Using API Calls During Development

Perhaps the best method of monitoring resources is via VB itself. By using the Debug window and a few Windows API functions, you can create a "status" procedure that will dynamically display the current memory and resource usage as your application is running. This is ideal for examining the "before" and "after" effects of loading control-laden forms, unloading objects, using one control vs. another, allocating arrays, etc.

The following code can be stored in a single code module called SHOWRES.BAS. Add this module to the project you wish to analyze, and sprinkle calls to the ShowResources procedure wherever you wish to check the memory and resource usage.

```
' Add these lines to general section of SHOWRES.BAS:
Global Const GFSR_GDIResources% = 1
Global Const GFSR_UserResources% = 2
Declare Function GetFreeSystemResources Lib "User" (ByVal fuSysResource
⇒ As Integer) As Integer
Declare Function GetFreeSpace Lib "Kernel" (ByVal wFlags As Integer)
⇒ As Long
' Add this procedure to SHOWRES.BAS:
Sub ShowResources ()
  Debug.Print "Free Memory:"; GetFreeSpace(0),
  Debug.Print "GDI:"; GetFreeSystemResources(GFSR_GDIResources%); "%",
  Debug.Print "User:"; GetFreeSystemResources(GFSR_UserResources%); "%"
End Sub
```

Here's a small example to illustrate the ShowResources procedure:

```
Sub LoadTest_Click ()
  Debug.Print "Before loading 15 picture controls:"
  ShowResources ' Show statistics in Debug window
  For l = 1 To 15
    Load MyPictures(l)
  Next
  Debug.Print "After loading 15 picture controls:"
  ShowResources ' Show statistics in Debug window
End Sub
```

The Debug window then shows the following results:

```
Before loading 15 picture controls:
Free Memory: 8237312        GDI: 75 %     User: 78 %
After loading 15 picture controls:
Free Memory: 8228512        GDI: 67 %     User: 77 %
```

As you can see, the GDI heap took a big hit once the picture controls were loaded. When the same example is modified to use image controls instead of picture controls, the Debug window shows the following:

```
Before loading 15 image controls:
Free Memory: 8228416        GDI: 75 %     User: 77 %
After loading 15 image controls:
Free Memory: 8225344        GDI: 75 %     User: 77 %
```

The drastic difference in GDI usage proves that image controls indeed offer significant savings over picture controls -- assuming that you can do without the additional features of picture controls (DDE, graphics methods, etc.)

Using The Graphical ("Light") Controls

Microsoft added the graphical controls (line, shape and image) in VB 2.0 for good reason. These controls differ from the other standard controls in that they are not truly windows -- that's why they lack the hWnd property. Therefore, they consume far less system resources than their true-window cousins. (Note: The label control is also a graphical control but of course is not new to VB 2).

For example, you may be currently using a custom control to display a status gauge like the ones you'll find in your garden-variety SETUP program. Here's how to make a "poor man's status gauge" from just two graphical controls and a few lines of code:

1. Add a label control called StatusLabel, with the following property values:

Property	Value
Alignment	2 (Center)
BorderStyle	1

2. Add a shape control called StatusBar, with the following property values:

Property	Value
Width	0
Top	Same as StatusLabel.Top
Left	Same as StatusLabel.Left
FillStyle	0 (Solid)
FillColor	Blue, or color of your choice
DrawMode	14 (Merge Pen Not)

To display a value in the "status bar", use the following code:

```
Progress% = some value
' The value above should be from 0 to 100
StatusBar.Width = StatusLabel.Width * (Progress% / 100)
StatusLabel.Caption = Str$(Progress%) + "%"
```

Another effective use for a graphical control is to use image controls as buttons.

Use graphics methods to simulate 3D controls.

Most 3D controls are more than just a pretty face, so substituting them with a cosmetic-only replacement will usually rob your application of certain features. Nevertheless, some 3D controls -- particularly 3D labels and frames -- are ripe for replacement with simulated 3D graphics. If you aren't sure how to accomplish a 3D effect using graphics method only, here's a simple function to get you started:

```
' Draw3DBorder Function: Draws a 3D border around the specified
' control, with "raised" or "lowered" shading, and user-defined
' border thickness.
Sub Draw3DBorder (TargetControl As Control, RaisedBorder%, BorderWidth%)
  ' Define how far the 3D lines are drawn from the
  ' outer edges of the control. Modify to your tastes:
  BorderOffset% = 8

  ' Define the four corners of the 3D box to draw:
  X1 = TargetControl.Left - BorderOffset%
  Y1 = TargetControl.Top - BorderOffset%
  X2 = X1 + TargetControl.Width + (BorderOffset% * 2)
  Y2 = Y1 + TargetControl.Height + (BorderOffset% * 2)

  ' We'll change the parent form's ForeColor and DrawWidth
  ' properties, so we'll save them first and restore when done
  OriginalForeColor = TargetControl.Parent.ForeColor
  OriginalDrawWidth = TargetControl.Parent.DrawWidth

  ' If RaisedBorder% is True, the white lines are drawn
  ' on the top and left sides; otherwise, they are drawn
  ' on the bottom and right sides.
  If RaisedBorder% Then
    UpperColor = QBColor(15)     ' white
    LowerColor = QBColor(8)      ' grey
  Else
    UpperColor = QBColor(8)      ' grey
    LowerColor = QBColor(15)     ' white
  End If
  ' Draw line on left
  TargetControl.Parent.DrawWidth = BorderWidth
  TargetControl.Parent.ForeColor = UpperColor
  TargetControl.Parent.Line (X1, Y2)-(X1, Y1)
  ' Draw line on top
  TargetControl.Parent.Line -(X2, Y1)
  ' Draw line on right
  TargetControl.Parent.ForeColor = LowerColor
  TargetControl.Parent.Line -(X2, Y2)
  ' Draw line on bottom
  TargetControl.Parent.Line -(X1, Y2)
  ' Restore the parent form's original properties
  TargetControl.Parent.ForeColor = OriginalForeColor
  TargetControl.Parent.DrawWidth = OriginalDrawWidth
End Sub
```

Using the Draw3DBorder function is simply a matter of specifying the name of the control (TargetControl), whether you want "raised" lines (RaisedBorder% = True) or "lowered" lines (RaisedBorder% = False), and the thickness of the 3D border (BorderWidth%). For example, to draw a 3D border around a label control, do the following:

```
' Draw a raised border around the MyLabel control:
RaisedBorder% = True
BorderWidth% = 2
Draw3DBorder MyLabel, RaisedBorder%, BorderWidth%
```

Note that when using this 3D effect on a form, you should set the form's AutoRedraw property to True in order to keep the 3D effect refreshed whenever the form is painted.

Use SndPlaySound API function instead of MCI.VBX

The MCI control provides a number of convenient services and sports an attractive cassette deck-style interface, but it can be overkill if you simply wish to play sound files without user interaction, or trigger the system event sounds defined in the Windows Control Panel. The SndPlaySound API function from the MMSYSTEM.DLL file (included with Windows 3.1 or the Microsoft Multimedia Extensions for Windows 3.0) allows you to play sounds without using a custom control. By avoiding the MCI control, your application will use less resources and will not require you to distribute MCI.VBX. In case you were wondering, MCI.VBX also makes calls to MMSYSTEM.DLL, so using SndPlaySound eliminates the overhead of the MCI control.

The code below shows how to play a WAV audio file with SndPlaySound:

```
' Add these lines to general section:
Declare Function SndPlaySound Lib "MMSYSTEM.DLL"
⇒ (ByVal lpszSoundName$, ByVal wFlags%) As Integer
' Flags used by SndPlaySound:
Const SND_ASYNC = &H0001

' Play a sound when a button is clicked:
Sub Command1_Click ()
  wFlags% = SND_ASYNC  ' Play asynchronously (returns immediately)
  x% = SndPlaySound("c:\windows\chord.wav", wFlags%)
End Sub
```