Make the Most of Resources

Click & Retrieve Source CODE!

Resource files are underappreciated in VB4.

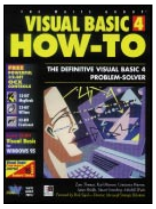
by Karl E. Peterson

One of the cooler enhancements of VB4 is perhaps one that is the least appreciated. Basic programmers have bemoaned the loss of the DATA statement since the introduction of VB1. With VB4, Microsoft added capabilities far exceeding old DOS Basics, incorporating resource (RES) files within the language. Yet I hardly ever hear of anyone taking advantage of this new tool—perhaps because of the archaic resource compiler that shipped with VB4? If so, have I got wonderful news for you! This month, I'll tell you about a great freeware tool you can use to create resource files, and show you a few tricks you can use once you've put a resource file together. These tricks are a bit different from what you will commonly see as the reasons for using resource files, such as providing localization support with multiple string resources in different languages. No, the uses of RES files are often limited only by your imagination.

Before you do anything, though, download VBRes from the Registered Level of The Development Exchange (for details, see the Code Online box at the end of this article). This is an incredible little tool that was written entirely with VB4 by Gregg Irwin, who coauthored *Visual Basic Controls Desk Reference CD: The Definitive Book of Third-Party VBX and DLL Controls*. VBRes allows you to build resource files using any sort of input data, assign resource types and IDs, and compile to both 16- and 32-bit RES files (see Figure 1). If you plan to build both 16- and 32-bit apps, it's important to load the proper RES file into your project, as it won't work in the wrong environment. VBRes' drag-and-drop MDI interface is a delight to work with compared to the alternative RC supplied with VB4 and other Microsoft products.

Here's just one example of how easy it is to add functionality VB programmers have been searching for since VB was introduced. Have you ever wanted to supply a choice of icons within your application so the user can pick which icon shows up in Program Manager or in a shortcut? To do so, simply compile a number of icons into a RES file and include the RES file in your project. There's only one rule you need to follow—VB reserves ID #1 for the application's icon. This is the default icon defined in the "Make EXE" dialog. As you can see in Figure 1, I've built a resource file with a series of moon icons (the ones placed in the \vb\icons\elements folder while installing VB). After compiling an application that uses this RES file, I can choose "Change Icon" while setting up a shortcut to it, after which I'm presented with all the icons in the RES file in addition to the default application icon (see Figure 2).

Karl E. Peterson is a GIS analyst with a regional transportation planning agency and a member of the Visual Basic Programmer's Journal *Technical Review Board. Based in Vancouver, Washington, he's also an independent programming consultant and a writer. Karl coauthored* Visual Basic 4 How-To, *from Waite Group Press. Online, he's a section leader in the* VBPJ *Forum 32-Bit Bucket, a Microsoft MVP@Large, and active in the chat area on The Development Exchange Web site. Contact Karl in the CompuServe forums at 72302,3707, on the Internet at karl@rtc.wa.gov, or on DevX.*

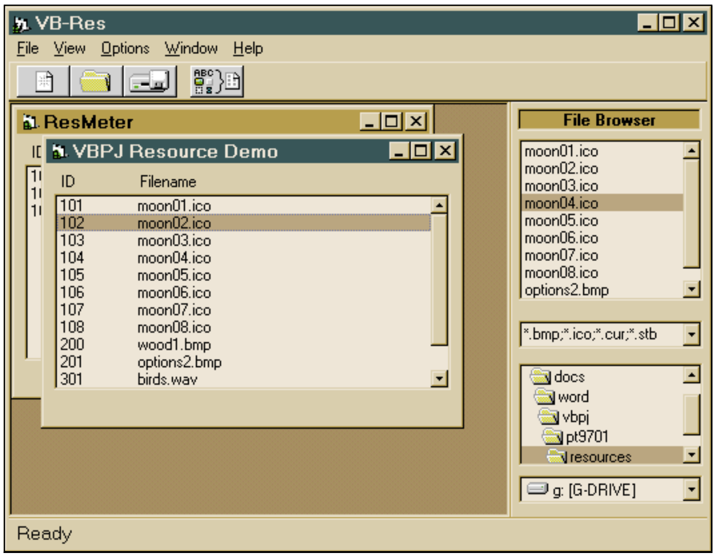


MARK PROGRESS WITH ANIMATION

Icons embedded in resource files provide you with the raw materials to communicate to your user that your application is busy with something. Simply by enabling a timer, you can provide the sort of animation used by Netscape and Internet Explorer when they're retrieving data from the Internet (see Figure 3). Begin by making sure that the "Standard OLE Types" reference is selected for your project, then declare a form-level Picture object in the Declarations section of your form. Place a Timer control on your form, and set its interval to around 100 milliseconds. On each Timer event, determine which icon to use next, load it from the RES file, and call DrawIcon to display it in the proper position on your form:

```
Private m_MoonIco As Picture

Private Sub Timer1_Timer()
```



ID	Filename
101	moon01.ico
102	moon02.ico
103	moon03.ico
104	moon04.ico
105	moon05.ico
106	moon06.ico
107	moon07.ico
108	moon08.ico
200	wood1.bmp
201	options2.bmp
301	birds.wav

moon01.ico
moon02.ico
moon03.ico
moon04.ico
moon05.ico
moon06.ico
moon07.ico
moon08.ico
options2.bmp

.bmp;.ico;*.cur;*.stb

docs
word
vbpj
pt9701
resources

g: [G-DRIVE]

FIGURE 1 ***A Windows-Based Resource Compiler.*** *VBRes is an MDI applet written entirely in VB4 by Gregg Irwin. It provides relief from the command-line resource compiler provided by Microsoft with VB4. You can download it from the Registered Level of The Development Exchange.*

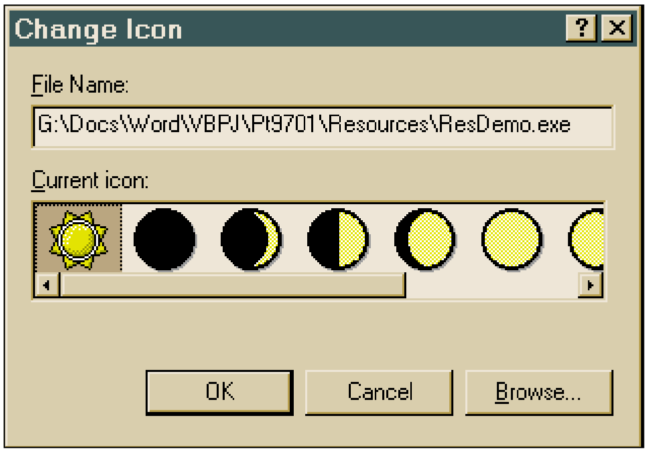


FIGURE 2 ***Offer a Choice of Icons For Your Application.*** *By including extra icons in your app's RES file, you can let your users pick one to represent your app in shortcuts or Program Manager. The only precaution is to ensure that all icons are assigned an ID greater than one.*

```
   Static Which As Integer
   Static X As Long, Y As Long

Which = ((Which + 1) Mod 8)
   Set m_MoonIco = LoadResPicture(Which _
      + 101, vbResIcon)
```

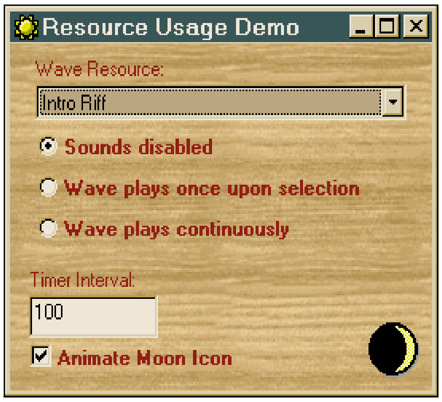


FIGURE 3 ***Multiple Effects Made Simpler with RES Files.*** *This shows how you can use DrawIcon to place an icon over a painted background, preserving the icon's transparency. By rapidly calling DrawIcon with a series of icons, you can effectively duplicate the method of indicating a busy state used by many Internet browsers. RES files are also useful for storing bitmaps used for any number of purposes, such as the background of this form and the radio and check box buttons.*

```
   ' If form is resizable, calc (X,Y)
   ' on each call.
   X = (Me.ScaleWidth \ Screen.TwipsPerPixelX) - 40
   Y = (Me.ScaleHeight \ Screen.TwipsPerPixelY) - 40
   Call DrawIcon(Me.hdc, X, Y, _
      m_MoonIco.Handle)
End Sub
```

I chose to have the Picture object exist at the form level because in a case such as this, minimizing overhead is the objective. The point is to inform the user that the program is busy, but to do so without adding tremendously to its burden. The DrawIcon API preserves the transparent areas of the icon, allowing you to draw over any sort of background. DrawIcon requires only a handle to the device context (in this case, the form itself), X and Y coordinates, and a handle to an icon. While Picture box or Image controls offer such a handle through their Picture properties, the Picture object also offers it, but by the more intuitive Handle property. When your application is through with the task at hand and you've disabled the Timer control, you'll probably want to either restore the default icon or clear the area used to display this progress indicator.

USE SOUND RESOURCES

Another neat way to use RES files is to embed sounds directly within your application. Although this is not a resource type directly supported by VB, it's a simple matter to load the sounds into memory. When compiling your RES file, assign a Resource Type of "SOUND," "WAVE," or some other unique string to the sound file. If you're using VBRes, you can do this by right-clicking on the file name and bringing up the Properties dialog. When you want to play one of these sounds, you can use the VB function LoadResData to load the sound resource into a byte array, and call the appropriate API:

```
Private Declare Function PlaySoundData _
   Lib "winmm.dll" Alias "PlaySoundA" _
   (lpData As Any, ByVal hModule As _
   Long, ByVal dwFlags As Long) As _
   LongPrivate m_snd() As Byte

Public Function PlaySoundResource(ByVal _
   SndID As Long) As Long
   Dim snd() As Byte
   Const Flags = SND_MEMORY Or SND_SYNC _
      Or SND_NODEFAULT
```

```
    snd = LoadResData(SndID, "SOUND")
    PlaySoundResource = _
       PlaySoundData(snd(0), 0, Flags)
End Function
```

If you're using VB4/16, you will be presented with a slight limitation here because LoadResData only returns up to 64K of data. It is up to you to ensure that no resources are larger than this, or you run the risk of an "Out of String Space" error. Even though the docs don't make the distinction, this is definitely not the case in 32-bit VB4, where you can load a resource of any size. You also may have noticed that this function plays sounds synchronously—that is, it doesn't return until the sound is finished playing. If you wish to start the sound and immediately return, another "interesting" situation presents itself. What becomes of the byte array when the function exits? As you most likely surmised, it goes out of scope and is released. That obviously won't do. For starters, it is nearly guaranteed that most sounds will not have had time to finish playing. Worse is the prospect of potential GPFs, as the system attempts to read memory that has already been discarded.

One potential workaround is to declare the byte array at the module level. Doing this prevents it from being deallocated as the function exits. But this approach raises the question, "What happens if PlaySoundResource is called again while a previous sound is still playing?" Again, the mind boggles at the possibility of GPFs. Amazingly, it does seem quite safe, at least from my experiments. To be entirely safe, you may wish to set up a scheme that uses two byte arrays, alternating on each call.

WHAT TO DO WITH A HANDLE?

I hate using controls when memory serves the same purpose. Controls just drag along all their associated baggage (slower form loads, extra resource consumption, and so on) and offer few benefits in many cases. For example, suppose you want to tile a bitmap across the back of your forms. The old approach would be to load a hidden Picture box with the bitmap at design time, then perform a series of BitBlts whenever the form needs

VB4 32-bit

```
VERSION 1.0 CLASS
BEGIN
  MultiUse = -1  'True
END
Attribute VB_Name = "CPictureDC"
Attribute VB_Creatable = False
Attribute VB_Exposed = False
Option Explicit
'
' Win32 API Declarations, Structures, and Constants
'
Private Declare Function CreateCompatibleDC Lib _
  "gdi32" (ByVal hdc As Long) As Long
Private Declare Function SelectObject Lib "gdi32" _
  (ByVal hdc As Long, ByVal hObject As Long) As Long
Private Declare Function DeleteObject Lib "gdi32" _
  (ByVal hObject As Long) As Long
Private Declare Function GetObj Lib "gdi32" Alias _
  "GetObjectA" (ByVal hObject As Long, _
  ByVal nCount As Long, lpObject As Any) As Long
Private Declare Function GetDesktopWindow Lib _
  "user32" () As Long
Private Declare Function GetDC Lib "user32" _
  (ByVal hWnd As Long) As Long
Private Declare Function ReleaseDC Lib "user32" _
  (ByVal hWnd As Long, ByVal hdc As Long) As Long
'
' Bitmap Header Definition
'
Private Type BITMAP '14 bytes
  bmType As Long
  bmWidth As Long
  bmHeight As Long
  bmWidthBytes As Long
  bmPlanes As Integer
  bmBitsPixel As Integer
  bmBits As Long
End Type
'
' Member variables
'
Dim m_bmp As BITMAP
Dim m_hDC As Long
Dim m_hBmp As Long
Dim m_hBmpTmp As Long
Dim m_pict As Picture
' *****************************************************
'  Initialization and Termination
' *****************************************************
Private Sub Class_Initialize()
  Dim hWndScn As Long
  Dim hDCScn As Long
  '
  ' Get desktop DC, and create compatable DC.
  '
  hWndScn = GetDesktopWindow()
  hDCScn = GetDC(hWndScn)
  m_hDC = CreateCompatibleDC(hDCScn)
  Call ReleaseDC(hWndScn, hDCScn)
End Sub

Private Sub Class_Terminate()
  '
  ' Clean up resources
  '
  If m_hBmp Then
     Call SelectObject(m_hDC, m_hBmpTmp)
  End If
  Call DeleteObject(m_hDC)
End Sub

' *****************************************************
'  Public Properties
' *****************************************************
Public Property Let hBitmap(NewVal As Long)
  Static PropSet As Boolean
  '
  ' Write-once handle to bitmap
  '
  If PropSet = False Then
     m_hBmp = NewVal
     m_hBmpTmp = SelectObject(m_hDC, m_hBmp)
     Call GetObj(m_hBmp, Len(m_bmp), m_bmp)
     PropSet = True
  End If
End Property

Public Property Get hDC() As Long
  hDC = m_hDC
End Property

Public Property Get bmType() As Long
  bmType = m_bmp.bmType
End Property
```

CONTINUED ON PAGE 122.

LISTING 1 ***CPictureDC Class Offers a Device Context for Picture Objects.*** *Many GDI calls, such as BitBlt, require a source device context as one of their parameters. If you store graphics in OLE Picture objects, it can be tricky to provide a device context handle for them. Using this class solves the problem because it creates a screen-compatible device context into which it selects the OLE Picture object's bitmap.*

CONTINUED FROM PAGE 121

```
Public Property Get Width() As Long
  Width = m_bmp.bmWidth
End Property

Public Property Get Height() As Long
  Height = m_bmp.bmHeight
End Property

Public Property Get WidthBytes() As Long
  WidthBytes = m_bmp.bmWidthBytes
End Property

Public Property Get Planes() As Integer
  Planes = m_bmp.bmPlanes
End Property

Public Property Get BitsPerPixel() As Integer
  BitsPerPixel = m_bmp.bmBitsPixel
End Property

Public Property Get Bits() As Long
  Bits = m_bmp.bmBits
End Property

' ****************************************************
' ****************************************************
```

repainting (see Figure 3). Depending on your needs, if you had a number of forms you wanted to use this effect with, it could either turn quite cumbersome to code or would suck up extra system resources as you placed this same Picture box on each form. With VB4, if you have the "Standard OLE Types" reference checked for your project, you can load bitmaps from a RES file into a Picture object in memory and potentially save a lot of system resources. As with many good things, trade-offs are often required. The problem with Picture objects is they don't expose (because they don't consist of) a device context and its handle.

The problem becomes how to use the Handle property exposed by Picture objects with the many APIs that require an hDC value. I've written a small class module that solves this problem by selecting a Picture object's bitmap into a temporary memory device context (see Listing 1). To use this class, declare a new instance of it within a subroutine and assign any hBitmap to its hBitmap property. You then can call any function that requires an hDC source by substituting the value returned from the class' hDC property:

```
Dim pic As New CPictureDC
Dim trans As Long

' Assign Picture object's Handle to
' class
pic.hBitmap = m_Options.Handle
TransColor = &HC0C0C0
Call TransBlt(Me.hdc, X, Y, 16, 16, pic.hdc, 16, 0, TransColor)
```

This example comes from the project shown in Figure 3. TransBlt is a routine with its origins in the Knowledge Base article Q94961, "How to Create a Transparent Bitmap Using Visual Basic." That code was presented in 16-bit only, but if you'd like a 32-bit translation that I also enhanced somewhat, download ResDemo from the Premier Level of The Development Exchange. As you can see, I simply assign the hBitmap to the class, and then I'm able to use its hDC property as I would if this were an actual control. Because this code is taken from a subroutine, you don't need to clean up by setting the class to Nothing—it conveniently terminates itself as it goes out of scope.

The CPictureDC class works by creating a memory bitmap that's compatible with the screen. First, GetDesktopWindow is called to obtain a handle to the screen window, then GetDC is used to obtain the screen window's device context handle. This hDC is used with CreateCompatibleDC to create a memory device context with the same characteristics (color depth and device capabilities) as the screen. The last preliminary step is to release the desktop DC so that other processes can use it.

The assignment of the hBitmap property is the guts of the class. It's written as a write-once property, allowing only one bitmap to be assigned per instance of the class. With the first assignment, the passed hBitmap is selected into the memory device context, and a reference to the bitmap that was previously stored in it is preserved. It is important that this original bitmap is restored before the memory device context is destroyed, which is why hBitmap is designed as a write-once property.

One oddity of the Picture object is that its Height and Width properties are expressed in HIMETRIC format. Because few VB programmers would choose this measurement system, the CPictureDC class takes this opportunity to obtain additional properties of the bitmap and expose them in addition to the hDC. A quick call to the GetObject API fills a BITMAP structure with a number of useful values, each of which are then exposed through corresponding read-only property routines.

The real beauty of class modules is that you can write them to do so much, and they require so little. In this case, CPictureDC cleans up after itself in its Terminate event by selecting the original bitmap back into the memory device context, then deleting the memory device context. Forgetting to do this will cause rapid system resource depletion! This step is intended to be performed automatically if you declare instances of CPictureDC at the local level. As your routine ends, a local class instance will go out of scope and its Terminate event will trigger. What could be easier?

SYSTRAY UPDATE

In my last column, "Stay in the Tray" [*VBPJ* November 1996], I presented a method to embed your applications within the taskbar tray, and it has since been pointed out that there was a slight problem with how the popup menu behaved. If the user clicked on some other object after the menu popped up, then the menu failed to disappear. The fix for this problem is to make a call to SetForegroundWindow immediately before calling the PopupMenu method, placing the invisible form that contains the menu into the foreground:

```
Case WM_RBUTTONUP
   Call SetForegroundWindow(Me.hwnd)
   Me.PopupMenu mPopup, , , , mPop(0)
```

While this workaround fixes the recalcitrant menu, it brings on yet another problem. If you load and show a form modally from the popup menu's Click event, you'll trigger a GPF. To get around this wrinkle, load the form invisibly, call SetForegroundWindow with the new form's hWnd, then show it modally. My thanks to Don Bradner for working through these issues with me.

Author's Note: VBPJ *is redefining its columns starting next month, so this is the last installment of the Programming Techniques column. It's been an assignment I've truly enjoyed, and I'm deeply grateful for both the opportunity to have worked on it and all the nice messages I've received from you over the last year and a half. Hopefully, Jonathan Wood and I will find new niches in the redesigned* VBPJ*, and we can continue exploring the back allies of VB together for some time to come!*

Code Online

You can find all the code published in this issue of* VBPJ *on The Development Exchange (DevX) at http://www.windx.com. All the listings and associated files essential to the articles are available for free to Registered members of DevX, in one ZIP file. This ZIP file is also posted in the Magazine Library of the* VBPJ *Forum on CompuServe. DevX Premier Club members ($20 for six months) can get each article's listings in a separate file, as well as additional code and utilities for selected articles, plus archives of all code ever published in* VBPJ *and* Microsoft Interactive Developer *magazines.

Make the Most of Resources

Locator+ Codes

Listings ZIP file plus VBRes.zip, which allows you to build resource files using any sort of input data, assign resource types and IDs, and compile to both 16- and 32-bit RES files (free Registered Level): VBPJ0197

★ Listings for this article plus ResDemo.zip, which is an enhanced 32-bit translation of TransBlt (subscriber Premier Level): PT0197P