Sounds Like Trouble

by Karl E. Peterson

Click & Retrieve Source CODE!

Q PLAYING SYSTEM SOUNDS

I want my app to play the wave (WAV) file that has been assigned to a particular Windows sound, such as Asterisk or Exclamation. Do I make an API call to return the file for a particular sound?

A Yes and no. All system sounds are defined in the registry, where you can look them up. Utilities and applications—such as Themes and Office—can assign sounds; these definitions are also stored in the same registry location. You can find them all under the key, HKEY_CURRENT_USER\AppEvents\Schemes\Apps. You can take one API-intensive approach of enumerating this key and its subkeys, searching for the sound of interest.

If you're mainly interested in using "standard" sounds, you can use a shortcut. By calling the PlaySound API, with the proper flags and a registered sound alias, you can avoid registry spelunking. Here's a brief example:

```
Private Declare Function PlaySound Lib "winmm.dll" Alias "PlaySoundA" (ByVal _
   lpszName As String, ByVal hModule As Long, ByVal dwFlags As Long) As Long
Private Const SND_ASYNC = &H1
Private Const SND_ALIAS = &H10000
Call PlaySound("SystemAsterisk", 0, SND_ASYNC Or SND_ALIAS)
```

This code sequence plays the sound associated with the Asterisk, which has been more recently referred to as an informational message—the sound played when you call MsgBox with the vbInformation flag. Similarly, you could use "SystemExclamation" or any of the other defined aliases. How do you determine the correct alias strings? The easiest way is simply to open your registry editor and browse around a bit. If your application requests an alias that isn't registered or doesn't currently have a sound associated with it, the default sound plays.

Q PLAYING MIDI FILES

I'd like to play a given MIDI file when my application starts up and also in response to certain user actions. I'd prefer not to ship an extra control for something that should be fairly easy to implement with code, but I can't seem to get any of the examples I've seen to work. Do you have a simple way to play such files?

A I can certainly understand your frustration. The Multimedia Control Interface (MCI) is one of the most confusing and ill-documented, and the related Knowledge Base articles leave pieces unexplained. I've written a couple short subroutines you can use to start and stop MIDI tunes (see Listing 1). The basic idea is to "open" and "play" a file, using the mciSendString API. Assigning an alias to the file when it's first opened allows you to individually identify this tune in future calls to mciSendString. The alias is important, for example, because it gives you a method to stop the tune if necessary. Only one MIDI file can play at any given time, and unlike WAV files, attempting to start a new one doesn't stop the previous one. To stop a MIDI file, make two more calls to mciSendString to "stop" and "close" the specified alias.

Q EXPOSING IMPLEMENTED METHODS

When using Implements with an abstract class, all the properties and methods of the abstract class must be in the class doing the implementing. Do they all need to be public?

A The quick answer is no. In fact, you generally would *not* want all the implemented methods to be public. Objects with an interface reference should have full access to the implemented methods, but objects unaware of

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. Based in Vancouver, Washington, he's also an independent programming consultant specializing in ActiveX controls. In addition to contributing to various journals, Karl coauthored* Visual Basic 4 How-To *from Waite Group Press. Online, he's a Microsoft MVP, and a section leader in several* VBPJ *online forums. Find more of Karl's VB samples at http://www.mvps.org/vb.*

Ask the VB Pro provides you with free advice on programming obstacles, techniques, and ideas. Read more answers from our crack VB pros on the Web at http://www.inquiry.com/thevbpro. You can submit your questions, tips, or ideas on the site, or access a comprehensive database of previously answered questions.

the secondary interface shouldn't. A more typical design might be to explicitly expose as a standard member of your class the methods you also want to expose publicly. For example, say you want to implement the abstract class Foo:

```
Option Explicit
Public Sub Bar()
End Sub
```

In the class implementing this interface, enter code like this:

```
Implements Foo
Private Sub Foo_Bar()
   ' whatever
End Sub
Public Sub Bar()
   Call Foo_Bar
End Sub
```

This design hides the implemented interface from objects that are unaware of it, yet still provides identical functionality for methods that need to be universally available. In the end, it's just a matter of style, and entirely up to you.

Q USING AN OCX'S SECONDARY INTERFACE

It seems an ActiveX control cannot usefully implement an abstract interface class. If you want to reuse code, it seems you need to design a fully functional dependent object of the control rather than define an abstract class. Instead, I'd like to reuse an interface. I plan to write an entire family of 10 or so controls, all of which implement the properties and methods of one or more abstract interfaces, in a polymorphic fashion. Can I do this with controls?

A Yes, you can do it, but the route is foggy. If you take the classic approach, you generate a type-mismatch error. Using the example interface, Foo, and a control named Fubar1 that Implements that interface, this code won't work:

```
Dim iFoo as Foo
Set iFoo = Fubar1
```

Although this syntax works fine if Fubar1 is a class, it generates a type-mismatch error when Fubar1 is an ActiveX control. Because controls can expose a default property and it's unlikely that this property resolves to an object also implementing the secondary interface, you're nearly assured of an error in this situation. A relatively unknown property of all ActiveX controls solves this problem. Alter your code and you're in business:

```
Dim iFoo as Foo
Set iFoo = Fubar1.Object
```

The Object property returns a reference to the actual control object, not the default property exposed by that control. This is another reason why support for default properties is a bad idea, even if they do save a few keystrokes.

Q DEALING WITH LOCALIZATION PROBLEMS

My application uses LoadResString to load strings from a resource

VB5

```
Private Declare Function mciSendString Lib "winmm.dll" Alias "mciSendStringA" _
  (ByVal lpstrCommand As String, ByVal lpstrReturnString As String, ByVal _
  uReturnLength As Long, ByVal hwndCallback As Long) As Long
Public Function PlayMidiFile(ByVal FileName As String, Optional ByVal _
  Alias As String = "tune") As Boolean
  Dim nRet As Long
  Call StopMidiFile(Alias)
  If mciSendString("open " & FileName & " alias " & Alias, vbNullString, _
     0, 0) = 0 Then
     nRet = mciSendString("play " & Alias & " from 0", vbNullString, 0, 0)
     PlayMidiFile = (nRet = 0)
  End If
End Function
Public Sub StopMidiFile(Optional ByVal Alias As String = "tune")
  Call mciSendString("stop " & Alias, vbNullString, 0, 0)
  Call mciSendString("close " & Alias, vbNullString, 0, 0)
End Sub
```

LISTING 1 ***Add MIDI to Your App.*** *The Multimedia Control Interface (MCI) is one of the most confusing there is. However, these two simple routines let you play MIDI files from within your app. Just be sure to stop any running MIDI tune before attempting to start a new one. Also note that MIDI tunes do not stop automatically when your application terminates, so this is something you should check on exit.*

VB4 32-bit VB5

```
' Win32 Locale functions
Private Declare Function GetSystemDefaultLangID Lib _
  "kernel32" () As Integer
Private Declare Function GetLocaleInfo Lib "kernel32" _
  Alias "GetLocaleInfoA" (ByVal Locale As Long, ByVal _
  LCType As Long, ByVal lpLCData As String, ByVal _
  cchData As Long) As Long
' Localized name of language
Private Const LOCALE_SLANGUAGE = &H2
Public Function SystemLanguage() As String
  Dim LCID As Long
  Dim nRet As Long
  Dim buf As String
  ' Cache language identifier.
  LCID = GetSystemDefaultLangID
  ' Determine buffer requirement for language.
  nRet = GetLocaleInfo(LCID, LOCALE_SLANGUAGE, buf, 0)
  buf = Space$(nRet)
  ' Obtain language description.
  Call GetLocaleInfo(LCID, LOCALE_SLANGUAGE, buf, _
     Len(buf))
  SystemLanguage = TrimNull(buf)
End Function
Public Function TrimNull(ByVal StrIn As String) As
String
  Dim nul As Long
  ' Truncate input string at first null.
  ' If no nulls, perform ordinary Trim.
  nul = InStr(StrIn, vbNullChar)
  Select Case nul
     Case Is > 1
     TrimNull = Left(StrIn, nul - 1)
     Case 1
        TrimNull = ""
     Case 0
        TrimNull = Trim(StrIn)
  End Select
End Function
```

LISTING 2 ***Determine the Default System Language.*** *Use this code to determine which language your application should use for its user-interface elements. The code returned by GetSystemDefaultLangID would be the best value to test for resource determination. However, once you have that language ID, you can determine many other values for the system—such as the currency symbol, thousands and decimal separators, or date formats—by simply calling GetLocaleInfo with the appropriate constant.*

VB4 32-bit VB5

```
Private Declare Function FormatMessage Lib "kernel32" _
  Alias "FormatMessageA" (ByVal dwFlags As Long, _
  lpSource As Any, ByVal dwMessageId As Long, ByVal _
  dwLanguageId As Long, ByVal lpBuffer As String, _
  ByVal nSize As Long, Arguments As Long) As Long
Private Const FORMAT_MESSAGE_FROM_SYSTEM As Long = _
  &H1000
Private Function ApiErrorText(ByVal ErrNum As Long) As _
  String
  Dim msg As String
  Dim nRet As Long

  msg = Space$(1024)
  nRet = FormatMessage(FORMAT_MESSAGE_FROM_SYSTEM, _
     ByVal 0&, ErrNum, 0&, msg, Len(msg), ByVal 0&)
  If nRet Then
     ApiErrorText = Left$(msg, nRet)
  Else
     ApiErrorText = "Error (" & ErrNum & ") not defined."
  End If
End Function
```

LISTING 3 ***Translate Error Code to Plain Text.*** *Calling FormatMessage with the appropriate constant and an API error code returns a plain text description of the indicated error. This utility, suitable for your desktop, lists all available API error messages, and optionally copies this routine to the Windows clipboard for inclusion in your applications. You can download this from the Tools page of my Web site at http://www.mvps.org/vb.*

file. The resource file is in English and Norwegian. When I set the regional settings to English, I expect the application to use the English part of the resource file. And when I switch the computer to Norwegian, I expect it to use the Norwegian part of the resource file. I always reboot the computer when I switch languages, and the application is then recompiled to an EXE file. Yet the resource strings used never vary. Why?

A The simple answer is that there's nothing automagic about it. My hunch is you've used the *VB Resource Editor*, a tool found on Microsoft's Web site, to build your resource file. Unfortunately, the help file for this tool implies that multiple string tables are the answer to your problem. What isn't mentioned is that this approach works only in Windows NT, and fails miserably under both Windows 95 and 98.

Instead, you need to either supply your resources in multiple DLLs or build them using offsets for each language. If you want your application to use English resources, you need to specify that. Similarly, if you want it to use Norwegian resources, specify those. So the question really becomes, which language should you use? A pair of API calls can answer this for you (see Listing 2).

Calling GetSystemDefaultLangID returns an identifier for the current language. As a side note, this is one of the few Win32 APIs you should declare as returning an Integer. If you declare the function to return a Long, the upper two bytes in the return value are meaningless and could be just about anything. You then need to manually split out the lower two bytes to get the data you're after. In this case, it's simpler to let VB split the lower two bytes for you by declaring the return As Integer.

Once you obtain the language identifier, pass that to GetLocaleInfo twice. This API returns results within a string buffer. If the buffer you supply isn't long enough, the return is the length of the required buffer. Calling GetLocaleInfo with zero for the buffer length supplies you with the required information to properly size your buffer. The second call to GetLocaleInfo returns a localized textual description of the current system language.

The text description might go farther than you want, and you would probably be better off selecting resources based on the return value of GetSystemDefaultLangID. But it's interesting just how much more information is available once you have that ID! For more details, see the SDK docs on GetLocaleInfo.

Q LOOK UP THE ERROR CODE!

I'm trying to register an ActiveX DLL using regsvr32, but I must be doing something wrong. At the command line, I type regsvr32 mydllname.dll, but I get this message:

```
LoadLibrary ("mydllname.dll") _
   failed GetLastError 0x00000485
```

The DLL is out there—I also tried unregistering an earlier one using /u, but that doesn't work either—and regsvr32.exe is in the c:\windows\system directory. What should I try next?

A When presented with an error code, the first thing you should do is look it up! In this case, the returned code, &H485, translates to "One of the library files needed to run this application cannot be found." Given only this to work with, my hunch is you didn't do a proper install. One or more dependencies cannot be found. If you simply copied the DLL to this machine, my advice is to build a setup program and use that. If you used a setup program and still missed dependencies, more detective work is called for. Sometimes, just opening a DLL in QuickView—in Explorer, right-click on the DLL and select Quick View—reveals missing files. Otherwise, the most thorough tool available is the Dependency Walker (Depends.exe), which ships with the Platform SDK.

For future reference, you can easily translate most API error codes into meaningful descriptions by calling the FormatMessage API, passing the desired error code (see Listing 3). Never call the GetLastError API when you suspect an API error. Instead, query the LastDllError property of VB's Err object. Some folks memorize some of the more common error codes, but nothing beats having the actual description handy! ☒

Code Online

You can find all the code published in this issue of* VBPJ *on The Development Exchange (DevX) at http://www.vbpj.com. For details, please see "Get Extra Code in DevX's Premier Club" in Letters to the Editor.

Sounds Like Trouble
Locator+ Codes

Listings for the entire issue (free Registered Level): VBPJ0898

✪Listings for this article only, plus complete source to a demo app that retrieves the names and plays all registered system sounds in addition to the MIDI examples shown in Listing 1 (subscriber Premier Level): AP0898