Link native libraries from a .NET application

Walter Hoolwerf
(Radboud University, The Netherlands
walto@vaag.cx)

Mark Jans
(Radboud University, The Netherlands
mjans@home.nl)

Abstract: To be written.
Key Words: static linking, dynamic linking, .NET, library, native library
Category: H.2.12

1 Introduction

Sharing libraries between different applications has been a common practice in software development for years. Many companies have put generic code into shared libraries so it can be used by all their programs without the need of installing the same code on one system multiple times.

All these libraries are platform dependent and have to be ported to every platform the application has to run on. But when a company decides to write an application in .Net, and they want to use existing non-.Net code, the platform dependent nature of their libraries and the platform independent nature of the .Net framework raises a problem. How libraries work on different platform and how native libraries can be used from within .Net code is covered in this article. Also, a statement about how portable this .Net code which uses native libraries is is made based on the information about native libraries and .Net.

2 Linking backgrounds

Most applications can be divided in a couple of components. Some components are such generic that its functionality can be used by more than one application. In this case one can decide to make a library of such a component. Libraries can be linked to an application, whereupon this application can use its functionality. Functionality of a library can be seen as an export of symbols (functions, structs, constants and so on), which can be called or used by an other application.

2.1 Linking in C

Linking in C works in pretty much the same way as linking in other languages, but we took C as our example language.

There are two ways to link a library to your application, static and dynamic, both are described below.

2.1.1 Static

In static linking, the library is adopted by your own program. If, for example, you would link some math library static to your application, the resulting executable will be the sum of your own code and the library.

Static linking is very common: If you divide your C application into two source files, both will be compiled and then linked together to create the final executable.

The way static linking works might differ a bit per language, but the main idea always remain that you will simply compile all separate source files and then add (link) them together to create the final executable.

An application which uses libraries in a static way can be easely transported to another system since all code from the library is adopted by the executable.

2.1.2 Dynamic

In dynamic linking, the code of the library isn't simply adopted by your application. Instead, with dynamic linking, the library gets loaded in memory at runtime. A dynamic library can be a shared library. This is when it can be used by multiple programs and only appear on your hard drive once. A dynamic library can also be a library you don't have at design time of a program, e.g. a plugin which can be written later than the program.

Beside the advantage of using less disk and memory space it has the disadvantage of re-entrance. Because the shared library can be in use by more than one program it has to be designed for multiple use. For example it can use global variables, but this must be done very carefully, so an other program using the same library at the same time doesn't notice anything of a global variable changement. This makes that most dynamic libraries perform less than static ones.

Most common libraries are dynamic linking libraries. Some examples are math libraries, decryption and encryption libraries, but also most operating system specific libraries like user interface libraries and libraries for IO.

In windows, dynamic libraries are usually represented as .dll files and live in the windows folder, in unix they are usually represented as .so files and live in the different lib folders. .Net libraries are called assemblies and live in the assemblies directory of the .Net subsystem.

Applications which use dynamic linking to libraries, need the library to be present on every platform it will run on. We call this being dependent on a library and the library is called a dependency of the application. Most operating system specific libraries are available on every system, but 3rd party libraries (for encoding and decoding audio for example) needs to be present before an application can run.

Libraries are different on other operating systems. This is because operating systems use other formats of executables. Windows uses mostly the EFI file format and Unix uses te Executable and Linkable Format ([ELF]) or the a.out format. One of these differences in construction is for example that the inner assembly code can be found on different physical places in the library. Further do most libraries depend on other dynamic libraries, which can be operating system dependent.

Windows

In windows, the operating system exports a lot of functionality for dynamic linking. A program which wants to use functionality from a dynamic library includes `windows.h` and loads the library into memory with the `LoadLibrary` function.

The Windows subsystem will load the library into shared memory if it isn't already loaded. If some other application already has requested this library to be loaded, nothing will be done. The function returns a pointer to the library.

Using the function `GetProcAddress`, you can now get an pointer to the right function from this library. You can now tell the Windows subsystem to execute the function which this pointer points to with the right parameters.

When you're done with the library, you have to tell the Windows subsystem so by calling the `FreeLibrary` function. If no other application is using the library, it will be unloaded from shared memory.

Unix

With Unix, loading and using dynamic libraries work in a similar way as with Windows. You have to include `dlfcn.h` and use `dlopen` to load the library, `dlsym` to get a pointer to the right function and `dlclose` to tell the Unix subsystem to unload the library if not in use by any other application.

With Mac OS X, linking works the same as with Unix, since the subsystem of Mac OS X is Unix based.

3 Linking a native library from .Net

In some cases, you may need to link your .Net application to a native library. For example, you want to encode ogg audio files in your .net applications, but the ogg vorbis library is only available in native format.

Since .Net is platform independent and linking to native libraries works different on each platform, .Net provides a wrapper to make native linking in .Net platform independent to the programmer.

To link to a native library, the same steps as usual need to be taken to make it happen. In .Net, these steps look a little different, but in fact, do the same.

To make a function from a native library available in a .Net application, the platform invoke (PInvoke) functionality is used:

```
[DllImport("msvcrt.dll")]
public static extern int puts(string c);
```

This loads the puts function from the msvcrt library. This function can now be used by the .Net application is if it were a .Net function.

```
puts("hello world");
```

The .Net garbage collector will automatically release the library when possible.

3.1 Marshalling

Marshalling is the process of convert data from one form into another, without changing the contents. This is often nessisary when you want to export data from one language to another.

Trivial marshalling is done by the .Net framework, but in some cases the default marshalling method isn't sufficient. For example, a .Net string can be marshalled into different forms of data. In this case, you can add marshalling hints to the PInvoke functionality. For example:

```
[DllImport("msvcrt.dll")]
public static extern int puts(
    [MarshalAs(UnmanagedType.LPStr)]
    string m);
```

imports the same puts function from the msvcrt dll, but this time the string argument is marshalled as a LPStr instead of the default type.

For structs, custom marshalling is also availble: ¡¡write text about how here¿¿

3.2 Wrapper class

To keep the native library as loosely coupled from the .Net application as possible, it is common to create a wrapper class for every native library used. All marshalling and PInvoke is done here and the rest of the application can use this class as if it were a normal .Net class.

This also has the advantage that when a .Net version of the library comes availble, updating the application to use this .Net library is very little work.

Another (and less trivial) advantage of wrapper classes are that they can (for a great deal) be generated, which takes a significant amount of work out of the hands of the developer.

3.3 Automate creation of wrapper classes

A wrapper class can be created by a script or program. To do this you need the interface to the library. In C/C++ for example this can be found in header files. Every define, function and struct is prototyped here. These prototypes can be translated to a wrapper class by a simple parser.

Though this sounds very simple there are some things you must take in consideration. So are much datatypes different in other languages. Some work with pointers and others with references or both. .Net does only work with references, so all pointer datatypes must be translated to reference ones. Also some datatypes like arrays are different in .Net because the size is not fixed. These can be translated very simple, but you have to tell .Net how to marshal this.

There are several ways to create a translation script, but you can also take one free available on the internet. E.g. you can use Simplified Wrapper and Interface Generator ([SWIG]). This program can create a .Net wrapper class from almost every programming language available by simply passing a header or source file. For example:

```
swig -csharp -module ogg ogg.h
```

In this case you tell swig to generate a C# wrapper class ogg (module name) which is stored as ogg.cs (also module name) from header file ogg.h. It will also create some more classes. These are structs in the original source, which are translated to classes to make everything more generic.

SWIG is available for almost every operating system.

4 Platform independency

Since .Net is platform independent, linking to native libraries should be platform independent as well. You would like to make a Windows dll and then port it to a Unix library with exactly the same interface and be able to use your .Net application on both platforms, linking to the right native library at both platforms and behaving exactly the same.

The PInvoke functionalllity is platform independent at a considerable length. If a Unix library has the same interface as a Windows library (or vise versa),

you will be able to write a .Net application which behaves the same on Windows as on Unix.

The only problem with PInvoke is the way you need to load a library. You use the DllImport function and have to use the full library name as parameter, including the extension. This raises a problem with portability, because a library will have a different extension on Unix (.so) then on Windows (.dll).

It will not be hard to create a work around for this, which checks at runtime which operating system you are running and then decide which filename to use as parameter, but it isn't completely platform independent.

5 Conclusions

Overall, platform invoke is a very powerful feature of the .Net framework. It takes a lot of work out of the developers hands with its default marshalling behavior and makes it possible to link against any non trivial library by allowing the developer to create it's own marshallng rules.

Virtually every native library can be linked against using PInvoke. A proof of fact has been given by the mono develop team (which creates an open source implementation of the .Net framework which runs on multiple platforms) who linked their mono code against the entire Gtk library, both the Unix and Windows variant. This enables developers to write graphical applications which both run on Windows and Unix using gtk# (the .Net wrapper for gtk).

A Windows-based runtime dynamic link example:

```
// A simple program that uses LoadLibrary and
// GetProcAddress to access myPuts from Myputs.dll.

#include <stdio.h>
#include <windows.h>

typedef int (*MYPROC)(LPTSTR);

VOID main(VOID) {
    HINSTANCE hinstLib;
    MYPROC ProcAdd;
    BOOL fFreeResult, fRunTimeLinkSuccess = FALSE;

    // Get a handle to the DLL module.
    hinstLib = LoadLibrary(TEXT("myputs"));

    // If the handle is valid, try to get the function address.
    if (hinstLib != NULL) {
        ProcAdd = (MYPROC) GetProcAddress(hinstLib, TEXT("myPuts"));

        // If the function address is valid, call the function.
        if (NULL != ProcAdd) {
            fRunTimeLinkSuccess = TRUE;
            (ProcAdd) (TEXT("Message via DLL function\n"));
        }

        // Free the DLL module.
        fFreeResult = FreeLibrary(hinstLib);
    }

    // If unable to call the DLL function, use an alternative.
    if (! fRunTimeLinkSuccess)
        printf("Message via alternative method\n");
}
```

For more information take a look at [Using Run-Time Dynamic Linking].

B Unix-based runtime dynamic link example:

```
// A simple program that uses dlopen and
// dlsym to access myPuts from Myputs.so.

#include <dlfcn.h>

int main(void) {
    void *lib_handle;
    struct local_file *(myputs)(const char *file_path);
    const char *error_msg;
    int freeResult, runTimeLinkSuccess = 0;

    // Get a handle to the Library module.
    lib_handle = dlopen("/full/path/to/Myputs.so", RTLD_LAZY);

    // If the handle is valid, try to get the function address.
    if (lib_handle) {
        myputs = dlsym(lib_handle, "myPuts");

        // If the function address is valid, call the function
        if (error_msg) {
            runTimeLinkSuccess = 1;
            (*myputs)("Message via a Library function\n"));
        )

        // Close the library
        freeResult = dlclose(lib_handle);
    }

    // If unable to call the Library function, use an alternative.
    if (!runTimeLinkSuccess == 0)
        printf("Message via alternative method\n");
}
```

For more information take a look at [Unix C libraries].

References

[Using Run-Time Dynamic Linking] Microsoft MSDN. Platform SDK: DLLs, Processes, and Threads. Using Run-Time Dynamic Linking. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dllproc/base/using_run_time_dynamic_linking.asp

[Unix C libraries] Keren, Guy: Building And Using Static And Shared "C" Libraries; http://users.actcom.co.il/ choo/lupg/tutorials/libraries/unix-c-libraries.html.

[.Net assemblies] MSDN page about .Net assemblies. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconassemblies.asp

[.Net PInvoke tutorial] MSDN tutorial about platform invoke. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/csref/html/vcwlkPlatformInvokeTutorial.asp

[SWIG] Homepage of Simplified Wrapper and Interface Generator http://www.swig.org/

[ELF] Tools Interface Standards(TIS); Portable Formats Specification, Version 1.1; Executable and Linkable Format (ELF). http://www.cs.northwestern.edu/ pdinda/ics-f02/doc/elf.pdf