Using Win32 and Other Libraries

Eric Gunnerson
Microsoft Corporation

September 16, 2002

Download Csharp09192002_sample.exe.

In the last column, I presented an overview on the different ways of using existing code from within C#. This time, we're going to delve into using Win32® and other existing libraries from within our code.

Two common questions from C# users are, "Why do I have to write special code to use things that are built into Windows®? Why isn't there something in the framework to do this for me?" When the frameworks team was building their part of .NET, they looked at what they needed to do to make Win32 available to .NET programmers, and found out that the Win32 API set is *huge*. They didn't have the resources to code, test, and document managed interfaces for all of Win32, so they had to prioritize and focus on the most important ones. Many common operations have managed interfaces, but there are whole sections of Win32 that aren't covered.

Platform Invoke (P/Invoke) is the most common way to do this. To use P/Invoke, you write a prototype that describes how the function should be called, and then the runtime uses this information to make the call. The other way to do this is by wrapping the functions using the Managed Extensions to C++, which I'll cover in a future column.

The best way to understand how to do this is by example. In some cases, I'll just present part of the code; the entire code is available as part of the download.

A Simple Example

In our first example, we'll call the `Beep()` API to make some sound. To start, I need to write the proper definition for `Beep()`. Looking at the definition in MSDN, I find that it has the following prototype:

```
BOOL Beep(
  DWORD dwFreq,       // sound frequency
  DWORD dwDuration    // sound duration
);
```

To write this in C#, I need to translate the Win32 types to appropriate C# types. Since `DWORD` is a 4-byte integer, we could either use `int` or `uint` as the C# analog. Since int is a CLS-compliant type (and can therefore be used by all .NET languages), it's the more common choice than `uint`, and in most cases, the difference isn't important. The `bool` type is the analog of `BOOL`. We therefore can write the following prototype in C#:

```
public static extern bool Beep(int frequency, int duration);
```

This is a fairly standard definition, except that we've used extern to indicate that the actual code for this function is somewhere else. This prototype will tell the runtime how to call the function; we now need to tell it where to find the function.

A trip back to MSDN is in order. In the reference information, we find that `Beep()` is defined in `kernel32.lib`. That means that the runtime code is contained in `kernel32.dll`. We put a DllImport attribute on our prototype to tell that to the runtime:

```
[DllImport("kernel32.dll")]
```

That's all we need to do. Here's a complete example that generates the kind of random notes that were so common

in bad 1960s science fiction movies.

```
using System;
using System.Runtime.InteropServices;

namespace Beep
{
   class Class1
   {
      [DllImport("kernel32.dll")]
      public static extern bool Beep(int frequency, int duration);

      static void Main(string[] args)
      {
         Random random = new Random();

         for (int i = 0; i < 10000; i++)
         {
            Beep(random.Next(10000), 100);
         }
      }
   }
}
```

This is pretty much guaranteed to annoy anybody within earshot. Because DllImport allows you to call arbitrary code in Win32, there is a chance of malicious code. The runtime therefore requires you to be a fully trusted user to make P/Invoke calls.

Enums and Constants

`Beep()` is fine to play an arbitrary sound, but sometimes we want to play the sound that's associated with a specific sound type. We'll use `MessageBeep()` instead. MSDN yields the following prototype:

```
BOOL MessageBeep(
  UINT uType   // sound type
);
```

That looks simple. Reading the remarks, however, yields two interesting facts.

First, the `uType` parameter really takes a set of predefined constants.

Second, one of the possible parameter value is -1, which means that although it's defined as taking a `uint`, an `int` is more appropriate.

Using an enum is the logical thing to do for `uType` parameter. MSDN lists the named constants, but it doesn't give any hints for what the values are. For that, we'll need to look at the actual APIs.

If you have Visual Studio® and you installed C++, the Platform SDK lives at **\Program Files\Microsoft Visual Studio .NET\Vc7\PlatformSDK\Include**.

To find the constants, I just did a findstr in that directory:

```
findstr "MB_ICONHAND" *.h
```

It located the constants in `winuser.h`, and I used them to create my enum and prototype:

```
public enum BeepType
{
   SimpleBeep = -1,
   IconAsterisk = 0x00000040,
   IconExclamation = 0x00000030,
   IconHand = 0x00000010,
   IconQuestion = 0x00000020,
   Ok = 0x00000000,
```

```
}

[DllImport("user32.dll")]
public static extern bool MessageBeep(BeepType beepType);
```

Now I can call it with:

```
MessageBeep(BeepType.IconQuestion);
```

Dealing with Structures

I've sometimes wanted to be able to figure out what the battery status is on my laptop. Win32 provides power management functions to obtain this information.

A bit of searching on MSDN leads to the `GetSystemPowerStatus()` function.

```
BOOL GetSystemPowerStatus(
  LPSYSTEM_POWER_STATUS lpSystemPowerStatus
);
```

This function takes a pointer to a structure, which is something we haven't dealt with yet. To work with structures, we need to define a structure in C#. We'll start with the unmanaged definition:

```
typedef struct _SYSTEM_POWER_STATUS {
  BYTE   ACLineStatus;
  BYTE   BatteryFlag;
  BYTE   BatteryLifePercent;
  BYTE   Reserved1;
  DWORD  BatteryLifeTime;
  DWORD  BatteryFullLifeTime;
} SYSTEM_POWER_STATUS, *LPSYSTEM_POWER_STATUS;
```

And then write a C# version by replacing the C types with C# ones.

```
struct SystemPowerStatus
{
   byte ACLineStatus;
   byte batteryFlag;
   byte batteryLifePercent;
   byte reserved1;
   int  batteryLifeTime;
   int  batteryFullLifeTime;
}
```

It is then simple to write the C# prototype:

```
[DllImport("kernel32.dll")]
public static extern bool GetSystemPowerStatus(
   ref SystemPowerStatus systemPowerStatus);
```

In this prototype, we use "`ref`" to indicate that we're passing a pointer to the structure rather than the structure by value. This is the normal method of dealing with structures that are passed by pointers.

This function works well, but the `ACLineStatus` and `batteryFlag` fields are better defined as enums:

```
   enum ACLineStatus: byte
   {
      Offline = 0,
      Online = 1,
      Unknown = 255,
   }

   enum BatteryFlag: byte
```

```
    {
       High = 1,
       Low = 2,
       Critical = 4,
       Charging = 8,
       NoSystemBattery = 128,
       Unknown = 255,
    }
```

Note that because the structure fields are bytes, we use `byte` as the base type for the enum.

Strings

While there is only one .NET string type, there are several flavors in the unmanaged world. There are character pointers and structures with embedded character arrays, each of which will need to be correctly marshaled.

There are also two different string representations used on Win32:

- ANSI
- Unicode

The original line of Windows used single-byte characters, which were economical in terms of storage space, but had a complex multi-byte encoding for many languages. When Windows NT® appeared, it used a two-byte Unicode encoding. To deal with this difference, the Win32 API uses a clever trick. It defines a `TCHAR` type, which is a single-byte char on Win9x platforms, and a two-byte Unicode char on WinNT platforms. For every function that takes a string or structure contain character data, it defines two versions of that structure, with an `A` suffix indicating that it is Ansi, and `W` indicating that it is wide (ie Unicode). If you compile a C++ program for single-byte, you get the `A` variant, and if you compile for Unicode, you get the `W` variant. The Win9x platforms contain the Ansi version, and the WinNT ones contain the `W` one.

Since the designers of P/Invoke didn't want you to have to deal with figuring out what platform you're on, they provided built-in support to use either the `A` or the `W` version automatically. If the function you call doesn't exist, the interop layer will look for the `A` or `W` version for you and use that instead.

There are a few subtleties to string support that are best demonstrated by example.

Simple Strings

Here's a simple example of a function that takes a string parameter:

```
BOOL GetDiskFreeSpace(
  LPCTSTR lpRootPathName,          // root path
  LPDWORD lpSectorsPerCluster,     // sectors per cluster
  LPDWORD lpBytesPerSector,        // bytes per sector
  LPDWORD lpNumberOfFreeClusters,  // free clusters
  LPDWORD lpTotalNumberOfClusters  // total clusters
);
```

The root path is defined as a `LPCTSTR`. This is the platform-independent version of a string pointer.

Since there is no function named `GetDiskFreeSpace()`, the marshaler will automatically look for the "`A`" or "`W`" variant, and call the appropriate function. We'll use an attribute to tell the marshaler what string type the API requires.

Here's the full definition for the function, as I first defined it:

```
[DllImport("kernel32.dll")]
static extern bool GetDiskFreeSpace(
   [MarshalAs(UnmanagedType.LPTStr)]
   string rootPathName,
   ref int sectorsPerCluster,
   ref int bytesPerSector,
```

```
    ref int numberOfFreeClusters,
    ref int totalNumberOfClusters);
```

Unfortunately, when I tried this, it didn't work. The problem is that by default, the marshaler tries to find the Ansi version of an API regardless of what system we're on, and since the `LPTStr` means that Unicode strings should be used on Windows NT platforms, we end up trying to call the Ansi function with a Unicode string, which doesn't work.

There are two ways to make this work. The simple way is just to remove the `MarshalAs` attribute. If you do this, you'll always be calling the `A` version of the function, which is fine if it exists on all platform variants you care about. If you do this, however, your code will be slower because the marshaler will convert your .NET string from Unicode to multi-byte, call the `A` version of the function, which will convert the string back to Unicode, and then call the `W` version of the function.

To prevent this, you'll need to tell the marshaler that you want it to look for the `A` version when it's on Win9x platforms and the W version on NT platforms. You do this by setting the `CharSet` as part of the `DllImport` attribute:

```
[DllImport("kernel32.dll", CharSet = CharSet.Auto)]
```

In my unscientific timings, I found that this was about 5 percent faster than the version the previous option.

Setting the `CharSet` attribute and using `LPTStr` for string types will work for most of the Win32 API. There are other functions, however, that don't use the `A`/`W` mechanism, for which you'll have to do something different.

String Buffers

The string type in .NET is an immutable type, which means that it will never have its value changed. For functions that will copy a string value into a string buffer, a string won't work. At best, doing so will corrupt the temporary buffer created by the marshaler when a string is translated. At worst, it will corrupt the managed heap, which generally causes bad things to happen. Either way, you are unlikely to get the correct value back.

To get this to work, we'll need to use a different type. The `StringBuilder` type is designed to act as a buffer, and we'll use that instead of string. Here's an example:

```
[DllImport("kernel32.dll", CharSet = CharSet.Auto)]
public static extern int GetShortPathName(
   [MarshalAs(UnmanagedType.LPTStr)]
   string path,
   [MarshalAs(UnmanagedType.LPTStr)]
   StringBuilder shortPath,
   int shortPathLength);
```

To use this function is simple:

```
StringBuilder shortPath = new StringBuilder(80);
int result = GetShortPathName(
@"d:\test.jpg", shortPath, shortPath.Capacity);
string s = shortPath.ToString();
```

Note that the `Capacity` of the `StringBuilder` is passed as the size of the buffer.

Structures with Embedded Char Arrays

Some functions take structures that have embedded character arrays. For example, the `GetTimeZoneInformation ()` function takes a pointer to the following struct:

```
typedef struct _TIME_ZONE_INFORMATION {
    LONG       Bias;
    WCHAR      StandardName[ 32 ];
    SYSTEMTIME StandardDate;
    LONG       StandardBias;
    WCHAR      DaylightName[ 32 ];
```

```
    SYSTEMTIME DaylightDate;
    LONG       DaylightBias;
} TIME_ZONE_INFORMATION, *PTIME_ZONE_INFORMATION;
```

Using this from C# requires two structures. The `SYSTEMTIME` one is simple to set up:

```
struct SystemTime
{
   public short wYear;
   public short wMonth;
   public short wDayOfWeek;
   public short wDay;
   public short wHour;
   public short wMinute;
   public short wSecond;
   public short wMilliseconds;
}
```

Nothing surprising there; the definition for `TimeZoneInformation` is a bit more complex:

```
[StructLayout(LayoutKind.Sequential, CharSet = CharSet.Unicode)]
struct TimeZoneInformation
{
   public int bias;
   [MarshalAs(UnmanagedType.ByValTStr, SizeConst = 32)]
   public string standardName;
   SystemTime standardDate;
   public int standardBias;
   [MarshalAs(UnmanagedType.ByValTStr, SizeConst = 32)]
   public string daylightName;
   SystemTime daylightDate;
   public int daylightBias;
}
```

There are two important details of this definition. The first is the `MarshalAs` attribute:

```
[MarshalAs(UnmanagedType.ByValTStr, SizeConst = 32)]
```

Taking a look at the documents for `ByValTStr`, we see that it's used for embedded character arrays. `SizeConst` is used to set the size of the arrays.

When I first coded this, I ran into execution engine errors. This usually means that you're overwriting some memory as part of the interop, which suggested the size of the structure was wrong. I used Marshal.SizeOf() to get the size the marshaller was using, and it was 108 bytes. I did a bit of investigation, and soon remembered that the default character type for interop is Ansi, or single-byte. The ones in the function def are typed as `WCHAR`, which is two bytes, therefore leading to the problem.

I fixed it by adding the `StructLayout` attribute. Sequential layout is the default for structs, and means that all fields are layed out in the order they're listed. The `CharSet` value is set to Unicode to always use the proper char type.

Once I did that, the function worked fine. You may wonder why I didn't use `CharSet.Auto` on this function. This is one of those functions that doesn't have `A` and `W` variants; it always uses Unicode strings, so I've hard-coded it that way.

Functions with Callbacks

When Win32 functions need to return more than one item of data, they typically do it through a callback mechanism. The developer passes a function pointer to the function, and the developer's function is called for each item that is enumerated.

Instead of having function pointers, C# has delegates, and they are used as a replacement for function pointers when calling Win32 functions.

An example of such a function is the `EnumDesktops()` function:

```
BOOL EnumDesktops(
  HWINSTA hwinsta,            // handle to window station
  DESKTOPENUMPROC lpEnumFunc, // callback function
  LPARAM lParam               // value for callback function
);
```

The `HWINSTA` type is replaced with an `IntPtr`, and `LPARAM` is replaced with an int. `DESKTOPENUMPROC` takes a bit more work. Here's the definition from MSDN:

```
BOOL CALLBACK EnumDesktopProc(
  LPTSTR lpszDesktop,  // desktop name
  LPARAM lParam        // user-defined value
);
```

We can convert this to the following delegate:

```
delegate bool EnumDesktopProc(
   [MarshalAs(UnmanagedType.LPTStr)]
   string desktopName,
   int lParam);
```

Once that's defined, we can write the definition for `EnumDesktops()`:

```
[DllImport("user32.dll", CharSet = CharSet.Auto)]
static extern bool EnumDesktops(
   IntPtr windowStation,
   EnumDesktopProc callback,
   int lParam);
```

That's enough to get the function up and running.

There's one important tip when using delegates in interop. The marshaler creates a function pointer that refers to the delegate, and it's that function pointer that's passed to the unmanaged function. The marshaler can't figure out, however, what the unmanaged function does with the function pointer, so it assumes that it only needs to be valid during the call to the function.

The upshot of this is that if you are calling a function such as `SetConsoleCtrlHandler()`, where the function pointer is saved for later use, you'll need to make sure that the delegate is referenced in your code. If you don't, the function may appear to work, but a future garbage collection will delete the delegate, and bad things will happen.

More Advanced Functions

All the examples I've done so far have been relatively straightforward, but there are many Win32 functions that are more complex. Here's an example:

```
DWORD SetEntriesInAcl(
  ULONG cCountOfExplicitEntries,           // number of entries
  PEXPLICIT_ACCESS pListOfExplicitEntries, // buffer
  PACL OldAcl,                             // original ACL
  PACL *NewAcl                             // new ACL
);
```

The first two parameters can be handled fairly easily. The `ulong` is easy, and the buffer can be marshaled using `UnmanagedType.LPArray`.

The third and fourth parameters present some problems. The problem is the way that an `ACL` is defined. The `ACL` structure only defines the header of an ACL, and the rest of the buffer is made up of `ACEs`. The `ACE` can be one of several different types of `ACEs`, and these `ACEs` are different lengths.

It's possible to deal with this in C#, if you're willing to do all the buffer allocation and use a fair bit of unsafe code. It's going to be a **ton** of work, however, and it will be very tough to debug. It's much easier to do this API in C++

Other Options for the Attributes

The `DLLImport` and `StructLayout` attributes have a number of options that are useful in using P/Invoke. I've listed all of them for completeness.

DLLImport

CallingConvention

You can use this to tell the marshaler what calling convention the function uses. You will want to set this to the calling convention of your function. In general, if you get this wrong, your code won't work. If, however, your function is a `Cdecl` function and you call it with `StdCall` (the default), your function will work, but the function parameters will never be removed from the stack, which can lead to filling up the stack.

CharSet

Controls whether the `A` or `W` variants are called.

EntryPoint

The property is used to set the name that the marshaler looks for in the DLL. When you set this property, you can then rename your C# function to whatever you want.

ExactSpelling

Set this to true, and the marshaler turns off the `A` and `W` lookup stuff. The documents on this attribute are backwards.

PreserveSig

COM interop makes a function with a final out parameter look like it returns that value. This property turns that off.

SetLastError

Makes sure that the Win32 API `SetLastError()` is called, so that you can find out what happened.

StructLayout

LayoutKind

The default layout for structs is sequential, and this works in most cases. If you need total control over where the struct members are placed, you can use `LayoutKind.Explicit` and then put a `FieldOffset` attribute on every member of the struct. This is typically done when you need to create a union.

CharSet

Controls what the default char type is for `ByValTStr` members.

Pack

Sets the packing size of the structure. This controls how the structure will be aligned. If the C structure uses a different packing, you may need to set this.

Size

Sets the size of the structure. Not generally used, but can be of some use if you need extra allocated space at the end of the structure.

Loading from Different Locations

There is no way to specify where you would like DLLImport to look for a file at runtime. There is, however, a trick that you can use to get this to work.

DllImport calls `LoadLibrary()` to do its work. If a specific DLL has already been loaded into a process, `LoadLibrary()` will succeed, even if the specified path for the load is different.

This means that if you call `LoadLibrary()` directly, you can load your DLL from wherever you want, and then the DllImport `LoadLibrary()` will use that version.

Because of this behavior, it's possible that `LoadLibrary()` can be called ahead of time to forward your calls to a different DLL. If you're writing a library, you can prevent this by calling `GetModuleHandle()` to make sure that the library hasn't been loaded previously before you make your first P/Invoke call.

Troubleshooting P/Invoke

If your P/Invoke calls are failing, it's usually because some types are defined incorrectly. Here are a few common problems:

- `Long` != `long`. In C++, `long` is a 4 byte integer, but in C#, it's an 8 byte integer.
- Not setting the string type correctly.

Next Month

Next month, we'll delve into wrapping functions in C++.

Eric Gunnerson is a Program Manager on the Visual C# team, a member of the C# design team, and the author of A Programmer's Introduction to C#. He's been programming long enough that he knows what 8-inch diskettes are and could once mount tapes with one hand. In his spare time, he's pursuing a doctorate in cat juggling with a concentration on Persians.