Chapter 1

Overcoming Holes in the .NET Framework

- Why Access the Win32 API?
- Win32 Access for C# Developers
- Win32 Access for Visual Basic Developers

There are few, if any, perfect programming platforms in the world and .NET is no exception. Developers who spend any time working in the unmanaged environment before they begin working with .NET will notice some distinct problems with functionality in the .NET environment. Some of these holes (such as those in the security area) are apparent and require an immediate fix; others (such as the lack of support for a `Beep()` function) are subtle and you might never notice them. The point is that the.NET Framework is new technology and there are bound to be some holes in coverage, and you'll notice them with regular frequency.

This chapter provides a quick overview of some major areas of omission in the .NET Framework. I want to say at the outset that I feel the .NET Framework is a big improvement over using unmanaged code, but it's new and lacks some features that most developers will need. Consequently, you'll find that this book discusses "holes" in coverage or augmentation of features. I'm not here to tell you that the .NET Framework is technically flawed. The goal of this chapter is to help you plan your development efforts to make the best use of existing .NET Framework functionality and to access the Win32 API when the .NET Framework proves less than perfect.

NOTE Visual C++ represents a unique language in .NET because it provides both managed and unmanaged coding environments. Consequently, you can access the Win32 API in its entirety from Visual C++ using the same unmanaged techniques you used before .NET appeared on the horizon. This unique functionality means that Visual C++ developers won't need the special techniques found in this book. It also means that you can use Visual C++ as a language bridge between managed and unmanaged environments.

Why Access the Win32 API?

Many of you have a lot of experience working with the Win32 API are already familiar with the programming requirements for unmanaged code. The idea of working with unmanaged code presents few problems for the seasoned developer. However, the .NET Framework that Microsoft has advertised is supposed to obviate the need to work with unmanaged code, so the first question on most developer's minds is: why they would even need to access the Win32 API? The short answer is that you'll find a lack of functionality in some areas such as DirectX, the Microsoft Management Console (MMC), and direct hardware access when working with the .NET Framework. You can only gain access to this functionality through the Win32 API.

The long answer requires a little more explanation. For example, not all .NET languages have an equal measure of missing functionality. The .NET Framework doesn't include a `Beep()` function, so you'll find this feature missing in C# as well. However, because Visual Basic includes a `Beep()` function as part of the language, it doesn't require any special programming to access

this Win32 API feature. In sum, the question of missing functionality is a matter of determining if the language you're using provides the feature and then determining the best method to access the feature if it is missing.

You'll find that the question of Win32 API access becomes more complex as you move from simple functions such as `Beep()` to major programming projects such as creating an MMC Snap-in. The level of Win32 API access varies by language, which is why this book addresses C# and Visual Basic. You'll find that some chapters, including this one, contain separate C# and Visual Basic sections because the two languages provide varying levels of Win32 API access. Consequently, the third issue in Win32 API access is whether the target language provides support for the required feature. It might not, which means you'll need to create wrappers for the missing functionality.

Now that you have a basic overview of the question of why you'd want to access the Win32 API, let's discuss the issues in more detail. The following sections describe needs and requirements for Win32 API access in a generic manner. You can apply this material equally to any language you might want to use with .NET.

A Case of Missing Functionality

As previously mentioned, the .NET Framework lacks functionality for some basic calls such as `Beep()`. This means that a C# developer who needs to create a sound within an application has to find some other means to do it. There's no doubt that the functionality is missing, but the technique used to create the desired functionality varies by language capability, environment, and flexibility. For example, when working with Visual Basic, you already have access to a basic `Beep()` function, so no additional coding is required if you require a simple beep. However, as shown in Listing 1.1, there are actually four ways to create a beep in C# and not all of them provide the same features. (You'll find the source code for this example in the `\Chapter 01\C#\MakeSound` folder of the CD; a Visual Basic version appears in the `\Chapter 01\VB\MakeSound` folder.)

Listing 1.1 Creating a Beep in C#

```
// Import the Windows Beep() API function.
[DllImport("kernel32.dll")]
private static extern bool Beep(int freq, int dur);

// Define some constants for using the PlaySound() function.
public const int SND_FILENAME = 0x00020000;
public const int SND_ASYNC = 0x0001;

// Import the Windows PlaySound() function.
[DllImport("winmm.dll")]
```

```
public static extern bool PlaySound(string pszSound,
                                    int hmod,
                                    int fdwSound);

[STAThread]
static void Main(string[] args)
{
   // Create a sound using an escape character.
   Console.Write("\a");
   Console.WriteLine("Press Any Key When Ready...");
   Console.ReadLine();

   // Create a sound using a Windows API call.
   Beep(800, 200);
   Console.WriteLine("Press Any Key When Ready...");
   Console.ReadLine();

   // Create a sound using a Visual Basic call.
   Microsoft.VisualBasic.Interaction.Beep();
   Console.WriteLine("Press Any Key When Ready...");
   Console.ReadLine();

   // Create a sound using a WAV file.
   PlaySound("BELLS.WAV",
             0,
             SND_FILENAME | SND_ASYNC);
   Console.WriteLine("Press Any Key When Ready...");
   Console.ReadLine();
}
```

It's important to note that using an escape character to produce a sound only works for a console application—you can't use this technique in a GUI application. However, this technique does enable you to circumvent the requirement to access the Win32 API just to create a beep. The technique is important because it provides you with another choice; one that doesn't rely on unmanaged code.

The Win32 API `Beep()` function has the advantage of providing the greatest flexibility for the smallest cost in resources. To use this technique, you must declare the Win32 API `Beep()` function as a DLL import using the `[DllImport]` attribute. In this case, you must use unmanaged code to achieve your objective, but you don't need a wrapper DLL—C# and Visual Basic both provide all the support required. Notice that the Win32 API `Beep()` function enables you to choose both the tone (frequency) and duration of the beep, which is something you won't get using an escape character or Visual Basic's built-in function.

Some developers might not realize that they are able to access other language features from within the current language by relying on a .NET Framework feature called Interaction. The third method, shown in Listing 1.1, simply calls the Visual Basic `Beep()` function. You need to

include a reference to the `Microsoft.VisualBasic.DLL` to make this portion of the example work. This technique requires a little more effort than making a direct Win32 API call, but it has the advantage of using pure managed code within the C# application.

Sometimes you don't want to use a plain beep within an application, so it's helpful to know how to access `WAV` files. The fourth technique, shown in Listing 1.1, has the advantage of complete sound source flexibility. However, this technique also has the dubious honor of being the most complicated way to produce a sound. The function call to `PlaySound()` is more complicated than the `Beep()` Win32 API call. You also need to define constants to use it.

The point of this section is that you'll find missing functionality within the .NET Framework, but you don't always have to rely on Win32 API calls to fill the gap. In many situations, you can rely on language interoperability or built-in operating system functionality. When you do need to rely on the Win32 API, you'll find that some functions are easier to use than others. It isn't always necessary to use the most complex method when a simple one will work. In fact, in some cases, you'll find that you can't use the full-featured function because the target language won't support it.

Win32 Function Types

One of the problems in determining if a piece of functionality is missing from the .NET Framework is that the framework is relatively large—not as large as the Win32 API, but large nonetheless. (At the time of this writing, the download size for the .NET Framework was 21 MB.) So it pays to know where you'll find holes in the .NET Framework most often. The following sections discuss the various places where other developers have found holes in the .NET Framework coverage of the Win32 API. You might find other areas when working with special Win32 API features, but these sections provide you with a fairly complete overview.

Hardware

Every time Microsoft releases a new technology, they find a way to add yet more layers of code between the developer and the hardware, and .NET is no exception. Any hope you entertained of direct hardware access will quickly fade as you make your way through a few programming tasks. You'll even find it difficult to access Windows driver and low-level DLL functionality—the access just isn't there. Generally, you'll find that the .NET Framework provides you with objects that indirectly relate to some type of hardware functionality, such as the use of streams for hard drive and printer access.

The lack of direct hardware access isn't always a negative, however. Once you get used to using the .NET Framework objects, you might find that direct hardware access is unnecessary or, at least, a rare event. Common hardware types, such as printers and hard drives, won't present a problem in most cases. Some developers have complained about the level of support provided for common low-level devices like the serial ports.

You'll also run into problems when working with hardware that Microsoft didn't anticipate. For example, accessing many USB devices is a frustrating experience when working with .NET. In most cases, you'll need to use unmanaged code and a third-party library to access new devices. We'll talk more about direct hardware access in Chapter 7.

Security

Microsoft's latest security craze is role-based security. It's true that role-based security is extremely easy to use and requires less effort on the part of the developer. In many cases, role-based security is also more flexible than the security that Microsoft provided in the past. However, role-based security is also less than appropriate if you need low-level control over the security features of your application.

There's a place for tokens, access control lists, and all of the other paraphernalia of Win32 API security in many applications, but you can't gain access to these features within the .NET Framework. To gain access to the low-level details of security within Windows, you still need to use the security calls provided by the Win32 API. We'll discuss security access within Chapter 8.

Operating System

It would seem that the operating system is the first thing you'd need to support as part of development platform, but this isn't necessarily true. Consider two existing types of application that don't rely very heavily on the operating system: browser-based applications and Java applications. Yes, both technologies require basic access to the operating system, but you'll find that for the most part you can't access the operating system as an entity. These development platforms rely on runtime engines that interact with the operating system in a predefined manner.

The .NET Framework is a modern development platform that will hopefully see implementation on other platforms. Consequently, you won't see any operating system support in the core namespaces, but will see some support in the Microsoft-specific namespaces. The separation of .NET functionality from operating system functionality is understandable, given Microsoft's stated goal of platform independence. However, unlike other platforms, the .NET Framework does provide limited operating system interfaces. In fact, there are three levels of operating system support that you need to consider when working with the .NET Framework—and .NET only supports one of them.

Upper-Level Interface This is the level of operating support that the .NET Framework does support. The support appears in several areas, but the two main namespaces are `System.Windows.Forms` and `Microsoft.Win32`. As the names imply, the first namespace helps you gain access to the GUI features that Windows provides, while the second namespace provides access to features like the registry. The level of support in both areas is extensive, but limited to features that Microsoft felt a developer would need to create business applications.

Low-Level Services There are a lot of low-level services that the .NET Framework doesn't even touch. For example, if you want to learn about the capabilities of the display, you'll need to use a Win32 API call to do it. Likewise, if you want to learn the status of the services on a remote machine, you'll have to resort to the Win32 API. We'll discuss low-level service access in greater detail in Chapter 10.

Version-Specific Features Generally, you'll find that any operating system features that the .NET Framework does support are also found in all versions of Windows since Windows NT. In some cases, you'll also find the new features originally found in the Windows 9*x* operating system interface. However, if you want to use the new graphical features found in Windows XP, you'll have to rely on the Win32 API. We'll discuss some of these special features and how to access them in Chapter 9.

Multimedia

Microsoft engineered the .NET Framework for business users. You won't find support for any sound capability and barely any functions for graphics. There isn't any support for devices such as joysticks. In short, if you want to work with multimedia, your only choices are using the Win32 API calls or employing DirectX. Both of these solutions currently require the use of unmanaged code. Microsoft has said they plan to create a managed version of DirectX, but it's not a high priority. We'll discuss multimedia issues in greater detail in Chapter 11. A joystick example appears in Chapter 7 as part of direct hardware access.

Utility

There are a number of utility applications within Windows that require special interfaces. The most prominent of these utility applications is the Microsoft Management Console (MMC), which acts as a container for special components called *snap-ins*. The MMC is a vital tool for network administrators (and even for the common user) because it enables you to perform tasks such as monitor computer performance and manage user security. Unfortunately, the .NET Framework doesn't include support for this necessary utility, despite constant requests from developers during the beta process. You'll find a comprehensive MMC example in Chapter 12, along with tips for working with other utility application types.

DirectX

It wasn't long ago that game developers fought with Microsoft over the need to access hardware directly in a way that would keep Windows in the loop without the performance-robbing penalty of actually using Windows. The result of this conflict is DirectX—an advanced programming technology for working with a wide range of multimedia hardware. Given Microsoft's goal of making the .NET Framework business friendly, it's not too surprising they failed to include any DirectX support.

Unfortunately, some business application developers rely on DirectX to produce complex reports and perform other business-related multimedia tasks. Part of the problem may be that Microsoft viewed the behemoth that is DirectX and decided they needed to implement it at a later date to get the .NET Framework out in a timely manner. Rumors abound that Microsoft plans to release a .NET Framework friendly version of DirectX sometime in the future, but for now, you need to rely on unmanaged programming techniques to work with DirectX.

DirectX is a complex topic—one that many books can't cover in detail. Consequently, we'll discuss DirectX in the chapters found in Part IV of the book. We won't discuss DirectX itself. I'm assuming you already know how to use DirectX (or will find another book to guide you). These chapters show how to make DirectX work with managed applications—no small undertaking, but definitely doable.

Win32 Access Requirements

It's important to know what you need to do in order to access the Win32 API once you decide that the .NET Framework doesn't provide a required level of support. Generally speaking, Win32 API access isn't difficult for general functions. If you look again at the `Beep()` example in Listing 1.1, you'll notice that gaining access to the required functions doesn't require a lot of code. However, you do need to know something about the function you want to access, including the fact that it exists. The following list details some of the information you need (we'll discuss this information in detail in Chapter 2).

- A knowledge of the function and its purpose
- A complete list of all function arguments and return values
- A description of any constants used with the function
- Complete details about any structures the function requires for data transfer
- The values and order of any enumeration used with the function

Not every function requires all of this information, but you need to at least verify what information the function does require. A simple function may require nothing more than a `[DLLImport]` entry and a call within your code. Complex functions might require structures, which means converting the data within the structure to match the language you're using within .NET. The most complex functions may have data structure elements such as unions that are impossible to replicate properly within a managed environment, which means creating a wrapper function in an unmanaged language such as Visual C++ (the language we'll use for this purpose throughout the book).

Sometimes what appears to be a single function call actually requires multiple functions. For example, the .NET Framework doesn't offer any way to clear the console screen, so you

need to perform this task using a Win32 API call. Unfortunately, clearing the screen means moving the cursor and performing other low-level tasks—a single call won't do. Listing 1.2 shows a typical example of a single task that required multiple function calls. (The source code for this example appears in the `\Chapter 01\C#\ClearScreen` and the `\Chapter 01\VB\ClearScreen` folders of the CD.)

NOTE Don't worry if this listing appears a bit on the complex side—you'll learn about all of the features in this example as you progress through the book. In fact, this example is just a good beginning point for the complex code that we'll discuss later.

Listing 1.2 Clearing the Screen Requires Multiple Function Calls

```
// This special class contains an enumeration of
// standard handles.
class StdHandleEnum
{
   public const int STD_INPUT_HANDLE   = -10;
   public const int STD_OUTPUT_HANDLE  = -11;
   public const int STD_ERROR_HANDLE   = -12;
};

// This sructure contains a screen coordinate.
[StructLayout(LayoutKind.Sequential, Pack=1)]
   internal struct COORD
{
   public short X;
   public short Y;
}

// This stucture contains information about the
// console screen buffer.
[StructLayout(LayoutKind.Sequential, Pack=1)]
   internal struct CONSOLE_SCREEN_BUFFER_INFO
{
   public COORD      Size;
   public COORD      p1;
   public short      a1;
   public short      w1;
   public short      w2;
   public short      w3;
   public short      w4;
   public COORD      m1;
}

// We need these four functions from kernel32.dll.
// The GetStdHandle() function returns a handle to any
// standard input or output.
```

```
[DllImport("kernel32.dll", SetLastError=true)]
public static extern IntPtr GetStdHandle(int nStdHandle);

// The GetConsoleScreenBufferInfo() returns information
// about the console screen buffer so we know how much to
// clear.
[DllImport("kernel32.dll", SetLastError=true)]
public static extern bool GetConsoleScreenBufferInfo(
   IntPtr hConsoleOutput,
   out CONSOLE_SCREEN_BUFFER_INFO lpConsoleScreenBufferInfo);

// The SetConsoleCursorPosition() places the cursor on the
// console screen.
[DllImport("kernel32.dll", SetLastError=true)]
public static extern bool SetConsoleCursorPosition(
   IntPtr hConsoleOutput,
   COORD dwCursorPosition);

// The FillConsoleOutputCharacter() allows us to place any character
// on the console screen. Using a space clears the display area.
[DllImport("kernel32.dll", SetLastError=true, CharSet=CharSet.Auto)]
public static extern bool FillConsoleOutputCharacter(
   IntPtr hConsoleOutput,
   short cCharacter,
   int nLength,
   COORD WriteCoord,
   out int lpNumberOfCharsWritten);

[STAThread]
static void Main(string[] args)
{
   // Needed ask Windows about the console screen
   // buffer settings.
   CONSOLE_SCREEN_BUFFER_INFO   CSBI;
   // Handle to the otuput device.
   IntPtr                       hOut;
   // Number of characters written to the screen.
   int                          CharOut;
   // Home cursor position.
   COORD                        Home;

   // Write some data to the screen.
   Console.Write("Some Text to Erase!" +
                 "\r\nPress any key...");
   Console.ReadLine();

   // Clear the screen.
   // Begin by getting a handle to the console screen.
   hOut = GetStdHandle(StdHandleEnum.STD_OUTPUT_HANDLE);
```

```
    // Get the required console screen buffer information.
    GetConsoleScreenBufferInfo(hOut, out CSBI );

    // Set the home position for the cursor (upper left corner).
    Home.X = 0;
    Home.Y = 0;

    // Fill the console with spaces.
    FillConsoleOutputCharacter(hOut,
                               (short) ' ',
                               CSBI.Size.X * CSBI.Size.Y,
                               Home,
                               out CharOut);

    // Place the cursor in the upper left corner.
    SetConsoleCursorPosition(hOut, Home);

    // Show the screen is clear.
    Console.ReadLine();
}
```

Notice that this example uses more of the elements typical of a Win32 API call, including an enumeration and two structures. The code requires an enumeration for standard output handles. An output handle is simply a pointer to a device such as the screen. The three standard devices are input, output, and error. We also need two structures to fulfill the needs of the Windows API calls used in the example. The code listing describes each structure's task.

The example code relies on four Windows API functions, all of which appear in the `KERNEL32.DLL`. All four perform some type of console screen manipulation. The code listing describes each function's task.

The short part of the code is actually demonstrating the console screen clearing process. `Main()` creates some output on screen. The `ReadLine()` call merely ensures the code will wait until you see the text. Press Enter and the clearing process begins.

The first thing we need is a handle to the console output. The handle tells Windows what device we want to work with. Once we have a handle to the output device, we need to ask Windows about its dimensions. The dimensions are important because you want to ensure the console screen erases completely. The `FillConsoleOutputCharacter()` function call fills the screen with spaces—the equivalent of erasing its content. Finally, we place the cursor in the upper left corner—the same place the CLS command would.

Of course, working with Windows means more than just making simple function calls; sometimes you need to work with COM as well. Once you get past simple functions and into the COM environment, development quickly gains a level or two of complexity. For example,

if you want to create a COM equivalent component, you'll also need to discover and implement the interfaces supported by the unmanaged component. Sometimes the interfaces can become complex and difficult to re-create, as we'll see in the MMC example in Chapter 12.

Win32 Access for C# Developers

C# developers have a number of advantages over other .NET languages when it comes to Win32 access. The most important is the ability to use unsafe code and pointers. Many developers find that C# is an outstanding choice for the low-level programming tasks required for Win32 access. Of course, there's no free lunch—you pay a price whenever you gain some level of flexibility in the development environment. The following sections provide you with an overview of the pros and cons of using C# as your development language. We'll discuss these issues in greater detail as the book progresses.

Understanding the Effects of Unsafe Code

The term "unsafe code" is somewhat ambiguous because it doesn't really tell you anything about the code. A better way to view unsafe code is unmanaged code that appears within a managed environment. Any code that relies on the use of manual pointers (* symbol) or addresses (& symbol) is unsafe code. Whenever you write code that uses these symbols, you also need to use the `unsafe` keyword in the method declaration as shown below. (This example appears in the `\Chapter 01\C#\Unsafe` folder of the CD.)

```
unsafe private void btnTest_Click(object sender, System.EventArgs e)
{
   int   Input = Int32.Parse(txtInput.Text); // Input string.

   // Convert the input value.
   DoTimeIt(&Input);

   // Display the result
   txtOutput.Text = Input.ToString();
}

unsafe private void DoTimeIt(int* Input)
{
   int   Output;  // Output to the caller.

   // Display the current minute.
   txtMinute.Text = System.DateTime.Now.Minute.ToString();
```

```
    // Create the output value.
    Output = *Input;
    Output = Output * System.DateTime.Now.Minute;

    // Output the result.
    *Input = Output;
}
```

This is a simple example that we could have created using other methods, but it demonstrates a principle you'll need to create applications that rely on the Win32 API later. The `btnTest_Click()` accesses the input value, converts it to an *`int`*, and supplies the address of the *`int`* to the `DoTimeIt()` method. Because we've supplied an address, rather than the value, any change in the supplied value by `DoTimeIt()` will remain when the call returns.

The `DoTimeIt()` method accesses the current time, multiplies it by the value of the input string, and then outputs the value. Notice the use of pointers in this method to access the values contained in the Input and Output variables. The reason this code is unsafe is that the compiler can't check it for errors. For example, you could replace the last line with `Input = &Output;` and the compiler would never complain, but you also wouldn't see the results of the multiplication.

Besides using the `unsafe` keyword, you also need to set your application to use unsafe code. Right-click the project name in Solution Explorer and choose Properties from the context menu. Select the Configuration Properties\Build folder and you'll see the Allow unsafe code blocks option shown in Figure 1.1. Set this option to True to enable use of unsafe code in your application.

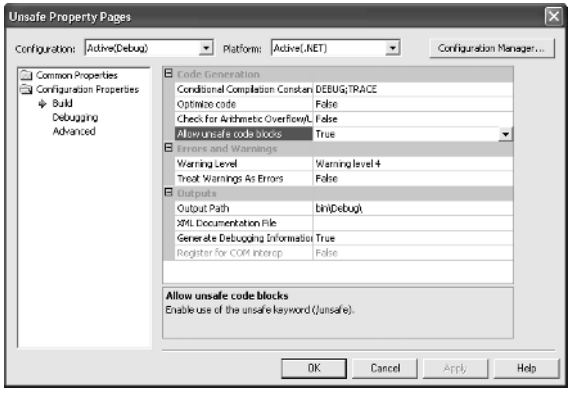

FIGURE 1.1: Using the Allow unsafe code blocks option to enable use of unsafe code in your application.

NOTE You can't use pointers on managed types, but you can use them on values. For example, you can't obtain the address of a `string` because a string is a managed type. The reason this example works in C# is that `int` is a value type. If you need to pass a string, then it's important to know other ways to mimic pointers. For example, you can pass a string using the `ref` or `out` keywords, or you can marshal it using various techniques.

Generally, you should avoid using unsafe code whenever possible, if only to get as much help as possible from the compiler. The "Understanding the Effects of Pointers" section tells you about managed alternatives that mimic pointers. In short, while unsafe code is a necessity when working with the Win32 API, you should avoid it whenever possible.

Understanding the Effects of Pointers

One of the first issues that you'll face when working with the Win32 API is the use of pointers—the Win32 API uses them by the gross. You'll find pointers as function arguments, within structures, and even nested within each other. The problem with pointers is that they aren't objects; they really aren't anything. A pointer is an abstraction, an address for something real. The pointer to your house is the street address found on letters and packages. The .NET Framework refrains from relying on pointers (from a developer's perspective) and uses the actual object whenever possible. The pointers are still there; CLR simply manages them for you.

As mentioned in the previous section, you can use actual pointers in C# if you're also willing to deal with the problems of unsafe code. Unlike other .NET languages, C# embraces C++ type pointers, which makes it ideal for creating low-level routines and even wrapper DLLs in many situations. However, there are many ways to mimic pointers so that you can gain the benefits of the Win32 API without losing the benefits of the managed environment.

The first thing to consider is that pointers aren't always necessary. For example, the code in Listing 1.1 works fine without pointers because we're passing values to the Win32 API and not expecting anything in return. Avoid pointers whenever possible by verifying the need for them first. In many situations, you can simply pass a value to the Win32 API when a return value isn't needed by your application.

Another issue to consider is the use of pointer substitutes. Look at the `FillConsoleOutputCharacter()` method declaration in Listing 1.2 and you'll notice that it relies on the `out` keyword to return the number of characters written to the screen. An `IntPtr` easily handles the console output handle (essentially a pointer to a pointer). In fact, you can place this use of an `IntPtr` in your rules of thumb book. Generally, you can replace a handle with an `IntPtr` for all Win32 API calls.

Sometimes you must use a pointer—there simply isn't any way around the issue. For example, you'll often find that COM calls require pointers to pointers, such as when you want to work with an interface. In this situation, you might find it impossible to develop a substitute for pointers. When this problem occurs, try to localize the pointer code to a special function, even if it might not make sense to create a separate function from a program flow perspective. Placing the pointer in its own function makes it simpler to work with the pointer, reduces the probability of missed pointer errors, and makes it easier to debug the application later.

Advantages for the C# Developer

C# developers have certain advantages when using the Win32 API. We've already discussed some of these advantages, but the most important is support for pointers and unsafe code. However, C# has some other advantages and I'd be remiss not to mention them.

C-like Language Structures Most of the information you'll need to access Win32 API is found in the C header files that come with Visual Studio. In fact, when you research a function in the Visual Studio help files, the information is often presented using C header file entries. While C# isn't C, it does have many of the same features, making conversion a lot easier than other languages.

Direct Language Conversion It's possible to recreate most C structures using C# without much effort. In fact, several of the examples in the book use the content from the C header files with small changes to account for language differences between C and C#. Because you don't have to interpret the structures, you'll find that writing the code to access the Win32 API from C# is relatively easy. The only time you'll run into problems is when you need to write code for complex COM interfaces and methods.

Less Language Baggage Generally, you'll find that if the .NET Framework doesn't support a Win32 API feature, then C# doesn't support it either. Knowing this fact saves time because you don't have to research the language to discover if it provides the required support. Of course, this could also be viewed as a negative because C# will require Win32 API function calls more often than languages that do provide robust language support for Windows features.

Better Microsoft Support It may be a quirk, but every time someone from Microsoft demonstrates a low-level language example for .NET, it appears in C# before it appears in any other language. C# is also the language of choice on newsgroups and on Web sites in many cases. Visual Basic is next on the list. Interestingly enough, the language most capable of handling Win32 API calls is the one that is seldom used—Visual C++.

Win32 Access for Visual Basic Developers

In the past, Advanced Visual Basic developers were used to accessing Win32 API functions because Visual Basic has always had certain holes in its coverage of Windows features. From this perspective, nothing has changed for Visual Basic .NET developers. What has changed is that you now have the additional hurdle of working with managed code when accessing the Win32 API, and this can make a significant difference.

WARNING Don't get the idea that you can use your old Visual Basic code directly in Visual Basic .NET. Some developers have stated that Microsoft created an entirely new language when they developed Visual Basic .NET. While this view might not be strictly true, it's true that your old code won't run as is—even the Win32 API access code. Your old code does provide a starting point, however, so make sure you use it as a reference as you develop your new Visual Basic .NET code.

Visual Basic still offers ease of use features that C# doesn't have. You can still prototype applications quickly using very little code. Unfortunately, the addition of managed code has put Visual Basic developers at a decided disadvantage in the Win32 API access arena. There are certain types of Win32 API access that you simply can't create using Visual Basic because it lacks support for unsafe code and pointers.

The following sections detail the advantages and disadvantages of using Visual Basic to access the Win32 API. At times you'll consider the disadvantages more important and may even decide to implement the Win32 API access using a wrapper DLL. However, Visual Basic does have features that make it the best language choice, in some cases, and we'll discuss them as well.

NOTE Throughout the book, you'll see examples in both Visual Basic and C#. If it's possible to perform an access task in both languages, you'll find the example in both languages on the CD, even if the Visual Basic source code doesn't appear in the text. You can create every example in the book using C#, but some examples are beyond the capabilities of Visual Basic .NET. Whenever an example fails in Visual Basic, the book will include an explanation of the problem and provide you with some alternatives whenever possible. There are some examples where you'll have to rely on C# or Visual C++ to perform the task.

Understanding Visual Basic Limitations

The biggest limitations for Visual Basic .NET developers are lack of unsafe code and lack of pointer support. You can get around some of these limitations using the techniques in the "Understanding the Effects of Pointers" section of the chapter. Essentially, you need to be

able to provide the input to the Win32 API call using something other than a pointer, which often means a either compromise or not using the call at all.

Visual Basic developers also have language problems to overcome. If you want to use the Win32 API, you also need to know how the C header files work, which means having some knowledge of the C language. Many Visual Basic developers lack this knowledge, making it difficult to create a Visual Basic version of a structure, function call, or other construct originally written in C.

In most cases, Visual Basic developers will find it difficult to re-create complex COM interfaces. For example, the MMC example defies implementation in Visual Basic because it relies heavily on COM interface simulation. In fact, this particular task is barely doable in C# and you still need to create a Visual C++ wrapper for certain MMC function calls. In short, some tasks will defy every effort to complete in Visual Basic because there's no conduit for communication with the Win32 API.

Another problem with Visual Basic is that you can't re-create some of the stranger Win32 API structures. For example, some structures include unions, which is a feature that Visual Basic doesn't support. Unfortunately, there isn't any workaround for this problem other than to emulate the union in some other manner. In many cases, there isn't any way to emulate the union, making it impossible to call the Win32 API function that relies on the structure in question.

One of the advantages of Visual Basic is also a disadvantage. Developers gain a significant development speed boost by using Visual Basic. It enables a developer to prototype applications quickly. Coding and debugging are equally fast in most cases. All of these features come with a price, however, a lack of contact with the lower-level functions of the operating system. Visual Basic hides a lot of the usual operating system plumbing from the developer—a bonus when you don't require such access and a problem when you do.

Advantages for the Visual Basic Developer

Visual Basic .NET does have some limitations when it comes to Win32 API access, but it also has some advantages. Faster development time is just one advantage we've discussed so far and it's an important issue in a world where speed is everything. However, there are other factors in favor of Visual Basic and the following list tells you about them.

Existing Code Even though you can't use existing Visual Basic code to access the Win32 API, you can use it as a source of information, and that's worth quite a bit to developers on a time schedule. The existing code is well understood, debugged, and ready to use. Simple Win32 API calls present the least number of problems for the Visual Basic developer. For example, the various beep function calls examined in Listing 1.1 present few problems because they require basic input and no output.

Stronger Language Support Remember that it's only necessary to call the Win32 API if the .NET Framework and the language lack support for a Windows feature. For example, C# lacks support for any type of beep function, so we need to create one. However, Visual Basic doesn't have this lack—it supplies a beep function, so you don't even need to use the Win32 API in this case. Visual Basic provides more built-in features than many other languages, making Win32 API access unnecessary in the first place.

Where Do You Go from Here?

This chapter has introduced you to the needs and requirements for Win32 API access from .NET languages. We've discussed some of the potential problems of working with the Win32 API and why you need to exercise care when making a Win32 call. This chapter has also pointed out some areas where the .NET Framework lacks certain types of support, so the need to use the Win32 API is very real.

Make sure you run the examples in this chapter because they demonstrate some of the essential principles we'll discuss in detail as the book progresses. It's also important to begin learning the rules of thumb presented throughout the chapter. For example, you should only use pointers when necessary in an application; otherwise, you might find it difficult to troubleshoot an errant pointer or figure out why an application misbehaves in some strange way.

As part of the preparation for this book, you'll want to know how to work in the .NET environment using either Visual C# or Visual Basic. It's important to know how the .NET Framework is put together and how you use it within an application. Consequently, you might want to use my previous book, *Visual C# .NET Developer's Handbook* (ISBN 0782140475, Sybex 2002) as an aid to learning C# at the intermediate level.

Chapter 2 begins the process of working with the Win32 API. We'll discuss data in its various forms. You'll learn about everything from simple variables to structures to enumerations. Remember that the Win32 API relies on unmanaged data, so you always need to consider data conversion a part of the calling process. In some cases, CLR will help you with the data conversion; but in many other situations, you'll need to create your own solutions. Chapter 2 is your key to making good data conversion decisions.