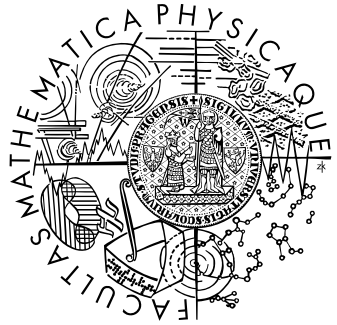
Charles University in Prague

Faculty of Mathematics and Physics

BACHELOR THESIS



Ondřej Vykouk

Native DLL Importer for C#

Department of Distributed and Dependable Systems

Supervisor: Mgr. Pavel Ježek

Study Program: General Computer Science

2009

I would like to thank my supervisor, Mgr. Pavel Ježek, for numerous valuable pieces of advice, corrections and time he spent guiding me through this thesis. Furthermore, I would like to thank my parents for confidence, patience and great support throughout my studies.

I declare that I have elaborated this bachelor thesis on my own and listed all used references. I agree with lending this bachelor thesis. The thesis may be reproduced for academic purposes.

In Prague on Ondřej Vykouk

Table of contents

Název práce: Importér nativních knihoven pro C#

Autor: Ondřej Vykouk

Katedra: Katedra distribuovaných a spolehlivých systémů

Vedoucí bakalářské práce: Mgr. Pavel Ježek

E-mail vedoucího: pavel.jezek@d3s.mff.cuni.cz

Abstrakt: Cílem předložené práce je návrh a implementace aplikace, která usnadní používání mechanismu P/Invoke, určeného pro import nativních DLL knihoven do prostředí .NET, v jazyce C#. Primárním cílem je podpora knihoven implementujících rozhraní Windows API. V práci je nejprve popsán mechanismus P/Invoke a následně je provedena analýza automatizovaného získávání informací pro tento mechanismus. Dále je popsána implementace samotné aplikace a v poslední části je provedeno zhodnocení dosažených cílů a jsou uvedena možná pokračování do budoucna.

Klíčová slova: Marshalling, interoperability, P/Invoke, DllImport, C#

Title: Native DLL Importer for C#

Author: Ondřej Vykouk

Department: Department of Software Engineering

Supervisor: Mgr. Pavel Ježek

Supervisor’s e-mail address: pavel.jezek@d3s.mff.cuni.cz

Abstract: The aim of this work is to design and implement application that facilitate the use of the mechanism of P/Invoke, which imports the native DLL libraries into .NET, in the C # language. The primary objective is to support libraries that implement the Windows API interface. The thesis first describes the mechanism of P/Invoke and then analyzes the automated gathering of information for this mechanism. The next part describes the implementation of the application and the last part provides the assessed achievements and possible work for the future.

Keywords: Marshalling, interoperability, P/Invoke, DllImport, C#

1 Introduction

1.1 Problem description

Today computers can be found almost everywhere in our lives. As they serve for various purposes, different variants of hardware and operating systems – collectively called platforms – have been developed. Applications for these platforms are written in various programming languages[1] and most of these languages have two disadvantages in common: the code has to be compiled separately for each platform and they are not designed to cooperate with other languages. In order to make the creation of multiplatform applications easier, Microsoft introduced the .NET framework [1].

> *"Imagine, if all of the sudden the universal translator from Star Trek were made available today enabling Russians to speak directly to Germans, to Dutch, to Spanish, to any language. Each using their own native tongue, yet each hearing in their own native tongue. That is exactly what .NET does for programming languages."[2]*

.NET is composed of *"a large library of coded solutions to common programming problems and a virtual machine that manages the execution of programs written specifically for the framework."* [1] Among the primary goal of .NET to offer multiplatform development and runtime environment there are secondary advantages arising from its architecture, e.g. opportunity to involve more than one programming language in one application, easy memory management with garbage collection, programming languages compliant with the .NET architecture are strong typed, which means they are safer, because it is possible to detect errors during compile time, while in low typed languages these errors are detected only at run time, etc.

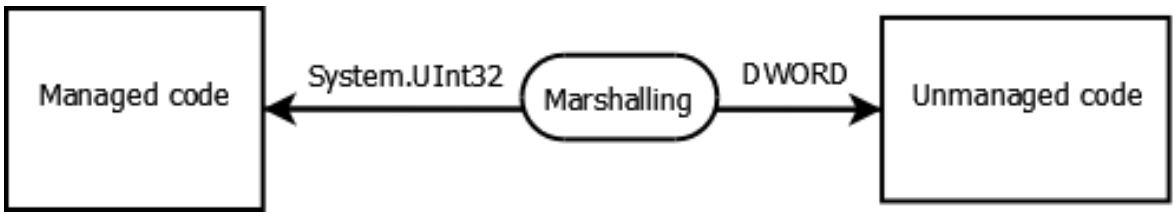
All these features and principles together create environment with unified methods for accessing functions of the underlying operating system or platform. When creating a real world application it is useful to have the opportunity to use specifics operating system features (e.g. hardware access) or existing implementation of some problem, which are otherwise unreachable from .NET. The price for this opportunity is platform portability, but

[1] Only compiled programming languages are meant.

other .NET advantages still can be used. For this purpose the .NET framework comes with platform invoke (P/Invoke [3]) feature that allows the .NET developers to invoke platform specific functions or to reuse code primary targeting Microsoft Windows platform. It is possible, because .NET runtime environment, called *common language runtime* or *CLR,* is also a program running on Windows platform and P/Invoke only allows the application to bypass the .NET environment and libraries to gain access directly to underlying system functionality. The code running under supervision of CLR is called *managed,* the code running outside CLR is called *unmanaged*. P/Invoke creates a bridge between managed and unmanaged code. Unmanaged code that can be invoked from managed code has to be distributed in form of native DLLs (dynamic link libraries) [4]. These libraries expose their functionality as a set of functions written in C or C++. A function in mentioned programming languages is declared via number of parameters and their data types. Data types are different in managed and unmanaged code, so in order to call a function stored in Native DLL from managed code, managed version of the function declaration and managed data types counterparts have to be created. These declarations are also called *P/Invoke signatures*.

The process of converting types from managed to unmanaged environment and vice versa is called *marshalling*. It is needed, because many types in unmanaged environment do not have equivalents in managed environment. These types need to be mapped to existing managed types or new managed equivalents have to be created.

As an example DWORD is taken – type extensively used in WinAPI. When assuming 32bit Windows system, DWORD is shortcut name for unsigned long and takes 4 bytes in memory. Thus equivalent type in C# is System.UInt32 and vice versa.



2.1.1-1 Marshalling overview

1.2 Thesis aim

The aim of this thesis is to implement a GUI that shows the data and functions declarations based on the given C header file and for the user selected declarations will show assumed

managed counterparts in C#. Code generated by the GUI should be able to compile with C# compiler, but it can generate some errors at runtime, because of lack of information about function parameters in the given header file.

2 Analysis

2.1.1 Platform, operating system

Platform: Microsoft .NET Framework 3.5

Language: C#

Operating system: Operating systems supporting .NET Framework 3.5

.NET platform offers wide range of classes for building the program without need to use additional external libraries. It has also good support for creating graphical user interface. .NET platform is widely used, so there are enough external tools targeting it that would be needed for lexical and syntactical analysis. Also previous author experience with the platform was considered.

2.1.2 Creating P/Invoke signatures

In order to use P/Invoke feature and thus gain access to functions in Native DLLs, P/Invoke signature for used functions have to be created. The process of creating correct signatures is not trivial, because it demands good understanding of C code approach to functions and their parameters and how various data types are stored in memory (memory layout). Knowledge of C# data types memory layout is also required, because proper passing the data between managed and unmanaged environment is crucial task for P/Invoke to work properly. This knowledge is not always necessary for programming applications in pure C or C#, so not all programmers have it. Three solutions, how to deal with creating P/Invoke signatures are presented in the following text.

1) P/Invoke signature database

The first solution consists in creating a central database that is accessible to everyone. Signatures in the database are mostly gathered from programmers, they used them in their code, so they are usually tested to be working properly. On the other side, the number of signatures in the database depends on the number of developers knowing of existence of the database and their will to contribute to it. The database would probably consist mainly of signatures of highly used functions, because they have higher probability that at least one

programmer created their signatures and contributed them to the database. The main drawback of this solution is that the database is never complete, usually does not contain signatures of less used functions and signatures of functions, which the developers have for their internal use. Also updating the database in case of some changes e.g. new and better ways of marshalling data, would be probably quite slow. Example of such database is *PInvoke.net* [5].

2) Writing signatures manually

P/Invoke signatures can be also written manually. This gives the programmer full control over the process of calling the functions and passing data to them and vice versa, but it requires good knowledge of marshalling process to write correctly working signatures. Another advantage is that with this manual approach signature for any function can be written. But when using PInvoke for many functions, many structures or just large structures, finding their original C declarations and writing their C# counterparts is tedious and error prone. An example of declaring managed counterpart is on the following figure.

```
typedef unsigned long DWORD;
typedef unsigned short WORD;
typedef unsigned char     BYTE;
typedef unsigned __int64 ULONGLONG;

typedef struct _IMAGE_DATA_DIRECTORY {
  DWORD VirtualAddress;
  DWORD Size;
} IMAGE_DATA_DIRECTORY;


struct _IMAGE_OPTIONAL_HEADER64 {
  WORD Magic;
  BYTE MajorLinkerVersion;
  BYTE MinorLinkerVersion;
  DWORD SizeOfCode;
  DWORD SizeOfInitializedData;
  DWORD SizeOfUninitializedData;
  20 declarations skipped
  ULONGLONG SizeOfHeapCommit;
  DWORD LoaderFlags;
  DWORD NumberOfRvaAndSizes;
  IMAGE_DATA_DIRECTORY DataDirectory[16];
};
```

```
using System;
using System.Runtime.Interop.Services;


[StructLayout(LayoutKind.Sequential]
public struct _IMAGE_DATA_DIRECTORY {
  public UInt32 VirtualAddress;
  public UInt32 Size;
};

[StructLayout(LayoutKind.Sequential]
public struct _IMAGE_OPTIONAL_HEADER64 {
  public UInt16 Magic;
  public Byte MajorLinkerVersion;
  public Byte MinorLinkerVersion;
  public UInt32 SizeOfCode;
  public UInt32 SizeOfInitializedData;
  public UInt32 SizeOfUninitializedData;
  20 declarations skipped
  public UInt64 SizeOfHeapCommit;
  public UInt32 LoaderFlags;
  public UInt32 NumberOfRvaAndSizes;
 [MarshalAs(UnmanagedType.ByValArray,  SizeConst = 16)]
  public IMAGE_DATA_DIRECTORY[] DataDirectory;
};
```

2.1.2-1: C code [left side] and equivalent C# code [right side]; added/changed code [red],type definitions for proper marshalling [green]

3) Generating signatures by a program

The last presented solution is to use a program that will get information about the function and its data as the input and generate the code with appropriate signatures automatically. The advantage of this approach is that the signatures can be generated without almost any knowledge of marshalling process and for any function, for which the program compliant input can be provided. Some knowledge of marshalling process can be needed for cases, when non standard way of marshalling is required. Generally this approach is as good as the program used. If the signatures provided by the program used are not working correctly and it can happen, because some exceptions are everywhere, they can be at least good starting point for manual writing approach. Example of such program is *P/Invoke Interop Assistant* (PIIA) from Microsoft [6]. This program was introduced during the writing this thesis. It provides better output then software developed with this thesis. Probably the only disadvantage of the PIIA is that it provides minimal configuration options and thus minimal control over the marshaling process.

Also a good knowledge of marshalling process is needed. Difficult cases, while understanding the context of the function is demanded for right marshalling can be handled with user support. Flexible automated process can be an advantage for many developers, because in many cases they could use PInvoke without a deep knowledge of marshalling.

As the database proposed in the first approach already exists and there is nothing to implement with the second approach and with respect to the fact that there was no such program that is presented in the last approach, the third variant – to implement a signature generator –was chosen. Interesting solution could also be achieved by combination the database and the program.

2.2 Gathering input information

As described in chapter Creating P/Invoke signatures solution 3), a signature generating program needs information about function, for which the signature is generated. As the native DLL only contain the exported function names, this information can be obtained from C or C++ *header files* that were used to compile libraries containing the desired functions.

The information consists of a number of parameters the function has and their data types and the declaration for data types that are not default language types. In order to obtain these information, the contents of header files have to be converted to a data structure, which is easily accessible from the program and can be completed with the derived information. This process is called *parsing*. Because parsing of C++ headers is much more difficult that only C header and main target of this thesis is the functionality from Windows API, which is described actually by C header files, the input for the program is restricted only to C header files.

2.2.1 Preprocessor

C header files usually contain C language code and lines beginning with *'#'* character. These lines are called preprocessor directives and need to be interpreted and removed before parsing the code itself. The program that interprets these lines is called preprocessor and is usually distributed together with C compiler, but standalone implementation of preprocessor also exists (e.g. mcpp [7]).

The main features of preprocessor are:

- text substitution (macro expansion)
- including contents of other files (usually header files)
- conditional compilation

Almost all C compilers are distributed together with a preprocessor. Widely used are GCC as the part of the GNU Compiler Collection [8] and Microsoft Visual Studio C++ (MSVS C++) [9]. GCC is a compiler targeting primary Unix-like systems, but version for Windows systems also exists. MSVS C++ is only for Windows systems. Both of them are freely distributed. With respect to chosen platform and focusing on Windows API, a preprocessor that comes in distribution with Microsoft's compiler is used. The main advantage of this preprocessor is the support for all Microsoft specific C extensions. On the other hand the size of the installed tools which comes with preprocessor is the main drawback of this solution. The disadvantage of all existing preprocessors is that they do not expose an API in .NET compliant programming language, thus they cannot be integrated with the program and have to be used as an external tool.

2.2.2 Parsing C header files

After the C header file is processed by preprocessor it can only contain C code and #pragma directives that were ignored by preprocessor, because according to the ANSI C standard[10], they can be ignored if they are not implemented. As they can carry semantic information, it is useful that the preprocessor skips them instead of erasing them. The code in header files mainly contains definitions of data types (constants, structures etc.), definitions of new type names (*typedefs*) and function declarations. In order to parse the code in C header and to build its inner representation, the *parser* has to be created. Each programming language can be described by a *grammar* [11], which is a set of structural rules that govern the composition of any expression or statement in the given language, i.e. C is also described by a grammar. The parser, which accepts the same language as the grammar describes, can be hand written or generated by a parser generator tool. As hand written parsers are hard to be modified when the grammar needs to be extended or changed, the approach of generated parser is used. Because the same language can be described by different grammars, the choice of the tool used to generate the parser depends on the used grammar and vice versa.
Parsing is usually split up into lexical and syntactical analysis. *"The first stage is the token generation, or lexical analysis, by which the input character stream is split into meaningful symbols defined by a grammar of regular expressions. The next stage is parsing or syntactic analysis, which is checking that the tokens form an allowable expression. This is usually done with reference to a context-free grammar which recursively defines components that can make up an expression and the order in which they must appear."* [12]

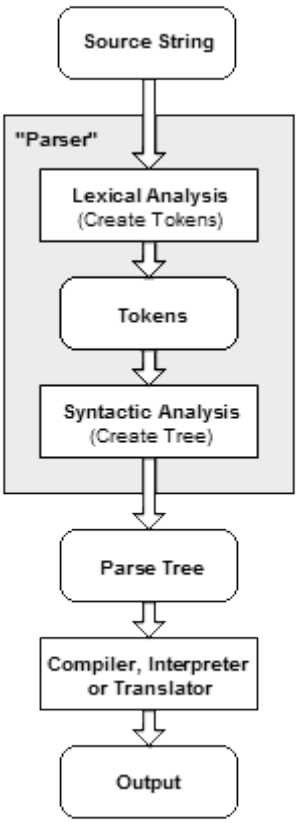


Figure 2-1 Flow of data in typical parser. Source: [12]

2.2.3 Tools for lexical and syntactical analysis

With respect to the chosen .NET platform and C# language, the tools for lexical and syntactical analysis have to be selected. GPPG[13] is LALR [14] parser generator with C# output. It accepts grammar in bison/yacc[2] fashion. Second option targeting C# language is *ANother Tool for Language Recognition* (ANTLR) generator, which generates LL parser [14]. It comes with ANTLR Works, the GUI for developing grammars that allows e.g. visualizing the grammar tree, but the input grammar is subjectively more complicated than grammar for GPPG, because bottom-up parsing technique (for details see [15]) used by LALR parser generators is subjectively more understandable to me than top-down parsing technique (for details see [15]) used by ANTLR, so the GPPG was chosen. As GPPG comes with no GUI that would simplify the process of modifying the grammar, ANTLR is probably a much better choice for those who like top-down parsers. Together with GPPG a lexer generator GPLEX

[2] Parser generators with output to the C or C++.

[16] was developed. It is designed to be used with GPPG. Other compatible lexer generator is e.g. CsLex [17], but it has no advantage over GPLEX, thus GPLEX is a suitable choice.

2.2.4 Grammar for C

To parse the C code correctly, a semantic analysis has to be made during parsing and information from this analysis have to be passed to lexer – this technique is known as *lexer hack* [18]. James A. Roskind rewrote yacc ANSI C grammar [19] to reduce the need of these semantic information (see original grammar file comments, e.g. /* DECLARATIONS */). So J. A. Roskind grammar[20] was used rather than yacc ANSI C grammar, because it makes the implementation of the parser easier.

2.2.5 Microsoft-specific extensions to the grammar

Microsoft specific extensions to ANSI C standard [10] have to be integrated into the grammar to correctly parse the input code, especially the header files defining Windows API, where these extensions are used. In order to be able to make modifications to the grammar, the grammar rules must have been understood. The GUI similar to ANTLR Works would be very helpful for this stage. The following keywords that are aliases for integral types of different size were added in the first step:

- __int8
- __int16
- __int32
- __int64

The second construct extending the grammar is "*the extended attribute syntax for specifying storage-class information that uses the __declspec keyword, which specifies that an instance of a given type is to be stored with a Microsoft-specific storage-class attribute."*[21]

The third and the last extension to the grammar that has to be made is processing #pragma directives skipped by the preprocessor. The only directive that has a semantic meaning for this thesis is *#pragma pack* others are skipped by lexer. It affects data alignment in memory. The effect of this directive can be seen in Figure 2-2.

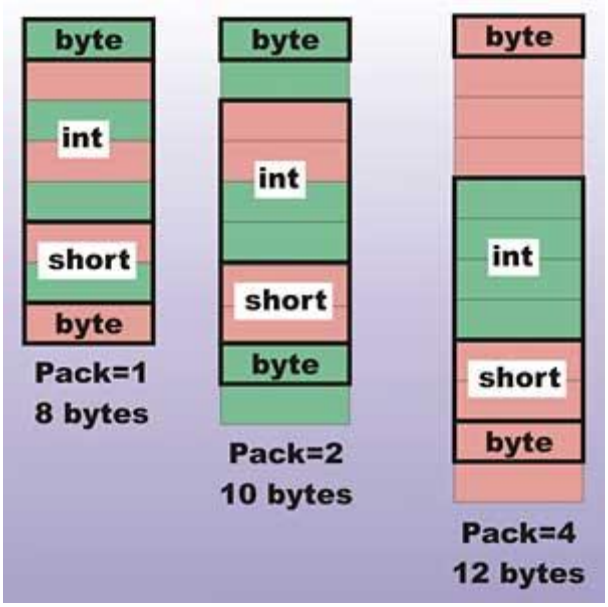


Figure 2-2 The effect of pack size on layout

The syntax for #pragma pack is as follows:

"#pragma pack([show] | [push | pop] [, identifier] , n)" [22],

where expressions in square brackets are optional and *n* defines the alignment value in bytes to be used for packing.

"The default value for *n* is 8. Valid values are 1, 2, 4, 8, and 16. The alignment of a member will be on a boundary that is either a multiple of *n* or a multiple of the size of the member, whichever is smaller. *N* can be used with *push* or *pop* for setting a particular stack value, or alone for setting the current value used by the compiler." [22]

2.2.6 What is in a DLL

A part of the P/Invoke signature is also DLL name, which is not a part of the parsed header file, but the signature already contains the function name, which can be used to match the list of exported function names from the DLL and therefore P/Invoke signature can be completed with DLL name according to this match. A conflict can arise when the function name is located in more than one DLL, because the P/Invoke signature must contain just one DLL name. This conflict has to be resolved by the user, because there are no means how to resolve it automatically. The following text briefly describes DLL files format and where the names of exported functions are located.

"DLL is Microsoft's implementation of the shared library concept in the Microsoft Windows and OS/2 operating systems. The file formats for DLLs are the same as for Windows EXE files — that is, Portable Executable (PE) for 32-bit and 64-bit Windows, and New Executable (NE) for 16-bit Windows. As with EXEs, DLLs can contain code, data, and resources, in any combination" [4].

DLL files are stored in the PE format, which is designed to be platform independent, but in the Windows it usually carries platform dependent code. The word *native* is added to differentiate the group of DLLs containing code that is able to run on the Windows platform. As the P/Invoke signature of a function contains the name of DLL, where the executable code of the function is stored, the options for identifying such DLL are described in the following text.

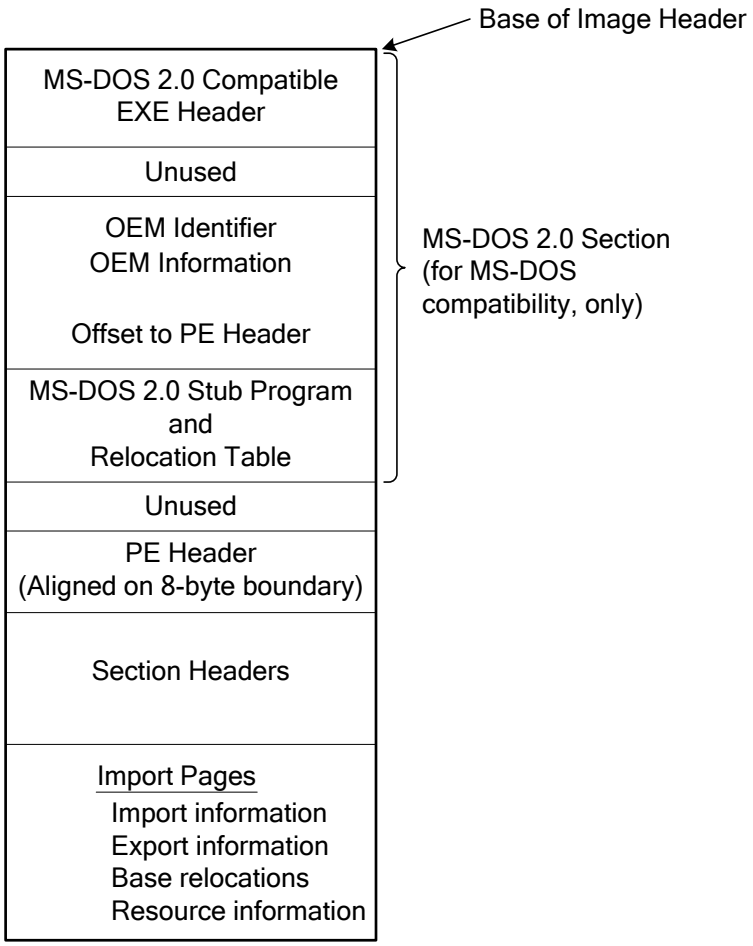


Figure 2-3 This figure illustrates the Microsoft PE format. Source: [23]

At the beginning the EXE header and the PE header are placed. According to the PE format documentation [23] the names of exported functions – only interesting data for this thesis – are located in the Import Pages part in the Export information section. *"The export data section, named .edata, contains information about symbols that other images can access through dynamic linking"*[23]**.** The information about the export data section location is stored in PE header in the form of a relative virtual address (RVA). RVA is the offset from beginning of the file and is only valid, when DLL is loaded into memory. But loading DLL into memory is time consuming and costs resources. To avoid that, the exported names are obtained only by parsing DLL file. Parsing implementation is described in detail in chapter 3.3.

2.3 Marshalling native types to .NET types

This chapter discusses the marshalling process. Data types are split up into simple and compound types. "*Simple types (integers, booleans, etc.) are those that are not made of other types. On the contrary, compound types (structures and classes) are those types that require special handling and made of other types"*[24]. Simple types are further divided into blittable and non-blittable. The chapter continues with marshalling of compound types (structures, unions), arrays and the last part is about marshalling callbacks (pointers to function).

2.3.1 Simple and compound types

Simple types are those which do not contain any other type in their definition. According to MSDN definition [25], simple types "*are identified through reserved words, but these reserved words are simply aliases for predefined struct types in the* `System` *namespace".* Terms *simple*, *primitive* and *basic* are interchangeable in this thesis. Simple types can be grouped in three categories: numeric, textual and handles. Examples of managed simple types are *System.Byte, System.Int32, System.Double* as numeric, *System.Char* as textual and *System.IntPtr* as handle. Although *string* is an alias for *System.String*, so it is compliant with above MSDN definition for simple types, it is considered as compound, because in fact it is array of characters.

Complex or *compound* types can be built up from simple and complex types. Structure or class is a typical representative of a compound type.

2.3.2 Blittable and non-blittable types

Most simple types do not require special handling when marshalling, because they have counterparts in unmanaged code. These types are called *blittable* types, because they do not need conversion when passed between managed and unmanaged environment. According to [26] the following simple types from *System* namespace are the blittable types:

- System.Byte
- System.SByte
- System.Int16
- System.UInt16
- System.Int32
- System.UInt32
- System.Int64
- System.UInt64
- System.IntPtr
- System.UIntPtr
- System.Single
- System.Double

And complex types fulfilling one of the following conditions are also blittable:

- One-dimensional arrays of blittable types, such as an array of integers.
- Formatted value types that contain only blittable types (and classes if they are marshalled as formatted types).

Other types are called *non-blittable* and they need special handling.

2.3.3 Structures

By default, native and managed structures have different in-memory representation. When passing managed structure type to unmanaged code, its memory layout has to be taken into consideration, because the Windows accesses the members of the structure via their addresses inside the memory, while the CLR accesses it by its name. The members of managed structures can be reordered for performance purposes, while unmanaged

structure members are placed sequentially in memory in the same way as in the declaration. The C# allows to modify types *metadata*, which are used for type description and a description of the deployment unit (the assembly) and also can affect in-memory representation of a type. The modification of metadata is possible by means of a text with the defined structure called an *attribute* [27]. To obtain the same memory layout for managed structures as the unmanaged structures have, the *StructLayoutAttribute* attribute has to be used with this structure declaration. There are two possible parameters:

- `LayoutKind.Sequential`
- `LayoutKind.Explicit`

The first one conforms to the unmanaged memory layout. The second one is used for user defined layout. In order to use explicit layout, each structure member has to have *FieldOffset* attribute, which sets the member's offset in bytes from the beginning of the structure in memory. This gives a better control over the layout than the first option. *Explicit* layout is used for C structure declarations, where the offset is defined for some or all members. Examples of use are shown in the following figure.

```
[StructLayout(LayoutKind.Sequential)]
public 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 wMillisecond;
}
[StructLayout(LayoutKind.Explicit)]
public struct SystemTime
{
    [FieldOffset(0)]
    public short wYear;
    [FieldOffset(2)]
    public short wMonth;
    [FieldOffset(4)]
    public short wDayOfWeek;
    [FieldOffset(6)]
    public short wDay;
    [FieldOffset(8)]
    public short wHour;
    [FieldOffset(10)]
    public short wMinute;
    [FieldOffset(12)]
    public short wSecond;
    [FieldOffset(14)]
    public short wMillisecond;
}
```

Figure 2.3.3 - 1: Examples of Sequential and Explicit LayoutKind.

2.3.4 Unions

Unions are similar to structures in C, because they create a sort of container for its members, but with the difference that all the members represent the same block of memory. It saves type casting and makes the code more readable. Union has the size of the largest member contained. The structure type with the attribute *[StructLayout(LayoutKind.Explicit)]* and setting the same offset to zero can be used to obtain C-like behaviour. As a simple example this C union definition can be taken:

```
union SOME_CHARACTER
{
    int i;
    char c;
}
```

Figure 2-4 SOME_CHARACTER: unmanaged definition

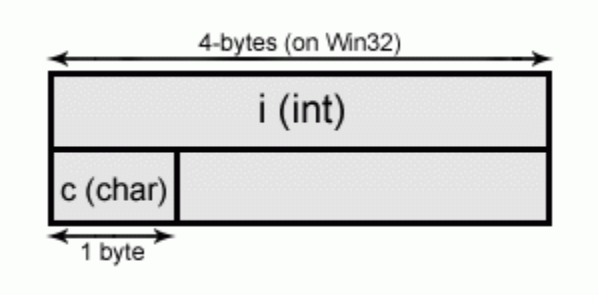


Figure 2-5 Memory representation of union SOME_CHARACTER defined in Figure 2-4. Source: [24]

Its C# counterpart with the same behavior is defined like this:

```
[StructLayoutAttribute(LayoutKind.Explicit)]
        public struct SOME_CHARACTER
        {
            [FieldOffset(0)] public int i;
            [FieldOffset(0)] public char c;
        }
```

Figure 2-6 Managed counterpart for union SOME_CHARACTER defined in Figure 2-4

Exception for this procedure are unions with both reference and value types, because .NET do not allow value and reference types to overlap. This restriction stems from the fact that .NET runtime environment stores value types in the different memory than reference types. E.g. array is a reference type and integer is a value type. An example is following.

```
        union UNION_WITH_ARRAY
        {
            int i;
            char c[128];
        }
```

Figure 2-7 UNION_WITH_ARRAY: unmanaged definition

Equivalent managed counterpart for union from *Figure 2-7 UNION_WITH_*ARRAY: unmanaged definition do not exist, however two separate managed structures can be defined, each containing one member and then used according to the needs. The function that takes this union as a parameter is then overloaded, one function for each structure definition.

```
[StructLayout(LayoutKind.Explicit)]
public struct UNION_WITH_ARRAY_1
{
[MarshalAs(UnmanagedType.ByValTStr, SizeConst = 128)]
    [FieldOffset(0)] public string charArray;
}
// StructLayoutAttribute.Size determines
// the size -in bytes- of the type.
// If the size specified is larger than
// members' size, the last member will be extended
[StructLayout(LayoutKind.Explicit, Size = 128)]
public struct UNION_WITH_ARRAY_2 {
    [FieldOffset(0)] public short number;
}
```

Figure 2-8 Managed definitions for unmanaged union with array

2.3.5 Arrays

Because the object that is responsible for passing the data between managed and unmanaged code needs to know about the sizes of passed types, marshalling arrays can be divided into *constant size arrays* and *arrays with variable length*. Constant size array declared as structure member or function parameter is marshaled with *MarshallAs* attribute, which allows to specify array size and optionally array subtype. It can be also marshalled the same way as variable length array, i.e. as System.IntPtr, which is equivalent to C pointer. In the second case the memory for the array have to be allocated and freed manually using the C# *Marshal* class from System.Runtime.InteropServices, which is uncomfortable.

2.3.6 Callbacks

Some functions may need to have other function as a parameter. In the C code it done by passing a pointer to the function (called callback) and its managed counterpart is a *delegate*. Because a callback is a pointer, it can be marshaled as *System.IntPtr,* but the function then cannot be called. In order to obtain working callback from unmanaged code it must be marshaled as a delegate. As unmanaged code does not know nothing about managed environment, the delegate to a managed function cannot be passed to unmanaged code, only the delegate obtained from the unmanaged code can be passed to unmanaged code.

2.3.7 Passing mechanism

"When passing an argument to a function, the function may require either passing the argument by value or by reference. If the function intends to change argument value, it requires it to be passed by reference, otherwise, by value. This is what called passing

mechanism. Value arguments (i.e. input/In arguments,) when passed to a function, a copy of the argument is sent to the function. Therefore, any changes to the argument do not affect the original copy. On the other hand, reference arguments, when passed to a function, the argument itself is passed to the function. Therefore, the caller sees any changes happen inside the function. Arguments passed by reference can be either In/Out (Input/Output) or only Out (Output.) In/Out arguments are used for passing input to the function and returning output. On the other hand, Out arguments used for returning output only. Therefore, In/Out arguments must be initialized before they are passed to the function. Conversely, Out arguments do not require pre-initialization. When passing an argument by value, no changes to the PInvoke method are required. Conversely, passing an argument by reference requires two additional changes. The first is adding the ref modifier to the argument if it is In/Out argument, or the out modifier if it is Out argument. The second is decorating your argument with both InAttribute and OutAttribute attributes if it is In/Out argument or only OutAttribute if it is Out argument. To be honest, applying those attributes is not required, the modifiers are adequate in most cases. However, applying them gives the CLR a notation about the passing mechanism. As you have seen, when marshaling a string, you can marshal it as a System.String or as a System.Text.StringBuilder. By default, StringBuilder is passed by reference (you do not need to apply any changes.) System.String on the other hand is passed by value. It is worth mentioning that Windows API does not support reference arguments. Instead, if a function requires an argument to be passed by reference, it declares it as a pointer so that caller can see the applied changes."[24]

3 Implementation

In order to create P/Invoke signature from header file containing C code with the declaration of a desired function and from list of DLL files, three main tasks must be solved: the analysis of C header file by a parser to obtain the information about the function for translation, the translation of the function and the relevant data type declarations and finding the DLL name, where the translated function is stored to complete the signature. The structure of the program is determined by these three steps and is presented in the following figure.

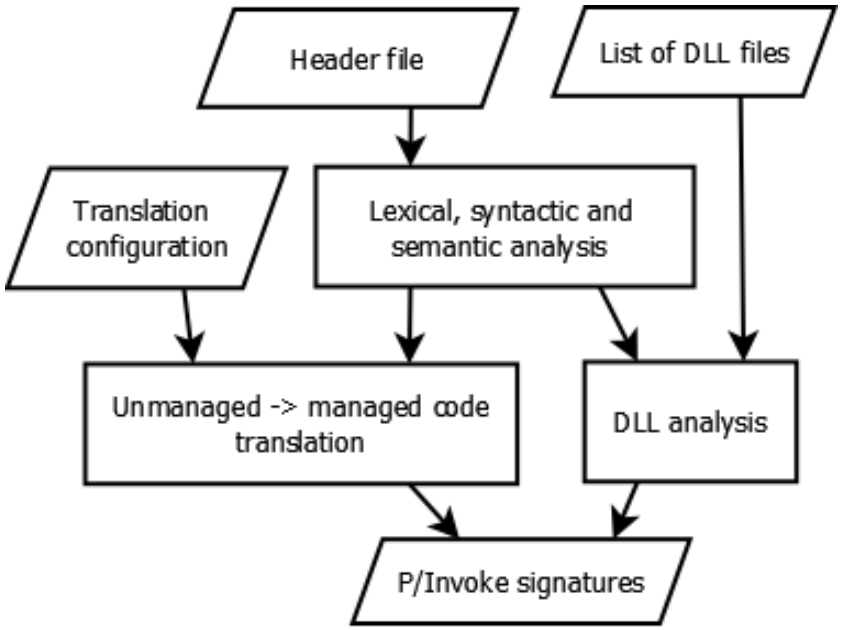


Figure 3-1 Program workflow overview

3.1 Parser construction

The first task of the program is to analysis C header file, which is done by parser. The goal of the parser is to the extract the information about the declarations of the functions and data type definition, which involves lexical, syntactic and semantic analysis. Lexical analysis is then divided into preprocessing and the processing of input by a lexer, which provides tokens for syntactic analysis done by a parser. The parser also does semantic analysis to provide necessary information about defined types to the lexer.

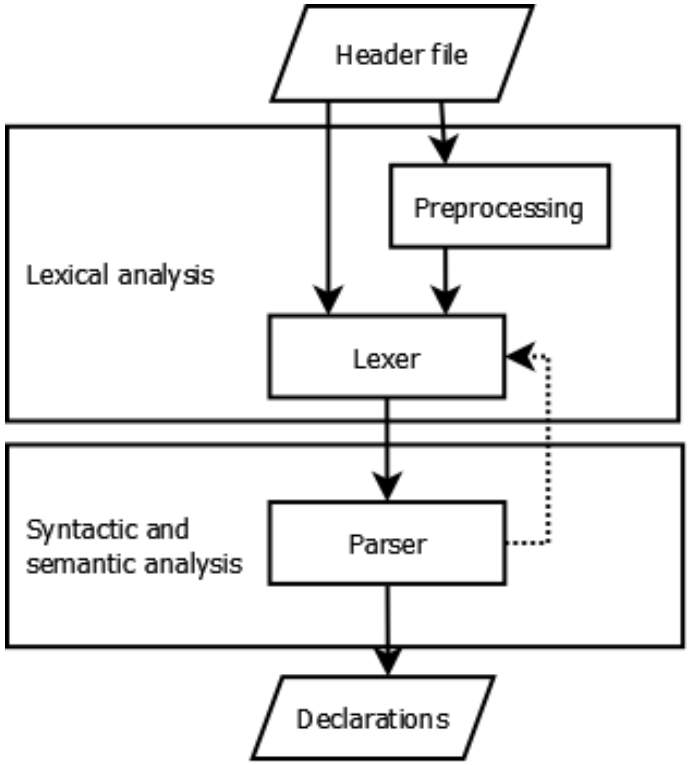


Figure 3-2 Parser overview

3.1.1 Preprocessing

As the input file is usually not preprocessed, preprocessing must be done to rid of preprocessor directives. As described in 2.2.1, Microsoft Visual Studio C++ (MSVS C++) compiler is used as external tool for preprocessing. It does not provide any API, thus it must be invoked via Windows command line. The invocation implies two tasks: find its location and set command line environment variables that are needed by the compiler to run correctly. The standard installation of MSVS C++ provides the script named *vsvars32.bat* that sets these required variables, so the location of the script has to be found or similar script

has to be created. Because the original script is created during MSVS C++ installation process and contains hardcoded strings, finding location of the script is easier than creating similar one. Microsoft does not provide any documentation on directory structure of MSVS C++ installation, but usually is the script located in the "Tools" directory under MSVS C++ installation directory. The most Windows programs store some information into the registry during their installation process, so it is one possible solution, how to locate "Tools" directory. Other options are locating the directory by searching the whole directory structure on the disk, which is slow and results can be ambiguous or let user set the directory manually. As location of either "Tools" or MSVS C++ installation directory is not stored in the registry, the path is derived from location of "IDE" directory, which is there and "IDE" directory is on the same level in directory structure as the "Tools" directory. If this approach fails, because of missing registry key or its invalid value, the user input is required to obtain the "Tools" directory. After location of the script is known, it is called to set desired environment variables for preprocessor. To call this script and then call the preprocessor, the batch file *runPreprocessor.bat* was created, which is invoked via .NET's *Process* class. The invocation also includes these parameters: Tools directory, the name of the script that sets the environment variables, output and input file and parameters for preprocessor. Exit code is retrieved to check that the preprocessing is done correctly.

3.1.2 Lexer

As there is no need to write the lexer by hand, the lexer generated by lexical grammar based on grammar written by J. A. Roskind [20] is used. The grammar mainly consists of set of regular expressions, which are matched against the character input stream and according to these matches tokens are created. The grammar is modified to reflect the Microsoft specific extensions described in 2.2.5. Because terms lexer and scanner are used interchangeably, the lexer is represented by *AnsiCScanner* class. Except the traditional lexer function that is translating input character stream to token stream for parser, the class also maintains the stack according to found *#pragma pack* directives (for details see 2.2.5) and provides the information of actual alignment value to the parser, which is important for translation phase done later.

3.1.3 Parser

A parser processing tokens generated by lexer from input C code is generated by GPPG tool from J. A. Roskind syntactic grammar [20]. GPPG accepts grammar in Yacc style [19] and allows to define semantic value type by using *%YYSTYPE* directive. The semantic value is used to create inner representation of parsed code in form of abstract syntax tree (AST) and it is usually set within the semantic blocks of code, which are placed after each grammar rule. In order to create AST that represents the code structure, semantic value type is set to *INode* interface, which represents any AST node. Because the *INode* actually carry syntactic value, it has a member called *SemanticValue* of type *object* to hold objects representing true semantic value of the Node. As the parser goes from the bottom of the grammar tree, these values are modified and merged as they are propagated towards top of the AST, where they conform a complete declaration that represents semantic information unit. The semantic information have to be stored, because they are used as input for translation phase. To do that each declaration is passed to *DeclarationHandler* object, which recognizes these types of declarations: new type name declaration, function declaration and structure, union or enum declaration. New type names are used for updating the symbol table, so lexer can recognize them. Function declarations are stored in the list and other declarations with type definitions are passed to *TypeHandler* object, which stores information about type definitions.

3.1.4 Updating symbol table

During the parsing have to be done the second main task of the parser that is updating symbol table names, which are used in lexer to distinguish identifier from name of the type. As defined in ANSI C standard [10] all symbols are valid only in block of code, where they are declared, so in order to implement this behavior, approach suggested in [28] is used for the structure of the table, but without maintaining the type stack as it is not needed due to the used grammar.

3.1.5 Completing the type information

The type objects stored in declarations mentioned in 3.1.3 does not have to have complete information, e.g. structure type does not have to know about its members, so this

information have to be completed as it is needed for marshalling. Recursive completing of these information with help of *TypeHandler* object passed as the parameter is used. This solution is definitely not perfect, because it complete the information for all types, i.e. even for those which are not marshaled, but it was easy to implement.

3.2 Marshalling process description

As soon as the types in stored declarations are completed, marshalling of selected functions and data types can be done. The marshalling process itself is implemented in the classes representing the function declaration and the data types.

3.2.1 Function declarations

Each function declared in the input header file is represented by one instance of the class *FunctionDeclaration* declared in AST assembly. This class is directly used by a GUI. The main method called *MarshalledDllImportOutput* returns the string with P/Invoke signature of the function. Other method called from GUI is *GetNonBasicTypesRecursive,* which returns a list of the types, which declarations should be marshalled with the function.

3.2.2 Type declarations

As C introduces different data types, classes representing these types are presented here. All types implements unified interface *ITypeDeclaration* used especially for translation phase.

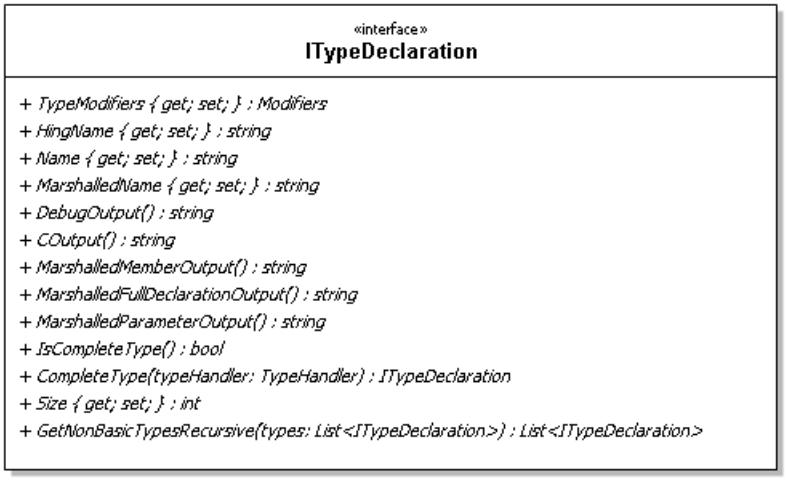


Figure 3-3 ITypeDeclaration interface

Classes implementing this interface are present in namespace AST.Types and represent these types: array, default C type, enum, function, pointer, structure, typedef and union. This division makes the marshalling process of the complex types complicated, e.g. class representing directly pointer to structure is available, etc.

3.2.3 Marshalling process configuration

Because marshalling types is not a process with strictly defined set of rules and can be done in various ways and also because there are many exceptions in this process, a configuration file with xml document structure is present to allow the user affect the process of marshalling these types. The configuration file provides option to modify the default marshalling behaviour of default C types, typedefs and also functions and their parameters.

3.3 Parsing Native DLL

The last step of creating P/Invoke signature involves adding DLL name based on exported function names from the DLL. The information contained in native DLL was described in chapter 2.2.6, the following text provides the procedure of obtaining these information. Parsing Native DLL means parsing PE format, but it is not straightforward process when DLL is not loaded in memory, because PE format extensively uses RVAs, which are only valid, when DLL is loaded in memory. Different structures created according to their unmanaged counterparts from the Windows API are used to read various headers in PE format, where each header contains information about the next header location. To read those headers from the file, unsafe code using generics introduced in the Codeproject article[29] in Chapter 4 was used for reading structures from byte array, which is created from input stream using .NET *BinaryReader* class. The code is shown in Figure 3-4.

```
protected static T ReadUsingMarshalUnsafe<T>(byte[] data) where T
: struct
{
    unsafe
    {
        fixed (byte* p = &data[0])
        {
            return      (T)Marshal.PtrToStructure(new      IntPtr(p),
                                                  typeof(T));
        }
    }
}
```

Figure 3-4: Reading structure from byte array

As mentioned in 2.2.6, native DLL contains names of exported functions, which are needed to determine DLL name that completes P/Invoke signature. The following figure ilustrates headers that must be read and their used members.

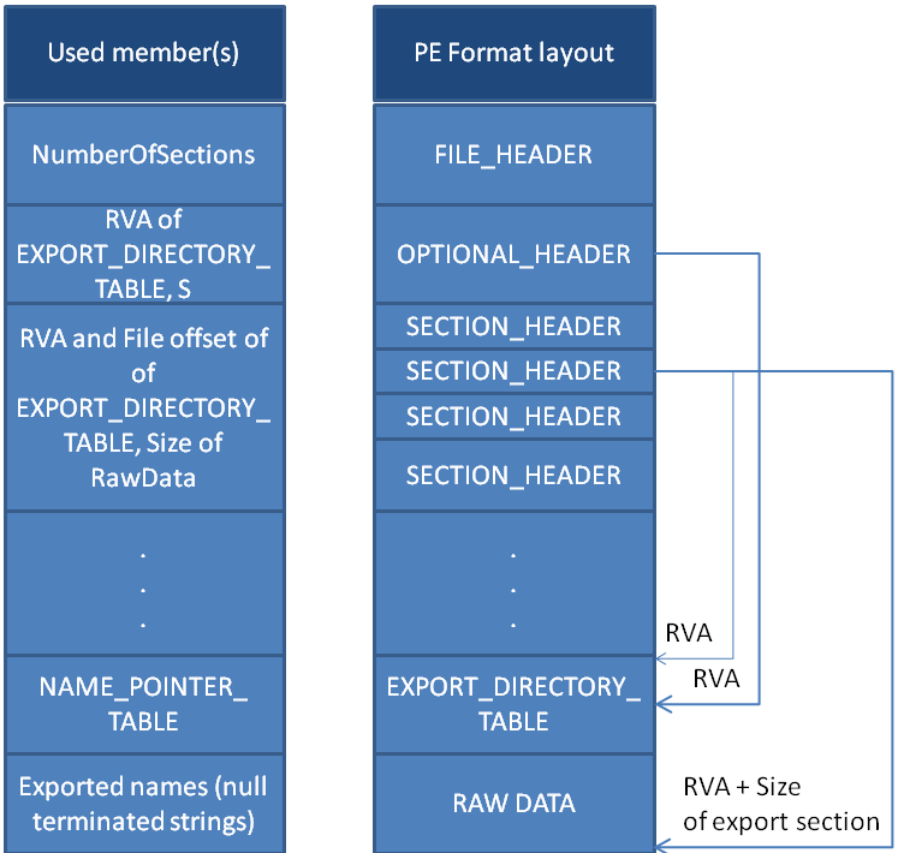


Figure 3-5: PE Format layout and used members from headers and sections

The goal is to read the NAME_POINTER_TABLE, which contains the file offsets to null terminated strings (exported names). It is a part of EXPORT_DIRECTORY_TABLE, which could be obtained by RVA from the OPTIONAL_HEADER, when in memory, but because the DLL is not loaded in memory, file offset from the SECTION_HEADER have to be used. Because the SECTION_HEADER that belongs to export section cannot be identified by the name it contains, all the SECTION_HEADERs are tested, whether the RVA pointing to the EXPORT_DIRECTORY_TABLE from the OPTIONAL_HEADER points somewhere to the region defined by the SECTION_HEADER RVA and size (see Figure 3-5). The other headers at the beginning of the file are read sequentially.

The above described procedure is implemented in the *PEReader* library, which provides only the functionality needed by the rest of the application, however it could be extented for use in other projects.

3.4 GUI Description

The GUI has a simple design, because the process does not need much user interaction. The input file path has to be provided and optionally the preprocessing can be skipped by checking the “Input file is preprocessed” option. The preprocessor options can be set in the settings tab, but it is usually necessary only in the case, when the preprocessor could not be located automatically. After the click on the “Parse” button, analysis of the given file is started. The user is informed about the process by the progressbar and by the text ouput in the area below.

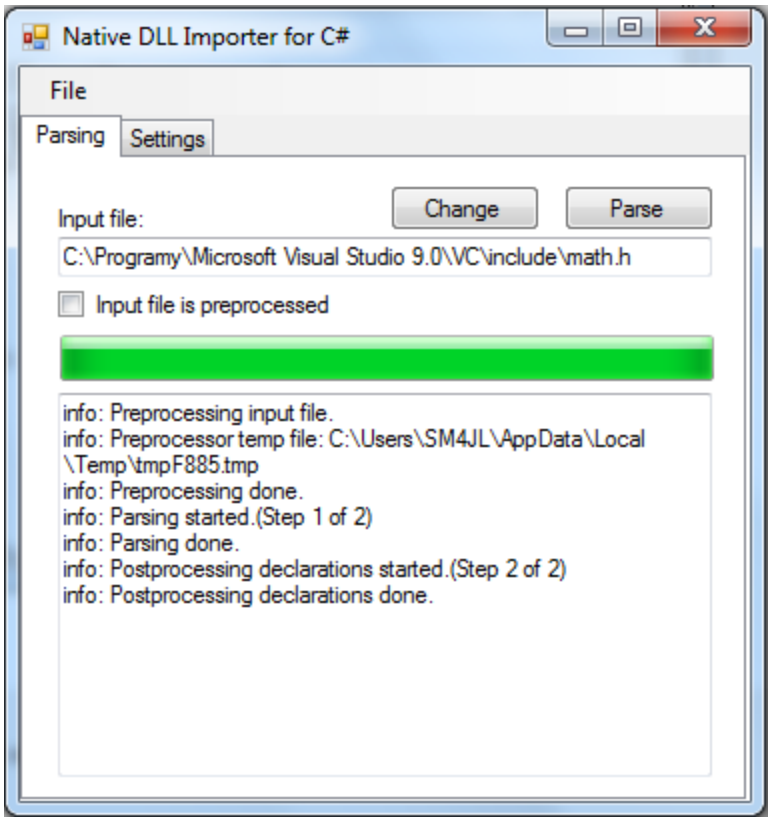


Figure 3-6 Main window screenshot

As soon as the analysis is complete the window illustrated in the Figure 3-7 is shown. The user can select the desired function from the list of functions, which were found during the analysis. By selecting the function three actions are invoked.

1. The types which are used in the function parameters are shown in the left bottom area.
2. The list of DLL libraries – configurable via “Config” button – is searched for the exported functions and the names of matched DLLs are offered in the “Dll name” combo box and the first one is automatically selected.
3. If “Auto show preview” option is checked, then the code preview of the selected function is shown in the right top text area.

Function list can be filtered by typing a part of the function name in the “Search” textbox. After all the desired functions are selected, the “Export checked” button is clicked to show the exported code.

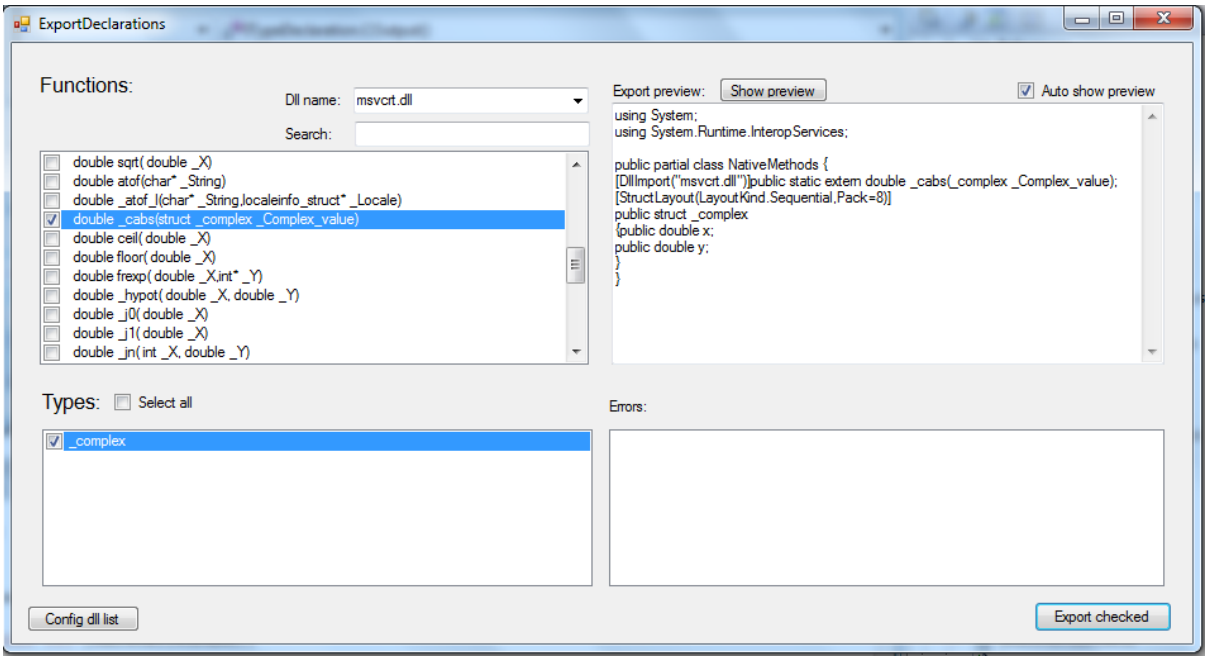


Figure 3-7 Export declarations window screenshot

4 Software requirements

- Microsoft .NET Framework 3.5 – needed to run the application
- Microsoft Visual Studio C++ 2008 and higher – needed to run the preprocessor

5 Conclusion

The aim of this thesis to generate managed counterparts for C functions and data types based on given C header file have been achieved. However, the solution is definitely not perfect. The header file information are analyzed by the written parser. The process of creating managed counterparts is simplified, thus the generated code is usually directly usable for common declarations and it is good starting point for more difficult and complex declarations, which need to be further modified manually. The configuration file provides some basic options for modifying the marshalling process, which are not sufficient in order to solved difficult cases, but these options provide a partial control over the marshalling process for simpler cases.

The improvement in the future could be the GUI for editing the configuration file, which would make the application more user-friendly. Also the configuration itself could provide more options, which would result in a better quality and usability of the exported code. Another useful improvement would be the connection of the program with the Microsoft Visual Studio environment.

7 Bibliography:

1. Wikipedia, the free encyklopedia: .NET Framework. *Wikipedia, the free encyklopedia.* [Online] http://en.wikipedia.org/wiki/.NET_Framework.

2. **Gediminas, Cibas.** Why .NET Framework? *Netvision.* [Online] Oktober 22, 2006. [Cited: August 2, 2010.] http://www.netvision.lt/en/why_net_framework.aspx.

3. Platform Invocation Services: Wikipedia, the free encyklopedia. *Wikipedia, the free encyklopedia.* [Online] [Cited: July 31, 2010.] http://en.wikipedia.org/wiki/Platform_Invocation_Services.

4. Dynamic link library. *Wikipedia.* [Online] 2010. [Cited: July 20, 2010.] http://en.wikipedia.org/wiki/Dynamic-link_library.

5. *pinvoke.net: the interop wiki!* [Online] http://pinvoke.net.

6. Managed, Native, and COM Interop Team: PInvoke Interop Assistant. *CodePlex.* [Online] July 20, 2010. [Cited: July 20, 2010.] http://clrinterop.codeplex.com/releases/view/14120.

7. mcpp - a portable C preprocessor. *Sourceforge.* [Online] November 30, 2008. [Cited: June 26, 2010.] http://mcpp.sourceforge.net/.

8. GCC, the GNU Compiler Collection. *GCC, the GNU Compiler Collection.* [Online] [Cited: March 10, 2010.] http://gcc.gnu.org/.

9. **Microsoft.** Downloads: Visual C++ Developer Center. *Visual C++ Developer Center.* [Online] 2010. [Cited: March 10, 2010.] http://msdn.microsoft.com/en-us/visualc/aa336402.aspx.

10. ANSI C Standard. [Online] http://flash-gordon.me.uk/ansi.c.txt.

11. Wikipedia, the free encyklopedia: Grammar. *Wikipedia, the free encyklopedia.* [Online] http://en.wikipedia.org/wiki/Grammar.

12. Parsing. *Wikipedia, The Free Encyclopedia.* [Online] July 31, 2010. [Cited: August 4, 2010.] http://en.wikipedia.org/wiki/Parser#Overview_of_process.

13. **Gough, J. and Kelly, W.** The GPPG Parser Generator. *Queensland University of Technology - Australia.* [Online] 2009. http://plas.fit.qut.edu.au/gppg/files/gppg.pdf.

14. **Dick, Grune and Jacobs, Ceriel.** *PARSING TECHNIQUES A Practical Guide.* Vrije Universiteit, Amsterdam : Printout by the Authors, 1998. Originally published by Ellis Horwood, Chichester, England, 1990. ISBN 0-13-651431-6.

15. Wikipedia, The Free Encyclopedia: Types of parsers. *Wikipedia, The Free Encyclopedia.* [Online] July 31, 2010. [Cited: August 4, 2010.] http://en.wikipedia.org/wiki/Parser#Types_of_parser.

16. **Gough, John.** The GPLEX Scanner Generator. *Queensland University of Technology - Australia.* [Online] March 3, 2009. [Cited: March 26, 2010.] http://plas.fit.qut.edu.au/gplex/files/gplex.pdf.

17. **Merrill, Brad.** CsLex: A lexical analyzer generator for C#(TM). [Online] Microsoft, September 24, 1999. [Cited: March 10, 2010.] http://www.cybercom.net/~zbrad/DotNet/Lex/Lex.htm.

18. Wikipedia, The Free Encyclopedia: The Lexer Hack. *Wikipedia, The Free Encyclopedia.* [Online] October 6, 2009. [Cited: August 4, 2010.] http://en.wikipedia.org/wiki/The_lexer_hack.

19. **Jutta, Degener.** ANSI C Yacc grammar. *The Questionable Utility Company.* [Online] November 2008. [Cited: March 3, 2010.] http://www.quut.com/c/ANSI-C-grammar-y.html.

20. **Roskind, James A.** [Online] [Cited: February 10, 2010.] ftp://ftp.iecc.com/pub/file/c++grammar/.

21. **Microsoft Corporation.** __declspec (C++). *MSDN: Microsoft Developers Network.* [Online] http://msdn.microsoft.com/en-us/library/dabb5z75%28VS.80%29.aspx.

22. —. MSDN: pack (C/C++). *Microsoft Developers Network (MSDN).* [Online] http://msdn.microsoft.com/en-us/library/2e70t5y1%28VS.80%29.aspx.

23. —. WHDC: Microsoft Portable Executable and Common Object File Format Specification. *Microsoft Hardware Developer Central (WHDC).* [Online] 2010. [Cited: February 15, 2010.] http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx.

24. **Elsheimy, Mohammad.** Marshalling with C# Pocket reference | Just Like a Magic. *Just Like a Magic.* [Online] March 9, 2010. [Cited: July 4, 2010.] http://justlikeamagic.wordpress.com/2010/03/09/marshaling/.

25. **Microsoft Corporation.** MSDN: Simple Types. *Microsoft Developer Network (MSDN).* [Online] 2010. [Cited: July 26, 2010.] http://msdn.microsoft.com/en-us/library/aa691144%28VS.71%29.aspx.

26. —. MSDN: Blittable and Non-Blittable Types. *Microsoft Developer Network (MSDN).* [Online] 2010. [Cited: July 26, 2010.] http://msdn.microsoft.com/en-us/library/75dwhxf7%28VS.85%29.aspx.

27. —. MSDN: C# Attributes. *Microsoft Developers Network (MSDN).* [Online] 2010. [Cited: June 25, 2010.] http://msdn.microsoft.com/en-us/library/aa287992%28VS.71%29.aspx.

28. Practical Parsing for ANSI C. *Dr. Dobb's.* [Online] December 6, 2006. [Cited: March 11, 2010.] http://www.ddj.com/cpp/196603535?pgno=4.

29. CodeProject: Reading Unmanaged Data Into Structures. *CodeProject.* [Online] May 8, 2008. [Cited: April 2, 2010.] http://www.codeproject.com/KB/dotnet/ReadingStructures.aspx?display=Print.

30. **Microsoft Corporation.** MSDN: Overview of the Windows API (Windows). *Microsoft Developer Network (MSDN).* [Online] 2010. [Cited: July 20, 2010.] http://msdn.microsoft.com/library/default.asp?url=/library/en-us/winprog/winprog/overview_of_the_windows_api.asp.