Thunk Before You Port

Click & Retrieve Source CODE!

BY STEVE JACKSON

Flat thunking techniques for calling 16-bit DLLs from 32-bit Win95 applications save time and money.

In the well-publicized push to move custom development to Windows 95 and VB4, an important technical trade-off has been overlooked by many programmers and the press. The move to a flat memory model in Windows 95 comes at a price: a 32-bit VB program cannot call a 16-bit DLL.

Consequently, developers face the daunting task of rewriting 16-bit DLLs as 32-bit DLLs. If you're on tight project deadlines like I am, you should seriously consider thunking as a technique for saving time and salvaging the hard work you've put into 16-bit DLLs. What's more, if you have third-party 16-bit DLLs without the source code, you can't rewrite them. Your options are to purchase 32-bit upgrades, or use the thunking techniques I'll describe to tap 16-bit third-party DLLs from 32-bit Win95 apps.

Here's the crux of the 16-/32-bit DLL dilemma. The way Windows 95 loads a DLL and passes function arguments is quite different in 16-bit and 32-bit modes. The 32-bit code uses data pointer types, stacks, CPU register settings, and function-calling conventions that differ from 16-bit code.

Porting a 16-bit DLL to 32-bit requires code changes to function declarations, pointer definitions, and memory-handling routines that will stretch your project deadlines if you attempt a rewrite. If you try to call your 16-bit DLL from a 32-bit VB program, you'll quickly find that VB (or any 32-bit Windows program) fails with an "Error loading DLL" message.

However, you can salvage the hard work you've put into 16-bit DLLs by calling them from your 32-bit applications using a Windows 95 technique called flat thunking. Thunking allows your 32-bit VB app to call a 32-bit DLL which in turn calls your 16-bit DLL through the thunking layer—a kind of hyperspace leap where code reaches through a 32-bit thunking layer to grab 16-bit DLL functions.

I'll show you how to create a simple 16-bit DLL, and then create a 32-bit DLL that calls the 16-bit functions using thunking. I'll explain all the code you need, saving you days' or weeks' worth of searching for the right tools. Along the way I'll correct a few errors and omissions in the Microsoft instructions. The project requires C compilers, assembly language, several utilities, and of course VB4.

Flat thunking converts flat 32-bit parameter memory addresses of Windows 95 to the 16-bit segment/offset equivalent of Windows 3.x. Thunking also does special fixes to the DLL loading code to create a 16-bit stack, switch stacks on the fly, and translate return values.

Unfortunately, Windows NT uses a different thunking model called generic thunking that allows 16-bit code to call 32-bit code, but generic thunking does not allow 32-bit code to call 16-bit code. Generic thunking loads a 32-bit DLL into the Virtual DOS Machine (VDM) where NT runs 16-bit applications. Flat thunking is not supported in Windows NT—it works only in Windows 95.

For information on the various thunk permutations, read the Knowledge Base article Q125710 on the MSDN Level 1 CD, "Types of Thunking Available in Win32 Platforms." Also read the Microsoft MSDN article Q125715 on the MSDN Starter Kit that comes with VB4, "Calling 16-bit Code from Win32-based Apps in Windows 95." While it's a good

```
//MYDLL16.C
#include <windows.h>
//  The standard 16 bit DLL opening
int FAR PASCAL LibMain (HANDLE hInstance, WORD
     wDataSeg, WORD wHeapSize, LPSTR lpszCmdLine)
{
  if (wHeapSize > 0)
    UnlockData (0) ;
  return 1 ;
}
//  Later add Thunking code here

WORD __export FAR PASCAL AddNums16(WORD a, WORD b)
{
  return (a + b);
}
void __export FAR PASCAL Beep16(void)
{
  MessageBeep(MB_OK);
}
void __export FAR PASCAL Ucase16(LPSTR pString)
{
  AnsiUpperBuff(pString, (UINT) lstrlen(pString));
}
```

LISTING 1 ***Build a Simple 16-Bit DLL.*** *This 16-bit DLL contains a few simple 16-bit functions called from the 32-bit thunking DLL. Thunking code will be added right after the standard LibMain entry routine. The three functions illustrate different types of parameter usage that must be handled by the thunk compiler.*

Steve Jackson works at a Southern California aerospace company, developing network-based applications in VB and C. He is a section leader on the VBPJ *CompuServe forum and can be reached at 72040,1640. Code for this article is available from the Magazine Library of the* VBPJ *Forum.*

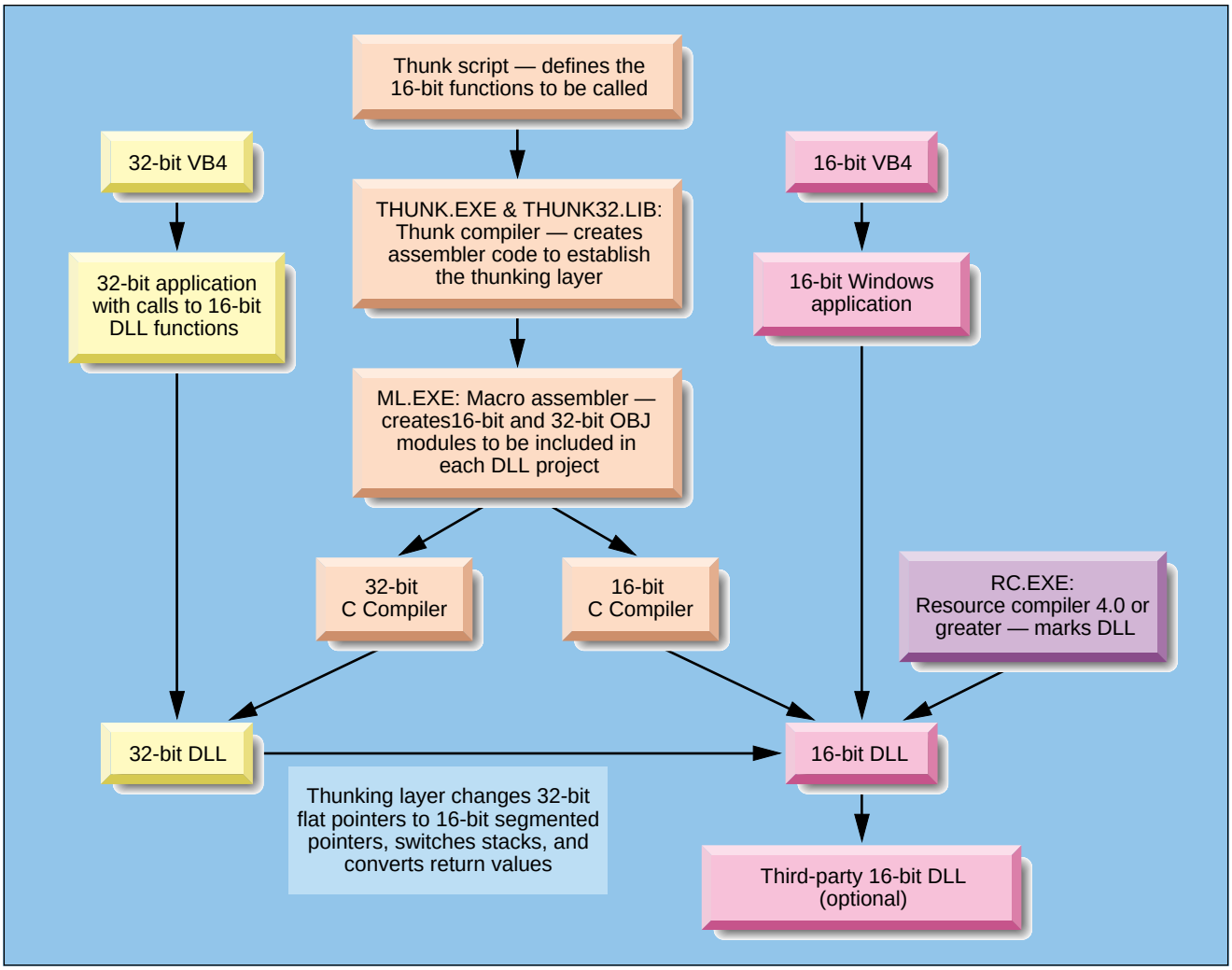

FIGURE 1 ***Assemble Your Tools.*** *You'll need 32-bit and 16-bit C compilers, a thunk compiler, MASM, and a number of utilities. The middle column shows the tools and process for creating the thunking layer. The thunk script is an ASCII text file placed into the thunk compiler. Assembler code produced from the thunk compiler is placed in the assembler, which produces OBJ files included in the C projects that produce the thunking layer.*

tutorial, it contains a number of errors and omits an important declaration. For example, Q125715 leaves out the semicolon in some code (such as: enablemapdirect3216=true;) but I noticed that other statements ended with C-style semicolons, so adding one solved the problem.

Before starting a thunking project you must have the right tools. It took me longer to determine which tools and utilities I needed, to find them, and to figure out when to use them, than it took to create the code. Of course the results of my search make your job much easier.

To compile the code in this article you'll need 16-bit and 32-bit C compilers that are compatible with Microsoft OBJ and LIB formats, and a number of utility programs to create the thunking DLLs (see Figure 1). I spent quite a bit of time searching the MSDN CDs while posting and re-posting CompuServe messages looking for the right assembler and resource compiler.

Because you are creating 32-VB apps, you'll need 32-bit tools for thunking. Obviously you need Windows 95 and a 32-bit C compiler. I used Microsoft Visual C++ 2.0 for the 32-bit compiler, and Visual C++ 1.51 for the 16-bit compiler, which both come on the same CD. You do not need the subscription versions or Visual C++ 4.0 to create the DLL thunking code I'll discuss. The other major C compilers on the market should compile the code properly, provided they are compatible with the THUNK32.LIB file and can assemble the code produced by the thunk compiler.

The thunk compiler is on the Win32 SDK CD, which comes in the MSDN Level 2 set of CDs. The Win32 SDK is not on the single CD Level 1 MSDN. Also, you will need the 16-bit resource compiler from the Win32 SDK, and the Macro Assembler from the Device Driver Kit CD in the MSDN Level 2 CD. I'll explain the directory names and file names as I discuss their use.

For the sake of illustrating thunking techniques, I'll discuss only the 32-bit-to-16-bit thunk in this article. However, I have added a 16-bit-to-32-bit thunk example in the code available for download on the *VBPJ* Forum on CompuServe.

CREATE YOUR 16-BIT DLL

When creating a thunking system it's best to start simple. To illustrate thunking techniques, I created a simple 16-bit DLL that doesn't have a thunking layer. It's important to ensure that the 16-bit functions work properly before adding the thunking layer.

I created this simple 16-bit DLL purely to demonstrate thunking techniques (see Listing 1). To build this DLL—called MYDLL16.C—start 16-bit Visual C++ 1.5x, start a new project, and select DLL from the project type list. Be sure the Microsoft Foundation Classes check box is off. Accept all defaults including the large memory model default, and let Visual C++

create the DEF file. Later you'll need to modify the DEF file to add thunking.

MYDLL16.C consists of four functions. LibMain() is the standard entry point found in all 16-bit DLL programs. The other three functions are trivial ones that will be called from 16-bit VB, and later from the 32-bit DLL after thunking is added. Specifically, Beep16 simply causes the speaker to beep, AddNums16 adds two integers together and returns the result, and Ucase16 converts a string to upper case. These functions demonstrate how thunking works with different parameter types.

To test the 16-bit DLL, start the 16-bit version of VB and open a new project. Remove the Form1 file and insert a new module to create a simple formless project that consists of declarations and the code in Sub Main (see Listing 2). Notice the use of the #If Win32 statement to change the DLL declarations for 16- and 32-bit modes. The 16-bit version will call the 16-bit DLL, and the 32-bit version will call the 32-bit DLL I'll describe later.

After compiling, move the compiled MYDLL16.DLL into your WINDOWS\SYSTEM directory. Alternatively, you can fully qualify the DLL path in the Declaration statement, but this will cause

VB4

```
Attribute VB_Name = "Module1"
Option Explicit
#If Win32 Then
   Declare Sub Beep16 Lib "MYDLL32.DLL" ()
   Declare Function AddNums16 Lib "MYDLL32.DLL" (ByVal a _
     As Integer, ByVal b As Integer) As Integer
   Declare Sub Ucase16 Lib "MYDLL32.DLL" _
     (ByVal s As String)
   Declare Function AddNums32 Lib "MYDLL32.DLL" (ByVal a _
     AsLong, ByVal b AsLong) As Long
#Else
   Declare Sub Beep16 Lib "MYDLL16.DLL" ()
   Declare Function AddNums16 Lib "MYDLL16.DLL" _
     (ByVal a As Integer, ByVal b As Integer) As Integer
   Declare Sub Ucase16 Lib "MYDLL16.DLL" _
     (ByVal s As String)
#End If

Sub Main()
   Dim x As Integer, s As String
   #If Win32 Then
     MsgBox "We are in 32 bit mode."
     '  Call 32 bit DLL
     x = AddNums32(2, 3)
     MsgBox "2 + 3 = " & Str$(x)
   #Else
     MsgBox "We are in 16 bit mode."
   #End If
   '  Call 16 bit DLL
   Call Beep16
   x = AddNums16(3, 4)
   MsgBox "3 + 4 = " & Str$(x)
   s = "My String"
   MsgBox "Before call: " & s
   Call Ucase16(s)
   MsgBox "After call: " & s
End Sub
```

LISTING 2 ***Call 16-Bit Functions.*** *This VB code calls 16-bit DLL functions from either 16-bit or 32-bit VB4. The #If Win32 statement changes the declarations so 16-bit VB programs will call the 16-bit DLLs, but 32-bit programs will call the same functions in the 32-bit DLL with thunking added. Hence the same VB code can be used for both 16-bit and 32-bit projects.*

VB3 VB4

User Tip

CLOSING ALL OPEN FILES IN AN APPLICATION

Use the Close statement without a file number and Visual Basic will close all files opened by your application.

—Douglas Haynes,
received on CompuServe

SEND YOUR TIP

If it's cool and we publish it, we'll pay you $25. If it includes code, limit code length to 10 lines if possible. Be sure to include a clear explanation of what it does and why it is useful. Send to 74774.305@compuserve.com or Fawcette Technical Publications, 209 Hamilton Ave., Palo Alto, CA, USA, 94301-2500.

problems later when the 32-bit DLL needs to find the 16-bit DLL, so I recommend that for your first thunking effort you copy all the DLLs to WINDOWS\SYSTEM.

COMPILE THE THUNK SCRIPT

Now that the 16-bit functions are finished, create a script for the thunk compiler. The thunk compiler uses script statements to generate a file containing assembler code functions. The assembler language functions are the core of thunking—they do the segment and stack fixes that let the 32-bit DLL call 16-bit functions. Although the assembler code could theoretically be coded by hand, it is quite complex and best left to the thunk compiler.

The parameters and return types of each 16-bit function are defined in the thunk script so the thunk compiler can generate the appropriate code for each function. Create a file with these script statements using Notepad or any ASCII text editor, and save it as 32TO16.THK:

```
enablemapdirect3216 = true;
typedef char *LPSTR;
void Beep16(void)
{
}
int AddNums16 (int a, int b)
{
}
void Ucase16 (LPSTR lpString)
{
   lpString = inout;
}
```

The first statement, enablemapdirect3216 = true;, indicates mapping from 32-bit to 16-bit. The typedef statement creates a long string pointer type that is used in the Ucase16() parameter list. The functions are declared C-style starting with return type, then function name, argument list, and a pair of braces.

Return type void indicates there is no return value—a SUB in Basic. The lpString=inout; statement tells the thunk compiler that data in the string will be passed to and from the 16-bit DLL, so the segment/offset conversions must occur both ways.

The 32-bit thunk compiler program THUNK.EXE can be found on the Win32 SDK CD of MSDN Level 2, in directory WIN32SDK\MSTOOLS\BIN\I386. Directory names sometimes change in SDKs, so if you have trouble finding this directory, do a file search and be sure you select the 32-bit Intel version that is usually in an I386 subdirectory.

While you have the Win32 SDK CD loaded, you should also copy THUNK32.LIB because you will need it later in your 32-bit DLL project. THUNK32.LIB is in the\WIN32SDK\MSTOOLS\LIB\I386 directory. To compile the thunk script, open a DOS Prompt window (yes, it's still there in Windows 95) and run the thunk compiler with this command line:

```
thunk -t thk 32to16.thk -o 32to16.asm
```

The -t thk option will prefix all the assembler routines with a thk_ prefix. The -o 32to16.asm option (lowercase "o") indicates the output file name for the assembly language routines.

ASSEMBLE THE THUNK OUTPUT

Now that you have an assembler language output file, you need to assemble the file into OBJ object code modules that can be included in each DLL project. Assemble the single ASM file twice with different command line switches to produce separate 32-bit and 16-bit OBJ files. You must use Microsoft Macro Assembler (MASM) 6.11 or later—MASM 5.0 won't do. When I got to this step, I found that, lo and behold, Visual C++ does not come with a separate assem-

bler. And an inline assembler won't cut it for this either.

Time to dig out the MSDN CD set again. Fortunately, MASM 6.11 can be found on the Device Driver Kit CD as ML.EXE in directory \DDK\BIN\I386\FREE. Again, directory names may change.

To confirm the version number, simply execute ml at the DOS prompt with no command-line arguments. I created a simple two-line BAT file to execute the assembler twice so the assembler will produce the THK32.OBJ and THK16.OBJ files that I added to the DLL projects (the underscore should be removed and the two lines combined into one):

```
ml /DIS_32 /c /W3 /nologo /coff /Fo  thk32.obj 32to16.asm
ml /DIS_16 /c /W3 /nologo /Fo _ thk16.obj 32to16.asm
```

Now create the 32-bit DLL called MYDLL32.C (see Listing 3). It's quite different from a 16-bit DLL. The LibMain entry point is replaced by DllMain. The 32-bit standard entry point contains a switch statement that checks whether the DLL is being loaded or unloaded for a new process or a new thread within a process.

For this program, simply break out of the switch in all cases. The DllMain entry code starts with a call to thk_ThunkCon-nect32(). This assembly-language function was produced by the thunk compiler. It loads the 16-bit DLL into memory and calls the corresponding 16-bit thunk code to establish the flat thunking layer.

If the 16-bit DLL cannot be loaded, the call returns a non-zero code, and MYDLL32 produces a beep and returns FALSE so the 32-bit DLL will not load. When this happens, VB reports it was unable to load the 32-bit DLL but does not indicate why. I added MessageBeep() so I know the 32-bit DLL loaded, but I get a beep if there is a problem loading the 16-bit DLL.

Function thk_ThunkConnect32() starts with the thk_ prefix specified on the thunk compiler command line -t switch. A declaration has been added for the function, with an "extern" keyword to tell the linker that this function can be found in an external module. This important declaration is missing from the MSDN Q125715 article, and if you leave it out the compile will fail with an "unresolved external reference" message. I also added a simple 32-bit function, AddNums32(), to illustrate the Win32 __declspec(dllexport) DWORD WINAPI keywords for exported functions.

You also need to modify the EXPORTS section of the default MYDLL32.DEF file created by Visual C++ to add thk_ThunkData32 and all the 16-bit functions your code will call. The ASM code produced by the thunk compiler contains 32-bit stubs with exported names that match the real 16-bit functions. These stubs do fix-up code, and then call the real 16-bit function. The new EXPORT section in MYDLL32.DEF is:

```
EXPORTS
   thk_ThunkData32
   Beep16
   AddNums16
   Ucase16
   AddNums32
```

To compile the 32-bit DLL, start 32-bit Visual C++ 2.0 or later and create a new project (File, New, Project) and choose project type DLL. Add file MYDLL32.C from Listing 3, add THK32.OBJ, which you created with the thunk compiler and assembler, and add THUNK32.LIB, which you copied from the Win32 SDK.

After your compile is complete, MYDLL32.DLL will be in a subdirectory named WINDEBUG created by Visual C++ under your project directory. Move this DLL to your WINDOWS\SYSTEM directory. If you get any "unresolved external" messages, recheck all the function names, recheck your thunk script, recheck that the OBJ and LIB files were properly added to the project, and recheck the thk_ThunkConnect32() function declaration.

MODIFY THE 16-BIT DLL

You must add code to MYDLL16.C to complete the 16-bit thunking layer (see Listing 4). DllEntryPoint calls function thk_ThunkConnect16() created by the thunk compiler. You can add the DllEntryPoint declaration and function anywhere to MYDLL16.C—a good place is right after LibMain. Notice that the standard LibMain function is unchanged. No other changes are needed in the C code, although some changes are required in the

```
//  declaration for function from thunk compiler
extern BOOL PASCAL thk_ThunkConnect16(LPSTR, LPSTR,
     WORD, DWORD);
//  required for thunking
BOOL FAR PASCAL __export DllEntryPoint(DWORD dwReason,
     WORD hInst, WORD wDS, WORD wHeapSize,
     DWORD dwReserved, WORD dwReserved2)
{
  if ( !thk_ThunkConnect16("MYDLL16.DLL",
       "MYDLL32.DLL", hInst, dwReason))
  {
    return FALSE;
  }
  return TRUE;
}
```

LISTING 4 ***Create 16-Bit Thunking.*** *Add this code to any 16-bit DLL that will be called through the flat thunking layer. DllEntryPoint is called by the 32-bit thunking layer. The thk_ThunkConnect16 function completes the link to and from the 32-bit DLL.*

```
// MYDLL32.C
#include <windows.h>
extern BOOL WINAPI thk_ThunkConnect32(LPSTR, LPSTR,
HINSTANCE, DWORD);
__declspec(dllexport) DWORD WINAPI AddNums32 (DWORD,
DWORD);
//  The standard 32 bit opening, with thunking added
BOOL WINAPI DllMain (HINSTANCE hDLL, DWORD dwReason,
LPVOID lpReserved)
{
  if (!thk_ThunkConnect32 ("MYDLL16.DLL",
    "MYDLL32.DLL",
    hDLL,
    dwReason))
  {
    MessageBeep(MB_ICONEXCLAMATION);
    return FALSE;
  } // end if
  switch (dwReason)
  {
    case DLL_PROCESS_ATTACH:
      break;
    case DLL_THREAD_ATTACH:
      break;
    case DLL_THREAD_DETACH:
      break;
    case DLL_PROCESS_DETACH:
      break;
  }  // end switch
return TRUE;
}
__declspec(dllexport) DWORD WINAPI AddNums32
    (DWORD a, DWORD b)
{
  return (a + b);
}
```

LISTING 3 ***Add a 32-Bit Thunking Layer.*** *Use this 32-bit DLL to call the 16-bit functions through the thunking stubs linked in from the thunk compiler. The thk_ThunkConnect32 function creates the thunking link to the 16-bit DLL. If this critical function call fails, the DLL beeps and fails to load.*

MYDLL16.DEF file. The EXPORTS section must be modified, and an IMPORTS section must be added. This code is identical for adding thunking to all 16-bit DLLs:

```
IMPORTS
   C16ThkSL01     = KERNEL.631
   ThunkConnect16 = KERNEL.651

EXPORTS
   WEP PRIVATE
   THK_THUNKDATA16 @1 RESIDENTNAME
   DllEntryPoint   @2 RESIDENTNAME
```

Finally, you must modify the 16-bit DLL project (click on Project, then Edit) to add THK16.OBJ that was produced by the thunk compiler and assembler. Compile MYDLL16 and you should get no errors and no warnings.

By now you're probably more than ready to test your thunking calls. The last step should take you five minutes (although it took me several days to find the right resource compiler necessary to complete this step). You must run the resource compiler program RC.EXE to mark the 16-bit DLL version as 4.0. Otherwise, the 32-bit DLL will not be able to load it. The trick is to find the right resource compiler.

The 16-bit and 32-bit resource compilers that come with VB4 won't do—you need a resource compiler version 4.0 or later that supports the "-40" command-line argument. The Win32 SDK CD contains five different copies of RC.EXE—you need the 16-bit version, which is located in \WIN32SDK\MSTOOLS\BINW16.

To confirm the version, run RC.EXE from the DOS prompt with no arguments, or with /? to be sure it supports the "-40" command- line argument. Then run it to mark your 16-bit DLL. You must repeat this step every time you recompile the 16-bit DLL:

```
rc -40 MYDLL16.DLL
```

Be sure to move both compiled DLLs to your WINDOWS\SYSTEM directory, or MYDLL32.DLL may have problems trying to load MYDLL16.DLL.

It's time to test! Start 32-bit VB4 and open the project created using Listing 2. Thanks to the #If Win32 statement, the code now calls the 32-bit DLL, and also calls an extra function, AddNums32(), which is in the 32-bit DLL only. VB loads both DLLs on the first call to AddNums32. If MYDLL32 is unable to load MYDLL16, it will fail at this point.

If you hear the beep, you know that MYDLL32 loaded, but was unable to load MYDLL16. If MYDLL16 fails to load, be sure you ran RC.EXE to mark it after your last compile, recheck the code to thk_ThunkConnect16(), and recheck all the DEF files in both DLLs, making sure all your 16-bit functions are in the EXPORTS section of the 32-bit DLL.

A good source of information for debugging thunks is the Microsoft Knowledge Base article Q133722 (GO MSKB on CompuServe). After VB loads a 32-bit DLL it only unloads it when the project is closed or you modify the Declare statement. If you recompile the 32-bit DLL, you won't be able to replace the DLL file until you close and reopen the VB project.

Well, there you have it. You can now call your 16-bit DLL from a 32-bit DLL. And here's one last tip. What if you don't have the source code for the 16-bit DLL you're trying to call? You can't add the needed thunk connect function, or link in the OBJ files, but you could write a 32-bit DLL that thunks to your own 16-bit DLL that in turn calls the third-party 16-bit DLL with normal 16-bit calls. But be careful with those pointers.

