

Platform Invoke (P/Invoke)

Article • 05/10/2024

P/Invoke is a technology that allows you to access structs, callbacks, and functions in unmanaged libraries from your managed code. Most of the P/Invoke API is contained in two namespaces: `System` and `System.Runtime.InteropServices`. Using these two namespaces give you the tools to describe how you want to communicate with the native component.

Let's start from the most common example, and that is calling unmanaged functions in your managed code. Let's show a message box from a command-line application:

C#

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

public partial class Program
{
    // Import user32.dll (containing the function we need) and define
    // the method corresponding to the native function.
    [LibraryImport("user32.dll", StringMarshalling =
StringMarshalling.Utf16, SetLastError = true)]
    private static partial int MessageBox(IntPtr hWnd, string lpText,
string lpCaption, uint uType);

    public static void Main(string[] args)
    {
        // Invoke the function as a regular managed method.
        MessageBox(IntPtr.Zero, "Command-line message box", "Atten-
tion!", 0);
    }
}
```

The previous example is simple, but it does show off what's needed to invoke unmanaged functions from managed code. Let's step through the example:

- Line #2 shows the `using` directive for the `System.Runtime.InteropServices` namespace that holds all the items needed.
- Line #8 introduces the `LibraryImportAttribute` attribute. This attribute tells the runtime that it should load the unmanaged binary. The string passed in is the unmanaged binary that contains the target function. Additionally, it specifies the encoding to use for marshalling the strings. Finally, it specifies that this function calls `SetLastError` and that the runtime should capture that error code so the user can retrieve it via `Marshal.GetLastPInvokeError()`.

- Line #9 is the crux of the P/Invoke work. It defines a managed method that has the **exact same signature** as the unmanaged one. The declaration uses the `LibraryImport` attribute and the `partial` keyword to tell a compiler extension to generate code to call into the unmanaged library.
 - Within the generated code and prior to .NET 7, the `DllImport` is used. This declaration uses the `extern` keyword to indicate to the runtime this is an external method, and that when you invoke it, the runtime should find it in the unmanaged binary specified in the `DllImport` attribute.

The rest of the example is invoking the method as you would any other managed method.

The sample is similar for macOS. The name of the library in the `LibraryImport` attribute needs to change since macOS has a different scheme of naming dynamic libraries. The following sample uses the `getpid(2)` function to get the process ID of the application and print it out to the console:

C#

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

namespace PInvokeSamples
{
    public static partial class Program
    {
        // Import the libSystem shared library and define the method
        // corresponding to the native function.
        [LibraryImport("libSystem.dylib")]
        private static partial int getpid();

        public static void Main(string[] args)
        {
            // Invoke the function and get the process ID.
            int pid = getpid();
            Console.WriteLine(pid);
        }
    }
}
```

It is also similar on Linux. The function name is the same, since `getpid(2)` is a standard [POSIX](#) system call.

C#

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

```

namespace PInvokeSamples
{
    public static partial class Program
    {
        // Import the libc shared library and define the method
        // corresponding to the native function.
        [LibraryImport("libc.so.6")]
        private static partial int getpid();

        public static void Main(string[] args)
        {
            // Invoke the function and get the process ID.
            int pid = getpid();
            Console.WriteLine(pid);
        }
    }
}

```

Invoking managed code from unmanaged code

The runtime allows communication to flow in both directions, enabling you to call back into managed code from native functions by using function pointers. The closest thing to a function pointer in managed code is a **delegate**, so this is what is used to allow callbacks from native code into managed code.

The way to use this feature is similar to the managed to native process previously described. For a given callback, you define a delegate that matches the signature and pass that into the external method. The runtime will take care of everything else.

C#

```

using System;
using System.Runtime.InteropServices;

namespace ConsoleApplication1
{
    public static partial class Program
    {
        // Define a delegate that corresponds to the unmanaged func-
        // tion.
        private delegate bool EnumWC(IntPtr hwnd, IntPtr lParam);

        // Import user32.dll (containing the function we need) and
        // define
        // the method corresponding to the native function.
        [LibraryImport("user32.dll")]
        private static partial int EnumWindows(EnumWC lpEnumFunc,
        IntPtr lParam);
    }
}

```

```

        // Define the implementation of the delegate; here, we simply
        output the window handle.
        private static bool OutputWindow(IntPtr hwnd, IntPtr lParam)
        {
            Console.WriteLine(hwnd.ToInt64());
            return true;
        }

        public static void Main(string[] args)
        {
            // Invoke the method; note the delegate as a first parameter.
            EnumWindows(OutputWindow, IntPtr.Zero);
        }
    }
}

```

Before walking through the example, it's good to review the signatures of the unmanaged functions you need to work with. The function to be called to enumerate all of the windows has the following signature: `BOOL EnumWindows (WNDENUMPROC lpEnumFunc, LPARAM lParam);`

The first parameter is a callback. The said callback has the following signature: `BOOL CALLBACK EnumWindowsProc (HWND hwnd, LPARAM lParam);`

Now, let's walk through the example:

- Line #9 in the example defines a delegate that matches the signature of the callback from unmanaged code. Notice how the LPARAM and HWND types are represented using `IntPtr` in the managed code.
- Lines #13 and #14 introduce the `EnumWindows` function from the user32.dll library.
- Lines #17 - 20 implement the delegate. For this simple example, we just want to output the handle to the console.
- Finally, in line #24, the external method is called and passed in the delegate.

The Linux and macOS examples are shown below. For them, we use the `ftw` function that can be found in `libc`, the C library. This function is used to traverse directory hierarchies and it takes a pointer to a function as one of its parameters. The said function has the following signature: `int (*fn) (const char *fpath, const struct stat *sb, int typeflag).`

```

C#

using System;
using System.Runtime.InteropServices;

namespace PInvokeSamples

```

```

{
    public static partial class Program
    {
        // Define a delegate that has the same signature as the native function.
        private delegate int DirClbk(string fName, ref Stat stat, int typeFlag);

        // Import the libc and define the method to represent the native function.
        [LibraryImport("libc.so.6", StringMarshalling = StringMarshalling.Utf16)]
        private static partial int ftw(string dirpath, DirClbk cl, int descriptors);

        // Implement the above DirClbk delegate;
        // this one just prints out the filename that is passed to it.
        private static int DisplayEntry(string fName, ref Stat stat, int typeFlag)
        {
            Console.WriteLine(fName);
            return 0;
        }

        public static void Main(string[] args)
        {
            // Call the native function.
            // Note the second parameter which represents the delegate (callback).
            ftw(".", DisplayEntry, 10);
        }
    }

    // The native callback takes a pointer to a struct. This type represents that struct in managed code.
    [StructLayout(LayoutKind.Sequential)]
    public struct Stat
    {
        public uint DeviceID;
        public uint InodeNumber;
        public uint Mode;
        public uint HardLinks;
        public uint UserID;
        public uint GroupID;
        public uint SpecialDeviceID;
        public ulong Size;
        public ulong BlockSize;
        public uint Blocks;
        public long TimeLastAccess;
        public long TimeLastModification;
        public long TimeLastStatusChange;
    }
}

```

macOS example uses the same function, and the only difference is the argument to the `LibraryImport` attribute, as macOS keeps `libc` in a different place.

C#

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

namespace PInvokeSamples
{
    public static partial class Program
    {
        // Define a delegate that has the same signature as the native function.
        private delegate int DirClbk(string fName, ref Stat stat, int typeFlag);

        // Import the libc and define the method to represent the native function.
        [LibraryImport("libSystem.dylib", StringMarshalling = StringMarshalling.Utf16)]
        private static partial int ftw(string dirpath, DirClbk cl, int descriptors);

        // Implement the above DirClbk delegate;
        // this one just prints out the filename that is passed to it.
        private static int DisplayEntry(string fName, ref Stat stat, int typeFlag)
        {
            Console.WriteLine(fName);
            return 0;
        }



        public static void Main(string[] args)
        {
            // Call the native function.
            // Note the second parameter which represents the delegate (callback).
            ftw(".", DisplayEntry, 10);
        }
    }

    // The native callback takes a pointer to a struct. This type represents that struct in managed code.
    [StructLayout(LayoutKind.Sequential)]
    public struct Stat
    {
        public uint DeviceID;
        public uint InodeNumber;
        public uint Mode;
        public uint HardLinks;
        public uint UserID;
        public uint GroupID;
    }
}
```

```
    public uint SpecialDeviceID;
    public ulong Size;
    public ulong BlockSize;
    public uint Blocks;
    public long TimeLastAccess;
    public long TimeLastModification;
    public long TimeLastStatusChange;
}
}
```

Both of the previous examples depend on parameters, and in both cases, the parameters are given as managed types. Runtime does the "right thing" and processes these into its equivalents on the other side. Learn about how types are marshalled to native code in our page on [Type marshalling](#).

More resources

- [Writing cross platform P/Invokes](#)
- [Source generated P/Invoke marshalling](#)
- [C#/Win32 P/Invoke source generator](#)  automatically generates definitions for Windows APIs.
- [P/Invoke in C++/CLI](#)
- [Mono documentation on P/Invoke](#) 

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

 .NET

.NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)