Platform Invoke Tutorial

Visual Studio .NET 2003

Platform Invocation Services (PInvoke) allows managed code to call unmanaged functions that are implemented in a DLL.

This tutorial shows you what you need to do to be able to call unmanaged DLL functions from C#. The attributes discussed in the tutorial allow you to call these functions and have data types be marshaled correctly.

Sample Files

See Platform Invoke Sample to download and build the sample files discussed in this tutorial.

Platform Invoke

Further Reading

- A Closer Look at Platform Invoke
- Using Attributes
- DllImportAttribute Class
- MarshalAsAttribute Class
- StructLayoutAttribute Class
- InAttribute Class
- OutAttribute Class

Tutorial

There are two ways that C# code can directly call unmanaged code:

- Directly call a function exported from a DLL.
- Call an interface method on a COM object (for more information, see COM Interop Part 1: C# Client Tutorial).

For both techniques, you must provide the C# compiler with a declaration of the unmanaged function, and you may also need to provide the C# compiler with a description of how to marshal the parameters and return value to and from the unmanaged code.

The tutorial consists of the following topics:

- Calling a DLL Export Directly from C#
- Default Marshaling and Specifying Custom Marshaling for Parameters to Unmanaged Methods
- Specifying Custom Marshaling for User-Defined Structs
- Registering Callback Methods

The tutorial includes the following examples:

- Example 1 Using DllImport
- Example 2 Overriding Default Marshaling
- Example 3 Specifying Custom Marshaling

Calling a DLL Export Directly from C#

To declare a method as having an implementation from a DLL export, do the following:

- Declare the method with the **static** and **extern** C# keywords.

- Attach the **DllImport** attribute to the method. The **DllImport** attribute allows you to specify the name of the DLL that contains the method. The common practice is to name the C# method the same as the exported method, but you can also use a different name for the C# method.
- Optionally, specify custom marshaling information for the method's parameters and return value, which will override the .NET Framework default marshaling.

Example 1

This example shows you how to use the **DllImport** attribute to output a message by calling puts from msvcrt.dll.

~~Copy~~

```
// PInvokeTest.cs

using System;

using System.Runtime.InteropServices;


class PlatformInvokeTest

{

    [DllImport("msvcrt.dll")]

    public static extern int puts(string c);

    [DllImport("msvcrt.dll")]

    internal static extern int _flushall();


    public static void Main()

    {

        puts("Test");

        _flushall();

    }

}
```

Output

~~Copy~~

```
Test
```

Code Discussion

The preceding example shows the minimum requirements for declaring a C# method that is implemented in an unmanaged DLL. The method PlatformInvokeTest.puts is declared with the **static** and **extern** modifiers and has the **DllImport** attribute which tells the compiler that the implementation comes from msvcrt.dll, using the

default name of puts. To use a different name for the C# method such as putstring, you must use the **EntryPoint** option in the **DllImport** attribute, that is:

~~Copy~~

```
[DllImport("msvcrt.dll", EntryPoint="puts")]
```

For more information on the syntax of the **DllImport** attribute, see ~~DllImportAttribute Class~~.

Default Marshaling and Specifying Custom Marshaling for Parameters to Unmanaged Methods

When calling an unmanaged function from C# code, the common language runtime must marshal the parameters and return values.

For every .NET Framework type there is a default unmanaged type, which the common language runtime will use to marshal data across a managed to unmanaged function call. For example, the default marshaling for C# string values is to the type LPTSTR (pointer to TCHAR char buffer). You can override the default marshaling using the **MarshalAs** attribute in the C# declaration of the unmanaged function.

Example 2

This example uses the **DllImport** attribute to output a string. It also shows you how to override the default marshaling of the function parameters by using the **MarshalAs** attribute.

~~Copy~~

```
// Marshal.cs
using System;
using System.Runtime.InteropServices;


class PlatformInvokeTest
{
    [DllImport("msvcrt.dll")]
    public static extern int puts(
        [MarshalAs(UnmanagedType.LPStr)]
        string m);
    [DllImport("msvcrt.dll")]
    internal static extern int _flushall();



    public static void Main()
```

```
    {
        puts("Hello World!");
        _flushall();
    }
}
```

Output

When you run this example, the string,

Copy

```
Hello World!
```

will display at the console.

Code Discussion

In the preceding example, the default marshaling for the parameter to the puts function has been overridden from the default of LPTSTR to LPSTR.

The **MarshalAs** attribute can be placed on method parameters, method return values, and fields of structs and classes. To set the marshaling of a method return value, place the **MarshalAs** attribute in an attribute block on the method with the return attribute location override. For example, to explicitly set the marshaling for the return value of the puts method:

Copy

```
...
[DllImport("msvcrt.dll")]
[return : MarshalAs(UnmanagedType.I4)]
public static extern int puts(
...
```

For more information on the syntax of the **MarshalAs** attribute, see MarshalAsAttribute Class.

Note The **In** and **Out** attributes can be used to annotate parameters to unmanaged methods. They behave in a similar manner to the **in** and **out** modifiers in MIDL source files. Note that the **Out** attribute is different from the C# parameter modifier, out. For more information on the **In** and **Out** attributes, see InAttribute Class and OutAttribute Class.

Specifying Custom Marshaling for User-Defined Structs

You can specify custom marshaling attributes for fields of structs and classes passed to or from unmanaged functions. You do this by adding **MarshalAs** attributes to the fields of the struct or class. You must also use the **StructLayout** attribute to set the layout of the struct, optionally to control the default marshaling of string members, and to set the default packing size.

Example 3

This example demonstrates how to specify custom marshaling attributes for a struct.

Consider the following C structure:

~~Copy~~

```
typedef struct tagLOGFONT

{

   LONG lfHeight;

   LONG lfWidth;

   LONG lfEscapement;

   LONG lfOrientation;

   LONG lfWeight;

   BYTE lfItalic;

   BYTE lfUnderline;

   BYTE lfStrikeOut;

   BYTE lfCharSet;

   BYTE lfOutPrecision;

   BYTE lfClipPrecision;

   BYTE lfQuality;

   BYTE lfPitchAndFamily;

   TCHAR lfFaceName[LF_FACESIZE];

} LOGFONT;
```

In C#, you can describe the preceding struct by using the **StructLayout** and **MarshalAs** attributes as follows:

~~Copy~~

```
// logfont.cs

// compile with: /target:module

using System;

using System.Runtime.InteropServices;


[StructLayout(LayoutKind.Sequential)]
```

```
public class LOGFONT

{

    public const int LF_FACESIZE = 32;

    public int lfHeight;

    public int lfWidth;

    public int lfEscapement;

    public int lfOrientation;

    public int lfWeight;

    public byte lfItalic;

    public byte lfUnderline;

    public byte lfStrikeOut;

    public byte lfCharSet;

    public byte lfOutPrecision;

    public byte lfClipPrecision;

    public byte lfQuality;

    public byte lfPitchAndFamily;

    [MarshalAs(UnmanagedType.ByValTStr, SizeConst=LF_FACESIZE)]

    public string lfFaceName;

}
```

For more information on the syntax of the **StructLayout** attribute, see ~~StructLayoutAttribute Class~~.

The structure can then be used in C# code as shown below:

~~Copy~~

```
// pinvoke.cs

// compile with: /addmodule:logfont.netmodule

using System;

using System.Runtime.InteropServices;


class PlatformInvokeTest

{
```

```
[DllImport("gdi32.dll", CharSet=CharSet.Auto)]
public static extern IntPtr CreateFontIndirect(
      [In, MarshalAs(UnmanagedType.LPStruct)]
      LOGFONT lplf   // characteristics
      );

[DllImport("gdi32.dll")]
public static extern bool DeleteObject(
      IntPtr handle
      );

public static void Main()
{
      LOGFONT lf = new LOGFONT();
      lf.lfHeight = 9;
      lf.lfFaceName = "Arial";
      IntPtr handle = CreateFontIndirect(lf);

      if (IntPtr.Zero == handle)
      {
            Console.WriteLine("Can't creates a logical font.");
      }
      else
      {

            if (IntPtr.Size == 4)
                  Console.WriteLine("{0:X}", handle.ToInt32());
            else
                  Console.WriteLine("{0:X}", handle.ToInt64());
```

```
                // Delete the logical font created.
                if (!DeleteObject(handle))
                     Console.WriteLine("Can't delete the logical font");
            }
      }
}
```

Sample Run

~~Copy~~

```
C30A0AE5
```

Code Discussion

In the preceding example, the CreateFontIndirect method is using a parameter of the type LOGFONT. The **MarshalAs** and **In** attributes are used to qualify the parameter. The program displays the numeric value returned by the method as a hexadecimal uppercase string.

Registering Callback Methods

To register a managed callback that calls an unmanaged function, declare a delegate with the same argument list and pass an instance of it via PInvoke. On the unmanaged side it will appear as a function pointer. For more information about PInvoke and callback, see ~~A Closer Look at Platform Invoke~~.

For example, consider the following unmanaged function, MyFunction, which requires callback as one of the arguments:

~~Copy~~

```
typedef void (__stdcall *PFN_MYCALLBACK)();

int __stdcall MyFunction(PFN_ MYCALLBACK callback);
```

To call MyFunction from managed code, declare the delegate, attach DllImport to the function declaration, and optionally marshal any parameters or the return value:

~~Copy~~

```
public delegate void MyCallback();

[DllImport("MYDLL.DLL")]

public static extern void MyFunction(MyCallback callback);
```

Also, make sure the lifetime of the delegate instance covers the lifetime of the unmanaged code; otherwise, the delegate will not be available after it is garbage-collected.