Platform Invocation Services in .NET Framework

Source	http://www.c-sharpcorner.com/UploadFile/dahuja/PInvokeServices11092005062837AM/PInvokeServices.aspx

This article will cover the details of Platform Invocation Services provided in .Net Framework. Platform Invoke Services acts as a bridge between the managed and unmanaged code. This service enables managed code to call unmanaged functions which are being exposed by any dynamic link libraries (DLL's) such as Win32 API's or any custom DLLs.

.Net Platform provides two services for interoperability with the unmanaged code. These are as follows:

1. Platform Invocation Service: Platform invocation services enable the managed code to call the functions which are being exposed by the dynamic link libraries such as Win32 API's and custom DLL's.
2. COM Interop Services: which enables the managed code to interact with the COM objects.

Interoperability Marshalling is responsible for exchanging the arguments and return values in the right format between the managed and the unmanaged code. Both PInvoke and COM Interop make an intensive use of Interoperability marshalling to exchange the arguments between the caller and the callee. The Interop marshaler marshals the data between the common language runtime heap and the unmanaged heap.

COM also supports the marshalling of data between the different COM apartments or between the different COM processes. COM supports three types of marshalling:

1. Automation Marshalling (Type library Marshalling);
2. Standard Marshalling.
3. Custom Marshalling.

Let's us have a brief look on the different types of marshalling supported by the COM. Marshalling concept is hard nut to crack but its is one of the most important concept of COM.

Marshalling

The packing/unpacking of parameters and return values across the COM apartments is called Marshalling. When a client wants to interact with a COM object, which is in a different apartment, COM will setup the proxy object in the calling thread's apartment (client of the COM object) and the stub will be setup in the object's apartment. These two objects handle the marshalling of data types across the apartments. The proxy packages the method parameters, sends them to the stub and unpackaged the results returned from the COM server method. The stub handles the actual call on the COM object.

There are three type of marshalling which is being supported by COM:

1. **Automation Marshalling (Type library Marshalling).**
2. **Standard Marshalling.**
3. **Custom Marshalling.**

Type library Marshalling (Automation Marshalling): Type library marshalling is used by automation interface (i.e. they must inherit from IDispatch Interface) and all parameters must be automation compatible. This marshalling will make the use of type library file, which describes the signature of the methods being exposed by the COM object. There will be no need of proxy-stub DLL for marshalling. This is because `OLEAUT32.DLL` **performs the marshalling for automation interfaces. This marshalling requires only type library files to be installed on client and sever machines. The** `[oleautomation]` **attribute to an interface definition enables the universal marshaler. Each interface which is marshaled via universal marshaler has the following entry for ProxyStubClsid32 and** `ProxyStubClsid`

`{00020424-0000-0000-C000-000000000046}.`

This `CLSID {00020424-0000-0000-C000-000000000046}` **corresponds to the universal marshaler. This CLSID has a default value of** `PSOAInterface` **(Proxy Stub OLE Automation) and their values for** `InprocServer32` **and** `InprocServer` **are as follows:**

InprocServer32 : Default value is oleaut32.dll.

InprocServer : Default value is ole2disp.dll.

Standard Marshalling: Standard Marshalling also makes a use of type library but with the much more freedom on the parameter types. This marshalling is not limited to automation compatible data types and automation interface. This supports the marshalling of non-automation data types and non-automation compatible interface.

Standard marshalling requires the stub-proxy DLL to be registered on the client and the server machines.

Custom Marshalling: The main advantage of the custom marshalling is that you have an option to choose the variety of inter-process or network communication protocols like TCP, UDP and HTTP.Standard marshalling uses only Microsoft's RPC.COM specifies the IMarshal interface, which is to support the customized marshalling. The practical use of customized marshalling is to implement the marshal-by-value. The basic concept of marshal-by-value is that the custom proxy should acts as a duplicate for the object. The CLSID for the custom proxy must be same as that of the original object.

When the exchange of data between the unmanaged code and the managed code lies in the same COM apartment, Interop Marshaler is only the marshaler, which is being involved. When the managed code and unmanaged code are in different COM apartments or in a different COM processes, both the Interop Marshalling and COM Marshaling will be involved.

Platform Invoke relies on metadata to locate the correct address of the exported function and to marshal their arguments at runtime. This marshalling (Interop marshalling) is being provided by common language runtime's marshalling services. Platform Invocation Services are responsible for locating and invoking the exported function and marshalling its arguments across the inter-operation boundaries. Platform Invocation Service performs the following actions to call the unmanaged exported function to be called properly.

1. **Locates the path of DLL which exposes the exported function.**

2. **Loads the DLL into memory.**
3. **Locates the address of the exported function in the memory and calling the function with the marshaled data.**
4. **Finally, Platform Invoke passes the control to the unmanaged code.**
5. **PInvoke Service causes the exception to be thrown from unmanaged code to managed code.**

To make the use of unmanaged DLL functions in the managed code, the client should know the name of the function and the name of the DLL which exports that function. With this information, you can provide a managed declaration for the unmanaged function implemented in a native DLLs.These declaration of the unmanaged code is language dependent.

Let's us consider the different ways to have a declaration of the unmanaged functions into a managed code.

Creating Prototype in Managed Code

We will be covering various languages which forms the part of .Net framework. We will cover various keywords/attributes in various languages which help in declaring the prototype of the unmanaged code in the managed code.

[VB.Net]

VB.Net makes an intensive use of Declare statement to declare the prototype of unmanaged DLL function into the managed code. This statement is used at the module level to the declare references to the DLL which implements or defines the unmanaged functions. The DLLs which exposes Windows APIs doesn't exposes type library. To make a call to these Window APIs,we must make a use of Declare statement which tells VBA about the whereabouts of function.

Here is a basic prototype :

```
Public Declare Function CreateDC Lib "gdi32" Alias "CreateDCA" (ByVal
lpDriverName As String, ByVal lpDeviceName As String, ByVal lpOutput As
String, lpInitData As DEVMODE) As Long
```

Public : This is a access specifier which provides a public access to the functions and hence no restriction on its use.

Function : This indicates that the procedure should return the value.

Name : CreateDC is the name of function. Note that these are case sensitive.

Lib : Mandatory. This indicates that the DLL provides the definition of the function being used.

LibName : "gdi32" is the name of DLL which will provide its definition.

Alias : This keyword indicates that the procedure has another name in the DLL.

AliasName : Another name for the function.

Alias helps to use the DLL functions which have an invalid or illegal identifier. Some of the procedures in the DLL start with the underscore and this will give an invalid identifier in VB.Net as the identifier in VB.Net can't start with the underscore. To solve this problem, use an Alias which will map any valid identifier with that invalid identifier.

Example :

[vb]

```
Declare Function Valid_Name Lib "kernel32" Alias "InValid_Name" (ByVal
lpPathName As String, ByVal iReadWrite As Long) As Long.
```

In this example ,`Valid_Name` **will identify the procedure in our VB code and** `InValid_Name` **is the actual name which is being provided by DLL.**

VB.Net can make a use of `DllImport` **attribute. The** `DllImport` **is an instance of the** `DllImportAttribute` **class being exposed by** `System.RunTime.InteropServices`**.The** `DllImport` **will indicate that the attributed function is implemented as an export from the native DLL.** `DllImport` **is equivalent to Declare statement but provides the better control over how the functions are to be called.** `DllImport` **attribute can be used only with those functions which are shared (i.e. static functions).This attribute can't be used on the member functions which requires an instance of a class.**

[C#]

`DllImportAttribute` **class can be used in C# to identify the DLL and function. In this case, mark the function with static and extern keyword.** `DllImportAttribute` **class is defined as a sealed, so it can't acts as a base class for other classes.**

Example for C#:

```
[DllImport("KERNEL32.DLL", EntryPoint="CopyFileA", SetLastError=true,
CharSet=CharSet.Ansicode, ExactSpelling=true,
CallingConvention=CallingConvention.StdCall)]
public static extern long CopyFile(String src, String dst, long Val);
```

`DllImportAttribute` **class exposes object field which provides a control on how the unmanaged code will be called.**

Let's us consider the few of these object fields:

1. **EntryPoint: An** `EntryPoint` **object field locates the function in the DLLs. These Entry Point could be a function name or an ordinal number corresponding to that function. Visual Basic makes the use of Function keyword in the Declare statement to set the EntryPoint object field. You can use** `EntryPoint` **object field to identify a function by a name or by an ordinal.**
2. **CharSet: CharSet object field controls the name mangling and the marshalling of string parameter to the function. The default value for this is** `CharSet.Ansi`**. In this case; the Platform Invoke marshals the string from its managed format to ANSI format. There can be two other values which can be associated with this object field.**

These are `CharSet.Unicode` and `CharSet.Auto`. In the case of Unicode, the Platform Invoke copies the string from its managed format to Unicode format.

3. **CallingConvention: This object field set up the calling convention while passing an arguments to the method. The default value for this object field is WinAPI which corresponds to the __`stdcall` convention.VB.Net can make a use of `DllImportAttribute` class to set the calling convention for the method. The Declare statement doesn't provides such flexibility over the control on calling convention, so `DllImportAttribute` class will be used in VB.Net. The value for this can be set as `CallingConvention=CallingConvention.Cdecl`. There are the various values for this object field such as Cdecl, FastCall, and ThisCall etc.**
4. **SetLastError : This field will indicate that whether the attribute method should call the `SetLastError` method before returning. The default value for this object field in C# and C++ is false.**

[C++]

`DllImportAttribute` class can also be used in C++ to identify the DLL and function. The function should be marked with extern "C" construct. The naming convention of C++ is different from C because of the name mangling (name decoration) in C++. This construct can be used only from C++.This extern "C" tells the compiler to use C calling convention and C name mangling (i.e. preceded by a single underscore).

There are some limitations associated with this construct.

1. **Member function can't be declared with this construct.**
2. **The concept of overloading is a part of C++ and C doesn't know about the overloading functions. Therefore, we can't use this construct to make all overloaded functions to have C calling convention.**

Here is an example for C++:

```
[DllImport("KERNEL32.DLL", EntryPoint="CopyFileA", SetLastError=true,
CharSet=CharSet.Ansicode, ExactSpelling=true,
CallingConvention=CallingConvention.StdCall)]
extern "C" long CopyFile(String* src, String* dst, long Val);
[DllImportAttribute("KERNEL32.DLL", EntryPoint="CopyFileA",
SetLastError=true,
CharSet=CharSet.Ansicode, ExactSpelling=true,
CallingConvention=CallingConvention.StdCall)]
extern "C" long CopyFile(String* src, String* dst, long Val);
```

The object fields which have been described above are also valid for the C++.

After creating a prototype of unmanaged function in managed code, we can provide a place holder for these functions. We can have a class which will wrap these functions. You must provide a static function for each function which you want to call. Once wrapped, these methods can be called as an ordinary static member functions are called. Platform Invoke Services will call an underlying function (in a native DLL) automatically.

Let's see the Platform Invoke Services into the action. We will be calling the functions defined in a native custom DLL via `PInvoke` services in managed code.

Here is the code for Win32 Application which is being implemented as a dynamic link library(DLL).

```
// DLL is saved as PInvoke.dll.
#include<iostream>
using namespace std;
extern "C" __declspec(dllexport) void __stdcall Display(char *p)
{
cout<<"Display Function Called From Unmanaged Code"<<endl;
cout<<p<<endl;
}
```

In the above case, `__declspec(dllexport)` **keyword is used to export data, functions, classes or class members from the DLL. If you are using this keyword, then there will be no need of .DEF file. To make the function exportable from the DLL, use this keyword to the left of the calling convention keyword.**

Now, we will have a wrapper class to make a call to the functions exposed by the native DLL.

```
using namespace System::Runtime::InteropServices;
__gc class CPInvoke
{
public:
[DllImportAttribute("PInvoke.dll")] static void Display(char *p);
};
__delegate void CallDisplay(char *);
int _tmain(void)
{
CPInvoke *pInvoke=new CPInvoke();
pInvoke->Display("Test Platform Invocation Service"); // Works fine.
CallDisplay *pDelegate=new CallDisplay (pInvoke,&CPInvoke::Display);
pDelegate->Invoke("Test Via Delegates"); // Works fine.
return 0;
}
```

In this example, we are calling the function directly via an instance of the class and via delegates.

Note: The `static` **keyword has to be used with each function in a wrapper class.**

If we have a wrapper class as follows:

```
__gc class CPInvoke
{
public:
[DllImportAttribute("PInvoke.dll")] void Display(char *p);
};
__delegate void CallDisplay(char *);
int _tmain(void)
{
CPInvoke *pInvoke=new CPInvoke();
pInvoke->Display("Test Platform Invocation Service"); // Run time error.
return 0;
}
```

The above code will give a run time error with description: "`Error: PInvoke item (field,method) must be Static`**".**

The unmanaged code can also be called without making a managed wrapper class around these functions. The managed wrapper class gives an OOPs concept to programming and hence provides the ease for maintenance.

Use of unmanaged code without a wrapper class

```
typedef void* HWND;
[DllImportAttribute("User32.DLL",CharSet=CharSet::Auto)]extern "C" int
MessageBox(HWND h_WND,String* pText,String* pCaption, unsigned int Type);
```

Here, `HWND` **is an old traditional window data type and is simply a** `typedef` **for** `void*` **pointer.**

Now, this unmanaged function can be called in the managed code.

```
int _tmain(void)
{
MessageBox(0,S"Hello Platform Invoke",S"PInvoke");
}
```

This article covers the basic concept of Platform Invocation Services and there are many other details which are yet to be explored in `PInvoke`**.**

***~~~ End of Article ~~~***