11/29/2005 - 5:00 pm - 6:30 pm Room:Swan 4 (Swan)

AUTODESK UNIVERSITY 2005
Walt Disney World Swan and Dolphin Resort
Orlando, Florida

Integration of AutoCAD® VBA with Microsoft Excel

dave espinosa-aguilar - Toxic Frog Multimedia

CP22-3 For years AutoCAD users have been trying to learn how to integrate the power of Microsoft Excel spreadsheets with AutoCAD drawings. This course introduces intermediate and advanced AutoCAD users to the programming concepts required to link and exchange data between these two products through AutoCAD's Visual Basic for Applications. Two sample applications will be discussed in depth which demonstrate how to survey an AutoCAD drawing and report the results in a spreadsheet and how to cost estimate a drawing based on an external spreadsheet table of dollar values.

About the Speaker:

As a consultant in CAD and multimedia for 18 years, dave trains professionals in architectural and engineering firms on the general use, customization, and advanced programming of Autodesk design and visualization software. He has authored facilities management applications for several Fortune 500 companies using ObjectARX, VBA, and AutoLISP technologies. dave also creates graphics applications and animations for Toxic Frog Multimedia and has coauthored several books including NRP's Inside 3D Studio MAX series. dave served on the Board of Directors for AUGI for 6 years, including serving as president in 1996.

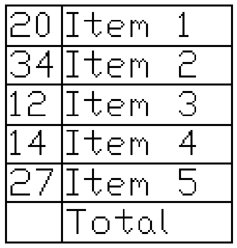
dave2@toxicfrogmultimedia.com

Hi, my name is dave espinosa-aguilar, and I've been using and training folks on AutoCAD since the early 1980's. I work with about 125 companies on average a year doing training and programming custom applications for them, and I get to see how a lot of people throughout the country use the program in new and creative ways. It's an interesting fact that most AutoCAD users have Microsoft Office installed on their systems these days, and applications like Microsoft Excel lend tremendous resources to extending the capabilities of AutoCAD, especially now that AutoCAD's VBA interface is "fully cooked."

This class is not an introduction to AutoCAD, Excel or VBA. Several classes at this year's Autodesk University treat the subject of introductory VBA programming in AutoCAD, and this course makes no attempt to cover that material in addition to discussing all the concepts involved in integrating AutoCAD and Excel. The sole purpose of this course is to build on existing knowledge of these applications and interfaces so that AutoCAD can exploit Microsoft Office spreadsheet functionality. These notes in previous years have become an AutoCAD VBA "bible" on this subject.

What does Microsoft Excel Offer an AutoCAD User?

If you've ever done a manual count of blocks or block attribute values, if you've ever had to generate a manual schedule of drawing components or a report of entitiy counts in a CAD drawing, you might already be able to appreciate the benefit of a spreadsheet application which can automate this process for you. Through AutoCAD's programming interfaces, tallies of AutoCAD entities and their values can be generated quickly to save time and make valuable information available to other applications.



20	Item 1
34	Item 2
12	Item 3
14	Item 4
27	Item 5
	Total

Consider the typical strengths of a spreadsheet: the ability to organize data, the ability to process data, the ability to query data and treat ranges of data, the ability to work with data types--- and best of all, pass that information in a grid format to other applications. This class is all about how to bring the power of a spreadsheet application right into AutoCAD drawings, how to export AutoCAD information to spreadhseets, and how to use spreadsheet data to evaluate AutoCAD drawings. Let's examine typical limitations of AutoCAD TEXT entities and how a spreadsheet application can circumvent them. Consider the figure to the left. Suppose this small schedule was drawn strictly using AutoCAD entities (TEXT and LINE entities) to report the counts for 5 different block types named ITEM1 through ITEM5.

- These numbers would have to be initially generated by manual counts since there are no functions in vanilla AutoCAD which will put together custom schedules like this.
- If new blocks are added to the drawing, the schedule would have to be updated manually since there is no relationship between the TEXT entities and the blocks.
- To get a total for all items, you would have to do a manual calculation (you can't add TEXT entities in AutoCAD). Maybe adding up five numbers isn't a problem, but imagine adding up 20, or 100, or 1000 items. Imagine trying to keep track of hundreds of item types as any design size increases.

In short, by using AutoCAD VBA with a spreadsheet application like Excel, not only can these counts be automated as new items get added to the drawing, but the counts are fast, accurate, and a total can be automatically generated. Add to these benefits the fact that it is easier to stretch the size of an OLE-pasted grid object than it is to redefine text styles, the fact that it is easier to insert a new row or new column to a grid object than it is to move existing text entities around in the drawing to accomodate new rows and columns, and add the fact that this schedule information can be shared with other applications like word processors and databases--- and you have some pretty convincing arguments for examining the integration of Microsoft Excel with AutoCAD.

AutoCAD VBA makes this all possible so that you can easily generate bill of materials, schedules, reports, and share all this information with other Windows applications. Not only can information from AutoCAD be tallied, but it can also be numerically processed (sums, averages, minimums maximums, etc) And since Excel is OLE-capable, you can drop-and-drag your integrated worksheets right back into AutoCAD as stretchable, linked (aka re-editable), evenly-spaced plot-able entities unto themselves.

Essential Preparations for Connecting AutoCAD with Excel

To take advantage of AutoCAD VBA, we obviously need to launch AutoCAD and use a command like VBAMAN or VBAIDE to bring up the VBA interface within AutoCAD. This class assumes you have Excel loaded on your system. By default, AutoCAD VBA does not recognize Excel objects, so you need to use the VBA Tools/References pulldown menu function and select **Microsoft Excel Object Library** to use Excel Objects and their properties in your VBA code. Public variables and non-private functions are used in these code examples for this class to keep things simple.

<u>AutoCAD VBA/Excel Coding Concepts</u>
The following General Declarations for treating the Excel Application and workbooks and worksheets within it are assumed for the code examples provided in this course:

```
Public excelApp As Object
Public wbkObj As Object
Public shtObj As Object
```

When working with any Microsoft Office application such as Excel, your VBA code must first establish a link with it. You cannot assume that a session of Excel is already running, so the code has to accomplish several things: it has to detect if Excel is already running, if Excel is already running then it needs to establish a link with it, and if Excel isn't already running, then a session of Excel needs to be launched. The GetObject function assumes that Excel is already running. If it isn't, an error (non-zero condition) is generated and we use the CreateObject function to launch Excel. If Excel cannot be launched for some reason, then another error is generated and we notify the user. If Excel can be launched, we add a workbook (which has 3 default sheets to it) and set the current worksheet to the first worksheet. A function to rename the first sheet is also provided. The current workbook and worksheet pointers are also set.

We start then by adding a Microsoft Excel Object Library reference, by adding Commandbutton1 to a UserForm1 and assigning it the code below, and by making sure the above public declarations are placed in the General declarations area:

```
Public excelApp As Object
Public wbkObj As Object
Public shtObj As Object

Sub CommandButton1_Click()
  On Error Resume Next 'make sure Excel reference is set!!
  Set excelApp = GetObject(, "Excel.Application")
  If Err <> 0 Then
    Err.Clear
    Set excelApp = CreateObject("Excel.Application")
    If Err <> 0 Then
      MsgBox "Could not start Excel", vbExclamation
      End
    End If
  End If
  excelApp.Visible = True
  Set wbkObj = excelApp.Workbooks.Add
  Set shtObj = excelApp.Worksheets(1)
End Sub
```

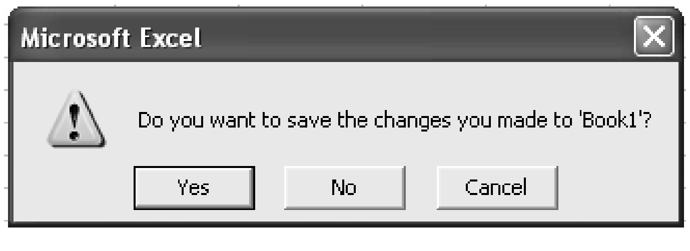
For ease, all spreadsheets referenced in the code examples will be kept in the root directory. If you wish to use an existing spreadsheet instead of creating a new one, you can swap the CommandButton1 line

```
  Set wbkObj = excelApp.Workbooks.Add
```

With

```
  Set wbkobj = Workbooks.Open(filename:="c:\filename.xls")
```

If the above code is run. Pressing this button will either launch Excel and link your VBA application to it, or link your VBA application to an existing session of Excel.


Microsoft Excel

Do you want to save the changes you made to 'Book1'?

The code below closes Excel. Depending on the version of Excel you're using and how it is pre-configured, a dialog may ask if you want to save changes to the current workbook when you quit the application.

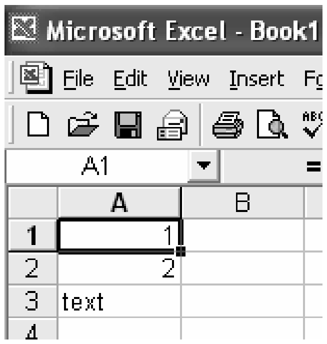
Take note, CommandButton2 closes Excel but it **does not** end your VBA application!
For now, create CommandButton2 and add the code below to it. Run the application, use CommandButton1 to launch Excel, then use CommandButton2 to shut Excel down.

```
Sub CommandButton2_Click()
  excelApp.Quit 'Unload me also an option.
End Sub
```

To send a value from your VBA application to a particular cell in the linked Excel application, first establish which worksheet you're sending information too and then then set the worksheet cell value by specifying the row and column using integer values. Be sure to use proper variable types when setting the values of cells. In the code below, the first worksheet is specificed, and an integer, a real number and a string as passed to it.

If you pull your values from textboxes and other object sources in your forms, this is especially important. For now, create CommandButton3, assign the code below to it, run the application, use CommandButton1 to launch Excel, then use CommandButton3 to pass the values to the spreadsheet.

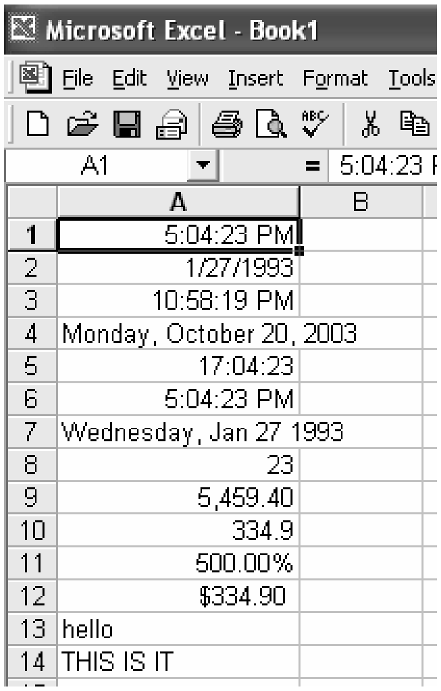
```
Sub CommandButton3_Click()
'don't forget to use CommandButton1 first 'before you use this
button
Set shtObj = wbkObj.Worksheets(1)
  ival% = 1
  rval& = 1.5
  sval$ = "text"
  shtObj.Cells(1, 1).Value = ival%
  shtObj.Cells(2, 1).Value = rval&
  shtObj.Cells(3, 1).Value = sval$
End Sub
```



	A	B
1	1	
2	2	
3	text	
4		

Notice that the real value 1.5 passed to cell A2 in the spreadsheet reports as a value of 2. This is not an error. You can set the format of any cell in the spreadsheet by formatting the data you are passing to it from your VBA application first.
Let's prove it. Create a new CommandButton4 and assign the code below to it. The code shows how various types of formatted values from your VBA application appear in the Excel spreadsheet once they are passed including system dates and times, system and date formats, decimal precisions, and capitalization functions. Take note the how pre-configured spreadsheets are setup will affect any data you pass to it unless you specify the formatting of every cell you're sending information to.

Reminder: when you test CommandButton4, don't forget to close all spreadsheets, use CommandButton1 to launch Excel and thenuse CommandButton4 to test the formatting of values you're sending. Also remember to use ALT+Tab to move back and forth between Excel and AutoCAD as you develop code for your applications. You may also need to expand the width of any cells you are passing values to or you'll get the infamous ####### where a value should be. Use the VBA editor to quit the VBA application if needed each time you test it.

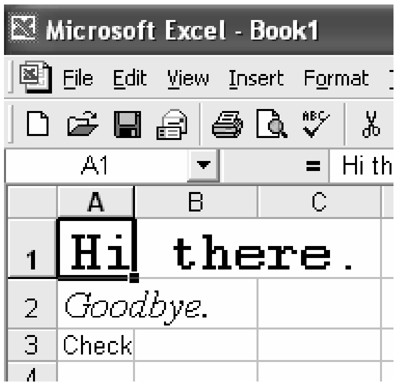


	A	B
1	5:04:23 PM	
2	1/27/1993	
3	10:58:19 PM	
4	Monday, October 20, 2003	
5	17:04:23	
6	5:04:23 PM	
7	Wednesday, Jan 27 1993	
8	23	
9	5,459.40	
10	334.9	
11	500.00%	
12	$334.90	
13	hello	
14	THIS IS IT	

```
Sub CommandButton4_Click()
'don't forget to use CommandButton1 first before you use
this button
  Set shtObj = wbkObj.Worksheets(1)
  Dim MyTime, MyDate, MyStr
  MyTime = #5:04:23 PM#
  shtObj.Cells(1, 1).Value = MyTime
  MyDate = #1/27/93#
  shtObj.Cells(2, 1).Value = MyDate
  MyStr = Format(Time, "Long Time")
  shtObj.Cells(3, 1).Value = MyStr
  MyStr = Format(Date, "Long Date")
  shtObj.Cells(4, 1).Value = MyStr
  MyStr = Format(MyTime, "h:m:s")
  shtObj.Cells(5, 1).Value = MyStr
  MyStr = Format(MyTime, "hh:mm:ss AMPM")
  shtObj.Cells(6, 1).Value = MyStr
  MyStr = Format(MyDate, "dddd, mmm d yyyy")
  shtObj.Cells(7, 1).Value = MyStr
  MyStr = Format(23)
  shtObj.Cells(8, 1).Value = MyStr
  MyStr = Format(5459.4, "##,##0.00")
  shtObj.Cells(9, 1).Value = MyStr
  MyStr = Format(334.9, "###0.00")
  shtObj.Cells(10, 1).Value = MyStr
  MyStr = Format(5, "0.00%")
  shtObj.Cells(11, 1).Value = MyStr
  MyStr = Format(334.9, "$###.##")
  shtObj.Cells(12, 1).Value = MyStr
  MyStr = Format("HELLO", "<")
  shtObj.Cells(13, 1).Value = MyStr
  MyStr = Format("This is it", ">")
  shtObj.Cells(14, 1).Value = MyStr
End Sub
```

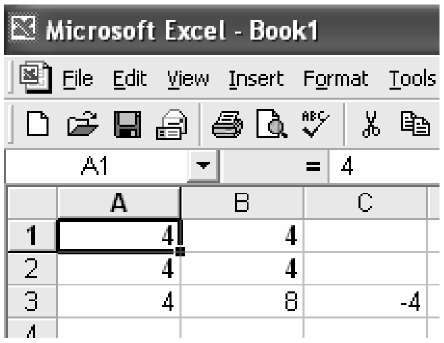
Font name, size, bold or italics, and columnwidths can also be controlled from your application. Create CommandButton5, assign the code below to it and run the application to test it.

```
Sub CommandButton5_Click()
'don't forget to use CommandButton1 first before you use
this button
  Set shtObj = wbkObj.Worksheets(1)
  shtObj.Cells(1, 1) = "Hi there."
  shtObj.Cells(1, 1).ColumnWidth = 30
  shtObj.Cells(1, 1).Font.Bold = True
  shtObj.Cells(1, 1).Font.Name = "Courier"
  shtObj.Cells(1, 1).Font.Size = 20
  shtObj.Cells(1, 1).Justify
  shtObj.Cells(2, 1) = "Goodbye."
  shtObj.Cells(2, 1).ColumnWidth = 20
  shtObj.Cells(2, 1).Font.Italic = True
  shtObj.Cells(2, 1).Font.Name = "Times Roman"
  shtObj.Cells(2, 1).Font.Size = 15
  shtObj.Cells(2, 1).Justify
  shtObj.Cells(3, 1) = "Check"
  shtObj.Cells(3, 1).ColumnWidth = 5
  shtObj.Cells(3, 1).Justify
End Sub
```



You can work with ranges and formulas in Excel from your VBA application as well. Create CommandButton6, assign the code below to it and run the application to test it:

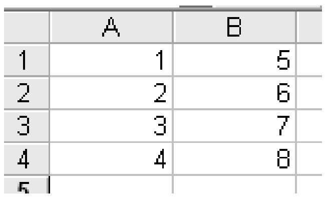
```
Sub CommandButton6_Click()
'don't forget to use CommandButton1 first before you use
this button
  Set shtObj = wbkObj.Worksheets(1)
  With shtObj.Range("A1:B2")
    .Font.Name = "Arial"
    .Font.Size = 10
    .Font.Bold = True
    .Value = 4
  End With
  Range("A3").Formula = "@average(A1:A2)"
  Range("B3").Formula = "@sum(B1:B2)"
  Range("C3").Formula = "= A3-B3"
End Sub
```



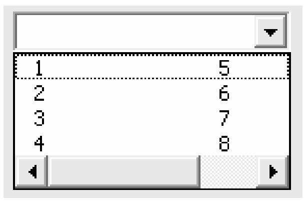
	A	B	C
1	4	4	
2	4	4	
3	4	8	-4

You can pull information from a cell into an object in your form. The code below shows how a value in a cell can be brought into a textbox object, or how values found in a range of cells (A1:B4) can be brought into a listbox or combobox object (notice the use of the ColumnCount property is required for proper display of cells involved in the imported range). Create ComboBox1. Create CommandButton7 and assign the code below to it. Run the application to verify it:

```
Sub CommandButton7_Click()
'don't forget to use CommandButton1 first before you use this button
  Set shtObj = wbkObj.Worksheets(1)
  shtObj.Cells(1, 1) = 1
  shtObj.Cells(2, 1) = 2
  shtObj.Cells(3, 1) = 3
  shtObj.Cells(4, 1) = 4
  shtObj.Cells(1, 2) = 5
  shtObj.Cells(2, 2) = 6
  shtObj.Cells(3, 2) = 7
  shtObj.Cells(4, 2) = 8
  ComboBox1.ColumnCount = 2
  ComboBox1.List = Worksheets("Sheet1").Range("A1:B4").Value
End Sub
```

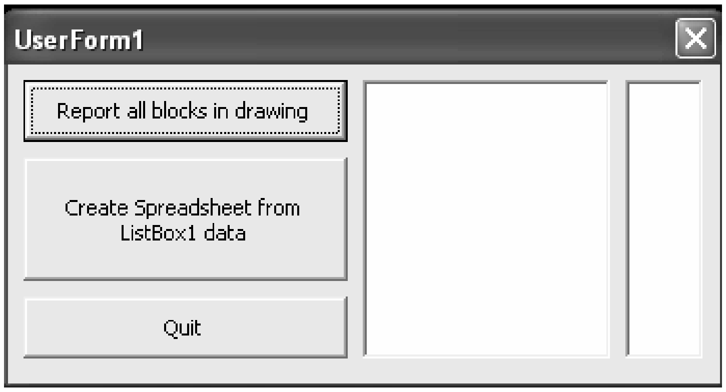


	A	B
1	1	5
2	2	6
3	3	7
4	4	8



```
1    5
2    6
3    7
4    8
```

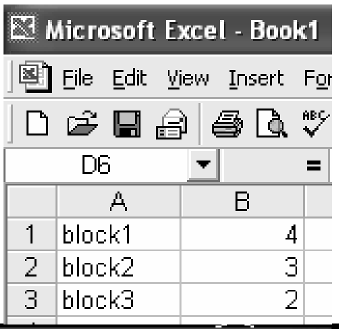
AutoCAD VBA/Excel Sample Application #1: Block Counter/Schedule Generator
The code below can be pasted directly into the General Declarations area of a new project. You can create the GUI shown at the right by inserting UserForm1 with CommandButton1, CommandButton2 and CommandButton3, ListBox1 and Listbox2 objects added. CommandButton1 cycles through all blocks found in the drawing block collection and adds their names (except for those beginning with an asterisk) to Listbox1. It then cycles through all Listbox1 names, totaling all entities in Modelspace which are both blocks and which are named the current Listbox1 value, and passes that total to Listbox2. CommandButton2 sends the Listbox data to a new spreadsheet. CommandButton3 exits the VBA application. To test this application, create several blocks with different names and insert a lot of them into the drawing. The more you add, the more impressive the totaling and reporting will be. After running this application, you can then copy/paste the resulting spreadsheet cells back into AutoCAD for a report of Block counts in the drawing.


UserForm1
Report all blocks in drawing
Create Spreadsheet from ListBox1 data
Quit

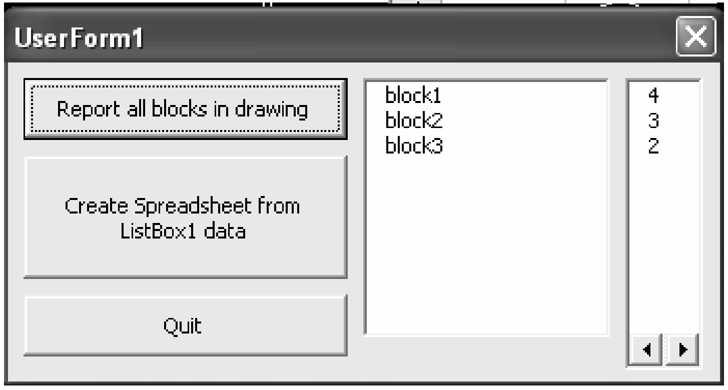
```
Public excelApp As Object
Public wkbObj As Object
Public shtObj As Object

Sub CommandButton1_Click()
  Dim i, j, btot As Integer
  Dim bnam As String
  Dim ent As Object
  btot = ThisDrawing.Blocks.Count
  For i = 0 To btot - 1
    bnam = ThisDrawing.Blocks.Item(i).Name
    If Not Mid$(bnam, 1, 1) = "*" Then ListBox1.AddItem bnam
  Next i
  For i = 0 To ListBox1.ListCount - 1
    bnam = ListBox1.List(i): btot = 0
    For j = 0 To ThisDrawing.ModelSpace.Count - 1
      Set ent = ThisDrawing.ModelSpace.Item(j)
      If ent.EntityType = acBlockReference And ent.Name = bnam Then btot = btot + 1
    Next j
    ListBox2.AddItem btot
  Next i
End Sub
Sub CommandButton2_Click()
  On Error Resume Next
  Set excelApp = GetObject(, "Excel.Application")
  If Err <> 0 Then
    Err.Clear
    Set excelApp = CreateObject("Excel.Application")
    If Err <> 0 Then
      MsgBox "Could not start Excel!", vbExclamation
      End
    End If
  End If
  excelApp.Visible = True
  Set wkbObj = excelApp.Workbooks.Add
  Set shtObj = wkbObj.Worksheets(1)
  shtObj.Name = "Block Count"
  Dim i, j, btot As Integer
  Dim bnam As String
  j = 1
  For i = 0 To ListBox1.ListCount - 1
    bnam = ListBox1.List(i)
    btot = ListBox2.List(i)
    shtObj.Cells(j, 1).Value = bnam
    shtObj.Cells(j, 2).Value = btot
    j = j + 1
  Next i
End Sub

Sub CommandButton3_Click()
```



	A	B
1	block1	4
2	block2	3
3	block3	2


UserForm1
Report all blocks in drawing
Create Spreadsheet from ListBox1 data
Quit

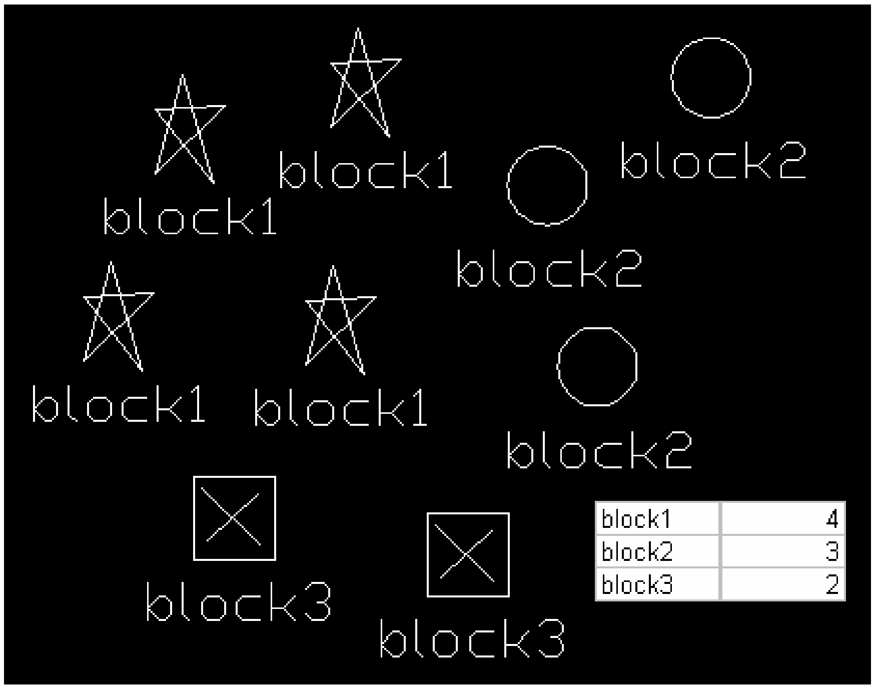
block1	4
block2	3
block3	2

```
  End ‘or ExcelApp.Quit Unload Me
End Sub
```

The images on the right show how the dialog reports the blocks and their totals found in the drawing found, how this information was passed to a spreadsheet and how that range of cells was pasted back into the drawing (shown below). With further VBA coding, you can have complete control over which blocks get reported (such as those with particular string wildcard values for their names, blocks with certain attribute tag names or attribute values that fall within numeric ranges,etc).

You can also calculate averages, minumum and maximum counts of blocks and values of block attributes in Excel, once the values are passed to it or once you've sent formulas to certain cells which do it for you. You could even update entities in AutoCAD according to cell values.

Once you've generated a spreadsheet report of the block counts, you can copy and paste cell ranges back into AutoCAD as a schedule report entity which can be stretched, replaced with new information, plotted/printed and even manually updated.



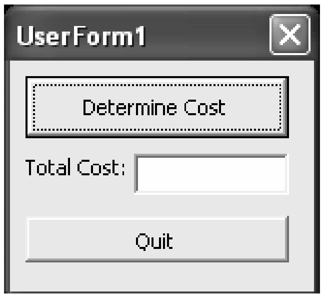
block1	4
block2	3
block3	2

AutoCAD VBA/Excel Sample
Application #2: Bill of Materials
The first sample application essentially shows how AutoCAD information can be passed to a spreadsheet so that a grid-like report (schedule) can be pasted back into AutoCAD. In this second application, entities which define areas are placed on layers with an associated cost per square inch. By pressing a single button, the user can know the overall cost of a design. For this application to work, as with many application, some standards have to be assumed:

On the spreadsheet side of things, an existing spreadsheet with layer names and costs already entered is assumed: column A cells are assigned string values reflecting layer names, and column B cells are assigned real (dollar) values reflecting cost of material per square inch for each layer in Column A. For example, The value of cell A1 might be "MATERIAL1" and the value of cell B1 might be $1.50. If a single rectangular 1x1 closed polyline were placed on layer “MATERIAL1” and this was the only entity in the drawing, running application would report a total cost of the drawing being $1.50. If this polyline were copied on the same layer, running this application would report a total cost of the drawing being $3.00.

On the AutoCAD side of things, only polylines in this sample application to define areas on layers whose names match values in column A of the spreadsheet. If a new layer is defined in column A with a new cost of material in column B, area-defining entities can now be created on that new layer in AutoCAD and be added to the overall cost when the routine is used. In other words, the application will read all values in Column A until it encounters a blank cell. By adding or deleting Column A values, the desired cost of the drawing evaluation can be controlled external to AutoCAD.

This application could easily be modified to associate block names with associated costs (instead of area-defining polyline entities) to drive the numbers as well.
Create the deceptively simple GUI shown to the right by creating UserForm1 with CommandButton1, CommandButton2, Label1 and TextBox1. Paste the code below into the General Declarations area. Take note that in this sample application, CommandButton2 is assigned the code statement:



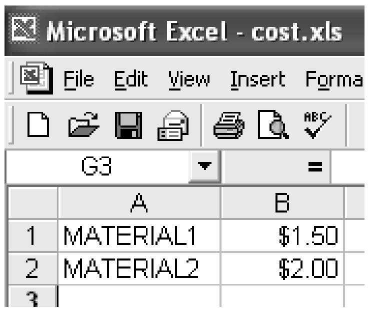
```
Unload Me
```

Instead of the End statement. Unload Me not only ends the currently loaded application but it also frees up memory. This is a good practice to get into for exiting your AutoCAD VBA programs.

When the application is launched and CommandButton1 is used, the routine opens an existing spreadsheet which abides by the above standards, loops through text values found in column A until it finds a blank cell and then exits. For each non-blank cell it finds in Column A, it pulls the price value in column B for that row. It then searches the drawing database for polylines (EntityType 24) on the column A layer name and multiplies each polyline entity area property with the column B value, adding this value to a total value for that layer. Each layer total is then added to a grand total. In this way, no matter how many polylines of any shape are drawn on layers represented with cost values in the spreadsheet, the press of one button will yield the total cost for all areas according to their layer pricings.

Now that's some serious estimating power!

Important reminders before you run this application: be sure your Microsoft Excel Object Library reference is created for this project, make sure you have a pre-existing spreadsheet named and pathed according to the code you're assigning to CommandButton1, and make sure you have properly created enclosed polyline boundaries on standardized layers which correspond to the spreadsheet Column A values before you run the application. Create 1x1 PLINEs for starters to verify correct cost estimates.

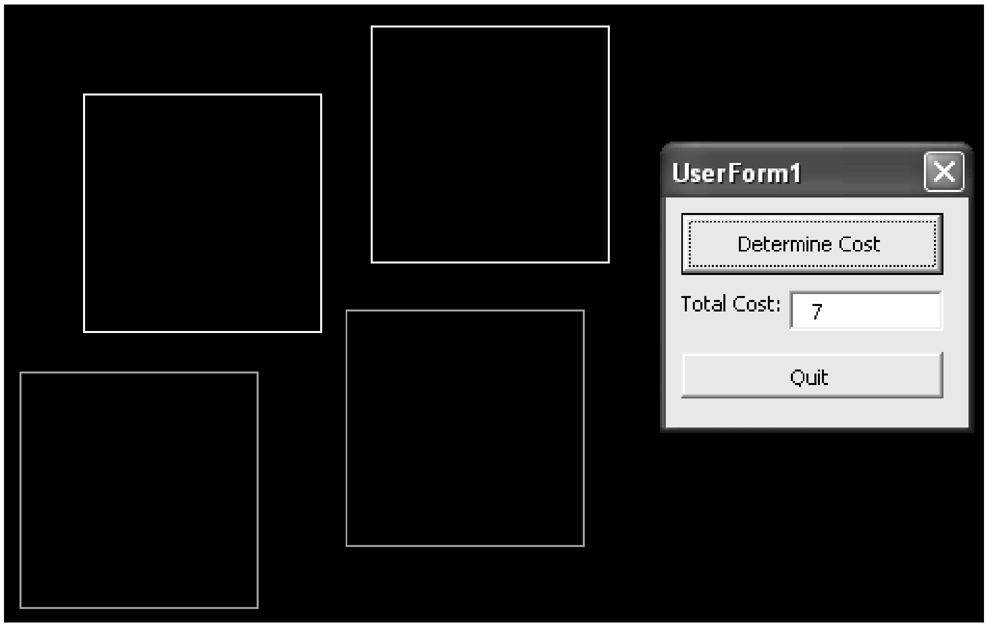


	A	B
1	MATERIAL1	$1.50
2	MATERIAL2	$2.00
3		

```
Public excelApp As Object
Public wbkObj As Object
Public shtObj As Object

Sub CommandButton1_Click()
  On Error Resume Next ' make sure pline zones on layers are created!!
  Set excelApp = CreateObject("Excel.Application")
  If Err <> 0 Then
    Err.Clear
    MsgBox "Could not start Excel!", vbExclamation
    End
  End If
  excelApp.Visible = True
  Set wbkObj = Workbooks.Open(filename:="c:\daveea\cp13-1cost.xls") ' create it!!
  Set shtObj = wbkObj.Worksheets("Sheet1")
  Dim i, j As Integer
  Dim pnum, anum, atot As Double
  Dim lnam, enam As String
  Dim ent As Object
  i = 1: anum1 = 0#: anum2 = 0#: atot = 0#
  lnam = shtObj.Cells(i, 1).Value
  pnum = shtObj.Cells(i, 2).Value
  Do While Not (lnam = "")
    For j = 0 To ThisDrawing.ModelSpace.Count - 1
      Set ent = ThisDrawing.ModelSpace.Item(j)
      If ent.EntityType = 24 And ent.Layer = lnam Then
        anum1 = ent.Area
        anum2 = anum2 + anum1
      End If
    Next j
    atot = atot + (anum2 * pnum)
    anum1 = 0#: anum2 = 0#
    i = i + 1
    lnam =
shtObj.Cells(i,
1).Value
    pnum =
shtObj.Cells(i,
2).Value
  Loop
  excelApp.Quit
  TextBox1.Text =
Format(atot,
"$###.##")
End Sub

Sub
CommandButton2_Click()
  Unload Me
End Sub
```



Things to note with this sample application:

- Excel runs transparently in the background, putting the columnated data to use for AutoCAD evaluation. You don't have to show the user Excel to be able to put Excel functionality to use for AutoCAD applications.

- The polyline boundaries can overlap and still report correct (area)x(cost) totals. This has highly practical applications for any type of application that uses surface areas (or 2D-representations of volumetric data. It could represent floor space or cubic space in a room, HVAC duct volumes or plumbing GPS values, wallspace calcs or concrete to be poured for foundations, landscaping zones with assigned vegetation, earthwork volumes of ore bodies or soil types. The analogies go on forever.

- A person who knows costing but doesn't know AutoCAD that well can customize a costing by simply adding or deleting Row values and running the application. You can setup different spreadsheets for different types of analysis!!!

(2) 1x1 polylineal rectangles are drawn on layers MATERIAL1 and MATERIAL2. The COST.XLS spreadsheet assigns the dollar value $1.50 for MATERIAL and $2.00 for MATERIAL2. Therefore, the total cost of the drawing at this point is 2x1.50 + 2x2.00 = 7.00 … yeah, it works.

Other Things to Keep in Mind about Working with Excel
The Task Manager (ALT+CTRL+DEL) can be very helpful in assessing if you have sessions of Excel left open while you're programming in VBA. A common error many beginning programmers make when linking to Microsoft Office applications is to forget to close those applications in addition to exiting your own routines. If you don't use excelAPP.Quit or Unload Me (excelApp is the object assigned to the Excel Application in the above code) for example, when finishing the command button code, you will leave a session of Excel running in the background even though your VBA routine has exited.

There may be times when you want to leave Excel running after your VBA routine is finished. But be aware that running the VBA routine again and again will launch session after session of Excel. If you're not sure how many sessions of Excel are open, or if Excel forbids you to edit a spreadsheet, use the Task Manager to see if Excel sessions are running and use it to close down an Excel session if Excel itself won't let you. Also, remember that new workbooks typically bring up three sheets by default. It is a good practice to either create your own sheets before pointing to them, or renaming existing sheets so that you can specify exactly what sheet you're passing and pulling information to and from. It is not a good idea to assume that everyone's Excel sessions will bring up the same default workbooks and worksheets you have on your system: many users rely on different default spreadsheet templates.