When you need capabilities that go beyond the core language and controls provided with Visual Basic, you can make direct calls to procedures contained in *dynamic-link libraries* (DLLs). By calling procedures in DLLs, you can access the thousands of procedures that form the backbone of the Microsoft Windows operating system, as well as routines written in other languages.

As their name suggests, DLLs are libraries of procedures that applications can link to and use at run time rather than link to statically at compile time. This means that the libraries can be updated independently of the application, and many applications can share a single DLL. Microsoft Windows itself is comprised of DLLs, and other applications call the procedures within these libraries to display windows and graphics, manage memory, or perform other tasks. These procedures are sometimes referred to as the Windows API, or application programming interface.

Note If you are using the Control Creation Edition of Visual Basic, some of the material covered in this document may not be applicable. With the full editions of Visual Basic you have the ability to create applications, ActiveX documents, and other types of components. Although some of the terminology may relate to application specific objects such as forms, in most cases the underlying concepts also apply to ActiveX control creation.

1

DLLs or Automation?

Another way to bring more power into Visual Basic is through Automation (formerly called OLE Automation). Using Automation is simpler than calling routines in a DLL, and it doesn't create the same level of risk that you'll hit when going straight to the Windows API. By using Automation, you can get programmatic access to a wide range of objects exposed by external applications.

Contents

- Using a DLL Procedure in Your Application
- Accessing the Microsoft Windows API
- Declaring a DLL Procedure
- Passing Strings to a DLL Procedure
- Passing Arrays to a DLL Procedure
- Passing User-Defined Types to a DLL Procedure
- Passing Function Pointers to DLL Procedures and Type Libraries
- Passing Other Types of Information to a DLL Procedure
- Converting C Declarations to Visual Basic

2

Using a DLL Procedure in Your Application

Because DLL procedures reside in files that are external to your Visual Basic application, you must specify where the procedures are located and identify the arguments with which they should be called. You provide this information with the Declare statement. Once you have declared a DLL procedure, you can use it in your code just like a native Visual Basic procedure.

Important When you call any DLLs directly from Visual Basic, you lose the built-in safety features of the Visual Basic environment. This means that you increase the risk of system failure while testing or debugging your code. To minimize the risk, you need to pay close attention to how you declare DLL procedures, pass arguments, and specify types. In all cases, save your work frequently. Calling DLLs offers you exceptional power, but it can be less forgiving than other types of programming tasks.

3

In the following example, we'll show how to call a procedure from the Windows API. The function we'll call, SetWindowText, changes the caption on a form. While in practice, you would always change a caption by using Visual Basic's Caption property, this example offers a simple model of declaring and calling a procedure.

Declaring a DLL Procedure

The first step is to declare the procedure in the Declarations section of a module:

```
Private Declare Function SetWindowText Lib "user32" _
Alias "SetWindowTextA" (ByVal hwnd As Long, _
ByVal lpString As String) As Long
```

4

You can find the exact syntax for a procedure by using the API Viewer application, or by searching the Win32api.txt file. If you place the Declare in a Form or Class module, you must precede it with the Private keyword. You declare a DLL procedure only once per project; you can then call it any number of times.

For More Information For more information on declare statements, see the topic "Declaring a DLL Procedure" later in this chapter.

Calling a DLL Procedure

After the function is declared, you call it just as you would a standard Visual Basic function. Here, the procedure has been attached to the Form Load event:

```
Private Sub Form_Load()
    SetWindowText Form1.hWnd, "Welcome to VB"
End Sub
```

5

When this code is run, the function first uses the hWnd property to identify the window where you want to change the caption (Form1.hWnd), then changes the text of that caption to "Welcome to VB."

Remember that Visual Basic can't verify that you are passing correct values to a DLL procedure. If you pass incorrect values, the procedure may fail, which may cause your Visual Basic application to stop. You'll then have to reload and restart your application. Take care when experimenting with DLL procedures and save your work often.

Note Very few API calls recognize the default Variant data type. Your API calls will be much more robust if you declare variables of specific types and use Option Explicit.

6

Accessing the Microsoft Windows API

You can gain access to the Windows API (or other outside DLLs) by declaring the external procedures within your Visual Basic application. After you declare a procedure, you can use it like any other language feature in the product.

The most commonly used set of external procedures are those that make up Microsoft Windows itself. The Windows API contains thousands of functions, subs, types, and constants that you can declare and use in your projects. These procedures are written in the C language, however, so they must be declared before you can use them with Visual Basic. The declarations for DLL procedures can become fairly complex. While you can translate these yourself, the easiest way to access the Windows API is by using the predefined declares included with Visual Basic.

The file Win32api.txt, located in the \Winapi subdirectory of the main Visual Basic directory, contains declarations for many of the Windows API procedures commonly used in Visual Basic. To use a function, type, or other feature from this file, simply copy it to your Visual Basic module. You can view and copy procedures from Win32api.txt by using the API Viewer application, or by loading the file in any text editor.

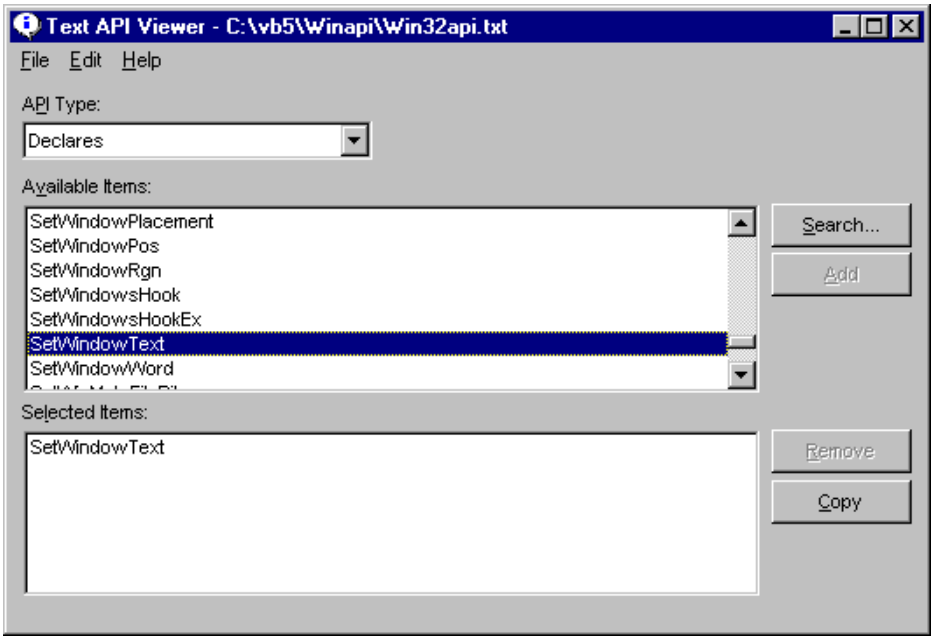
Note The Windows API contains a vast amount of code. To find reference information on the procedures and other details included in this API set, refer to the Win32 SDK, included in the \Tools directory of your Visual Basic CD.

7

Using the API Viewer Application

The API Viewer application enables you to browse through the declares, constants, and types included in any text file or Microsoft Jet database. After you find the procedure you want, you can copy the code to the Clipboard and paste it into your Visual Basic application.

Figure 1.1 The API Viewer application



1

To load the Win32api.txt file, choose Load Text File from the File menu. After the file is loaded, you can view entries in the Available Items list box by selecting Declares, Constants, or Types in the API Type drop-down list box. To search for specific items in the file, use the Search button.

To load a Jet database API file, choose Load Database File from the File menu. You can then search for specific items in the database by typing the first letter of the item you want to find.

Adding Procedures to Your Visual Basic Code

Once you have found a procedure you want, choose the Add button to add the item to the Selected Items box. You can add as many items as you like. To remove an entry from the Selected Items box, select the item, then choose the Remove button.

To copy the items from the Selected Items list box to the Clipboard, choose the Copy button. All of the items in the list will be copied. You can then open your Visual Basic project and go to the module in which you want to place the API information. Position the insertion point where you want to paste the declarations, constants, and/or types, and then choose Paste from the Edit menu.

Converting Text Files to Jet Database Files

To optimize speed, you can convert the Win32api.txt file into a Jet database file, because it is much faster to display the list when opening a database than when opening a text file.

To convert the text file, start the API Viewer application, then choose the Load Text File command from the File menu and open the .txt file. A message will appear asking if you want to convert the .txt file to a database file. Choose Yes to confirm the conversion. If you choose No, you can still convert the file later by choosing the Convert Text to Database command from the File menu.

Loading an API File Automatically from the Command Line

You can specify a text or database file on the command line for Apilod32.exe so that the file is automatically loaded when you start API Viewer. Use the following syntax to load the file you choose when you start the API Viewer application:

Apilod32.exe {/T|/D} *filename*

8

Argument	Description
/T	API Viewer will load the file as a text file. /T must be uppercase.
/D	API Viewer will load the file as a database file. /D must be uppercase.
filename	The path of the file you want to open.

9

There must be a space between /T or /D and the *filename* argument. An error message will be displayed if the file is not found. If you specify a file that is not a database or text file, an error message will be displayed when you try to load the file.

For example, you might enter the following command line for API Viewer in the Windows Program Item Properties dialog box (by choosing Properties from the File menu):

C:\VB\Winapi\Apilod32.exe /D C:\VB\Winapi\Win32api.mdb

10

Viewing the Win32api.txt file with a Text Editor

You can also load the Win32api.txt file in a text editor, such as Microsoft Word or WordPad, to locate the procedures you want to use. Again, you just copy the procedures from the file to a Visual Basic module to use them in your application.

Tip Don't load the Win32api.txt file into a module. This is a large file, and it will consume a lot of memory in your application. You will generally use only a handful of declarations in your code, so selectively copying the declarations you need is much more efficient.

11

Using Procedures from Other Sources

If you are attempting to call a procedure in a DLL that is not part of the operating system, you must determine the proper declaration for it. The topic "Declaring a DLL Procedure" explains the syntax of the Declare statement in detail.

Declaring a DLL Procedure

Even though Visual Basic provides a broad set of predefined declares in the Win32api.txt file, sooner or later you'll want to know how to write them yourself. You might want to access procedures from DLLs written in other languages, for example, or rewrite Visual Basic's predefined declares to fit your own requirements.

To declare a DLL procedure, you add a Declare statement to the Declarations section of the code window. If the procedure returns a value, write the declare as a Function:

Declare Function *publicname* **Lib** "*libname*" **[Alias** "*alias*"**] [([[ByVal]** *variable* **[As** *type***] [,[ByVal]** *variable* **[As** *type***]]...])] As Type**

12

If a procedure does not return a value, write the declare as a Sub:

Declare Sub *publicname* **Lib** "*libname*" **[Alias** "*alias*"**] [([[ByVal]** *variable* **[As** *type***] [,[ByVal]** *variable* **[As** *type***]]...])]**

13

DLL procedures declared in standard modules are public by default and can be called from anywhere in your application. DLL procedures declared in any other type of module are private to that module, and you must identify them as such by preceding the declaration with the Private keyword.

Procedure names are case-sensitive in 32-bit versions of Visual Basic. In previous, 16-bit versions, procedure names were not case-sensitive.

Specifying the Library

The Lib clause in the Declare statement tells Visual Basic where to find the .dll file that contains the procedure. When you're referencing one of the core Windows libraries (User32, Kernel32, or GDI32), you don't need to include the file name extension:

```
Declare Function GetTickCount Lib "kernel32" Alias _
"GetTickCount" () As Long
```

14

For other DLLs, the Lib clause is a file specification that can include a path:

```
Declare Function lzCopy Lib "c:\windows\lzexpand.dll" _
(ByVal S As Integer,      ByVal D As Integer) As Long
```

15

If you do not specify a path for *libname*, Visual Basic will search for the file in the following order:

1. Directory containing the .exe file
2. Current directory
3. Windows 32-bit system directory (often but not necessarily \Windows\System32)

4. Windows 16-bit system directory (often but not necessarily \Windows\System)
5. Windows directory (not necessarily \Windows)
6. Path environment variable

2

The following table lists the common operating environment library files.

Dynamic Link Library	Description
Advapi32.dll	Advanced API services library supporting numerous APIs including many security and Registry calls
Comdlg32.dll	Common dialog API library
Gdi32.dll	Graphics Device Interface API library
Kernel32.dll	Core Windows 32-bit base API support
Lz32.dll	32-bit compression routines
Mpr.dll	Multiple Provider Router library
Netapi32.dll	32-bit Network API library
Shell32.dll	32-bit Shell API library
User32.dll	Library for user interface routines
Version.dll	Version library
Winmm.dll	Windows multimedia library
Winspool.drv	Print spooler interface that contains the print spooler API calls

16

Working with Windows API Procedures that Use Strings

When working with Windows API procedures that use strings, you'll need to add an Alias clause to your declare statements to specify the correct character set. Windows API functions that contain strings actually exist in two formats: ANSI and Unicode. In the Windows header files, therefore, you'll get both ANSI and Unicode versions of each function that contains a string.

For example, following are the two C-language descriptions for the SetWindowText function. You'll note that the first description defines the function as SetWindowTextA, where the trailing "A" identifies it as an ANSI function:

```
WINUSERAPI
BOOL
WINAPI
SetWindowTextA(
    HWND hWnd,
    LPCSTR lpString);
```

17

The second description defines it as SetWindowTextW, where the trailing "W" identifies it as a wide, or Unicode function:

```
WINUSERAPI
```

```
BOOL
WINAPI
SetWindowTextW(
    HWND hWnd,
    LPCWSTR lpString);
```

18

Because neither function is actually named "SetWindowText," you need to add an Alias clause to the declare to point to the function you want to reference:

```
Private Declare Function SetWindowText Lib "user32" _
Alias "SetWindowTextA" (ByVal hwnd As Long, ByVal _
lpString As String) As Long
```

19

Note that the string that follows the Alias clause must be the true, case-sensitive name of the procedure.

Important For API functions you use in Visual Basic, you should specify the ANSI version of a function, because Unicode versions are only supported by Windows NT — not Windows 95. Use the Unicode versions only if you can be certain that your applications will be run only on Windows NT-based systems.

20

Passing Arguments by Value or by Reference

By default, Visual Basic passes all arguments *by reference*. This means that instead of passing the actual value of the argument, Visual Basic passes a 32-bit address where the value is stored. Although you do not need to include the ByRef keyword in your Declare statements, you may want to do so to document how the data is passed.

Many DLL procedures expect an argument to be passed *by value*. This means they expect the actual value, instead of its memory location. If you pass an argument by reference to a procedure that expects an argument passed by value, the procedure receives incorrect data and fails to work properly.

To pass an argument by value, place the ByVal keyword in front of the argument declaration in the Declare statement. For example, the InvertRect procedure accepts its first argument by value and its second by reference:

```
Declare Function InvertRect Lib "user32" Alias _
"InvertRectA" (ByVal hdc As Long, _
lpRect As RECT) As Long
```

21

You can also use the ByVal keyword when you call the procedure.

Note When you're looking at DLL procedure documentation that uses C language syntax, remember that C passes all arguments except arrays by value.

22

String arguments are a special case. Passing a string by value means you are passing the address of the first data byte in the string; passing a string by reference means you are passing the memory address where another address is stored; the second address actually refers to the first data byte of the string. How you determine which approach

to use is explained in the topic "Passing Strings to a DLL Procedure" later in this chapter.

Nonstandard Names

Occasionally, a DLL procedure has a name that is not a legal identifier. It might have an invalid character (such as a hyphen), or the name might be the same as a Visual Basic keyword (such as GetObject). When this is the case, use the Alias keyword to specify the illegal procedure name.

For example, some procedures in the operating environment DLLs begin with an underscore character. While you can use an underscore in a Visual Basic identifier, you cannot begin an identifier with an underscore. To use one of these procedures, you first declare the function with a legal name, then use the Alias clause to reference the procedure's real name:

```
Declare Function lopen Lib "kernel32" Alias "_lopen" _
(ByVal lpPathName As String, ByVal iReadWrite _
As Long) As Long
```

23

In this example, lopen becomes the name of the procedure referred to in your Visual Basic procedures. The name _lopen is the name recognized in the DLL.

You can also use the Alias clause to change a procedure name whenever it's convenient. If you do substitute your own names for procedures (such as using WinDir for GetWindowsDirectoryA), make sure that you thoroughly document the changes so that your code can be maintained at a later date.

Using Ordinal Numbers to Identify DLL Procedures

In addition to a name, all DLL procedures can be identified by an *ordinal number* that specifies the procedure in the DLL. Some DLLs do not include the names of their procedures and require you to use ordinal numbers when declaring the procedures they contain. Using an ordinal number consumes less memory in your finished application and is slightly faster than identifying a procedure in a DLL by name.

Important The ordinal number for a specific API will be different with different operating systems. For example, the ordinal value for GetWindowsDirectory is 432 under Win95, but changes to 338 under Window NT 4.0. In sum, if you expect your applications to be run under different operating systems, don't use ordinal numbers to identify API procedures. This approach can still be useful when used with procedures that are not APIs, or when used in applications that have a very controlled distribution.

24

To declare a DLL procedure by ordinal number, use the Alias clause with a string containing the number sign character (#) and the ordinal number of the procedure. For example, the ordinal number of the GetWindowsDirectory function has the value 432 in the Windows kernel; you can declare the DLL procedure as follows:

```
Declare Function GetWindowsDirectory Lib "kernel32" _
Alias "#432" (ByVal lpBuffer As String, _
ByVal nSize As Long) As Long
```

25

Notice that you could specify any valid name for the procedure in this case, because Visual Basic is using the ordinal number to find the procedure in the DLL.

To obtain the ordinal number of a procedure you want to declare, you can use a utility application, such as Dumpbin.exe, to examine the .dll file. (Dumpbin.exe is a utility included with Microsoft Visual C++.) By running Dumpbin on a .dll file, you can extract information such as a list of functions contained within the DLL, their ordinal numbers, and other information about the code.

For More Information For more information on running the Dumpbin utility, refer to the Microsoft Visual C++ documentation.

Flexible Argument Types

Some DLL procedures can accept more than one type of data for the same argument. If you need to pass more than one type of data, declare the argument with As Any to remove type restrictions.

For example, the third argument in the following declare (lppt As Any) could be passed as an array of POINT structures, or as a RECT structure, depending upon your needs:

```
Declare Function MapWindowPoints Lib "user32" Alias _
"MapWindowPoints" (ByVal hwndFrom As Long, _
ByVal hwndTo As Long, lppt As Any, _
ByVal cPoints As Long) As Long
```

26

While the As Any clause offers you flexibility, it also adds risk in that it turns off all type checking. Without type checking, you stand a greater chance of calling the procedure with the wrong type, which can result in a variety of problems, including application failure. Be sure to carefully check the types of all arguments when using As Any.

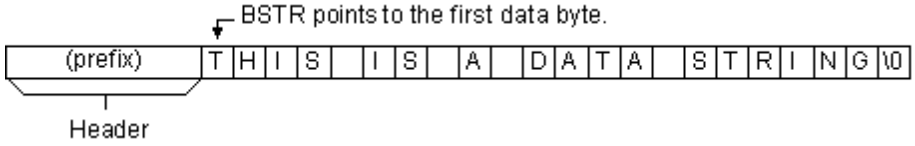
When you remove type restrictions, Visual Basic assumes the argument is passed by reference. Include ByVal in the actual call to the procedure to pass arguments by value. Strings are passed by value so that a pointer to the string is passed, rather than a pointer to a pointer. This is further discussed in the section "Passing Strings to a DLL Procedure."

Passing Strings to a DLL Procedure

In general, strings should be passed using ByVal, unless you are passing them to an OLE 2.0 API or a Visual Basic procedure. Visual Basic uses a String data type known as a BSTR, which is a data type defined by Automation (formerly called OLE Automation). A BSTR is comprised of a header, which includes information about the length of the string, and the string itself, which may include embedded nulls. A BSTR

is passed as a pointer, so the DLL procedure is able to modify the string. (A *pointer* is a variable that contains the memory location of another variable, rather than the actual data.) Most BSTRs are Unicode, which means that each character takes two bytes. BSTRs typically end with a two-byte two null character.

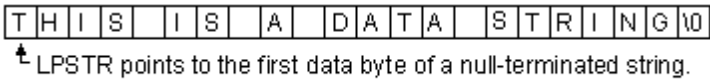
Figure 1.2 The BSTR type



3

The procedures in most DLLs (and in all procedures in the Windows API) recognize LPSTR types, which are pointers to standard null-terminated C strings (also called ASCIIZ strings). LPSTRs have no prefix. The following figure shows an LPSTR that points to an ASCIIZ string.

Figure 1.3 The LPSTR type



4

If a DLL procedure expects an LPSTR (a pointer to a null-terminated string) as an argument, pass the BSTR by value. Because a pointer to a BSTR is a pointer to the first data byte of a null-terminated string, it looks like an LPSTR to the DLL procedure.

For example, the sndPlaySound function accepts a string that names a digitized sound (.wav) file and plays that file.

```
Private Declare Function sndPlaySound Lib "winmm.dll" _
Alias "sndPlaySoundA" (ByVal lpszSoundName As String, _
ByVal uFlags As Long) As Long
```

27

Because the string argument for this procedure is declared with ByVal, Visual Basic passes a BSTR that points to the first data byte:

```
Dim SoundFile As String, ReturnLength As Long
SoundFile = Dir("c:\Windows\System\" & "*.wav")
Result = sndPlaySound(SoundFile, 1)
```

28

In general, use the ByVal keyword when passing string arguments to DLL procedures that expect LPSTR strings. If the DLL expects a pointer to an LPSTR string, pass the Visual Basic string by reference.

When passing binary data to a DLL procedure, pass a variable as an array of the Byte data type, instead of a String variable. Strings are assumed to contain characters, and

binary data may not be properly read in external procedures if passed as a String variable.

If you declare a string variable without initializing it, and then pass it by value to a DLL, the string variable is passed as NULL, not as an empty string (""). To avoid confusion in your code, use the vbNullString constant to pass a NULL to an LPSTR argument.

Passing Strings to DLLs that Use Automation

Some DLLs may be written specifically to work with Automation data types like BSTR, using procedures supplied by Automation.

Because Visual Basic uses Automation data types as its own data types, Visual Basic arguments can be passed by reference to any DLL that expects Automation data types. Thus, if a DLL procedure expects a Visual Basic string as an argument, you do not need to declare the argument with the ByVal keyword, unless the procedure specifically needs the string passed by value.

Some DLL procedures may return strings to the calling procedure. A DLL function cannot return strings unless it is written specifically for use with Automation data types. If it is, the DLL probably supplies a type library that describes the procedures. Consult the documentation for that DLL.

For More Information For information on Automation data types, see the *OLE 2 Programmer's Reference,* published by Microsoft Press.

Procedures That Modify String Arguments

A DLL procedure can modify data in a string variable that it receives as an argument. However, if the changed data is longer than the original string, the procedure writes beyond the end of the string, probably corrupting other data.

You can avoid this problem by making the string argument long enough so that the DLL procedure can never write past the end of it. For example, the GetWindowsDirectory procedure returns the path for the Windows directory in its first argument:

```
Declare Function GetWindowsDirectory Lib "kernel32" _
Alias "GetWindowsDirectoryA" (ByVal lpBuffer As _
String, ByVal nSize As Long) As Long
```

29

A safe way to call this procedure is to first use the String function to set the returned argument to at least 255 characters by filling it with null (binary zero) characters:

```
Path = String(255, vbNullChar)
ReturnLength = GetWindowsDirectory(Path, Len(Path))
Path = Left(Path, ReturnLength)
```

30

Another solution is to define the string as fixed length:

```
Dim Path As String * 255
ReturnLength = GetWindowsDirectory(Path, Len(Path))
```

31

Both of these processes have the same result: They create a fixed-length string that can contain the longest possible string the procedure might return.

Note Windows API DLL procedures generally do not expect string buffers longer than 255 characters. While this is true for many other libraries, always consult the documentation for the procedure.

32

When the DLL procedure calls for a memory buffer, you can either use the appropriate data type, or use an array of the byte data type.

Passing Arrays to a DLL Procedure

You can pass individual elements of an array the same way you pass a variable of the same type. When you pass an individual element, it will be passed as the base type of the array. For example, you can use the sndPlaySound procedure to play a series of .wav files stored in an array:

```
Dim WaveFiles(10) As String
Dim i As Integer, worked As Integer
    For i = 0 to UBound(WaveFiles)
        worked = sndPlaySound(WaveFiles(i), 0)
    Next i
```

33

Sometimes you may want to pass an entire array to a DLL procedure. If the DLL procedure was written especially for Automation, then you may be able to pass an array to the procedure the same way you pass an array to a Visual Basic procedure: with empty parentheses. Because Visual Basic uses Automation data types, including SAFEARRAYs, the DLL must be written to accommodate Automation for it to accept Visual Basic array arguments. For further information, consult the documentation for the specific DLL.

If the DLL procedure doesn't accept Automation SAFEARRAYs directly, you can still pass an entire array if it is a numeric array. You pass an entire numeric array by passing the first element of the array by reference. This works because numeric array data is always laid out sequentially in memory. If you pass the first element of an array to a DLL procedure, that DLL then has access to all of the array's elements.

As an example, consider how you can use an API call to set tab stops within a text box There are internal tab stops in multiple-line (but not single-line) text box controls: If the text in the text box contains tab characters (character code 9), the text following the tab character is aligned at the next tab stop. You can set the position of these tab stops by calling the SendMessage function in the Windows API and passing an array that contains the new tab stop settings.

```
Private Declare Function SendMessageSetTabs Lib _
"user32" Alias "SendMessageA" (ByVal hwnd As Long, _
ByVal wMsg As Long, ByVal wParam As Long, _
lParam As Any) As Long
Const EM_SETTABSTOPS = &HCB
```

```
Sub ChangeTabs(anyText As TextBox, tabcount As Integer)
Dim i As Integer
Dim alngTabs() As Long
Dim lngRC As Long
ReDim alngTabs(tabcount - 1)
    For i = 0 To UBound(alngTabs)
        alngTabs(i) = (i + 1) * 96
        ' Set value to specify tabs in "dialog units."
    Next i
    ' Call with null pointer to empty existing
    ' tab stops.
    lngRC = SendMessageSetTabs(anyText.hwnd, _
    EM_SETTABSTOPS, 0, vbNullString)
    ' Pass first element in array; other elements
    ' follow it in memory.
    lngRC = SendMessageSetTabs(anyText.hwnd, _
    EM_SETTABSTOPS, tabcount, alngTabs(0))
    anyText.Refresh
End Sub
```

34

When you call this procedure, you specify the name of the text box and the number of tab stops you want to use for the indent. For example:

```
Private Sub Command1_Click()
    ChangeTabs Text1, 4
End Sub
```

35

This approach will also work for string arrays. A DLL procedure written in C treats a string array as an array of pointers to string data, which is the same way Visual Basic defines a string array.

For More Information For more information on SAFEARRAYs and other Automation data types, see the Microsoft Press book, *OLE 2 Programmer's Reference*.

Passing User-Defined Types to a DLL Procedure

Some DLL procedures take user-defined types as arguments. (User-defined types are referred to as "structures" in C and as "records" in Pascal.) As with arrays, you can pass the individual elements of a user-defined type the same way you would pass ordinary numeric or string variables.

You can pass an entire user-defined type as a single argument if you pass it by reference. User-defined types cannot be passed by value. Visual Basic passes the address of the first element, and the rest of the elements of a user-defined type are stored in memory following the first element. Depending on the operating system, there may also be some padding.

For example, several procedures in the operating environment DLLs accept a user-defined type for a rectangle, which has the following structure:

```
Type RECT
    Left As Long
    Top As Long
    Right As Long
    Bottom As Long
End Type
```

36

Two of the procedures that accept a rectangle are DrawFocusRect, which draws a dotted outline around the specified rectangle, and InvertRect, which inverts the colors of the specified rectangle. To use the procedures, place these declarations in the Declarations section of a form or standard module:

```
Declare Function DrawFocusRect Lib "User32" Alias _
"DrawFocusRect" (ByVal hdc As Long, _
lpRect As RECT) As Long

Declare Function InvertRect Lib "User32" Alias _
"InvertRect" (ByVal hdc As Long, _
lpRect As RECT) As Long

Dim MouseRect As RECT
```

37

Now you can use the following Sub procedures to call the DLLs:

```
Private Sub Form_MouseDown (Button As Integer, _
Shift As Integer, X As Single, Y As Single)
    ScaleMode = 3
    If Button And 1 Then
        MouseRect.Left = X
        MouseRect.Top = Y
        MouseRect.Right = X
        MouseRect.Bottom = Y
    End If
End Sub

Private Sub Form_MouseUp (Button As Integer, _
Shift As Integer, X As Single, Y As Single)
    ScaleMode = 3
    If Not (Button And 1) Then
        MouseRect.Right = X
        MouseRect.Bottom = Y
        InvertRect hDC, MouseRect
    End If
End Sub

Private Sub Form_MouseMove (Button As Integer, _
Shift As Integer, X As Single, Y As Single)
    ScaleMode = 3
    If Button And 1 Then
        DrawFocusRect hDC, MouseRect
        MouseRect.Right = X
        MouseRect.Bottom = Y
```

```
        DrawFocusRect hDC, MouseRect
    End If
End Sub
```

38

User-defined types can contain objects, arrays, and BSTR strings, although most DLL procedures that accept user-defined types do not expect them to contain string data. If the string elements are fixed-length strings, they look like null-terminated strings to the DLL and are stored in memory like any other value. Variable-length strings are incorporated in a user-defined type as pointers to string data. Four bytes are required for each variable-length string element.

Note When passing a user-defined type that contains binary data to a DLL procedure, store the binary data in a variable of an array of the Byte data type, instead of a String variable. Strings are assumed to contain characters, and binary data may not be properly read in external procedures if passed as a String variable.

39

Passing Function Pointers to DLL Procedures and Type Libraries

If you're familiar with the C programming language, function pointers may be familiar to you. If you're not, the concept merits some explanation. A *function pointer* is a convention that enables you to pass the address of a user-defined function as an argument to another function you've declared for use within your application. By using function pointers, you can now call functions like EnumWindows to list the open windows on the system, or EnumFontFamilies to catalog all of the current fonts. You can also use them to gain access to many other functions from the Win32 API that have not previously been supported in Visual Basic.

For Visual Basic 5.0, several limitations apply to the use of function pointers. For details, see "Limitations and Risks with Function Pointers" later in this topic.

Learning About Function Pointers

The use of function pointers is best illustrated with an example. To start, look at the EnumWindows function from the Win32 API:

```
Declare Function EnumWindows lib "user32" _
(ByVal lpEnumFunc as Long, _
ByVal lParam as Long ) As Long
```

40

EnumWindows is an enumeration function, which means that it can list the handle of every open window on your system. EnumWindows works by repeatedly calling the function you pass to its first argument (lpEnumFunc). Each time EnumWindows calls the function, EnumWindows passes it the handle of an open window.

When you call EnumWindows from your code, you pass a user-defined function to this first argument to handle the stream of values. For example, you might write a function to add the values to a list box, convert the hWnd values to window names, or take whatever action you choose.

To specify that you're passing a user-defined function as an argument, you precede the name of the function with the AddressOf keyword. Any suitable value can be passed to the second argument. For example, to pass the function MyProc as an argument, you might call the EnumWindows procedure as follows:

```
x = EnumWindows(AddressOf MyProc, 5)
```

41

The user-defined function you specify when you call the procedure is referred to as the *callback function*. Callback functions (or "callbacks," as they are commonly called) can perform any action you specify with the data supplied by the procedure.

A callback function must have a specific set of arguments, as determined by the API from which the callback is referenced. Refer to your API documentation for information on the necessary arguments and how to call them.

Using the AddressOf Keyword

Any code you write to call a function pointer from Visual Basic 5.0 must be placed in a standard .BAS module — you can't put the code in a class module or attach it to a form. When you call a declared function using the AddressOf keyword, you should be aware of the following conditions:

- AddressOf can only be used immediately preceding an argument in an argument list; that argument can be the name of a user-defined sub, function, or property.
- The sub, function, or property you call with AddressOf must be in the same project as the related declarations and procedures.
- You can only use AddressOf with user-defined subs, functions, or properties — you cannot use it with external functions declared with the Declare statement, or with functions referenced from type libraries.
- You can pass a function pointer to an argument that is typed As Any or As Long in a declared Sub, Function, or user-defined type definition.

5

Storing a Function Pointer in a Variable

At times, you may need to store a function pointer in an intermediate variable before passing it to the DLL. This is useful if you want to pass function pointers from one Visual Basic function to another. It's required if you are calling a function like RegisterClass, where you need to pass the pointer through an argument to a structure (WndClass), which contains a function pointer as one of its elements.

To assign a function pointer to an element in a structure, you write a wrapper function. For example, the following code creates the wrapper function FnPtrToLong, which can be used to put a function pointer in any structure:

```
Function FnPtrToLong (ByVal lngFnPtr As Long) As Long
    FnPtrToLong = lngFnPtr
End Function
```

42

To use the function, you first declare the type, then call FnPtrToLong. You pass AddressOf plus your callback function name for the second argument.

```
Dim mt as MyType
mt.MyPtr = FnPtrToLong(AddressOf MyCallBackFunction)
```

43

Subclassing

Subclassing is a technique that enables you to intercept Windows messages being sent to a form or control. By intercepting these messages, you can then write your own code to change or extend the behavior of the object. Subclassing can be complex, and a thorough discussion of it is beyond the scope of this book. The following example offers a brief illustration of the technique.

Important When Visual Basic is in break mode, you can't call vtable methods or AddressOf functions. As a safety mechanism, Visual Basic simply returns 0 to the caller of an AddressOf function without calling the function. In the case of subclassing, this means that 0 is returned to Windows from the WindowProc. Windows requires nonzero return values from many of its messages, so the constant 0 return may create a deadlock situation between Windows and the Visual Basic, forcing you to end the process.

44

This application consists of a simple form with two command buttons. The code is designed to intercept Windows messages being sent to the form and to print the values of those messages in the Immediate window.

The first part of the code consists of declarations for the API functions, constant values, and variables:

```
Declare Function CallWindowProc Lib "user32" Alias _
"CallWindowProcA" (ByVal lpPrevWndFunc As Long, _
    ByVal hwnd As Long, ByVal Msg As Long, _
    ByVal wParam As Long, ByVal lParam As Long) As Long

Declare Function SetWindowLong Lib "user32" Alias _
"SetWindowLongA" (ByVal hwnd As Long, _
ByVal nIndex As Long, ByVal dwNewLong As Long) As Long

Public Const GWL_WNDPROC = -4
Global lpPrevWndProc As Long
Global gHW As Long
```

45

Next, two subroutines enable the code to hook into the stream of messages. The first procedure (Hook) calls the SetWindowLong function with the GWL_WNDPROC index to create a subclass of the window class that was used to create the window. It then uses the AddressOf keyword with a callback function (WindowProc) to intercept the messages and print their values in the Immediate window. The second procedure

(Unhook) turns off subclassing by replacing the callback with the original Windows procedure.

```
Public Sub Hook()
    lpPrevWndProc = SetWindowLong(gHW, GWL_WNDPROC, _
    AddressOf WindowProc)
End Sub

Public Sub Unhook()
    Dim temp As Long
    temp = SetWindowLong(gHW, GWL_WNDPROC, _
    lpPrevWndProc)
End Sub

Function WindowProc(ByVal hw As Long, ByVal uMsg As _
Long, ByVal wParam As Long, ByVal lParam As Long) As _
Long
    Debug.Print "Message: "; hw, uMsg, wParam, lParam
    WindowProc = CallWindowProc(lpPrevWndProc, hw, _
    uMsg, wParam, lParam)
End Function
```

46

Finally, the code for the form sets the initial hWnd value, and the code for the buttons simply calls the two subroutines:

```
Private Sub Form_Load()
    gHW = Me.hwnd
End Sub

Private Sub Command1_Click()
    Hook
End Sub

Private Sub Command2_Click()
    Unhook
End Sub
```

47

Limitations and Risks with Function Pointers

Working with function pointers can be unforgiving. You lose the stability of Visual Basic's development environment any time you call a DLL, but when working with function pointers, it can be especially easy to cause the application to fail and to lose your work. Save often and back up your work as necessary. Following are notes on some areas that require special attention when working with function pointers:

- Debugging. If your application fires a callback function while in break mode, the code will be executed, but any breaks or steps will be ignored. If the callback function generates an exception, you can catch it and return the current value. Resets are prohibited in break mode when a callback function is on the stack.
- Thunks. *Thunking* is the way that Windows enables relocatable code. If you delete a callback function in break mode, its thunk is modified to return 0. This value will be correct most of the time — but not all of the time. If you delete a

callback function in break mode and then type it again, it's possible that some callees will not know about the new address. Thunks aren't used in the .exe ¾ the pointer is passed directly to the entry point.

- Passing a function with the wrong signature. If you pass a callback function that takes a different number of arguments than the caller expects, or mistakenly calls an argument with ByRef or ByVal, your application may fail. Be careful to pass a function with the correct signature.
- Passing a function to a Windows procedure that no longer exists. When subclassing a window, you pass a function pointer to Windows as the Windows procedure (WindowProc). When running your application in the IDE, however, it's possible that the WindowProc will be called after the underlying function has already been destroyed. This will likely cause a general protection fault and may bring down the Visual Basic development environment.
- "Basic to Basic" function pointers are not supported. Pointers to Visual Basic functions cannot be passed within Visual Basic itself. Currently, only pointers from Visual Basic to a DLL function are supported.

6

Passing Other Types of Information to a DLL Procedure

Visual Basic supports a wide range of data types, some of which may not be supported by the procedures in certain dynamic-link libraries. The following topic describes how to handle some of the special cases you may find when using Visual Basic variables with DLL procedures.

Passing Null Pointers

Some DLL procedures may sometimes expect to receive either a string or a null value as an argument. If you need to pass a null pointer to a string, declare the argument As String and pass the constant vbNullString.

For example, the FindWindow procedure can determine if another application is currently running on your system. It accepts two string arguments, one for the class name of the application, and another for the window title bar caption:

```
Declare Function FindWindow Lib "user32" Alias _
"FindWindowA" (ByVal lpClassName As String, _
ByVal lpWindowName As String) As Long
```

48

Either of these arguments can be passed as a null value. Passing a zero-length string ("") does not work, however, as this passes a pointer to a zero-length string. The value of this pointer will not be zero. You instead need to pass an argument with the true value of zero. The easiest way to do this is by using the constant value vbNullString for the appropriate argument:

```
hWndExcel = FindWindow(vbNullString, "Microsoft Excel")
```
49

Another way to handle this situation is to rewrite the declare to substitute a Long data type for the argument that you want to pass as null, and then call that argument with the value 0&. For example:

```
Declare Function FindWindowWithNull Lib "user32" -
Alias "FindWindowA" (ByVal lpClassName As Long, _
ByVal lpWindowName As String) As Long

hWndExcel = FindWindow(0&, "Microsoft Excel")
```
50

Passing Properties

Properties must be passed by value. If an argument is declared with ByVal, you can pass the property directly. For example, you can determine the dimensions of the screen or printer in pixels with this procedure:

```
Declare Function GetDeviceCaps Lib "gdi32" Alias _
"GetDeviceCaps" (ByVal hdc As Long, _
ByVal nIndex As Long) As Long
```
51

You can also pass the hDC property of a form or the Printer object to this procedure to obtain the number of colors supported by the screen or the currently selected printer. For example:

```
Private Sub Form_Click ()
Const PLANES = 14, BITS = 12
    Print "Screen colors ";
    Print GetDeviceCaps(hDC, PLANES)* 2 ^ _
    GetDeviceCaps(hDC, BITS)
    Print "Printer colors ";
    Print GetDeviceCaps(Printer.hDC, PLANES) * _
    2 ^ GetDeviceCaps(Printer.hDC, BITS)
End Sub
```
52

To pass a property by reference, you must use an intermediate variable. For example, suppose you want to use the GetWindowsDirectory procedure to set the Path property of a file list box control. This example will not work:

```
ReturnLength = GetWindowsDirectory(File1.Path,_
Len(File1.Path))
```
53

Instead, use the following code to set the property:

```
Dim Temp As String, ReturnLength As Integer
Temp = String(255, 0)
ReturnLength = GetWindowsDirectory(Temp, Len(Temp))
Temp = Left(Temp, ReturnLength)
File1.Path = Temp
```
54

Use this technique with numeric properties if you want to pass them to DLL procedures that accept arguments by reference.

Using Handles with DLLs

A *handle* is a unique Long value defined by the operating environment. It is used to refer to objects such as forms or controls. The operating environment DLL procedures make extensive use of handles — handles to windows (hWnd), handles to device contexts (hDC), and so on. When a procedure takes a handle as an argument, always declare it as a ByVal Long. DLL functions that return a handle can be declared as Long functions. Handles are identifier (ID) numbers, not pointers or numeric values; never attempt mathematical operations on them.

The hWnd property of forms and nongraphical controls and the hDC property of forms and picture box controls supply valid handles that you can pass to DLL procedures. Like any other property passed to a DLL procedure, they can be passed only by value.

Passing Variants

Passing an argument of type Variant is similar to passing any other argument type, as long as the DLL procedure uses the Automation VARIANT data structure to access the argument data. To pass Variant data to a argument that is not a Variant type, pass the Variant data ByVal.

Converting C Declarations to Visual Basic

The procedures in DLLs are most commonly documented using C language syntax. To call these procedures from Visual Basic, you need to translate them into valid Declare statements and call them with the correct arguments.

As part of this translation, you must convert the C data types into Visual Basic data types and specify whether each argument should be called by value (ByVal) or implicitly, by reference (ByRef). The following table lists common C language data types and their Visual Basic equivalents for 32-bit versions of Windows.

C language data type	In Visual Basic declare as	Call with
ATOM	ByVal *variable* As Integer	An expression that evaluates to an Integer
BOOL	ByVal *variable* As Long	An expression that evaluates to a Long
BYTE	ByVal *variable* As Byte	An expression that evaluates to a Byte
CHAR	ByVal *variable* As Byte	An expression that evaluates to a Byte
COLORREF	ByVal *variable* As Long	An expression that evaluates to a Long
DWORD	ByVal *variable* As Long	An expression that evaluates to a Long
HWND, HDC, HMENU,	ByVal *variable* As Long	An expression that

etc. (Windows handles)		evaluates to a Long
INT, UINT	ByVal *variable* As Long	An expression that evaluates to a Long
LONG	ByVal *variable* As Long	An expression that evaluates to a Long
LPARAM	ByVal *variable* As Long	An expression that evaluates to a Long
LPDWORD	*variable* As Long	An expression that evaluates to a Long
LPINT, LPUINT	*variable* As Long	An expression that evaluates to a Long
LPRECT	*variable* As *type*	Any variable of that user-defined type
LPSTR, LPCSTR	ByVal *variable* As String	An expression that evaluates to a String
LPVOID	*variable* As Any	Any variable (use ByVal when passing a string)
LPWORD	*variable* As Integer	An expression that evaluates to an Integer
LRESULT	ByVal *variable* As Long	An expression that evaluates to a Long
NULL	As Any or ByVal *variable* As Long	ByVal Nothing or ByVal 0& or vbNullString
SHORT	ByVal *variable* As Integer	An expression that evaluates to an Integer
VOID	Sub *procedure*	Not applicable
WORD	ByVal *variable* As Integer	An expression that evaluates to an Integer
WPARAM	ByVal *variable* As Long	An expression that evaluates to a Long