a VBA procedure that will execute when the button is clicked. In the later topics, you will see other callback attributes, such as getLabel, getVisible, and getEnabled.

5. Click the Validate button on the toolbar to check for errors.
6. Click the Generate Callbacks button on the toolbar.

Since there are two callback attributes (one for each of the two buttons) in the XML code in Step 4, two callbacks are generated with the following signatures:

```
'Callback for BtnInsert0 onAction
Sub Insert0(control As IRibbonControl)
End Sub

'Callback for BtnInsert1 onAction
Sub Insert1(control As IRibbonControl)
End Sub
```

Copy the callbacks. Later, you will paste them into a standard VBA module in the workbook.

7. Save and close the file.
8. Open the file in Excel.
9. Press Alt+11 to activate VBE.
10. Insert a standard VBA module and paste the callbacks you copied in Step 6.
11. Add an MsgBox statement to each of the two procedures to test the buttons.

See below for an example:

```
'Callback for BtnInsert0 onAction
```

```
Sub Insert0(control As IRibbonControl)
 With control
    MsgBox .ID & " in " & .Context.Caption & _
      " was clicked."
 End With
End Sub
```

The MsgBox statement displays “BtnInsert0 in buttons.xlsm was clicked.”, if the user clicks the Insert 0 button on the Ribbon and if the filename of the Excel workbook is buttons.xlsm.

The argument (named control) in the Insert0 callback procedure has three properties:

- ID: The id of the control specified in the XML code.
- Context: The active window containing the Ribbon. Context.Caption is the name that appears on the title bar of the window. In this case, the name of the workbook.
- Tag: The value of the tag attribute of an element (the button element, in this case) in the XML code. It is to store arbitrary strings. Often tags are used to identify controls with common tags before performing some common action on them. For an example, see the subtopic Add a split button.

The following figure shows how the two button controls appear in the Attn Sh group on the Insert tab:

Add toggle buttons

Repeat the steps in the subtopic Add buttons with the sample XML code in Step 4 replaced with the following:

```
<customUI xmlns="http://schemas.microsoft.com/office/2006/01/customui">
 <ribbon>
  <tabs>
   <tab id="MyTab" label="Custom" insertBeforeMso="TabHome">
    <group id="Group1" label="toggleButton">
     <toggleButton id="toggleBtn1"
```

```
                label ="Toggle button 1"
                imageMso="Breakpoint"
                onAction="Macro1"/>
        <toggleButton id="toggleBtn2"
                imageMso="Superscript"
                onAction="Macro2"
                supertip="no label"/>
      </group>
    </tab>
  </tabs>
 </ribbon>
</customUI>
```

If to verify whether the generated callbacks are working when the buttons are clicked, you may add MsgBox statements to the callback procedures as follow:

```
'Callback for toggleBtn1 onAction
Sub Macro1(control As IRibbonControl, pressed As Boolean)
  MsgBox "The status of " & control.ID & " is pressed: " & pressed
End Sub
```

The MsgBox statement displays “The status of toggleBtn1 is pressed: True”, when the user clicks the toggle button (labelled Toggle button 1) that was initially unpressed.

```
'Callback for toggleBtn2 onAction
Sub Macro2(control As IRibbonControl, pressed As Boolean)
  MsgBox "The status of " & control.ID & " is pressed: " & pressed
End Sub
```

The MsgBox statement displays “The status of toggleBtn2 is pressed: False”, when the user clicks the toggle button (with no label) that was initially pressed.

The figure below shows how the two toggle button controls appear on the Custom tab.

Add a split button

A split button control is a composite control with a one-click button and a drop-down list of buttons. A user can select either the one-click button or one of buttons from the drop-down list.

Repeat the steps in the subtopic Add buttons with the sample XML code in Step 4 replaced with the following:

```
<customUI xmlns="http://schemas.microsoft.com/office/2006/01/customui">
 <ribbon>
  <tabs>
   <tab id="MyTab" label="Custom" insertBeforeMso="TabHome">
    <group id="Group1" label="splitButton">
     <splitButton id="splitBtn1" size="large">

      <button id="Btn1"
              label="Button1"
              imageMso="Copy"
              onAction="Macro1"
              supertip="This is a split button control."
              tag="Button1"/>
              <!-- See Macro1 in Excel on how tag is used -->

       <menu id="splitMenu">
        <button id="menuButton1"
                label="Button1"
                imageMso="Copy"
                onAction="Macro1"
                tag="Button1"/>
                <!-- the same Macro1 as for one-click Btn1 -->
                <!-- See Macro1 in Excel on how tag is used -->
        <button id="menuButton2"
                label="Button2"
                onAction="Macro2"/>
                <!-- no imageMso means no image for the button -->
        <button id="menuButton3"
                label="Button3"
                imageMso="FormatPainter"
                onAction="Macro3"
                supertip="The 3rd button in the menu."/>
       </menu>
      </splitButton>
    </group>
   </tab>
  </tabs>
 </ribbon>
</customUI>
```

If to verify whether the generated callbacks are working, you may add MsgBox statements to the callback procedures as follow:

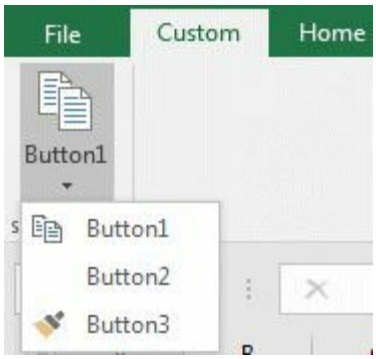
```
'Callback for Btn1 and menuButton1 onAction
Sub Macro1(control As IRibbonControl)
  MsgBox control.Tag & " was clicked."
End Sub
```

The MsgBox statement displays “Button1 was clicked.”, if the user clicks either the one-click button or the menuButton1 button in the drop-down list of buttons. When either button is clicked, the same Macro1 procedure is executed and the same message is displayed because the tag attributes of both buttons are with the same value (which is, Button1).

```
'Callback for menuButton2 onAction
Sub Macro2(control As IRibbonControl)
  MsgBox "Marco2 is executed."
End Sub
```

```
'Callback for menuButton3 onAction
Sub Macro3(control As IRibbonControl)
  MsgBox "Marco3 is executed."
End Sub
```

The figure below shows how the split button control appears on the Custom tab.



Add a combo box

The sample XML code:

```
<customUI xmlns="http://schemas.microsoft.com/office/2006/01/customui">
 <ribbon>
  <tabs>
   <tab id="MyTab" label="Custom" insertBeforeMso="TabHome">
```

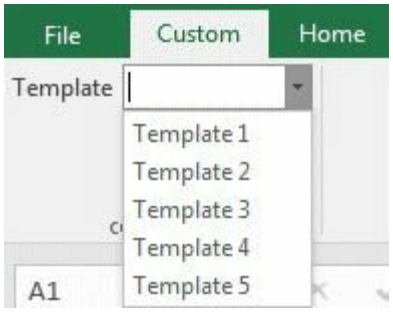
```
      <group id="Group1" label="comboBox">
        <comboBox id="comboBox1"
                  label="Template"
                  onChange="Combo1_onChange">
          <item id="Template1" label="Template 1"/>
          <item id="Template2" label="Template 2"/>
          <item id="Template3" label="Template 3"/>
          <item id="Template4" label="Template 4"/>
          <item id="Template5" label="Template 5"/>
        </comboBox>
      </group>
    </tab>
  </tabs>
 </ribbon>
</customUI>
```

The edited callback procedure in a standard VBA module:

```
'Callback for comboBox1 onChange
Sub Combo1_onChange(control As IRibbonControl, text As String)
  MsgBox "The displayed text in the combo box: " & text
End Sub
```

The MsgBox statement displays “The displayed text in the combo box: Template 1”, if the user selects the item labelled Template 1 from the combo box.

The appearance of the combo box control on the Custom tab:



Add a drop-down control

Unlike a combo box control, which can also accept user-entered text, a drop-down control only allows the user to make a selection from the listed items in the control.

The sample XML code:

```
<customUI xmlns="http://schemas.microsoft.com/office/2006/01/customui">
 <ribbon>
```

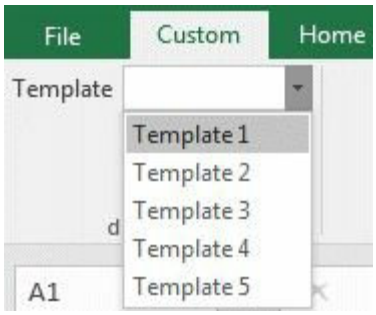
```
  <tabs>
    <tab id="MyTab" label="Custom" insertBeforeMso="TabHome">
      <group id="Group1" label="dropDown">
        <dropDown id="dropDown1"
                  label="Template"
                  onAction="SelectedItem">
          <item id="Template1" label="Template 1"/>
          <item id="Template2" label="Template 2"/>
          <item id="Template3" label="Template 3"/>
          <item id="Template4" label="Template 4"/>
          <item id="Template5" label="Template 5"/>
        </dropDown>
      </group>
    </tab>
  </tabs>
 </ribbon>
</customUI>
```

The edited callback procedure in a standard VBA module:

```
'Callback for dropDown1 onAction
Sub SelectedItem(control As IRibbonControl, _
    id As String, index As Integer)
 MsgBox "You selected Template " & index + 1
End Sub
```

The MsgBox statement displays “You selected Template 1”, if the user selects the item labelled Template 1 from the drop-down control.

The appearance of the drop-down control on the Custom tab:



Add a gallery control

You can insert images into a custom gallery. The extensions of the image files can be bmp, png, jpg, and tif.

To add a gallery control to a custom Ribbon tab, execute the following steps:

1. Create a new workbook and save it as a macro-enabled workbook.
2. Close the workbook and open it in Custom UI Editor.
3. In Custom UI Editor, choose Insert | Office 2007 Custom UI Part.

If you choose Office 2010 Custom UI Part, simply replace the namespace of the xmlns attribute with the following:

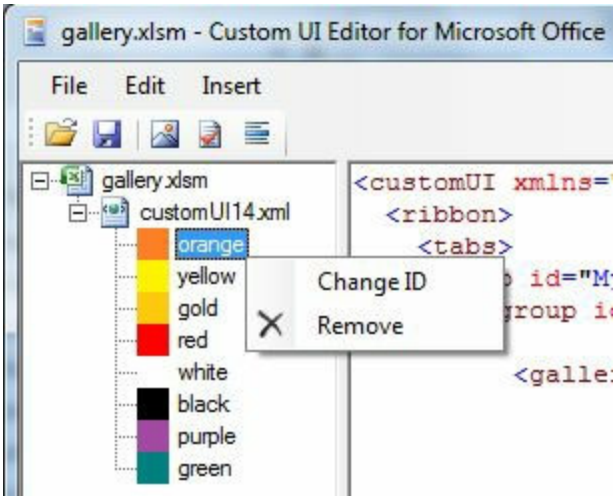
```
xmlns=http://schemas.microsoft.com/office/2009/07/customui
```

4. Choose Insert | Icon to display the Insert Custom Icons dialog box.
5. Locate and select your image file(s) and click Open.

A copy of the image file(s) is inserted. Deleting the original files has no effect on the inserted files.

Note: Avoid using space characters in the file name of your image file. A space character is interpreted as %20 in the editor and the file cannot then be referred correctly.

6. Right-click one of the inserted images in Custom UI Editor to change its IDs, if you like. See figure below for an example.



7. Copy and paste the following XML code:

```
<customUI xmlns="http://schemas.microsoft.com/office/2006/01/customui">
 <ribbon>
  <tabs>
   <tab id="MyTab" label="Custom" insertBeforeMso="TabHome">
    <group id="Group1" label="gallery">
     <gallery id="gallery1"
              label="Color gallery"
              imageMso="CellFillColorPicker"
              columns="4" rows="2"
              itemWidth="20"
              itemHeight="15"
```

```
                onAction="SelectedColor">
            <item id="black" image="black"/>
            <item id="white" image="white"/>
            <item id="red" image="red"/>
            <item id="green" image="green"/>
            <item id="orange" image="orange"/>
            <item id="yellow" image="yellow"/>
            <item id="purple" image="purple"/>
            <item id="gold" image="gold"/>
          </gallery>
        </group>
      </tab>
    </tabs>
  </ribbon>
</customUI>
```

The item element:

The value of the image attribute is the ID of the inserted image. See figure above.

The value of the id attribute is not necessarily the same as the ID of the inserted image. It can be any arbitrary unique text string.

8. Click the Validate button on the toolbar to check for errors.

9. Click the Generate Callbacks button on the toolbar.

One callback procedure for the onAction callback attribute is generated. The following is the signature of the callback:

```
'Callback for gallery1 onAction
Sub SelectedColor(control As IRibbonControl, id As String, _
    index As Integer)
End Sub
```

Copy the callback. Later, you will paste it into a standard VBA module in the workbook.

10. Save and close the file.

11. Open the file in Excel.

12. Press Alt+11 to activate VBE.

13. Insert a standard VBA module and paste the callback you copied in Step 9.

14. Add an MsgBox statement to the procedure to test the control.

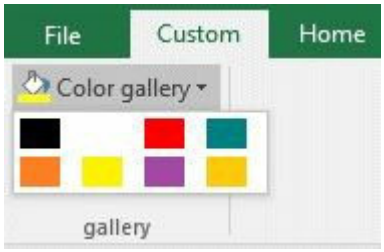
See below for example:

```
'Callback for gallery1 onAction
Sub SelectedColor(control As IRibbonControl, id As String, _
```

```
  index As Integer)
 MsgBox "You selected " & id
End Sub
```

The MsgBox statement displays "You selected red", if the user clicks the red item in the gallery.

The following figure shows how the items appear in the gallery control on the Custom tab:



Instead of inserting images manually as in Steps 4 and 5 above, alternatively, you can populate the gallery with the same images by using a VBA procedure. This can be done by assigning the name of the VBA procedure to the loadImage callback attribute. The following are the XML code and the VBA code:

The sample XML code:

```
<customUI xmlns="http://schemas.microsoft.com/office/2006/01/customui"
 loadImage="LoadImage">
 <ribbon>
  <tabs>
   <tab id="MyTab" label="Custom" insertBeforeMso="TabHome">
    <group id="Group1" label="gallery">
     <gallery id="gallery1"
              label="Color gallery"
              imageMso="CellFillColorPicker"
              columns="4" rows="2"
              itemWidth="20"
              itemHeight="15"
              onAction="SelectedColor">
      <item id="black" image="black.jpg"/>
      <item id="white" image="white.jpg"/>
      <item id="red" image="red.jpg"/>
      <item id="green" image="green.jpg"/>
      <item id="orange" image="orange.jpg"/>
      <item id="yellow" image="yellow.jpg"/>
      <item id="purple" image="purple.jpg"/>
      <item id="gold" image="gold.jpg"/>
     </gallery>
```

```
      </group>
    </tab>
  </tabs>
 </ribbon>
</customUI>
```

The LoadImage VBA procedure, which is referenced by the loadImage attribute, is executed only once when the workbook is opened in Excel and the gallery control is clicked for the first time.

The VBA code in a standard VBA module:

```
'Callback for customUI.loadImage
Sub LoadImage(ImageName As String, ByRef Image)
  Set Image = LoadPicture("C:\Icons\" & ImageName)
  'MsgBox ImageName
End Sub
```

The procedure loops through every item element in the XML code for the filename of the image and gets the image loaded to the gallery control. Include an MsgBox statement, as shown above, to see how the procedure loops through every item.

```
'Callback for gallery1 onAction
Sub SelectedColor(control As IRibbonControl, id As String, _
   index As Integer)
  MsgBox "You selected " & id
End Sub
```

Add a label control

Repeat the steps in the subtopic Add buttons with the sample XML code in Step 4 replaced with the following:

```
<customUI xmlns="http://schemas.microsoft.com/office/2006/01/customui">
 <ribbon>
  <tabs>
   <tab id="MyTab" label="Custom" insertBeforeMso="TabHome">
    <group id="Group1" label="label">
     <labelControl id="label1" getLabel="getLabel1"/>
    </group>
   </tab>
  </tabs>
 </ribbon>
</customUI>
```

getLabel is a callback attribute. It is assigned the name of a VBA procedure, in this

case, the getLabel1 callback procedure. The procedure is executed when the workbook is first opened or when the control is invalidated.

After saving the file in Custom UI Editor and opening it in Excel for the first time, you will be prompted with the error message about getLabel1, as expected, because the getLabel1 callback procedure has yet to be found in a standard VBA module. Click OK to the error message.

There are three methods to invalidate controls as follow:

- InvalidateControlMso is to invalidate a particular built-in control. (not for Excel 2007)
- InvalidateControl is to invalidate a particular custom control.
- Invalidate is to invalidate all of the built-in and custom controls on the Ribbon.

Not only the above three methods can invalidate controls, but also groups and tabs. Let me call them (controls, groups, and tabs) elements. An element can possibly have more than one callback attribute. See the sample program at the end of the book. Once an element is invalidated, any data associated with that element is destroyed, and it is automatically refreshed by calling all the VBA procedures that are referenced by the callback attributes of the element stated in the XML code. Hence, to improve efficiency, only invalidate those elements that are necessary. I discuss further the examples of invalidating elements later in the topics Hide and unhide controls, groups, and tabs and Disable and enable controls, groups, and tabs.

The edited callback procedure in a standard VBA module:

```
'Callback for label1 getLabel
Sub getLabel1(control As IRibbonControl, ByRef returnedVal)
  If Time() < 0.5 Then
    returnedVal = "Good morning, " & Application.UserName
  Else
    returnedVal = "Good day, " & Application.UserName
  End If
End Sub
```

The appearance of the label control on the Custom tab:

Add a check box

The sample XML code:

```
<customUI xmlns="http://schemas.microsoft.com/office/2006/01/customui">
 <ribbon>
  <tabs>
   <tab id="MyTab" label="Custom" insertBeforeMso="TabHome">
    <group id="Group1" label="check box">
     <checkBox id="checkBox1"
               label="Check box 1"
               onAction="Checkbox1_Change"/>
    </group>
   </tab>
  </tabs>
 </ribbon>
</customUI>
```

The edited callback procedure in a standard VBA module:

```
'Callback for checkBox1 onAction
Sub Checkbox1_Change(control As IRibbonControl, pressed As Boolean)
  MsgBox "Check box is checked: " & pressed
End Sub
```

The MsgBox statement simply displays “Check box is checked: True”, if the user ticks the check box.

The appearance of the check box on the Custom tab:

Add an edit box

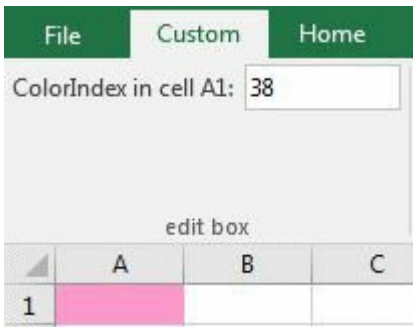
The sample XML code:

```
<customUI xmlns="http://schemas.microsoft.com/office/2006/01/customui">
 <ribbon>
  <tabs>
   <tab id="MyTab" label="Custom" insertBeforeMso="TabHome">
    <group id="Group1" label="edit box">
     <editBox id="EditBox1"
              label="ColorIndex in cell A1:"
              onChange="EditBox1_onChange"/>
    </group>
   </tab>
  </tabs>
 </ribbon>
</customUI>
```

The edited callback procedure in a standard VBA module:

```
'Callback for EditBox1 onChange
Sub EditBox1_onChange(control As IRibbonControl, text As String)
  On Error Resume Next
  Range("A1").Interior.ColorIndex = text
  If Err.Number <> 0 Then _
    MsgBox "Enter an integer between 0 and 56."
End Sub
```

The appearance of the edit box control on the Custom tab:



Add a menu control

The sample XML code:

```
<customUI xmlns="http://schemas.microsoft.com/office/2006/01/customui">
 <ribbon>
```

```
  <tabs>
    <tab id="MyTab" label="Custom" insertBeforeMso="TabHome">
      <group id="Group1" label="menu">
        <menu id="Menu1"
              label = "Menu"
              imageMso = "CreateShortcutMenuFromMacro"
              itemSize="normal"> <!-- use itemSize, not size -->
          <button id="button1"
                  label = "Button 1"
                  imageMso = "_1"
                  onAction="Macro1"/>
          <button id="button2"
                  label = "Button 2"
                  imageMso = "_2"
                  onAction="Macro2"/>
          <menuSeparator id="menuSep1"/>
          <button id="button3"
                  label = "Button 3"
                  imageMso = "_3"
                  onAction="Macro3"/>
          <menu id="subMenu1"
                label = "Button 4"
                imageMso="_4">
            <button id="button4a"
                    label = "Button 4A"
                    imageMso = "A"
                    onAction="Macro4A"/>
            <button id="button4b"
                    label = "Button 4B"
                    imageMso = "B"
                    onAction="Macro4B"/>
          </menu>
          <button id="button5"
                  label = "Button 5"
                  imageMso = "_5"
                  onAction="Macro5"/>
        </menu>
      </group>
    </tab>
  </tabs>
 </ribbon>
</customUI>
```

The edited callback procedures in a standard VBA module:

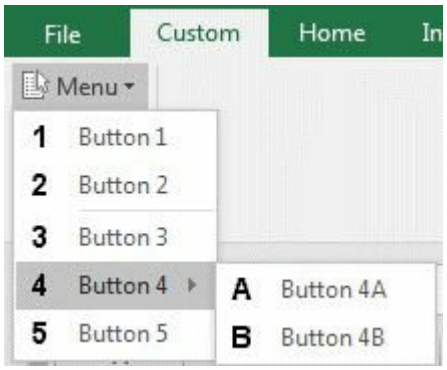
```
'Callback for button1 onAction
Sub Macro1(control As IRibbonControl)
```

```
  MsgBox "Button 1 clicked"
End Sub
```

The MsgBox statement simply displays “Button 1 clicked”, if the user clicks Button 1 in the menu control.

```
'Callback for button2 onAction
Sub Macro2(control As IRibbonControl)
  MsgBox "Button 2 clicked"
End Sub

'Callback for button3 onAction
Sub Macro3(control As IRibbonControl)
  MsgBox "Button 3 clicked"
End Sub

'Callback for button4a onAction
Sub Macro4A(control As IRibbonControl)
  MsgBox "Button 4A clicked"
End Sub

'Callback for button4b onAction
Sub Macro4B(control As IRibbonControl)
  MsgBox "Button 4B clicked"
End Sub

'Callback for button5 onAction
Sub Macro5(control As IRibbonControl)
  MsgBox "Button 5 clicked"
End Sub
```

The appearance of the menu control on the Custom tab:



Note: Other than adding button and menu controls to a menu control with the XML code, you can also add other types of controls, namely toggle button, split button, gallery, check box, general control, and dynamic menu. A dynamic menu is a special control and will only be discussed in the topic Add a dynamic menu.

Add various types of controls

The sample XML code:

```
<customUI xmlns="http://schemas.microsoft.com/office/2006/01/customui">
 <ribbon>
  <tabs>
   <tab id="MyTab" label="Custom Controls" insertAfterMso="TabInsert">
    <group id="Group1" label="button | toggleButton | splitButton">
     <button id="button1"
             label = "Button 1"
             imageMso = "HappyFace"
             onAction="Macro1"/>
     <button id="button2"
             label = "Button 2"
             imageMso = "PenComment"
             onAction="Macro2"/>
     <button id="button3"
             label = "Button 3"
             imageMso = "Clear"
             onAction="Macro3"/>
     <button id="button4"
             label = "Button 4"
             imageMso = "_4"
             onAction="Macro4"/>
     <buttonGroup id="ButtonGrp1"> <!-- Buttons are in horizontal -->
      <button id="button5"
              imageMso = "HappyFace"
              description = "no label"
              onAction="Macro5"/>
      <button id="button6"
              imageMso = "PenComment"
              onAction="Macro6"/>
      <button id="button7"
              imageMso = "Clear"
              onAction="Macro7"/>
     </buttonGroup>

     <separator id="separator1"/>  <!-- Begins a new column -->

     <toggleButton id="toggleBtn1"
                   imageMso = "Breakpoint"
```

```
                onAction="Macro8"/>
    <toggleButton id="toggleBtn2"
                imageMso = "Superscript"
                onAction="Macro9"/>

    <separator id="separator2"/>

    <splitButton id="splitBtn1" size="large">
      <button id="Btn1"
            label = "Button 10"
            imageMso = "Copy"
            onAction = "Macro10"
            supertip = "This is a split button control."
            tag="Button 10"/>
            <!-- See Macro10 in Excel on how tag is used -->

      <menu id="splitMenu">
        <button id="menuButton1"
              label = "Button 10"
              imageMso = "Copy"
              onAction = "Macro10"
              tag="Button 10"/>
              <!-- the same macro as for the one-click Btn1 -->
        <button id="menuButton2"
              label = "Button 11"
              onAction="Macro11"/>
              <!-- no imageMso means no image for the button -->
        <button id="menuButton3"
              label = "Button 12"
              imageMso = "FormatPainter"
              onAction = "Macro12"
              supertip="The 3rd button in the menu."/>
      </menu>
    </splitButton>
  </group>


  <group id="Group3" label="comboBox, dropDown, menu | gallery">
    <comboBox id="comboBox1"
            label = "Template"
            onChange="Combo1_onChange">
      <item id="Template1" label="Template 1"/>
      <item id="Template2" label="Template 2"/>
      <item id="Template3" label="Template 3"/>
      <item id="Template4" label="Template 4"/>
      <item id="Template5" label="Template 5"/>
    </comboBox>
```

```
<dropDown id="dropDown1"
          label = "Template"
          onAction="SelectedItem">
  <item id="Template01" label="Template 1"/>
  <item id="Template02" label="Template 2"/>
  <item id="Template03" label="Template 3"/>
  <item id="Template04" label="Template 4"/>
  <item id="Template05" label="Template 5"/>
</dropDown>

<menu id="Menu1"
      label = "Menu"
      imageMso = "CreateShortcutMenuFromMacro"
      itemSize="normal">  <!-- use itemSize, not size -->
  <button id="button14"
          label = "Button 14"
          onAction="Macro14"/>
  <button id="button15"
          label = "Button 15"
          onAction="Macro15"/>
  <menuSeparator id="menuSep1"/>
  <button id="button16"
          label = "Button 16"
          onAction="Macro16"/>
  <menu id="subMenu1"
        label = "Button 17">
    <button id="button17a"
            label = "Button 17A"
            imageMso = "A"
            onAction="Macro17A"/>
    <button id="button4b"
            label = "Button 17B"
            imageMso = "B"
            onAction="Macro17B"/>
    </menu>
  <button id="button18"
          label = "Button 18"
          onAction="Macro18"/>
</menu>

<separator id="separator3"/>

<gallery id="gallery1"
         imageMso = "ViewAppointmentInCalendar"
         label = "Select a module:"
         rows="3" columns="2"
         onAction="ModuleSelected">
  <item id="module1"
```

```
            label = "Module 1"
            imageMso="ObjectBringForward"/>
      <item id="module2"
            label = "Module 2"
            imageMso="ObjectBringForward"/>
      <item id="module3"
            label = "Module 3"
            imageMso="ObjectBringForward"/>
      <item id="module4"
            label = "Module 4"
            imageMso="ObjectBringForward"/>
      <item id="module5"
            label = "Module 5"
            imageMso="ObjectBringForward"/>
    </gallery>

    <gallery id="gallery2"
             label = "Color gallery"
             imageMso = "CellFillColorPicker"
             columns="4" rows="2"
             itemWidth = "20"
             itemHeight = "15"
             onAction="SelectedColor">
      <item id="black" image="black"/>
      <item id="white" image="white"/>
      <item id="red" image="red"/>
      <item id="green" image="green"/>
      <item id="orange" image="orange"/>
      <item id="yellow" image="yellow"/>
      <item id="purple" image="purple"/>
      <item id="gold" image="gold"/>
    </gallery>
  </group>


  <group id="Group4" label="labelControl, editBox | checkBox">
    <labelControl id="label1" getLabel="getLabel1"/>
    <labelControl id="label2" getLabel="getLabel2"/>

    <editBox id="EditBox1"
             label = "ColorIndex in cell A1:"
             onChange="EditBox1_onChange"/>

    <separator id="separator4"/>

    <checkBox id="checkBox1"
              label = "Check box 1"
              onAction="Checkbox1_Change"/>
```

```
      </group>
    </tab>
  </tabs>
 </ribbon>
</customUI>
```

The edited callback procedures in a standard VBA module:

```
'Callback for button1 onAction
Sub Macro1(control As IRibbonControl)
  MsgBox "Button 1 clicked."
End Sub

'Callback for button2 onAction
Sub Macro2(control As IRibbonControl)
  MsgBox "Button 2 clicked."
End Sub

'Callback for button3 onAction
Sub Macro3(control As IRibbonControl)
  MsgBox "Button 3 clicked."
End Sub

'Callback for button4 onAction
Sub Macro4(control As IRibbonControl)
  MsgBox "Button 4 clicked."
End Sub

'Callback for button5 onAction
Sub Macro5(control As IRibbonControl)
  MsgBox "Button 5 clicked."
End Sub

'Callback for button6 onAction
Sub Macro6(control As IRibbonControl)
  MsgBox "Button 6 clicked."
End Sub

'Callback for button7 onAction
Sub Macro7(control As IRibbonControl)
  MsgBox "Button 7 clicked."
```

```
End Sub

'Callback for toggleBtn1 onAction
Sub Macro8(control As IRibbonControl, pressed As Boolean)
  MsgBox "Status of the toggle button: " & pressed
End Sub

'Callback for toggleBtn2 onAction
Sub Macro9(control As IRibbonControl, pressed As Boolean)
  MsgBox "Status of the toggle button: " & pressed
End Sub

'Callback for Btn1 onAction
Sub Macro10(control As IRibbonControl)
  MsgBox control.Tag & " clicked."
End Sub

'Callback for menuButton2 onAction
Sub Macro11(control As IRibbonControl)
  MsgBox "Button 11 clicked."
End Sub

'Callback for menuButton3 onAction
Sub Macro12(control As IRibbonControl)
  MsgBox "Button 12 clicked."
End Sub

'Callback for comboBox1 onChange
Sub Combo1_onChange(control As IRibbonControl, text As String)
 MsgBox "The displayed text in the combo box: " & text
End Sub

'Callback for dropDown1 onAction
Sub SelectedItem(control As IRibbonControl, id As String, index As Integer)
 MsgBox "You selected Template " & index + 1
End Sub

'Callback for button14 onAction
Sub Macro14(control As IRibbonControl)
  MsgBox "Button 14 clicked"
End Sub
```

```
'Callback for button15 onAction
Sub Macro15(control As IRibbonControl)
  MsgBox "Button 15 clicked"
End Sub

'Callback for button16 onAction
Sub Macro16(control As IRibbonControl)
  MsgBox "Button 16 clicked"
End Sub

'Callback for button17a onAction
Sub Macro17A(control As IRibbonControl)
  MsgBox "Button 17A clicked"
End Sub

'Callback for button4b onAction
Sub Macro17B(control As IRibbonControl)
  MsgBox "Button 17B clicked"
End Sub

'Callback for button18 onAction
Sub Macro18(control As IRibbonControl)
  MsgBox "Button 18 clicked"
End Sub

'Callback for gallery1 onAction
Sub ModuleSelected(control As IRibbonControl, id As String, index As Integer)
 MsgBox " You selected Module " & index + 1
End Sub

'Callback for gallery2 onAction
Sub SelectedColor(control As IRibbonControl, id As String, index As Integer)
 MsgBox "You selected " & id
End Sub

'Callback for label1 getLabel
Sub getLabel1(control As IRibbonControl, ByRef returnedVal)
  If Time() < 0.5 Then
    returnedVal = "Good morning, " & Application.UserName
  Else
```

```
    returnedVal = "Good day, " & Application.UserName
  End If
End Sub
```

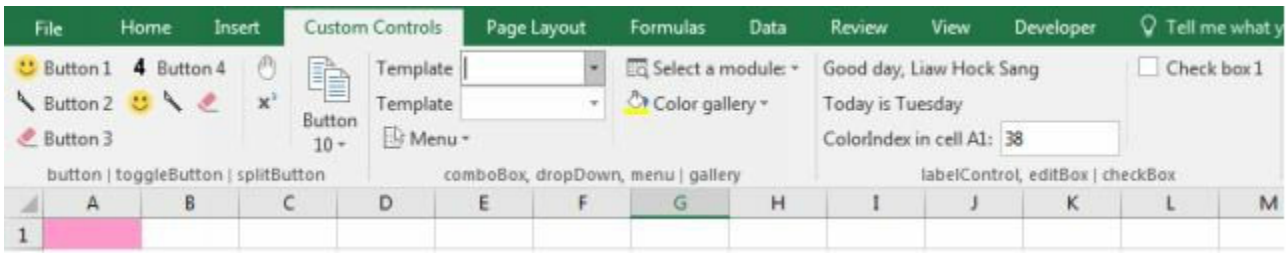
```
'Callback for label2 getLabel
Sub getLabel2(control As IRibbonControl, ByRef returnedVal)
  returnedVal = "Today is " & Format(Date, "dddd")
End Sub
```

The getLabel1 and getLabel2 callback procedures are executed when the workbook is first opened or when the label controls are invalidated.

```
'Callback for checkBox1 onAction
Sub Checkbox1_Change(control As IRibbonControl, pressed As Boolean)
  MsgBox "Check box is checked: " & pressed
End Sub
```

```
'Callback for EditBox1 onChange
Sub EditBox1_onChange(control As IRibbonControl, text As String)
  On Error Resume Next
  Range("A1").Interior.ColorIndex = text
  If Err.Number <> 0 Then _
    MsgBox "Enter an integer between 0 and 56."
End Sub
```

The appearance of the various types of controls on the Custom Controls tab:



Hide and unhide controls, groups, and tabs

Built-in controls (not allowed)

You cannot hide individually the built-in controls in a built-in group. However, you can hide the built-in group and hence all the controls in that group are hidden. Hiding built-in groups will be discussed in the next subtopic. Alternatively, you can disable (and

enable) individually the controls in the group.

The available attributes of the command element in the table below explain why you *cannot hide*, but *can disable* (and enable) the built-in controls individually. On the other hand, the available attributes of group and tab elements explain why you *can hide* (and unhide), but *cannot disable* groups and tabs.

Elements	Available attributes
command	idMso, enabled, getEnabled, onAction
group	id, idMso, label, getLabel, visible, getVisible, image, getImage, imageMso, getImageMso, , keytip, getKeytip, screentip, getScreentip, supertip, getSupertip, insertAfterMso, insertBeforeMso, idQ, insertAfterQ, insertBeforeQ
tab	id, idMso, label, getLabel, visible, getVisible, keytiptag, getKeytip, insertAfterMso, insertBeforeMso, idQ, insertAfterQ, insertBeforeQ

The table above also illustrates the general idea that some attributes (enabled, label, visible …) can have their values set at design time and some (getEnabled, getLabel, getVisible …, which are known as callback attributes) can have their values set dynamically at startup (when Excel starts) and can have their values changed at runtime (by using VBA callback procedures when the elements are invalidated).

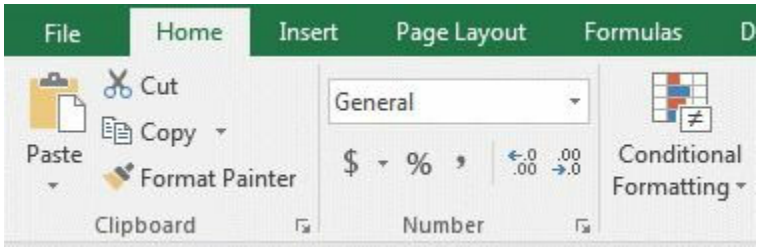
Built-in groups of controls

You can hide groups of controls permanently at design time by using the visible attribute. Alternatively, you can hide (and unhide) them dynamically by using the getVisible callback attribute. visible is a design-time attribute, and getVisible is a runtime attribute.

For example, the following sample XML code hides permanently the Font and Alignment groups on the Home tab:

```
<customUI xmlns="http://schemas.microsoft.com/office/2006/01/customui">
 <ribbon>
  <tabs>
   <tab idMso="TabHome">
    <group idMso="GroupFont" visible="false"/>
    <group idMso="GroupAlignmentExcel" visible="false"/>
   </tab>
  </tabs>
 </ribbon>
</customUI>
```

The appearance of the Home tab with the Font and Alignment groups hidden:



Although the controls in the groups are hidden, you can still execute some of their underlying commands, by using short-key combinations and context menus. For examples, selecting a worksheet cell and pressing Ctrl+B will get the content in the cell bold, and right-clicking a cell will display the Cell context menu and the Mini Toolbar.

As mentioned earlier, you can also to hide (and unhide) built-in groups dynamically at runtime when certain conditions are met. Examples of conditions are a chart sheet is selected, a particular worksheet is selected, a particular item from a combo box is selected, and the Gridlines check box is ticked.

For example, the following sample XML code and the VBA code in a standard VBA module can hide (and unhide) the Alignment group when certain condition at runtime is met:

```
<customUI xmlns="http://schemas.microsoft.com/office/2006/01/customui"
        onLoad="Initialize">
 <ribbon>
  <tabs>
   <tab idMso="TabHome">
    <group idMso="GroupFont" visible="false"/>
    <group idMso="GroupAlignmentExcel"
          getVisible="HideAlignmentGroup"/>
   </tab>
  </tabs>
 </ribbon>
</customUI>
```

The customeUI element includes the onLoad attribute with the Initialize callback procedure. The callback is executed when the workbook is opened.

The GroupAlignmentExcel group element includes the getVisible attribute. The HideAlignmentGroup VBA procedure, which is referenced by the attribute, is executed when the workbook is opened or when the control is invalidated. The condition whether to hide or unhide the group is then evaluated in the procedure.

After saving the file in Custom UI Editor and opening it in Excel for the first time, you will be prompted with the error messages about the Initialize and HideAlignmentGroup procudures, as expected, because the two procedures have yet to be found in a standard VBA module. Click OK to the error messages

The VBA code in a standard VBA module:

```
Public myRibbon As IRibbonUI

'Callback for customUI.onLoad
Sub Initialize(ribbon As IRibbonUI)
  Set myRibbon = ribbon
End Sub

'Callback for GroupAlignmentExcel getVisible
Sub HideAlignmentGroup(control As IRibbonControl, ByRef Visible)
  Visible = TypeName(ActiveSheet) = "Worksheet"
End Sub
```

In the HideAlignmentGroup procedure, the Visible argument is set to True if the active sheet is a worksheet. The result is that the Alignment group is visible. If the Visible argument is set to False, the group is hidden.

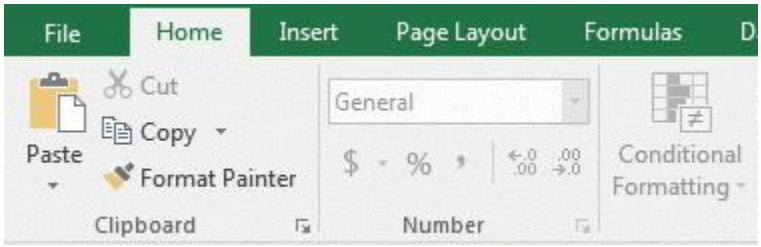
The SheetActivate event handler in the ThisWorkbook module:

```
Private Sub Workbook_SheetActivate(ByVal Sh As Object)
  'In Excel 2010 and later, use the following code statement:
  myRibbon.InvalidateControlMso "GroupAlignmentExcel"

  'Since Excel 2007 does not have the InvalidateControlMso method,
  'use the following statement to invalidate the Ribbon:
  'myRibbon.Invalidate
End Sub
```

The SheetActivate event handler is executed when a different sheet is activated. In Excel 2010 and later, the InvalidateControlMso method invalidates only the Alignment group. In Excel 2007, the Invalidate method invalidates the Ribbon. Subsequently, the HideAlignmentGroup procedure is called. The group is then hidden if the active sheet is not a worksheet. Otherwise, it is visible.

Note: The ribbon object is created when the workbook is opened. Editing your VBA code may possibly destroy the newly created object. Attempting to invalidate controls that associated with the destroyed object is not possible. The only way to re-create the ribbon object is to reopen the workbook.

The appearance of the Home tab with the Alignment group hidden when a chart sheet is activated:



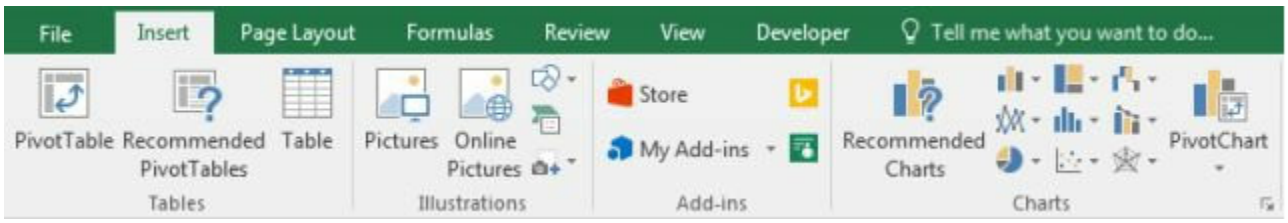
In fact, you can hide more than a group with just one callback procedure. This idea of using just one callback can be extended to tabs and controls. I discuss the idea of changing the appearance of a few controls with the same callback procedure later in the subtopics Hide and unhide custom controls and Disable and enable (built-in and custom) controls.

Built-in tabs

For example, the following sample XML code hides the Home and Data tabs:

```
<customUI xmlns="http://schemas.microsoft.com/office/2006/01/customui">
  <ribbon>
    <tabs>
      <tab idMso="TabHome" visible="false"/>
      <tab idMso="TabData" visible="false"/>
    </tabs>
  </ribbon>
</customUI>
```

The appearance of the Ribbon with the Home and Data tabs hidden:



Although the controls on the tabs are hidden, you can still execute some of their underlying commands, by using short-key combinations and context menus. For examples, selecting a non-empty worksheet cell and pressing the menu-key-sequence Alt+D+S will display the Sort dialog box, pressing Ctrl+H in a worksheet will display the Find and Replace dialog, and right-clicking a cell will display the Cell context menu and the Mini Toolbar.

Similar to hiding (and unhiding) built-in groups, you can hide (and unhide) built-in tabs dynamically at runtime when certain conditions are met. For example, the following

sample XML code and the VBA code in a standard VBA module can hide (and unhide) the Home tab when certain condition at runtime is met:

```
<customUI xmlns="http://schemas.microsoft.com/office/2006/01/customui"
        onLoad="onLoad">
 <ribbon startFromScratch="false">
  <tabs>
   <tab idMso="TabHome" getVisible="HideHomeTab"/>
   <tab idMso="TabData" visible="false"/>
  </tabs>
 </ribbon>
</customUI>
```

The VBA code in a standard VBA module:

```
Public myRibbon As IRibbonUI

'Callback for customUI.onLoad
Sub onLoad(ribbon As IRibbonUI)
  Set myRibbon = ribbon
End Sub

'Callback for TabHome getVisible
Sub HideHomeTab(control As IRibbonControl, ByRef Visible)
  Visible = TypeName(ActiveSheet) = "Worksheet"
End Sub
```

When the HideHomeTab procedure is called, it hides the Home tab if the active sheet is not a worksheet. Otherwise, it gets the Home tab visible.

The SheetActivate event handler in the ThisWorkbook module:

```
Private Sub Workbook_SheetActivate(ByVal Sh As Object)
 'In Excel 2010 and later, use the following code statement:
  myRibbon.InvalidateControlMso "TabHome"

  'Since Excel 2007 does not have the InvalidateControlMso method,
  'use the following statement to invalidate the Ribbon:
  'myRibbon.Invalidate
End Sub
```

The SheetActivate event handler is executed when a different sheet is activated. The InvalidateControlMso method then invalidates only the Home tab in Excel 2010 and later – Only the tab is invalidated; the controls on the Home tab are actually not invalidated. In Excel 2007, the Invalidate method will invalidate all the controls on the Ribbon. Subsequently, the HideHomeTab procedure is called. The Home tab is

hidden if the active sheet is not a worksheet. Otherwise, it is visible.

All built-in tabs

The sample XML code:

```
<customUI xmlns="http://schemas.microsoft.com/office/2006/01/customui">
 <ribbon startFromScratch ="true">
 </ribbon>
</customUI>
```

The appearance of the Ribbon with all tabs hidden:

Although all the built-in tabs are hidden, you can still access some of the built-in controls by using short-key combinations, context menus, and contextual tabs. For examples, pressing the menu-key-sequence Alt+I+S will display the Symbol dialog box, pressing Alt+F1 will insert an empty embedded chart and display the Chart Tools contextual tab, and right-clicking a worksheet cell will display the Cell context menu and the Mini Toolbar.

Custom controls

You cannot hide individually the built-in controls, but you can hide the custom controls individually. You can hide (and unhide) the custom controls either permanently at design time or dynamically at runtime. However, it is more sensible doing it dynamically. You can set your own conditions whether to have the controls hidden.

For example, the following sample XML code adds three buttons to the Home tab before the Font group:

```
<customUI xmlns="http://schemas.microsoft.com/office/2006/01/customui"
        onLoad="Initialize">
 <ribbon>
  <tabs>
   <tab idMso="TabHome">
    <group id="Group1" label="Custom Group" insertBeforeMso="GroupFont">
     <button id="BtnA" imageMso="A"/>
     <button id="BtnB" imageMso="B" getVisible="getVisibleBtnBC"/>
     <button id="BtnC" imageMso="C" getVisible="getVisibleBtnBC"/>
    </group>
```

```
    </tab>
   </tabs>
  </ribbon>
</customUI>
```

Note: The getVisible attributes of the both button elements use the same getVisibleBtnBC callback procedure. The callback is executed when the workbook is opened or when either one or both controls are invalidated.

The VBA code in a standard VBA module:

```
Public myRibbon As IRibbonUI

'Callback for customUI.onLoad
Sub Initialize(ribbon As IRibbonUI)
  Set myRibbon = ribbon
End Sub

'Callback for BtnB and BtnC getVisible
Sub getVisibleBtnBC(control As IRibbonControl, ByRef Visible)
  Visible = ActiveSheet.Name = "Sheet1"
End Sub
```

When the getVisibleBtnBC procedure is called, it gets the BtnB and BtnC buttons visible if the name of the active sheet is Sheet1. Otherwise, it gets the buttons hidden.

The SheetActivate event handler in the ThisWorkbook module:

```
Private Sub Workbook_SheetActivate(ByVal Sh As Object)
  myRibbon.InvalidateControl "BtnB"
  myRibbon.InvalidateControl "BtnC"
End Sub
```

The SheetActivate event handler is executed when a different sheet is activated. The event handler invalidates both BtnB and BtnC buttons. Subsequently, the same getVisibleBtnBC procedure is called. It loops through all the invalidated controls (in this case, the two buttons) that use the same getVisibleBtnBC procedure for their getVisible attributes. Both buttons are visible if the name of the active sheet is Sheet1. Otherwise, both are hidden.

Custom groups and tabs

The way of hiding (and unhiding) custom groups and tabs is the same as the way of hiding (and unhiding) built-in groups and tabs. The following shows an example of

hiding a custom tab when the active sheet is not a worksheet.

The sample XML code:

```
<customUI xmlns="http://schemas.microsoft.com/office/2006/01/customui"
          onLoad="Initialize">
  <ribbon>
    <tabs>
      <tab id="CustomTab" label="Custom" getVisible="HideCustomTab">
        <group id="Group1" label="Custom Group">
          <button id="BtnA" imageMso="A"/>
        </group>
      </tab>
    </tabs>
  </ribbon>
</customUI>
```

The VBA code in a standard VBA module:

```
Public myRibbon As IRibbonUI

'Callback for customUI.onLoad
Sub Initialize(ribbon As IRibbonUI)
   Set myRibbon = ribbon
End Sub

'Callback for CustomTab getVisible
Sub HideCustomTab(control As IRibbonControl, ByRef Visible)
   Visible = TypeName(ActiveSheet) = "Worksheet"
End Sub
```

The SheetActivate event handler in the ThisWorkbook module:

```
Private Sub Workbook_SheetActivate(ByVal Sh As Object)
   myRibbon.InvalidateControl "CustomTab"
End Sub
```

Disable and enable controls, groups, and tabs

Built-in controls

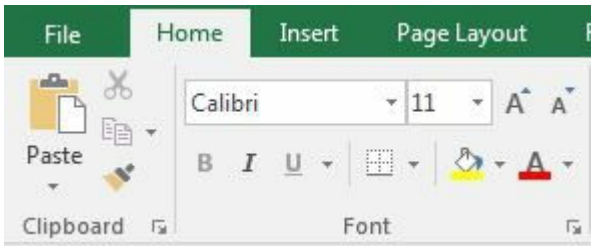
You can disable (and enable) built-in controls either permanently at design time or

dynamically at runtime by using the enabled and getEnabled attributes, respectively. A disabled control then appears greyed out on the Ribbon.

For example, the following sample XML code disables the Copy, Cut, Bold, and Underline controls:

```
<customUI xmlns="http://schemas.microsoft.com/office/2006/01/customui">
 <commands>
  <command idMso="Copy" enabled = "false"/>
  <command idMso="Cut" enabled = "false"/>
  <command idMso="Bold" enabled = "false"/>
  <command idMso="Underline" enabled = "false"/>
 </commands>
</customUI>
```

The appearance of the greyed out Copy, Cut, Bold, and Underline controls on the Ribbon:



Although the disabled controls on the Ribbon are greyed out, you can still execute some, but not all, of their underlying commands, by using short-key combinations. For examples, Ctrl+C for copying and Ctrl+X for cutting still work, but Ctrl+B for getting something bold and Ctrl+U for underling do not work.

You can also set your own conditions whether to disable certain built-in controls at runtime. For example, the following sample XML code and the VBA code in a standard VBA module can get the Bold and Underline controls disabled (and enabled) when certain condition at runtime is met:

```
<customUI xmlns="http://schemas.microsoft.com/office/2006/01/customui"
        onLoad="Initialize">
 <commands>
  <command idMso="Bold" getEnabled="getEnabledBU"/>
  <command idMso="Underline" getEnabled="getEnabledBU"/>
 </commands>
</customUI>
```

Note: The getEnabled attributes of the both command elements refer to the same getEnabledBU procedure. The procedure is called when the workbook is opened or when either one or both controls are invalidated

The VBA code in a standard VBA module:

```
Public myRibbon As IRibbonUI

'Callback for customUI.onLoad
Sub Initialize(ribbon As IRibbonUI)
  Set myRibbon = ribbon
End Sub

'Callback for Bold and Underline getEnabled
Sub getEnabledBU(control As IRibbonControl, ByRef Enabled)
  Enabled = ActiveSheet.Name = "Sheet1"
End Sub
```

In the getEnabledBU procedure, the Enabled argument is set to True if the name of the active sheet is Sheet1. This gets the invalidated controls enabled. Otherwise, the controls are disabled.

The SheetActivate event handler in the ThisWorkbook module:

```
Private Sub Workbook_SheetActivate(ByVal Sh As Object)
  'In Excel 2010 and later, use the following code statements:
  myRibbon.InvalidateControlMso "Bold"
  myRibbon.InvalidateControlMso "Underline"

  'Since Excel 2007 does not have the InvalidateControlMso method,
  'use the following statement to invalidate the Ribbon:
  'myRibbon.Invalidate
End Sub
```

The SheetActivate event handler invalidates both Bold and Underline controls when a different sheet is activated. Subsequently, the same getEnabledBU procedure is called. Both controls are then enabled if the name of the active sheet is Sheet1. Otherwise, both are disabled.

Built-in- and custom groups and built-in- and custom tabs (not allowed)

You cannot disable groups of controls and tabs because the group and tab elements do not have the enabled and getEnabled attributes that allow you to do so. Please refer to the table in the subtopic Hide and unhide built-in controls (not allowed) or the technical article Customizing the Office (2007) Ribbon User Interface for Developers (Part 2 of 3) on the available elements and attributes. Nevertheless, you can still disable (and

enable) the controls individually, as discussed earlier.

Certain Custom controls

The way to disable (and enable) custom controls by using the getEnabled attribute is the same as the way to hide (and unhide) custom controls by using the getVisible attribute. Instead of repeating what have been discussed, I will discuss on how to disable (and enable) certain custom controls based on their ids.

The sample XML code:

```
<customUI
 xmlns="http://schemas.microsoft.com/office/2006/01/customui"
 onLoad="Initialize">
 <ribbon>
  <tabs>
   <tab idMso="TabHome">
    <group id="Group1" label="Attn Sh" insertBeforeMso="GroupFont">
     <button id="BtnInsert0"
             label = "Insert 0"
             imageMso = "_0"
             onAction = "Insert0"
             getEnabled="GetEnabledAttnSh"/>
     <button id="BtnInsert1"
             label = "Insert 1"
             imageMso="_1"
             onAction = "Insert1"
             getEnabled="GetEnabledAttnSh"/>
     <button id="BtnUpdateRcd"
             label = "Update Rcd"
             imageMso="SelectAll"
             onAction="UpdateRcd"
             getEnabled="GetEnabledAttnSh"/>
    </group>
   </tab>
  </tabs>
 </ribbon>
</customUI>
```

When the workbook is opened in Excel, the Initialize and GetEnabledAttnSh callbacks are automatically executed.

After saving the file in Custom UI Editor and opening it in Excel for the first time, you will be prompted with the error messages about the Initialize and GetEnabledAttnSh procedures, as expected, because the two callback procedures have yet to be found in a standard VBA module. Click OK to the error messages.

The VBA code in a standard VBA module:

```
Public myRibbon As IRibbonUI
Public myID As String

'Callback for customUI.onLoad
Sub Initialize(ribbon As IRibbonUI)
  Set myRibbon = ribbon
End Sub

'Callback for all buttons getEnabled
Sub GetEnabledAttnSh(control As IRibbonControl, ByRef Enabled)
  If control.ID Like myID Then
      Enabled = True
  Else
     Enabled = False
  End If
End Sub

'Callback for BtnInsert0 onAction
Sub Insert0(control As IRibbonControl)
   MsgBox "Insert 0 clicked"
End Sub

'Callback for BtnInsert1 onAction
Sub Insert1(control As IRibbonControl)
   MsgBox "Insert 1 clicked"
End Sub

'Callback for BtnUpdateRcd onAction
Sub UpdateRcd(control As IRibbonControl)
   MsgBox "Update Rcd clicked"
End Sub
```

To disable (and enable) certain custom controls based on their ids in the XML code, include the following code in the existing standard VBA module or in a new standard VBA module:

```
Sub EnableAll()
  Call RefreshRibbon("*")
```

```
End Sub

Sub DisableAll()
  Call RefreshRibbon("")
End Sub

Sub EnableInsert()
  Call RefreshRibbon("*Insert*")
End Sub

Sub EnableInsert1()
  Call RefreshRibbon("*Insert1")
End Sub
```

Execute each of the procedures above to see the effect. Each of these procedures calls the RefreshRibbon procedure to invalidate all the three controls. See below for the RefreshRibbon procedure. Whether a control is enabled (or disabled) depends on the value passed by the argument in RefreshRibbon. Once the controls are invalidated, the GetEnabledAttnSh procedure is called. It loops through all the invalidated controls that share the same callback. If the id of the control matches the value of the argument, the control is enabled. Otherwise, it is disabled.

```
Sub RefreshRibbon(ID As String)
  myID = ID

  'Invalidate all controls on the Ribbon
  'myRibbon.Invalidate

  'Alternatively, it is best to invalidate only the three controls
  myRibbon.InvalidateControl "BtnInsert0"
  myRibbon.InvalidateControl "BtnInsert1"
  myRibbon.InvalidateControl "BtnUpdateRcd"
End Sub
```

If to enable all the three controls when the active sheet is a worksheet and to disable them when the active sheet is not a worksheet, you may simply include the following event handler in the ThisWorkbook module:

```
Private Sub Workbook_SheetActivate(ByVal Sh As Object)
  If TypeName(Sh) = "Worksheet" Then
   Call EnableAll
  Else
    Call DisableAll
  End If
```

```
End Sub
```

For example, the following shows the appearance of the Attn Sh group with two enabled controls after the EnableInsert procedure is executed:

Alternatively, you can also disable (and enable) particular custom controls based on the values of their tag attributes, instead of the id attributes.

Repurpose certain built-in controls

You can override the functionality of a built-in control either permanently or temporarily by using the command element and its onAction attribute.

For example, the following sample XML code repurposes the Cut and Bold controls:

```
<customUI xmlns="http://schemas.microsoft.com/office/2006/01/customui">
 <commands>
   <command idMso="Cut" onAction = "MyCut"/>
   <command idMso="Bold" onAction = "MyBold"/>
 </commands>
</customUI>
```

The following edited callback procedure in a standard VBA module *permanently* repurposes the function of the Cut control:

```
'Callback for Cut onAction
Sub MyCut(control As IRibbonControl, ByRef cancelDefault)
  MsgBox "The Cut button has been repurposed permanently."
End Sub
```

Clicking the repurposed Cut control executes the MyCut procedure.

However, a repurposed control does not always work as intended if a user does not click the control or use the appropriate Alt+keystrokes to trigger the control (in this case, Alt+H+X to trigger the Cut control). For example, you can still access the cutting functionality by using the Ctrl+X key combination.

You can also *temporarily* override the functionality of a built-in control and restore its

functionality by simply setting the cancelDefault argument in the callback procedure to False. For example, the following edited callback procedure restores the function of the Bold control after displaying a message to a user:

```
'Callback for Bold onAction
Sub MyBold(control As IRibbonControl, pressed As Boolean, _
    ByRef cancelDefault)
  MsgBox "The Bold button was temporarily repurposed." & Chr(10) & _
        "Click OK to restore its normal functionality."
  cancelDefault = False
End Sub
```

Caution: The Bold control is a toggle button. The callback for its onAction attribute must have three arguments, as shown above. As I was writing this book, Custom UI Editor for Microsoft Office that I installed generated for me a callback signature with a wrong number of arguments, as shown below:

```
'Callback for Bold onAction (wrong number of arguments)
Sub MyBold(control As IRibbonControl, ByRef cancelDefault)
End Sub
```

Monitor a built-in control

Repurposing a built-in control can be used to monitor the control. Sometimes you may want to take necessary action when a particular built-in control is clicked. For example, you want to intercept a printing if the total number of pages to be printed is greater than 100.

Alternatively, it is best to intercept the printing by the Workbook_BeforePrint event handler. See below for an example:

```
Private Sub Workbook_BeforePrint(Cancel As Boolean)
  If ActiveSheet.PageSetup.Pages.Count > 100 Then
    If MsgBox("To print " & ActiveSheet.PageSetup.Pages.Count & _
          " pages!", vbYesNo) = vbNo Then
      Cancel = True   'Intercept printing
    End If
  End If
  'By default, Cancel = False, i.e.: continue to print
End Sub
```

Add a dynamic menu

In the sample program later at the end of the book, you will see how to populate a down-drop control with items and a gallery control with images dynamically at runtime by using VBA procedures, which are referenced by the getItemLabel and getItemImage callback attributes, respectively. Another control that allows you to populate its content dynamically (just like the drop-down control) is the combo box control.

A dynamic menu control can do even more at runtime. It is the only one control whose content's structure can be changed at runtime. It can contain both custom and built-in controls – including other dynamic menus. The VBA procedure, referenced by the getContent attribute of the control, is the one that builds the XML code for the content of the menu at runtime.

We will go through a simple example of using a dynamic menu control that displays a different menu for each of the three worksheets (named Data, Analysis, and Reports) in a workbook. To create the example, execute the following steps:

1. Create a new workbook and save it as a macro-enabled workbook.
2. Add additional worksheets if needed and rename a worksheet Data, a worksheet Analysis, and another worksheet Reports.
3. Close the workbook and open it in Custom UI Editor.
4. In Custom UI Editor, click Insert and choose Office 2007 Custom UI Part.
5. Copy and paste the following XML code:

```
<customUI
 xmlns="http://schemas.microsoft.com/office/2006/01/customui"
 onLoad="Initialize">
 <ribbon>
  <tabs>
   <tab idMso="TabHome">
    <group id="Group1" label="Dynamic Menu vs Static Controls"
          insertBeforeMso="GroupClipboard">
     <dynamicMenu id="DynamicMenu"
                  imageMso = "HappyFace"
                  label = "Dynamic Menu"
                  size = "large"
                  getContent="GetMenuContent"/>

     <separator id="sep1"/>
     <button id="CustomBtn1" imageMso="A"
             label="Custom Button"
             onAction="MacroCustomButton"/>
     <button idMso="Connections"/>
     <menu idMso="WhatIfAnalysisMenu"/>
    </group>
```

```
    </tab>
   </tabs>
  </ribbon>
</customUI>
```

A group element can contain not only a dynamic menu, but also other controls.

6. Click the Validate button on the toolbar to check for errors.

7. Save and close the file.

8. Open the file in Excel.

Click OK to the error messages.

9. Press Alt+11 to activate VBE.

10. Insert a standard VBA module and copy and paste the following VBA code into the module:

```
Public myRibbon As IRibbonUI

'Callback for customUI.onLoad
Sub Initialize(ribbon As IRibbonUI)
  Set myRibbon = ribbon
End Sub

'Callback for DynamicMenu getContent
Sub GetMenuContent(control As IRibbonControl, ByRef content)
  Dim xml As String

  xml = "<menu xmlns=" & _
      """http://schemas.microsoft.com/office/2006/01/customui"">"
  Select Case ActiveSheet.Name
    Case "Data"
      xml = xml & "<button id=""Btn1"" imageMso=""Cut""" & _
                  " label=""Reformat""" & _
                  " onAction=""Reformat""/>"
      xml = xml & "<checkBox id=""checkBox1""" & _
                   " label=""Include OEM""" & _
                   " onAction=""Checkbox1_Change""/>"
     xml = xml & "<menu id=""submenu1"" label=""Optional"">"
      xml = xml & "  <button id=""Btn2""" & _
                   " imageMso=""PenComment""" & _
                   " label=""TouchUp""" & _
                   " onAction=""TouchUp""/>"
      xml = xml & "  <button id=""Btn3""" & _
                   " imageMso=""Breakpoint""" & _
                   " label=""Polish""" & _
```

```
                    " onAction=""Polish""/>"
    xml = xml & "  <menuSeparator id=""div2""/>"
    xml = xml & "  <dynamicMenu id=""subMenu""" & _
                    " label=""Submenu""" & _
                    " getContent=""GetSubContent""/>"
    xml = xml & "</menu>"
    xml = xml & "<button idMso=""SortDialog""/>"

  Case "Analysis"
    xml = xml & "<button id=""Btn1"" imageMso=""_1""" & _
                " label=""Analysis 1""" & _
                " onAction=""Analysis1""/>"
    xml = xml & "<button id=""Btn2"" imageMso=""_2""" & _
                " label=""Analysis 2""" & _
                " onAction=""Analysis2""/>"
    xml = xml & "<button id=""Btn3"" imageMso=""_3""" & _
                " label=""Analysis 3""" & _
                " onAction=""Analysis3""/>"
    xml = xml & "<menuSeparator id=""div2""/>"
    xml = xml & "<dynamicMenu id=""subMenu""" & _
                  " label=""Submenu""" & _
                  " getContent=""GetSubContent""/>"

  Case "Reports"
    xml = xml & "<button id=""Btn1"" imageMso=""A""" & _
                " label=""Report A""" & _
                " onAction=""ReportA""/>"
    xml = xml & "<button id=""Btn2"" imageMso=""B""" & _
                " label=""Report B""" & _
                " onAction=""ReportB""/>"
    xml = xml & "<button id=""Btn3"" imageMso=""C""" & _
                " label=""Report C""" & _
                " onAction=""ReportC""/>"
    xml = xml & "<menuSeparator id=""div2""/>"
    xml = xml & "<dynamicMenu id=""subMenu""" & _
                  " label=""Submenu""" & _
                  " getContent=""GetSubContent""/>"
  Case Else
    'Empty dynamic menu
End Select
xml = xml & _
    "</menu>"
```

```
    content = xml

    'To view the XML code in the Immediate window
    'Debug.Print xml
End Sub
```

The GetMenuContent callback procedure is executed when the workbook is first opened or when the dynamic menu control is invalidated. This procedure builds the XML code for the content of the dynamic menu.

Note: The VBA code above is indented in a way that it looks like the one in Custom UI Editor. For a better view, send the constructed XML code to the Immediate window by using the Debug.Print statement. Copy and paste the code into Notepad and press Enter after each opening tag (such as <menu … >) and after each closing tag (such as <button … />).

```
'Callback for Sub Dynamic Menu getContent
Sub GetSubContent(control As IRibbonControl, ByRef SubContent)
    Dim xml As String

    xml = "<menu xmlns=" & _
          """http://schemas.microsoft.com/office/2006/01/customui"">"
    xml = xml & "<button id=""subBtn1"" label=""P""" & _
                  " onAction=""MacroSubBtn1""/>"
    xml = xml & "<button id=""subBtn2"" label=""Q""" & _
                  " onAction=""MacroSubBtn2""/>"
    xml = xml & "<button id=""subBtn3"" label=""R""" & _
                  " onAction=""MacroSubBtn3""/>"
    xml = xml & _
          "</menu>"

    SubContent = xml
End Sub
```

For simplicity, all the three different sets of menu (for the three different worksheets) use the same sub dynamic menu.

```
'Callbacks for the controls in the dynamic menu
'when the Data sheet is activated
Sub Reformat(control As IRibbonControl)
    MsgBox "Reformat"
End Sub
```

```
Sub Checkbox1_Change(control As IRibbonControl, _
    pressed As Boolean)
  MsgBox "OEM check box is checked: " & pressed
End Sub

Sub TouchUp(control As IRibbonControl)
  MsgBox "TouchUp"
End Sub

Sub Polish(control As IRibbonControl)
  MsgBox "Polish"
End Sub

'Callbacks for the controls in the dynamic menu
'when the Analysis sheet is activated
Sub Analysis1(control As IRibbonControl)
  MsgBox "Analysis 1"
End Sub

Sub Analysis2(control As IRibbonControl)
  MsgBox "Analysis 2"
End Sub

Sub Analysis3(control As IRibbonControl)
  MsgBox "Analysis 3"
End Sub

'Callbacks for the controls in the dynamic menu
'when the Reports sheet is activated
Sub ReportA(control As IRibbonControl)
  MsgBox "Report A"
End Sub

Sub ReportB(control As IRibbonControl)
  MsgBox "Report B"
End Sub

Sub ReportC(control As IRibbonControl)
  MsgBox "Report C"
End Sub
```

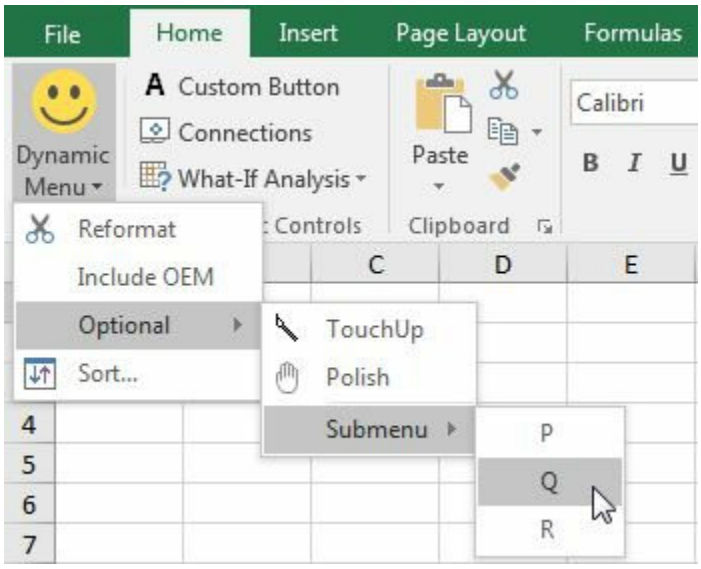
```
'Callbacks for the controls in the sub dynamic menu
Sub MacroSubBtn1(ontrol As IRibbonControl)
  MsgBox "P"
End Sub

Sub MacroSubBtn2(ontrol As IRibbonControl)
  MsgBox "Q"
End Sub

Sub MacroSubBtn3(ontrol As IRibbonControl)
  MsgBox "R"
End Sub

'Callback for CustomBtn1 onAction
Sub MacroCustomButton(control As IRibbonControl)
  MsgBox "Custom Button"
End Sub
```

11. Insert the following VBA code into the ThisWorkbook module:

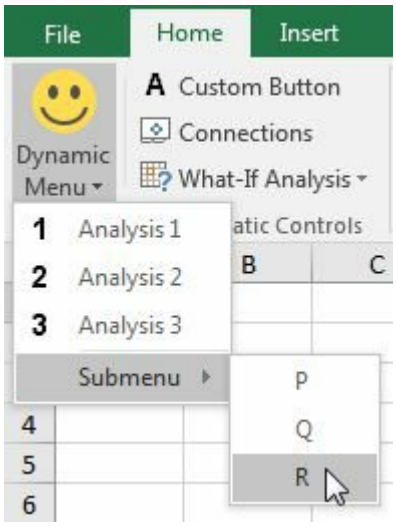
```
Private Sub Workbook_SheetActivate(ByVal Sh As Object)
  myRibbon.InvalidateControl "DynamicMenu"
End Sub
```

12. Save, close, and reopen the workbook in Excel.

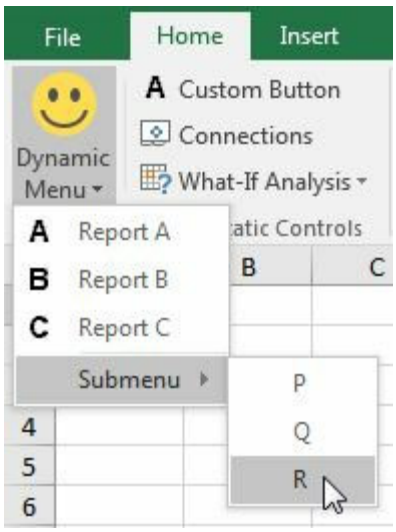
The content of the dynamic menu on the Home tab when the active sheet is the Data worksheet:



The content of the dynamic menu when the active sheet is the Analysis worksheet:



The content of the dynamic menu when the active sheet is the Reports worksheet:



Retain the ticked condition of a custom check box

In the sample XML and VBA code above, when a user in the Data worksheet clicks the custom check box on the dynamic menu, the check box is graphically ticked and unticked accordingly.

However, if the user after getting the check box ticked activates other worksheet by clicking its sheet tab, the dynamic menu is invalidated. Any data (including the ticked condition of the check box) associatied with the menu is destroyed. When the Data worksheet is reactivated, the menu is then rebuilt by calling the GetMenuContent procedure, but the check box resets to its default state (that is, the unticked condition).

If to retain its condition, you may store its state before it is invalidated and restore its state when the menu is rebuilt. This can be done by using a module-level variable and the getPressed callback attribute. To accomplish this task, modify the existing VBA code by executing the following steps:

1. Declare a module-level variable to store the state of the check box:

```
Public myRibbon As IRibbonUI
Dim Checkbox1Pressed As Boolean

```

Note: The greyed out VBA code is to indicate the location where you should declare the Checkbox1Pressed module-level variable.

2. Include the VBA code for the getPressed attribute in the GetMenuContent procedure:

```
Select Case ActiveSheet.Name
  Case "Data"
    xml = xml & "<button id=""Btn1"" imageMso=""Cut""" & _
                " label=""Reformat""" & _
                " onAction=""Reformat""/>"
    xml = xml & "<checkBox id=""checkBox1""" & _
                 " label=""Include OEM""" & _
                 " getPressed=""CheckBox1getPressed """ & _
                 " onAction=""Checkbox1_Change""/>"
```

3. Include an additional code statement in the Checkbox1_Change procedure, which is referenced by the onAction attribute of the checkBox element:

```
Sub Checkbox1_Change(control As IRibbonControl, _
   pressed As Boolean)
  MsgBox "OEM check box is checked: " & pressed
  Checkbox1Pressed = pressed
End Sub
```

When a user clicks the check box, the Checkbox1_Change is exexcuted and the state of the check box is then stored in the Checkbox1Pressed variable.

4. Insert the CheckBox1getPressed procedure, which is referenced by the getPressed attribute, into the same standard VBA module that stores the Checkbox1_Change procedure:

```
Sub CheckBox1getPressed(control As IRibbonControl, _
   ByRef returnedVal)
  returnedVal = Checkbox1Pressed
End Sub
```

When the user reactivates the Data worksheet and clicks the dynamic menu, the menu is rebuilt and the CheckBox1getPressed procedure is executed. The state of the check box is then reassigned by the Checkbox1Pressed variable,

which stores the state the check box before it is invalidated.

5. Save, close, and reopen the workbook in Excel.

   The check box is now able to retain its state even after the dynamic menu is invalidated and rebuilt. As you can see, the Checkbox1Pressed module-level variable retains its value between procedure calls.

In general, public-level variables, module-level variables, and procedure-level *static* variables in a workbook retain their values even after the code execution in the workbook has ended. You can purge the values stored by these variables using the following four methods:

- You execute the End statement in a procedure or in the Immediate window.
- In VBE, you choose Run | Reset.
- When VBE displays the standard error message box (because an unhandled runtime error occurrs), you click the End button on the message box.
- You close the workbook file.

If there is no unhandled error, only you as a programmer can execute the first two methods, and the user can only execute the last method. Hence, the Checkbox1Pressed variable can properly reflect the state of the check box as long as the file remains open.

If to retain the state of the check box even after the user closes and reopens the file, you may store its state in a cell of a hidden worksheet or in the Windows Registry. You can do the former easily. I will discuss the latter in my future book.

Cell context menu (in Excel 2010 and later)

Add a button control

Suppose that you need to perform Excel calculations on those values preceded with currency symbols in a worksheet. However, those values are interpreted as text. You have then written a VBA procedure to remove the currency symbols in a range of selected cells. To make the procedure more accessible, you want to place a shortcut to the procedure on the Cell context menu. The following are the XML code and VBA code to accomplish the task.

The sample XML code:

```
<customUI xmlns="http://schemas.microsoft.com/office/2009/07/customui">
```

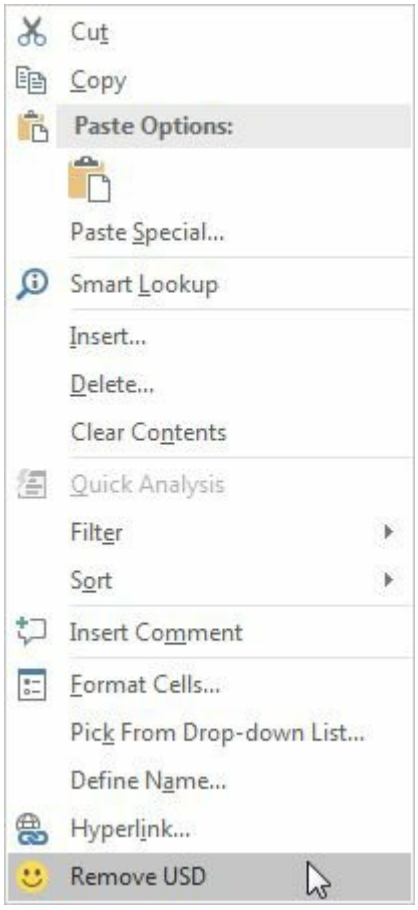
```
  <contextMenus>
    <contextMenu idMso="ContextMenuCell">
      <button id="RemoveUSD"
              label="Remove USD"
              insertAfterMso="HyperlinkInsert"
              onAction="RemoveUSD"
              imageMso="HappyFace"/>
    </contextMenu>
  </contextMenus>
</customUI>
```

Note: In Custom UI Editor, click Insert and choose Office 2010 Custom UI Part, not Office 2007 Custom UI Part because the latter does not have contextMenus as its child element.

The procedure in a standard VBA module:

```
Sub RemoveUSD(control As IRibbonControl)
'To remove the currency symbols in a range of selected cells
'For example, USD 500.25 becomes 500.25
  Dim workRng As Range
  Dim Item As Range

  On Error Resume Next
  Set workRng = Intersect(Selection, _
                Selection.Cells.SpecialCells _
                (xlCellTypeConstants, xlTextValues))
  If Not workRng Is Nothing Then
    For Each Item In workRng
      If UCase(Left(Item, 3)) = "USD" Then _
        Item = Right(Item, Len(Item) - 3)
    Next Item
  End If
End Sub
```

The following is the appearance of the Remove USD button on the Cell context menu:



Add other types of controls

Other than adding a button control to the Cell context menu with the XML code, you can also add six other types of built-in and custom controls: toggle button, split button, menu, gallery, check box, and dynamic menu.

The XML code and VBA code for the additional six types of controls added to the Cell content menu are compiled from the previously discussed topics in customizing the Ribbon.

The sample XML code:

```
<customUI xmlns="http://schemas.microsoft.com/office/2009/07/customui"
 onLoad="Initialize">
 <contextMenus>
   <contextMenu idMso="ContextMenuCell">
     <button id="RemoveUSD"
             label = "Remove USD"
             insertAfterMso = "HyperlinkInsert"
             onAction = "RemoveUSD"
             imageMso="HappyFace"/>
```

```
<menu id="menu1"
      label = "Various Types"
      insertBeforeMso = "Cut"
      imageMso="HappyFace">

    <toggleButton idMso="Bold"/>

    <splitButton id="splitBtn1">
      <button id="Btn0"
              label = "Button1"
              imageMso = "Copy"
              onAction = "Macro1s"
              supertip = "This is a split button control."
              tag="Button1" />
              <!-- See Macro1s in Excel on how tag is used -->

      <menu id="splitMenu">  <!-- use itemSize, not size -->
        <button id="menuButton1"
                label = "Button1"
                imageMso = "Copy"
                onAction = "Macro1s"
                tag="Button1" />
                <!-- the same macro as for the one-click Btn0 -->
        <button id="menuButton2"
                label = "Button2"
                onAction="Macro2s" />
                <!-- no imageMso means no image for the button -->
        <button id="menuButton3"
                label = "Button3"
                imageMso = "FormatPainter"
                onAction = "Macro3s"
                supertip="The 3rd button in the menu."  />
      </menu>
    </splitButton>

    <menu id="Menu1"
          label = "Menu"
          imageMso = "CreateShortcutMenuFromMacro"
          itemSize="normal">
      <button id="button1"
          label = "Button 1"
          imageMso = "_1"
          onAction="Macro1m"/>
      <button id="button2"
          label = "Button 2"
          imageMso = "_2"
          onAction="Macro2m"/>
```

```
      <menuSeparator id="menuSep1"/>
      <button id="button3"
              label = "Button 3"
              imageMso = "_3"
              onAction="Macro3m"/>
      <menu id="subMenu1"
            label = "Button 4"
            imageMso="_4">
        <button id="button4a"
                label = "Button 4A"
                imageMso = "A"
                onAction="Macro4Am"/>
      <button id="button4b"
              label = "Button 4B"
              imageMso = "B"
              onAction="Macro4Bm"/>
      </menu>
    <button id="button5"
            label = "Button 5"
            imageMso = "_5"
            onAction="Macro5m"/>
    </menu>

    <gallery id="gallery1"
             label = "Color gallery"
             imageMso = "CellFillColorPicker"
             columns="4" rows="2"
             itemWidth = "20"
             itemHeight = "15"
             onAction="SelectedColor">
      <item id="black" image="black" />
      <item id="white" image="white" />
      <item id="red" image="red" />
      <item id="green" image="green" />
      <item id="orange" image="orange" />
      <item id="yellow" image="yellow" />
      <item id="purple" image="purple" />
      <item id="gold" image="gold" />
    </gallery>

    <checkBox idMso="GridlinesExcel"/>

    <dynamicMenu id="DynamicMenu"
                 imageMso = "HappyFace"
                 label = "Dynamic Menu"
                 getContent="GetMenuContent"/>
  </menu>
</contextMenu>
```

```
  </contextMenus>
</customUI>
```

The VBA code in a standard VBA module:

```
Public myRibbon As IRibbonUI
Dim Checkbox1Pressed As Boolean

'Callback for customUI.onLoad
Sub Initialize(ribbon As IRibbonUI)
  Set myRibbon = ribbon
End Sub

'Callback for DynamicMenu getContent
Sub GetMenuContent(control As IRibbonControl, ByRef content)
  Dim xml As String

  xml = "<menu xmlns=" & _
      """http://schemas.microsoft.com/office/2006/01/customui"">"
  Select Case ActiveSheet.Name
    Case "Data"
      xml = xml & "<button id=""Btn1"" imageMso=""Cut""" & _
                  " label=""Reformat""" & _
                  " onAction=""Reformat""/>"
      xml = xml & "<checkBox id=""checkBox1""" & _
                  " label=""Include OEM""" & _
                  " getPressed=""Checkbox1getPressed""" & _
                  " onAction=""Checkbox1_Change""/>"
      xml = xml & "<menu id=""submenu1"" label=""Optional"">"
      xml = xml & "  <button id=""Btn2""" & _
                  " imageMso=""PenComment""" & _
                  " label=""TouchUp""" & _
                  " onAction=""TouchUp""/>"
      xml = xml & "  <button id=""Btn3""" & _
                  " imageMso=""Breakpoint""" & _
                  " label=""Polish""" & _
                  " onAction=""Polish""/>"
      xml = xml & "  <menuSeparator id=""div2""/>"
      xml = xml & "  <dynamicMenu id=""subMenu""" & _
                    " label=""Submenu""" & _
                    " getContent=""GetSubContent""/>"
      xml = xml & "</menu>"
```

```
        xml = xml & "<button idMso=""SortDialog""/>"

    Case "Analysis"
        xml = xml & "<button id=""Btn1"" imageMso=""_1""" & _
                    " label=""Analysis 1""" & _
                    " onAction=""Analysis1""/>"
        xml = xml & "<button id=""Btn2"" imageMso=""_2""" & _
                    " label=""Analysis 2""" & _
                    " onAction=""Analysis2""/>"
        xml = xml & "<button id=""Btn3"" imageMso=""_3""" & _
                    " label=""Analysis 3""" & _
                    " onAction=""Analysis3""/>"
        xml = xml & "<menuSeparator id=""div2""/>"
        xml = xml & "<dynamicMenu id=""subMenu""" & _
                      " label=""Submenu""" & _
                      " getContent=""GetSubContent""/>"

    Case "Reports"
        xml = xml & "<button id=""Btn1"" imageMso=""A""" & _
                    " label=""Report A""" & _
                    " onAction=""ReportA""/>"
        xml = xml & "<button id=""Btn2"" imageMso=""B""" & _
                    " label=""Report B""" & _
                    " onAction=""ReportB""/>"
        xml = xml & "<button id=""Btn3"" imageMso=""C""" & _
                    " label=""Report C""" & _
                    " onAction=""ReportC""/>"
        xml = xml & "<menuSeparator id=""div2""/>"
        xml = xml & "<dynamicMenu id=""subMenu""" & _
                      " label=""Submenu""" & _
                      " getContent=""GetSubContent""/>"
    Case Else
        'Empty dynamic menu
  End Select
  xml = xml & _
        "</menu>"

  content = xml

  'To view the XML code in the Immediate window
  'Debug.Print content
End Sub
```

```
'Callback for Sub Dynamic Menu getContent
Sub GetSubContent(control As IRibbonControl, ByRef SubContent)
  Dim xml As String

  xml = "<menu xmlns=" & _
       """http://schemas.microsoft.com/office/2006/01/customui"">"
  xml = xml & "<button id=""subBtn1"" label=""P""" & _
                " onAction=""MacroSubBtn1""/>"
  xml = xml & "<button id=""subBtn2"" label=""Q""" & _
                " onAction=""MacroSubBtn2""/>"
  xml = xml & "<button id=""subBtn3"" label=""R""" & _
                " onAction=""MacroSubBtn3""/>"
  xml = xml & _
       "</menu>"

  SubContent = xml
End Sub

'Callbacks for the controls in the dynamic menu
'when the Data sheet is activated
Sub Reformat(control As IRibbonControl)
  MsgBox "Reformat"
End Sub

Sub Checkbox1_Change(control As IRibbonControl, _
   pressed As Boolean)
  MsgBox "OEM check box is checked: " & pressed
  Checkbox1Pressed = pressed
End Sub

Sub CheckBox1getPressed(control As IRibbonControl, ByRef returnedVal)
  returnedVal = Checkbox1Pressed
End Sub

Sub TouchUp(control As IRibbonControl)
  MsgBox "TouchUp"
End Sub

Sub Polish(control As IRibbonControl)
  MsgBox "Polish"
End Sub
```

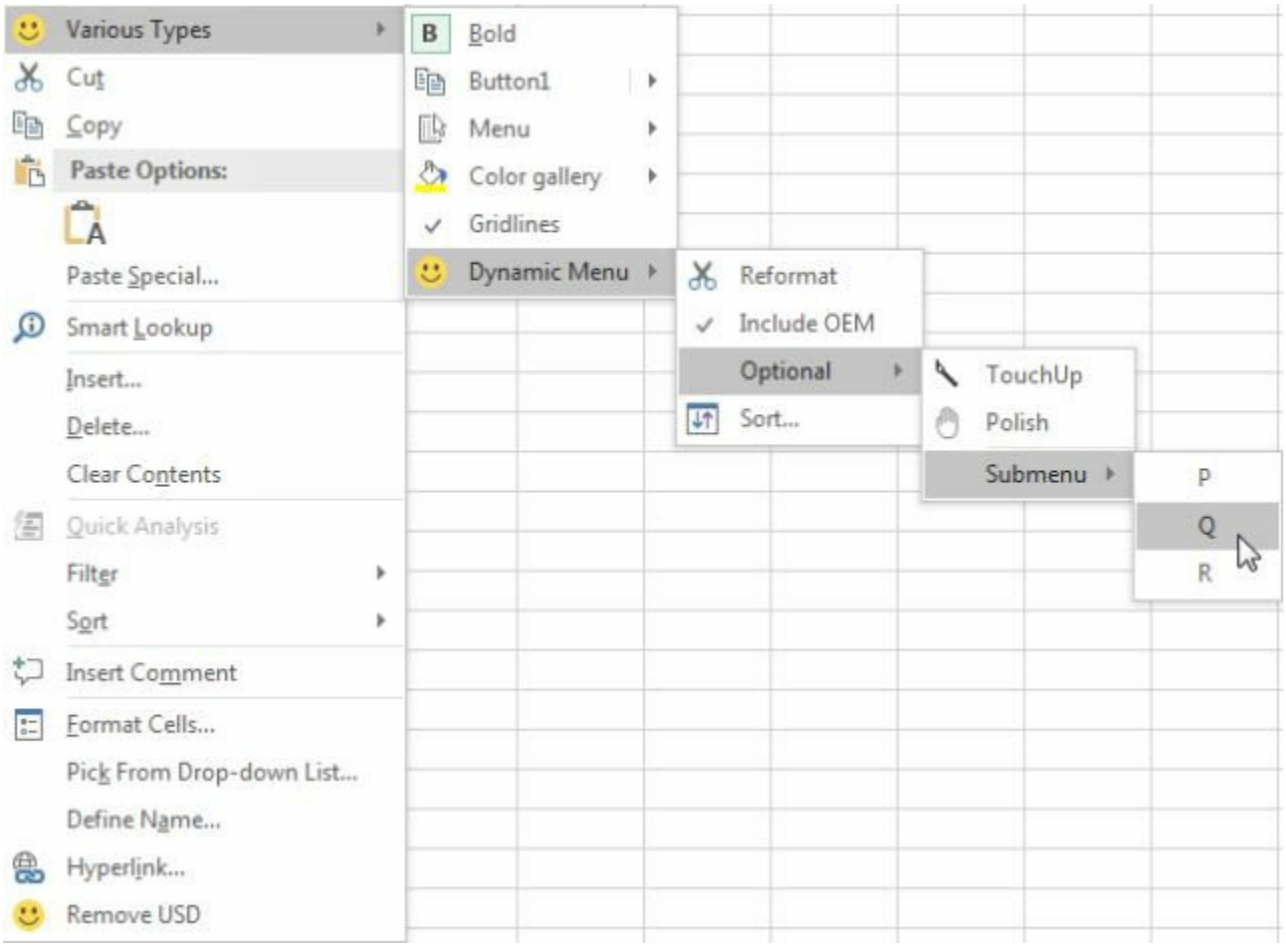
```
'Callbacks for the controls in the dynamic menu
'when the Analysis sheet is activated
Sub Analysis1(control As IRibbonControl)
  MsgBox "Analysis 1"
End Sub

Sub Analysis2(control As IRibbonControl)
  MsgBox "Analysis 2"
End Sub

Sub Analysis3(control As IRibbonControl)
  MsgBox "Analysis 3"
End Sub

'Callbacks for the controls in the dynamic menu
'when the Reports sheet is activated
Sub ReportA(control As IRibbonControl)
  MsgBox "Report A"
End Sub

Sub ReportB(control As IRibbonControl)
  MsgBox "Report B"
End Sub

Sub ReportC(control As IRibbonControl)
  MsgBox "Report C"
End Sub

'Callbacks for the controls in the sub dynamic menu
Sub MacroSubBtn1(ontrol As IRibbonControl)
  MsgBox "P"
End Sub

Sub MacroSubBtn2(ontrol As IRibbonControl)
  MsgBox "Q"
End Sub

Sub MacroSubBtn3(ontrol As IRibbonControl)
  MsgBox "R"
End Sub
```

```
'Callback for CustomBtn1 onAction
Sub MacroCustomButton(control As IRibbonControl)
  MsgBox "Custom Button"
End Sub

'Callback for Btn1 and menuButton1 onAction
Sub Macro1s(control As IRibbonControl)
  MsgBox control.Tag & " was clicked."
End Sub

'Callback for menuButton2 onAction
Sub Macro2s(control As IRibbonControl)
  MsgBox "Marco2s executes."
End Sub

'Callback for menuButton3 onAction
Sub Macro3s(control As IRibbonControl)
  MsgBox "Marco3s executes."
End Sub

'Callback for button1 onAction
Sub Macro1m(control As IRibbonControl)
  MsgBox "Button 1 clicked"
End Sub

'Callback for button2 onAction
Sub Macro2m(control As IRibbonControl)
  MsgBox "Button 2 clicked"
End Sub

'Callback for button3 onAction
Sub Macro3m(control As IRibbonControl)
  MsgBox "Button 3 clicked"
End Sub

'Callback for button4a onAction
Sub Macro4Am(control As IRibbonControl)
  MsgBox "Button 4A clicked"
End Sub

'Callback for button4b onAction
Sub Macro4Bm(control As IRibbonControl)
```

```
  MsgBox "Button 4B clicked"
End Sub

'Callback for button5 onAction
Sub Macro5m(control As IRibbonControl)
  MsgBox "Button 5 clicked"
End Sub

'Callback for gallery1 onAction
Sub SelectedColor(control As IRibbonControl, id As String, _
   index As Integer)
  MsgBox "You selected " & id
End Sub

Sub RemoveUSD(control As IRibbonControl)
'To remove the currency symbols in a range of selected cells
'For example, USD 500.25 becomes 500.25
  Dim workRng As Range
  Dim Item As Range

  On Error Resume Next
  Set workRng = Intersect(Selection, _
              Selection.Cells.SpecialCells _
              (xlCellTypeConstants, xlTextValues))
  If Not workRng Is Nothing Then
    For Each Item In workRng
      If UCase(Left(Item, 3)) = "USD" Then _
        Item = Right(Item, Len(Item) - 3)
    Next Item
  End If
End Sub
```

In the topic Add a dynamic menu to the Ribbon, when a user activates a different sheet, the menu is invalidated explicitly in the Workbook_SheetActivate event handler (in order to rebuild the menu). However, if the dynamic menu is on the Cell context menu, you need not to write VBA code to invalidate the menu. When a user right-clicks a worksheet cell, the dynamic menu rebuilds itself during the process of displaying its contents on the Cell context menu.

The following shows the appearance of the Cell context menu with various types of (custom and built-in) controls:



Note: It is not possible to add controls to the Cell context menu in Excel 2007 and earlier with XML code. Nevertheless, it is still possible to do so with VBA code. However, this book does not discuss the VBA-code method.

A sample program

In the previous topics, you have seen numerous tiny working parts of VBA code in changing the appearance of a workbook. We will go through a sample program that adopts most of the discussed ideas with the following characteristics:

- When the workbook is opened,
  - A particular worksheet (named Sample) is activated.
  - The first three rows are frozen.
  - A particular row (row 50) is scrolled up and becomes the top row in the unfrozen pane.
  - The scroll area for the active sheet is limited to certain range (A4:H100).
  - A custom tab (named Custom) is activated.
  - A drop-down control is dynamically filled with items (which are labelled

as: All Groups, Group 1, Group 2, Group 3, Groups 1 and 2, Groups 1 and 3, and Groups 2 and 3) at runtime.
    - A gallery control is dynamically filled with images at runtime.
- When the user selects a different item from the drop-down control on the Custom tab,
    - Only certain groups of controls on the tab (either all groups, Group 1, Group 2, Group 3, Groups 1 and 2, Groups 1 and 3, or Groups 2 and 3) are displayed accordingly.
    - The status bar displays the currently selected item.
    - If a particular item (namely, Group 2) is selected, a particular worksheet (named Sheet2) is then activated with the following changes to its appearance:
        - The worksheet is displayed in page layout view.
        - The row and column headings are hidden.
        - The gridlines in the worksheet are removed.
        - The formula bar is hidden.
- The Custom tab is visible if the activated sheet is a worksheet.
- A custom control (namely, the G5B1 button on the View tab) is disabled (or enabled) if a particular built-in checkbox (namely, the Formula Bar check box on the View tab) is unticked (or ticked).
- A set of control buttons in different groups on the Custom tab (namely: G1B1 in Group 1, G2B2 in Group 2, G3B3 in Group 3, and G4B3 in Group 4) is enabled if the activated worksheet (named Sheet1) has a particular worksheet-level named range (namely, a range named MyRange). Otherwise, the set of buttons is disabled.
- A custom control (named Remove USD) can be accessed from the Cell context menu

To build the program, execute the following steps:

1. Create a new workbook and save it as a macro-enabled workbook.
2. Right-click a sheet tab and choose Insert to add a chart sheet.
3. Add additional worksheets if needed and rename a worksheet Sample, a worksheet Sheet1, and another worksheet Sheet2.
4. Activate the Sheet1 worksheet, select a range of cells, and in the Name box enter Sheet1!MyRange to name the range a worksheet-level name of MyRange.

   If the Name box is not visible, tick the Formula Bar check box on the View tab.

5. Close the workbook and open it in Custom UI Editor.
6. In Custom UI Editor, click Insert and choose Office 2010 Custom UI Part.
7. Copy and paste the following XML code:

```
<customUI
 xmlns="http://schemas.microsoft.com/office/2009/07/customui"
 onLoad="Initialize">
 <commands>
  <command idMso="ViewFormulaBar"
          onAction="MonitorViewFormulaBar"/>
 </commands>

 <ribbon>
  <tabs>
   <tab id="CustomTab" label="Custom"
        insertBeforeMso="TabView"
        getVisible="getVisibleCustomTab">
    <group id="GroupGallerye" label="Gallery" >
     <gallery id="gallery1"
              label="Photos"
              imageMso="Camera"
              size="large"
              itemWidth="100"
              itemHeight="100"
              onAction="SelectedPhoto"
              getItemCount="getGalleryItemCount"
              getItemImage="getGalleryItemImage"/>
    </group>

    <group id="GroupModule" label="Modules" >
     <dropDown id="dropDown1"
               label="Module:"
               sizeString="xxxxxxxxxxxxxxxxxx"
               onAction="SelectedItem"
               getItemCount="getDropDownItemCount"
               getItemLabel="getDropDownItemLabel">
     </dropDown>
    </group>

    <group id="Group1" label="Group 1"
           getVisible="getVisibleGrp">
     <button id="G1B1" label="G1B1" imageMso="Copy"
             onAction="MacroG1B1" getEnabled="getEnabledBs"/>
    </group>

    <group id="Group2" label="Group 2"
           getVisible="getVisibleGrp" >
     <button id="G2B1" label="G2B1" imageMso="Bold"
```

```
            onAction="MacroG2B1" />
        <button id="G2B2" label="G2B2" imageMso="Italic"
            onAction="MacroG2B2" getEnabled="getEnabledBs"/>
      </group>

      <group id="Group3" label="Group 3"
          getVisible="getVisibleGrp" >
        <button id="G3B1" label="G3B1" imageMso="_1"
            onAction="MacroG3B1" />
        <button id="G3B2" label="G3B2" imageMso="_2"
            onAction="MacroG3B2" />
        <button id="G3B3" label="G3B3" imageMso="_3"
            onAction="MacroG3B3" getEnabled="getEnabledBs"/>
      </group>

      <group id="Group4" label="Group 4" >
        <button id="G4B1" label="G4B1" imageMso="A"
            onAction="MacroG4B1" />
        <button id="G4B2" label="G4B2" imageMso="B"
            onAction="MacroG4B2" />
        <button id="G4B3" label="G4B3" imageMso="C"
            onAction="MacroG4B3" getEnabled="getEnabledBs"/>
      </group>

    </tab>

    <tab idMso="TabView" >
      <group id="Group5" label="Group 5"
          insertAfterMso="GroupViewShowHide">
        <button id="G5B1" label="G5B1" imageMso="Camera"
            size="large" onAction="MacroG5B1"
            getEnabled="getEnabledG5B1"/>
      </group>
    </tab>

  </tabs>
 </ribbon>

 <contextMenus>
  <contextMenu idMso="ContextMenuCell">
    <button id="RemoveUSD"
          label="Remove USD"
          insertBeforeMso="Cut"
          onAction="RemoveUSD"
          imageMso="HappyFace"/>
  </contextMenu>
 </contextMenus>
</customUI>
```

8. Click the Validate button on the toolbar to check for errors.
9. Save and close the file.
10. Open the file in Excel.

Click OK to the error messages.

11. Press Alt+11 to activate VBE.
12. Insert a standard VBA module and copy and paste the following VBA code into the module:

```
Public myRibbon As IRibbonUI
Dim ImageCount As Long  'Number of images in the gallery
Dim ImageFilenames() As String    'Filenames of the images
Dim ItemLabels(0 To 6) As String  'Item labels for the drop-down
Dim VisGrpNm1 As String  'Store the names of groups to be visible
Dim VisGrpNm2 As String  'when an item from the drop-down is selected

'Callback for customUI.onLoad
Sub Initialize(ribbon As IRibbonUI)
   Set myRibbon = ribbon

   'Activate the Custom tab
   myRibbon.ActivateTab "CustomTab"
   'Do not place the above line of code in Workbook_Open
   'because myRibbon is still Nothing

   'Prepare the filenames of the images for the gallery
   Call PrepareItemImages

   'Prepare the labels of the items for the drop-down
   Call PrepareItemLabels
End Sub

Private Sub PrepareItemImages()
'To build an array of filenames of the images for the gallery
   Dim Filename As String
   Filename = Dir("C:\Photos\*.jpg")

   'To loop through all the jpg files in the folder and populate
   'the ImageFilenames array with the filenames of the jpg files
   Do While Filename <> ""
      ImageCount = ImageCount + 1
```

```
      ReDim Preserve ImageFilenames(1 To ImageCount)
      ImageFilenames(ImageCount) = Filename
      Filename = Dir
    Loop
    'Dir() returns a zero-length string ("")
    'when no more file in the folder matches
    'the pathname, "C:\Photos\*.jpg"
End Sub

'Callback for gallery1 getItemCount
Sub getGalleryItemCount(control As IRibbonControl, ByRef Count)
'To specify the number of times to call
'the getGalleryItemImage procedure
    Count = ImageCount
End Sub

'Callback for gallery1 getItemImage
Sub getGalleryItemImage(control As IRibbonControl, _
     index As Integer, ByRef Image)
    'Each time this procedure is called, index is increased by one
    Set Image = LoadPicture("C:\Photos\" & ImageFilenames(index + 1))
End Sub

'Callback for gallery1 onAction
Sub SelectedPhoto(control As IRibbonControl, id As String, _
     index As Integer)
    MsgBox "You selected Photo " & index + 1
End Sub

Private Sub PrepareItemLabels()
'To build an array of item labels for the drop-down
    Dim i As Long
    ItemLabels(0) = "All Groups"
    ItemLabels(1) = "Group 1"
    ItemLabels(2) = "Group 2"
    ItemLabels(3) = "Group 3"
    ItemLabels(4) = "Groups 1 and 2"
    ItemLabels(5) = "Groups 1 and 3"
    ItemLabels(6) = "Groups 2 and 3"
End Sub

'Callback for dropDown1 getItemCount
```

```
Sub getDropDownItemCount(control As IRibbonControl, ByRef Count)
'To specify the total number of items in the drop-down control
  Count = 7
End Sub

'Callback for dropDown1 getItemLabel
Sub getDropDownItemLabel(control As IRibbonControl, _
   index As Integer, ByRef ItemLabel)
  'Set the item labels in the drop-down control
  ItemLabel = ItemLabels(index)

  'Alternatively, if the item labels are in stored
  'the range A1:A7 on Sheet1, use the following code:
  'ItemLabel = Worksheets("Sheet1").Cells(index + 1, 1).Value
End Sub

'Callback for dropDown1 onAction
Sub SelectedItem(control As IRibbonControl, id As String, _
   index As Integer)
  'Determine which group(s) to be visible
  VisGrpNm1 = "": VisGrpNm2 = ""
  Select Case index
    Case 0
      VisGrpNm1 = "*"
    Case 1
      VisGrpNm1 = "*1"
    Case 2
      VisGrpNm1 = "*2"
      'Change the appearance of Sheet2 if 3rd item is selected
      Call ChangeSheet2Appearance
    Case 3
      VisGrpNm1 = "*3"
    Case 4
      VisGrpNm1 = "*1"
      VisGrpNm2 = "*2"
    Case 5
      VisGrpNm1 = "*1"
      VisGrpNm2 = "*3"
    Case 6
      VisGrpNm1 = "*2"
      VisGrpNm2 = "*3"
  End Select
```

```
    'Invalidate Group1, Group2, and Group3
    'Once invalidated, getVisibleGrp is executed
    myRibbon.InvalidateControl "Group1"
    myRibbon.InvalidateControl "Group2"
    myRibbon.InvalidateControl "Group3"

    'Update the status bar
    Application.StatusBar = " Module: " & ItemLabels(index)
End Sub

'Callback for Group1 getVisible
Sub getVisibleGrp(control As IRibbonControl, ByRef Enabled)
'Hide and unhide certain groups of 1, 2, and 3
'based on the selected item from the drop-down control
    If control.id Like VisGrpNm1 Or control.id Like VisGrpNm2 Then
        Enabled = True  'Visible
    Else
        Enabled = False 'Hidden
    End If
End Sub

Private Sub ChangeSheet2Appearance()
    Application.ScreenUpdating = False

    Sheets("Sheet2").Activate
    With ActiveWindow
        'To display the active worksheet in page layout view
        .View = xlPageLayoutView

        'To hide the row and column headings
        .DisplayHeadings = False

        'To hide the gridlines
        .DisplayGridlines = False
    End With

    'To hide the formula bar
    Application.DisplayFormulaBar = False

    Application.ScreenUpdating = True
End Sub
```

```
'Callback for G1B1, G2B2, G3B3, and G4B3 getEnabled
Sub getEnabledBs(control As IRibbonControl, ByRef Enabled)
  'The set of G1B1, G2B2, G3B3, and G4B3 buttons is enabled
  'if the active sheet has a range named MyRange
  'In this program, this procedure is called when the Ribbon
  'is invalidated in the Workbook_SheetActivate event handler
  Enabled = RngNameExists(ActiveSheet, "MyRange")
End Sub

Function RngNameExists(ws As Worksheet, RngName As String) As Boolean
'To return whether a named range exists in a worksheet
  Dim rng As Range
  On Error Resume Next
  Set rng = ws.Range(RngName)
  RngNameExists = Err.Number = 0
End Function

'Callback for ViewFormulaBar onAction
Sub MonitorViewFormulaBar(control As IRibbonControl, _
   pressed As Boolean, ByRef cancelDefault)
  cancelDefault = False  'Restore the functionality of the control
  myRibbon.InvalidateControl "G5B1"
End Sub

'Callback for G5B1 getEnabled
Sub getEnabledG5B1(control As IRibbonControl, ByRef Enabled)
  'The GSB1 button is enabled if the formula bar is visible
  Enabled = Application.DisplayFormulaBar
End Sub

'Callback for CustomTab getVisible
Sub getVisibleCustomTab(control As IRibbonControl, _
   ByRef CustomTabVisible)
  CustomTabVisible = TypeName(ActiveSheet) = "Worksheet"
End Sub
```

The Custom tab is visible if the active sheet is a worksheet. In this program, this procedure is called when the Ribbon is invalidated in the Workbook_SheetActivate event handler

```
'Callback for G1B1 onAction
```

```
Sub MacroG1B1(control As IRibbonControl)
  MsgBox "MacroG1B1"
End Sub

'Callback for G2B1 onAction
Sub MacroG2B1(control As IRibbonControl)
  MsgBox "MacroG2B1"
End Sub

'Callback for G2B2 onAction
Sub MacroG2B2(control As IRibbonControl)
  MsgBox "MacroG2B2"
End Sub

'Callback for G3B1 onAction
Sub MacroG3B1(control As IRibbonControl)
  MsgBox "MacroG3B1"
End Sub

'Callback for G3B2 onAction
Sub MacroG3B2(control As IRibbonControl)
  MsgBox "MacroG3B2"
End Sub

'Callback for G3B3 onAction
Sub MacroG3B3(control As IRibbonControl)
  MsgBox "MacroG3B3"
End Sub

'Callback for G4B1 onAction
Sub MacroG4B1(control As IRibbonControl)
  MsgBox "MacroG4B1"
End Sub

'Callback for G4B2 onAction
Sub MacroG4B2(control As IRibbonControl)
  MsgBox "MacroG4B2"
End Sub

'Callback for G4B3 onAction
Sub MacroG4B3(control As IRibbonControl)
  MsgBox "MacroG4B3"
```

```
End Sub

'Callback for G5B1 onAction
Sub MacroG5B1(control As IRibbonControl)
  MsgBox "MacroG5B1"
End Sub

Sub RemoveUSD(control As IRibbonControl)
'To remove the currency symbols in a range of selected cells
'For example, USD 500.25 becomes 500.25
  Dim workRng As Range
  Dim Item As Range

  On Error Resume Next
  Set workRng = Intersect(Selection, _
                Selection.Cells.SpecialCells _
                (xlCellTypeConstants, xlTextValues))
  If Not workRng Is Nothing Then
    For Each Item In workRng
      If UCase(Left(Item, 3)) = "USD" Then _
        Item = Right(Item, Len(Item) - 3)
    Next Item
  End If
End Sub
```

13. Insert the following VBA code into the ThisWorkbook module:

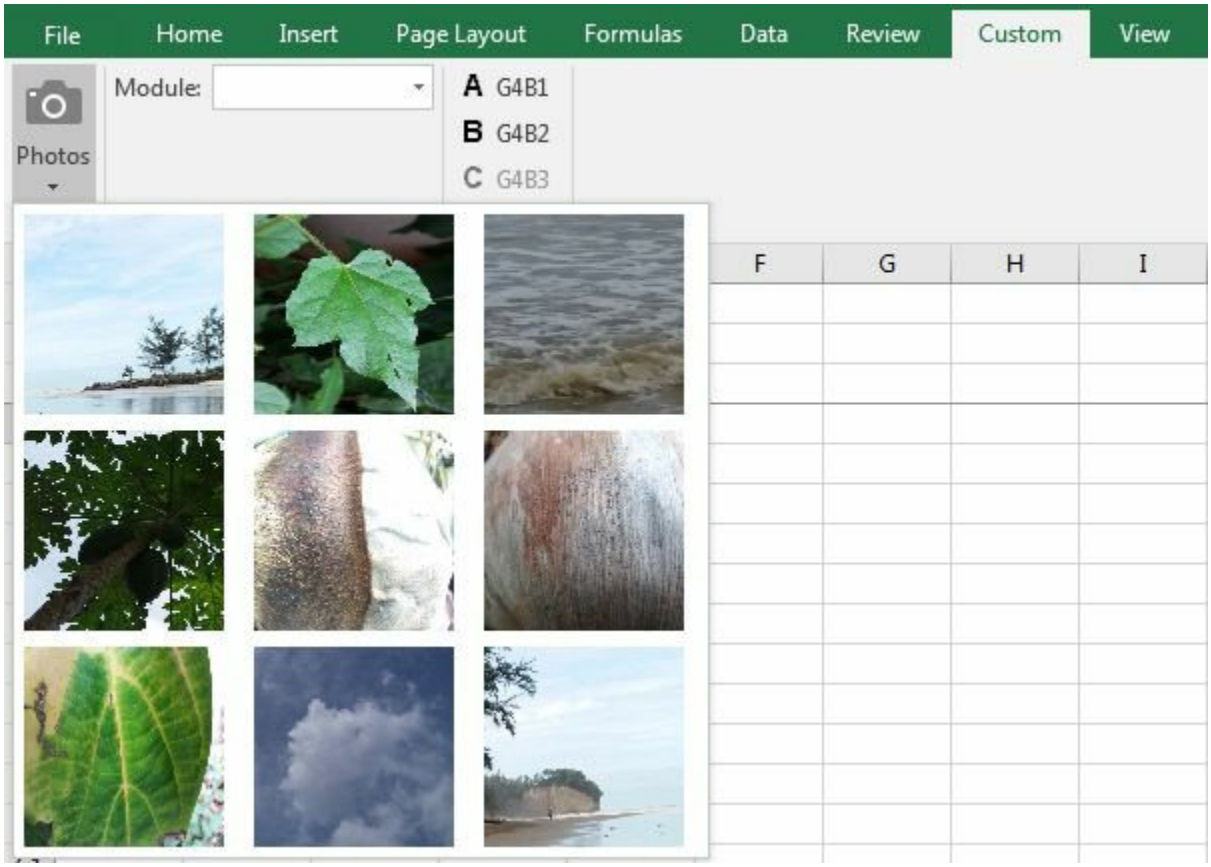
```
Private Sub Workbook_Open()

  With Application
    'Disable Workbook_SheetActivate because
    'myRibbon is still Nothing
    .EnableEvents = False
    .ScreenUpdating = False
  End With

  'Activate a particular worksheet
  Worksheets("Sample").Activate

  'Freeze the first three rows
  With ActiveWindow
    If .View = xlPageLayoutView Then _
      .View = xlNormalView
```

```
    .SplitRow = 3
    .SplitColumn = 0
    .FreezePanes = True

  End With

  'Set row 50 to be the top row in the unfrozen pane
  ActiveWindow.ScrollRow = 50

  'A message for the user
  With Range("A50")
    .Value = "Scroll up to see other info"
    .Font.Bold = True
    .Activate
  End With

  'Set the scroll area for the active sheet
  ActiveSheet.ScrollArea = "A4:H100"  'Limit to the range A4:H100

  With Application
    .EnableEvents = True
    .ScreenUpdating = True
  End With
End Sub
```

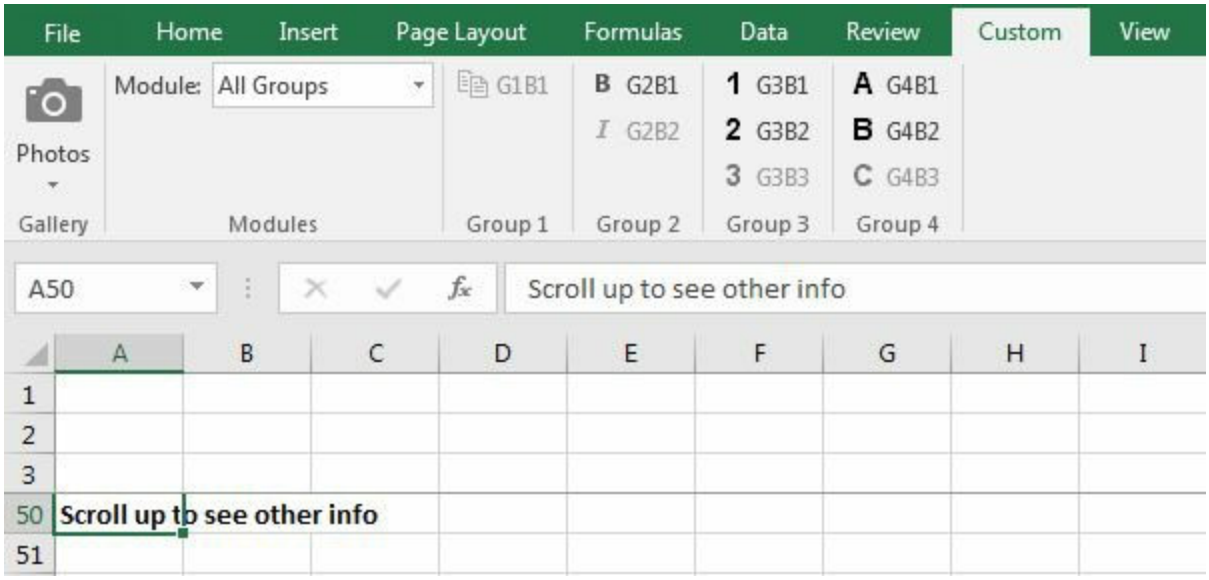
```
Private Sub Workbook_SheetActivate(ByVal Sh As Object)
'To invalidate all controls
  myRibbon.Invalidate
End Sub
```

14. Save, close, and reopen the workbook in Excel.

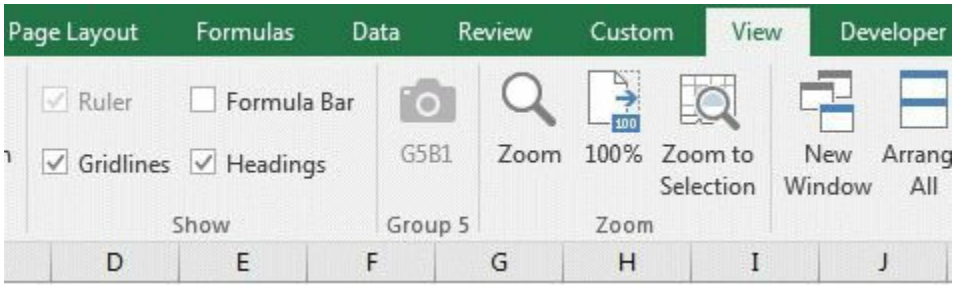
The appearance of the dynamically loaded images in the gallery control on the Custom tab:



The appearance of the custom controls on the Custom tab when the first item from the drop-down control is selected and the active sheet is the Sample worksheet:



The appearance of the greyed out G5B1 button in Group 5 when the Formula Bar check box on the View tab is unselected:



The appearance of the Remove USD button on the Cell context menu:

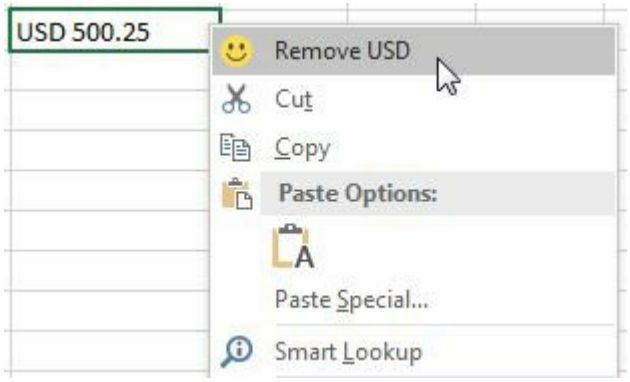


Table of Contents