INTERMEDIATE Ask the VB Pro

Mine Your Resources

by Karl E. Peterson

Q. Enumerate Outside Resources

I need to extract resources out of DLLs—and possibly EXEs—that aren't actually a part of my application. I didn't put together the original files, so how do I determine what IDs to use for these resources? It looks as though I need to use the EnumResourceNames API, but all my attempts have failed.

ABOUT THIS COLUMN
Ask the VB Pro provides you with free advice on programming obstacles, techniques, and ideas. Read more answers from our crack VB pros. You can submit your questions, tips, or ideas on the site, or access a comprehensive database of previously answered questions.

A. You're definitely on the right track. EnumResourceNames passes the names or IDs of all resources of a specific type within a given module to a designated procedure within your app. If you're after, say, bitmaps or icons, EnumResourceNames is the call to make. But if you first want to determine exactly what types of resources are available, you can call EnumResourceTypes, which provides a callback for each type of resource within the module, then call EnumResourceNames on each of these callbacks. Together, these enumerations provide a list of all present resources (see Figure 1).

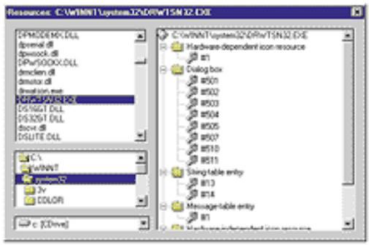

Figure 1 Uncover Hidden Resources. Click here.

Before you can enumerate a file's resources, you must first load that module into your process's address space using LoadLibraryEx. This API takes three parameters, but uses only the first and last; the second, hFile, is reserved and must be zero. Pass the filename you want to load as the first parameter to LoadLibraryEx and use LOAD_LIBRARY_AS_DATAFILE for the dwFlags parameter. If LoadLibraryEx succeeds, it returns an hModule you can use with later resource-related APIs (see Listing 1). Don't forget to call FreeLibrary to release the module handle after you're through enumerating its resources.

Begin the actual enumeration with a call to EnumResourceTypes, passing it the hModule retrieved by LoadLibraryEx and the address of a callback procedure. Optionally, the third EnumResourceTypes parameter accepts a Long, which is later passed directly to the callback. You can use it to indicate how the callback should process other incoming data.

The function prototypes for your callback procedures are defined precisely. EnumResTypeProc expects to receive three Longs—a handle to the module being enumerated, a pointer to a string that describes the resource type, and the optional value that you might have passed as the last parameter to EnumResourceTypes. Of these, the most interesting is the resource description.

About 20 resource types are predefined, but developers are free to define their own as well. If the current resource is one of the standard types, lpszType contains a value that maps to one of the RT_* constants that now range from 1 to 23. More standard types will be added as time goes on, no doubt. But if the current resource is user-defined, not standard, then lpszType is a pointer to a resource type's string description.

How can you tell what the callback passed? If the hiword of lpszType is zero, the loword

Original Code
Get the code for this article **here**.

Updated Code
EnumRes

contains a numeric ID. If the hiword isn't zero, it passed a pointer, and you need to dereference it into a standard VB string variable (see Listing 2). In either case, simply enumerating all resources of this type requires that you pass hModule and lpszType directly to EnumResourceNames, along with the address of your desired callback routine and an optional Long user data value.

As before, you must define EnumResNameProc precisely. The parameters are similar to EnumResTypeProc, with one addition that provides the resource ID. Decode both lpszType and lpszName, as before. If the hiword is zero, you have a numeric ID; otherwise, you need to dereference a string pointer.

Q. Implement Callbacks in Classes and Forms

Windows offers all kinds of cool callbacks, which have been accessible since VB5's introduction of the AddressOf operator. But it bothers me to no end that I can direct these callbacks only into standard BAS modules, instead of where I'd really like to send them—into my classes and forms. This seems to go against all the principles of encapsulation I strive for. How can I reroute these callbacks to where they're really needed?

A. I agree: AddressOf's limitations are frustrating, but they're not insurmountable. My own preference is to define a custom callback interface. Forms and classes can use Implements to offer this interface to the system callback procedures. This means you're now stuck with two extra modules instead of just one, but it also means you can move callback processing back to where it really belongs.

Let's use the above resource enumeration as an example. First, define your interface class, IEnumResources:

```
Option Explicit
Public Sub EnumResourceSink(ByVal _
   ResName As String, ByVal ResType _
   As String, Continue As Boolean)
   ' This routine is called by
   ' the MEnumResources module
   ' once for each resource type
   ' found and once for each
   ' named resource of that type.
   '
   ' Allows enumeration to be handled
   ' in class or form modules.
   ' Enumeration continues until all
   ' resources have been found or
   ' Continue is set to False.
End Sub
```

Open the code window to your class or form module and add this to the Declarations section:

```
Implements IEnumResources
```

Now drop the Objects box, which displays the names of all objects associated with the module, and select IEnumResources. This interface includes only one method, so the EnumResourceSink prototype is added to your code window immediately:

```
Private Sub IEnumResources_EnumResourceSink _
   (ByVal hModule As Long, ByVal _
   ResName As String, ByVal ResType _
   As String, Continue As Boolean)

End Sub
```

Next, expand on the EnumResources function in Listing 1, adding a new function that supports an additional argument—a pointer to your implemented interface:

```
' Reference to callback interface
Private m_Callback As IEnumResources

Public Function _
   EnumResourcesEx(Callback As _
```

```
   IEnumResources, Optional ByVal _
   ModuleName As String = "") As Boolean
   ' This routine is *not* re-entrant!
   If m_Callback Is Nothing Then
      Set m_Callback = Callback
         EnumResourcesEx = EnumResources(ModuleName)
      Set m_Callback = Nothing
   End If
End Function
```

Call EnumResourcesEx, passing Me as a pointer to your implemented interface:

```
Call EnumResourcesEx(Me, FileSpec)
```

The last step is to add support for interface notification to your callback routines. For example, EnumResNameProc tests whether the module-level m_Callback object variable has been set to point to an IEnumResources interface. If so, EnumResNameProc notifies that interface each time it's called by the system (see Listing 3).

This probably seems like a tortuous path simply to maintain some semblance of encapsulation; however, it's worthwhile because you now have the full decision and action power back where it belongs.

*Karl E. Peterson is a GIS analyst with a regional transportation planning agency and serves as a member of the **Visual Basic Programmer's Journal** Technical Review and Editorial Advisory Boards. Online, he's a Microsoft MVP and a section leader on several **VBPJ** forums. Find more of Karl's VB samples at www.mvps.org/vb.*