by Keith R Pleas

Control Excel 5.0 With OLE Automation

Microsoft Excel version 5.0 provides developers with a rich library of programmable objects. In previous versions of Excel, much of this functionality was there but unexposed to applications outside Excel. With Excel 5.0, Microsoft has implemented an OLE Automation interface that makes it possible to create and manipulate objects inside Excel from other applications. This feature is especially attractive to Visual Basic programmers because it allows access to Excel's library of objects, which is probably second to none for building spreadsheet applications.

Perhaps you're wondering whether the OLE Automation interface is capable, and if you'll run into any of the performance issues that plagued developers who tried in the past to integrate applications using DDE. Well, the interface is the method by which VBA is integrated into its host application and the answer is that the OLE Automation interface is truly capable—it's a stronger, faster, and more powerful interface than DDE ever was.

THE OLE AUTOMATION INTERFACE IS STRONGER, FASTER, AND MORE POWERFUL THAN DDE EVER WAS.

EXCEL REGISTRATION

You may already be familiar with the Windows registration database where OLE objects are registered. This file, which you can view using the REGEDIT.EXE utility included with Windows, stores a variety of information of interest to programmers wishing to integrate applications using OLE and OLE Automation (while the list is useful, it is also necessary to understand how objects are contained by and accessed through other objects—see Figure 1).

While the Windows registration database will be a topic by itself in a future column, there are a couple of points that you should be aware of now. First, under 16-bit Windows the registration database is limited to 64K for keys and 64K for values. Excel 5.0 registers 130 different objects for exposure to OLE Automation and Microsoft Graph 5.0 (shipped with Excel 5.0) registers another 38; together they consume 20K of that heap. These entries are needed to do type checking for early-bound method invocations performed in the VB (and VBA) design environment.

Second, there are a couple of possible syntaxes for referencing Excel in CreateObject and GetObject functions: "Excel.Application.5" will always give you Excel 5; "Excel.Application" will always give you the default VBA-aware installation of Excel on the system, which would usually be the most recent version.

This will only be useful after Excel version 6.0 ships, but should be kept in mind for forward compatibility.

USING AN EXCEL SPREADSHEET

You are probably already familiar with Excel's spreadsheets. Figure 2 shows the AMOR-TIZE.XLS worksheet included with the Excel samples. The important variables (and their cell locations and predefined range names) in calculating an amortization table are:

C7	=	Loan_amount
C8	=	Annual_interest_rate
C9	=	Term_in_years
C10	=	Payments_per_year
C14	=	Calculated_payment

Here's a code sample that opens AMORTIZE.XLS, plugs in values for the variables, and pops up the calculated payment in a message box (it is also straightforward to bring back the whole table, if desired):

```
Dim APPXL As object
Dim XL As object
Dim ws As object

Set APPXL = GetObject(,"Excel.Application")
Set XL = APPXL.Application
XL.WorkBooks.Open "F:\TEMP\AMORTIZE.XLS"
Set ws = XL.ActiveSheet
ws.Range("Loan_amount").Value = 100000
ws.Range("Annual_interest_rate").Value = .075
ws.Range("Term_in_years").Value = 30
ws.Range("Payments_per_year").Value = 12
MsgBox Format$(ws.Range("Calculated_payment")._
        Value, "currency"), , "Payment"
XL.Workbooks(1).[Close] (False)

Set ws = Nothing
Set XL = Nothing
Set APPXL = Nothing
```

There's one line in this sample I'd like to draw your attention to:

```
XL.Workbooks(1).[Close] (False)
```

Even though the workbook was opened via "WorkBooks.Open," it can't be closed from VB via a corresponding "WorkBooks(1).Close." Trying to do this results in the message "Method not applicable for this object." The problem is that, in VB, the Close method is only applicable to a database. To force VB to pass the Close method on to Excel for evaluation, enclose it in brackets. In cases that require brackets—or example, using the Show, AddItem, and RemoveItem methods—it is often a requirement that the parameters be enclosed in parentheses.

EXCEL DIALOGS

Keith R. Pleas is an independent system consultant. He is the author of the forthcoming book, Implementing DDE & OLE, from Ziff-Davis Press. He can be reached via CompuServe at 72331,2150, or via the Internet at keithp@curlew.wa.com.

If you can build your user interface within Excel, why work with VB at all? Well, the differences between VB and VBA are most apparent in the user interface. In the first release of VBA, there are no forms or controls; likewise, there is no place to plug in the add-on custom controls that are so much a part of VB.

And if you have no controls, it stands to reason you have no control events. Instead, VBA (again, this is only in the first release) programmatically builds the user interface using the host application's tools. Instead of VB forms, you have Excel dialogs that are constructed on dialog sheets and can contain a variety of traditional Windows controls. And you can (sort of) write event code and attach it to the "controls," though the method for doing this is nowhere near as straightforward as it is for VB.

Assuming that you're already familiar with the forms capabilities of VB, here's a list of the major restrictions and limitations of using Excel dialogs for your user interface.

I'm not trying to discourage you from using them, but you might appreciate knowing about the roadblocks before you run into them:

- •All dialogs are application modal.
- •Excel doesn't respond to DDE or OLE messages when a dialog is showing or a menu is dropped down.
- •You are limited to one dialog per sheet (though a workbook can have many).
- •Moving a control in code (must be done before the dialog is shown) actually moves it on the dialog sheet (analogous to VB's design environment).
- •Dialogs don't really go away until the calling subroutine ends (they're just hidden).
- •Only two dialogs can be displayed at the same time; anything more (or if the dialogs are complex) runs out of stack space.
- •Because of the stack space issue, dialogs must be chained together using OnTime events to kick off subsequent macros.
- •The OnTime event involves a tolerance factor. If you miss it for some reason (perhaps your system is busy with another task), you're dead. In addition, you can have only one OnTime event running at any given time. Further, because Windows timers require callbacks (which are unavailable in Excel, much as they are in native VB) you can't use Windows APIs to get additional timer capabilities.
- •When a dialog is showing, you can't move a control on it.
- •Text boxes are limited to 255 characters.
- •Only one macro can be assigned to a control. For example, buttons have only a Click event, and list boxes only a select event.
- •The text box Keypress event occurs *after* the character shows up in the text box; it doesn't tell you what key is pressed.

Are these restrictions something to worry about? Excel developers have been turning out useful applications for years with roughly this same set of limitations. If all you want to do is throw up a couple of list boxes and buttons (something as complex as the Windows File | Open common dialog, for example), you'll probably be OK. But developers used to the VB forms model will be extremely frustrated.

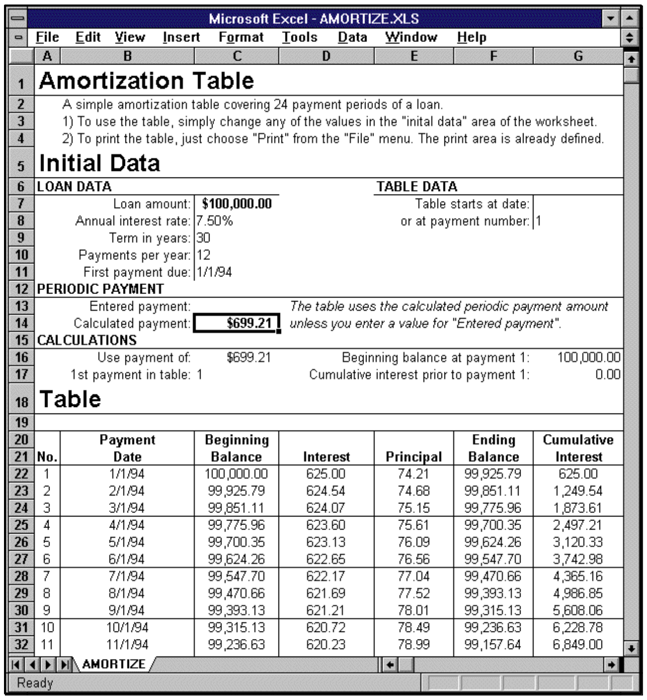
Here's an example of calling an Excel dialog from VB. First, create an Excel workbook called SHEETS.XLS, open a dialog sheet, and add a list box and a list box. Then run the following code from VB:

```
Dim APPXL As object
Dim XL As object
Dim wb As object
Dim dlg As object
Dim dlgList As object
Dim objList As object

Set APPXL = GetObject(,"Excel.Application")
Set XL = APPXL.Application
XL.Workbooks.Open "SHEETS.XLS"
Set wb = XL.ActiveWorkbook
Set Dlg = wb.DialogSheets("dialog1")
Set DlgList = dlg.ListBoxes("sheetsList")
Set objList = wb.Sheets
dlgList.RemoveAllItems
dlg.DialogFrame.Caption = "List of Sheets"
For ix = 1 To objList.count
    dlgList.[AddItem] (objList(ix).name)
Next
dlg.[Show]

Set dlg = Nothing
Set dlgList = Nothing
Set objList = Nothing
Set wb = Nothing
```

How Excel Objects Flow Through Excel. Visual Basic developers gain access to a peerless library of objects designed for building business spreadsheet applications. This chart shows how Excel's objects are contained by and accessed through other objects.



Amortization Table

A simple amortization table covering 24 payment periods of a loan.
1) To use the table, simply change any of the values in the "inital data" area of the worksheet.
2) To print the table, just choose "Print" from the "File" menu. The print area is already defined.

Initial Data

LOAN DATA
Loan amount: $100,000.00
Annual interest rate: 7.50%
Term in years: 30
Payments per year: 12
First payment due: 1/1/94

TABLE DATA
Table starts at date:
or at payment number: 1

PERIODIC PAYMENT
Entered payment:
Calculated payment: $699.21
The table uses the calculated periodic payment amount unless you enter a value for "Entered payment".

CALCULATIONS
Use payment of: $699.21
1st payment in table: 1
Beginning balance at payment 1: 100,000.00
Cumulative interest prior to payment 1: 0.00

Table

	Payment	Beginning			Ending	Cumulative
No.	Date	Balance	Interest	Principal	Balance	Interest
1	1/1/94	100,000.00	625.00	74.21	99,925.79	625.00
2	2/1/94	99,925.79	624.54	74.68	99,851.11	1,249.54
3	3/1/94	99,851.11	624.07	75.15	99,775.96	1,873.61
4	4/1/94	99,775.96	623.60	75.61	99,700.35	2,497.21
5	5/1/94	99,700.35	623.13	76.09	99,624.26	3,120.33
6	6/1/94	99,624.26	622.65	76.56	99,547.70	3,742.98
7	7/1/94	99,547.70	622.17	77.04	99,470.66	4,365.16
8	8/1/94	99,470.66	621.69	77.52	99,393.13	4,986.85
9	9/1/94	99,393.13	621.21	78.01	99,315.13	5,608.06
10	10/1/94	99,315.13	620.72	78.49	99,236.63	6,228.78
11	11/1/94	99,236.63	620.23	78.99	99,157.64	6,849.00

The AMORTIZE.XLS worksheet sample included with Excel. A simple code string is used to open AMORTIZE.XLS, plug in values for the variables, and pop up the calculated payment in a message box. If desired, the whole table can be easily brought back.

```
Set XL = Nothing
Set APPXL = Nothing
```

The Excel 5 INI File

When programming Excel from external applications, it's important to understand at least one aspect of Excel's INI file: the section that manages add-ins. This is necessary for at least two reasons: First, as a developer you naturally want to minimize the resource usage of your app and make it as fast and responsive as possible. Second, it's sometimes necessary to have precise control over Excel's state. A related issue is that if you change the default state of Excel, you'll almost certainly want to restore the user's previous state when you're through.

Excel add-ins are an important variable in the startup state of Excel. Individual add-ins are shown in entries in the "[Microsoft Excel]" section and begin with the word "OPEN:"

```
[Microsoft Excel]
OPEN=/F C:\WINDOWS\EXCEL\LIBRARY\SOLVER\SOLVER.XLA
OPEN1=/F C:\WINDOWS\EXCEL\LIBRARY\CROSSTAB\CROSSFNC.XLA
OPEN2=/F /R C:\WINDOWS\EXCEL\LIBRARY\ADDINFNS.XLA

[Recent File List]

[Init Commands]

[Converters]
```

Note the /F switch in each line. This option, new with version 5.0, tells Excel to fast load the add-in: This means the add-in is examined for menu items and selected text strings, but is not yet loaded into memory. While this approach results in much quicker loads times (and a significantly smaller main memory footprint) than if each of the add-ins were fully loaded, it still takes time for Excel to manage. Also, these menu items that are read in for each add-in consume resources from the Windows menu heap. The menu heap (added in Windows 3.1) is one of the three system heaps (the others being GDI and User) involved in calculating free System Resources; while the amount of free menu heap has customarily been higher than GDI, which is usually the first to get used up, adding a slew of menu items can have a dramatic affect. Of course, the problem with 64K heaps goes away in 32-bit Windows.

So if you want to integrate with Excel 5.0, be aware of Excel's add-in state. With Excel running and after setting an object variable xl to point to an Excel application, enter the following code in VB's debug (easily my favorite window in Windows):

```
for x = 1 to xl.addins.count :?xl.addins(x).name, _
                    xl.addins(x).installed:next
```

This generates a listing similar to the following (your mileage may vary):

```
ANALYSIS.XLL    0
AUTOSAVE.XLA    0
CROSSFNC.XLA   -1
SCENARIO.XLA    0
ADDINFNS.XLA   -1
ANALYSIS.XLA    0
ANALYSF.XLA     0
PROOF.XLA       0
REPORTS.XLA     0
SOLVER.XLA     -1
UPDTLINK.XLA    0
VIEWS.XLA       0
```

Though Excel establishes and maintains this list in EXCEL5.INI and

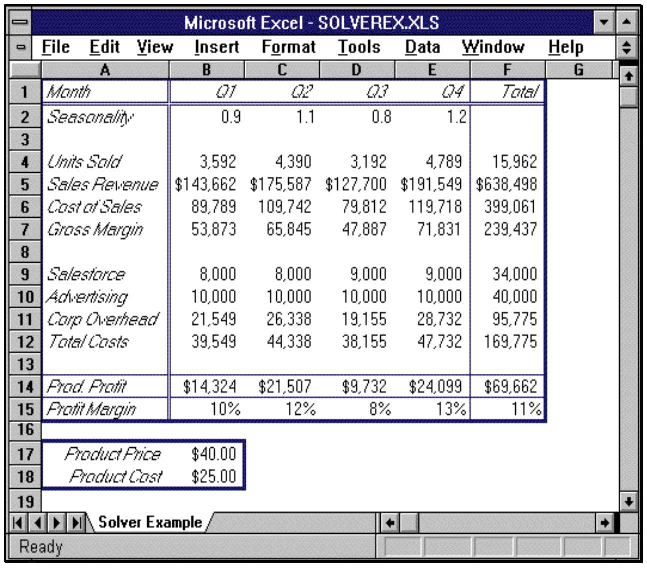
Go To Next Page.

Go To Next Page.

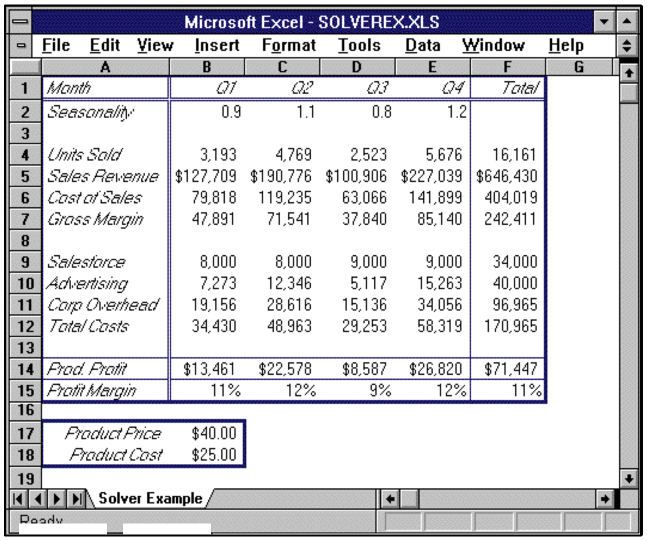
Go To Next Page.

will renumber the entries so that they are sequential, a programmer shouldn't rely on this. The proper technique involves reading in all the keys in that section and searching for anything beginning with "OPEN." The following VB code stuffs the contents of these strings into a default list box (what you want to do with these string will vary):

```
Declare Function GetPrivateProfileString Lib "Kernel"
Alias "GetPrivateProfileString"
(ByVal lpApplicationName As String,
ByVal lpKeyName  As String,
ByVal lpDefault As String,
ByVal lpReturnedString As String,
ByVal nSize As Integer,
ByVal lpFileName As String) As Integer
Declare Function GetPrivateProfileStringSec _
                    Lib "Kernel"
```


Microsoft Excel - SOLVEREX.XLS

	A	B	C	D	E	F	G
1	Month	Q1	Q2	Q3	Q4	Total	
2	Seasonality	0.9	1.1	0.8	1.2		
3							
4	Units Sold	3,592	4,390	3,192	4,789	15,962	
5	Sales Revenue	$143,662	$175,587	$127,700	$191,549	$638,498	
6	Cost of Sales	89,789	109,742	79,812	119,718	399,061	
7	Gross Margin	53,873	65,845	47,887	71,831	239,437	
8							
9	Salesforce	8,000	8,000	9,000	9,000	34,000	
10	Advertising	10,000	10,000	10,000	10,000	40,000	
11	Corp Overhead	21,549	26,338	19,155	28,732	95,775	
12	Total Costs	39,549	44,338	38,155	47,732	169,775	
13							
14	Prod. Profit	$14,324	$21,507	$9,732	$24,099	$69,662	
15	Profit Margin	10%	12%	8%	13%	11%	
16							
17	Product Price	$40.00					
18	Product Cost	$25.00					
19							


Microsoft Excel - SOLVEREX.XLS

	A	B	C	D	E	F	G
1	Month	Q1	Q2	Q3	Q4	Total	
2	Seasonality	0.9	1.1	0.8	1.2		
3							
4	Units Sold	3,193	4,769	2,523	5,676	16,161	
5	Sales Revenue	$127,709	$190,776	$100,906	$227,039	$646,430	
6	Cost of Sales	79,818	119,235	63,066	141,899	404,019	
7	Gross Margin	47,891	71,541	37,840	85,140	242,411	
8							
9	Salesforce	8,000	8,000	9,000	9,000	34,000	
10	Advertising	7,273	12,346	5,117	15,263	40,000	
11	Corp Overhead	19,156	28,616	15,136	34,056	96,965	
12	Total Costs	34,430	48,963	29,253	58,319	170,965	
13							
14	Prod. Profit	$13,461	$22,578	$8,587	$26,820	$71,447	
15	Profit Margin	11%	12%	9%	12%	11%	
16							
17	Product Price	$40.00					
18	Product Cost	$25.00					
19							

Using the Solver add-in to analyze a spreadsheet. Excel defines an object for handling add-ins, with properties and methods like other Excel objects. Unfortunately, add-ins are not necessarily objects and must be programmed through the Application object. Figures 3 and 4 show the effects of running Solver on the SOLVEREX.XLS sample provided with Excel, using a scenario covered in Chapter 30 of the Microsoft Excel User Guide.

```
Alias "GetPrivateProfileString"
(ByVal lpApplicationName As String,
ByVal lpKeyName As Long,
ByVal lpDefault As String,
ByVal lpReturnedString As String,
ByVal nSize As Integer,
ByVal lpFileName As String) As Integer
Dim sBuff As String * 2048
Dim sBuff2  As String * 128
Z$=Chr$(0)

iBuff = GetPrivateProfileStringSec("Microsoft Excel", 0&, "none", sBuff, Len(sBuff),
"EXCEL5.INI")
Ini$ = Z$ & Left$(sBuff, iBuff)

Foo$=Z$ & "OPEN"
y%= 1

Do
    x% = InStr(y%, Ini$, Foo$)
    If x% = 0 Then Exit Do
    y% = InStr(x% + 1, Ini$, Z$)
    Tmp$ = Trim$(Mid$(Ini$, x% + 1, y% - x% - 1))
    iBuff = GetPrivateProfileString("Microsoft Excel", Tmp$, "none", sBuff2,
Len(sBuff2), "EXCEL5.INI")
    List 1.AddItem Left(sBuff2, iBuff)
Loop
```

USING EXCEL ADDINS (SOLVER)

After building a spreadsheet model of a problem, you can use Solver to optimize the value of a target cell in that model by changing the values of a related range of cells. Solver allows you to set constraints for any cell in the model. Figures 3 and 4 show the effects of running Solver on the SOLVEREX.XLS sample provided with Excel and using the scenario covered in Chapter 30 of the Microsoft Excel User Guide. Specifically, a constraint is specified for total advertising expenditures (F10 <= $40,000) and total profit (F14) is optimized by varying the quarterly advertising estimate (B10 to E10).

Excel defines an object for handling add-ins, with properties and methods just like other Excel objects:

•Properties: Application, Author, Comments, Count, Creator, FullName, Installed, Keywords, Name, Parent, Path, Subject, Title
•Methods: Add, Item

Unfortunately, the add-ins themselves are not necessarily objects and must be programmed through the Execute-Excel4Macro method of the Application object. For example, to run the Solver analysis described above, the following code is run from VB:

```
Dim APPXL As object
Dim XL As object
Dim ws As object

Set APPXL = GetObject(, "Excel.Application")
Set XL = APPXL.Application
XL.Workbooks.Open
"C:\WINDOWS\EXCEL\EXAMPLES\SOLVER\SOLVEREX.XLS"

Set ws = XL.ActiveSheet
oldP$ = ws.Range("$F$14").Value
XL.ExecuteExcel4Macro "[SOLVER.XLA]SOLVER!SOLVER.OK(!R10C6,1,0,)"
XL.ExecuteExcel4Macro
"[SOLVER.XLA]SOLVER!SOLVER.ADD(!R10C6,1,""=40000"")"
XL.ExecuteExcel4Macro
"[SOLVER.XLA]SOLVER!SOLVER.OK(!R14C6, 1, 0,(!R10C2:R10C5))"
XL.ExecuteExcel4Macro "[SOLVER.XLA]SOLVER!SOLVER.SOLVE(True)"
```

Go To Next Page.

Go To Next Page.

```
newP$=ws.Range("$F$14").Value
MsgBox "Old: " & Format(oldP$, "currency") & Chr$(10) & "New: " &
Format(newP$, "currency"), , "Profit"
XL.Workbooks(1).[Close] (False)

Set ws = Nothing
Set XL = Nothing
Set APPXL = Nothing
```

Interestingly, Excel add-ins can also be programmed (from VBA only) via:

References Command (Tools Menu)

This command adds, deletes, or makes available for editing Visual Basic references to libraries or other workbooks that are specified for the active workbook. You must be in a Visual Basic module to use this command.

```
SOLVEROK(!R10C6,1,0,)
SOLVERADD(!R10C6,1,""=40000"")
SOLVEROK(!R14C6,1,0,(!R10C2:R10C5))
SOLVERSOLVE(True)
```

A couple of notes about this behavior: first of all, Excel's built-in Macro recorder, when set for VBA format, records (at least in the version I'm using) generates "Application.ExecuteExcel4Macro" code. This isn't as bad as it seems since the code is much more portable to VB than it would be otherwise. Second, Excel 5 (at least for this version) is actually translating OLE Automation calls into traditional XLM macro code anyway, so there wouldn't be an appreciable performance improvement if add-ins were programmable via conventional syntax. ■