.NetAsm

A JIT Native Code Injection Library

NETASM USER GUIDE

How to use native code injection with .NET

Alexandre MUTEL - alexandre_mutel [at] yahoo.fr

Date	Contributor	NetAsm Version	Comment
24 July 2008	Alexandre MUTEL	1.0	Initial version

This document is a short user guide to help you use NetAsm native code injection. Feel free to send your comments and improvement requests at http://www.codeplex.com/netasm

TABLE OF CONTENTS

1 INTRODUCTION

NetAsm provides a **hook to the .NET JIT compiler** and enables to inject your own native code in replacement of the default CLR JIT compilation. With this library, it is possible, at runtime, to **inject x86 assembler code in CLR methods** with the speed of a pure CLR method call and without the cost of Interop/PInvoke calls.

The main features of NetAsm are:

- Runs on x86 32bit Microsoft .NET platform with 2.0+ CLR runtime (x64 may be supported in the future).
- Provides three different native **code injection techniques**: **Static**, **DLL**, and **Dynamic**.
- **Supports for debugging** static and dynamic code injection.
- Supports for StdCall, FastCall, ThisCall, Cdecl. Default calling convention is CLRCall.
- NetAsm can be used inside **any .NET language**.
- **Very small library** <100Ko.

1.1 Hello World Code Injection sample

Using native code injection with NetAsm is a very simple task that can be achieved in two steps:

1) Specify the native code injection in a class
2) Install the hook in the main program of your application

1.1.1 Setup native code injection in a class

The following code is our first native code injection using static code injection technique:

```
using System;
using System.Runtime.CompilerServices;
using NetAsm;

namespace NetAsmDemo
{
    [AllowCodeInjection]
    class TestHelloWorld
    {
        [CodeInjection(new byte[] { 0xC3 }), MethodImpl(MethodImplOptions.NoInlining)]
        static public void NetAsmReturn()
        {
            throw new Exception("With NetAsm, You should not have an exception!");
        }
    }
}
```

The steps to allow and use code injection on a method in a class are:

1. Set [`AllowCodeInjection`] attribute on the class you want to do code injection.
2. Set [`CodeInjection`] attribute on the method that will be injected with native code
3. For `void NetAsmReturn()` method in `TestHelloWorld`, the native code used is : `new byte[] { 0xC3 }`. In x86 assembler, it's the "RET" (return) command. This method does nothing more than immediately returning after a call.
4. In our example, we have set the attribute `MethodImpl(MethodImplOptions.NoInlining)` : This attribute force the JIT to not inline the IL code inside the method. This is for the purpose of the demonstration but should be used with caution (see usage recommendation chapter).

That's all to use code injection!

In NetAsm, this kind of native code injection is called **static native code injection**. `TestHelloWorld` only use static code injection at the method level. We will see later that NetAsm provides other code injection techniques.

Now, to run this code injection test, we need to install NetAsm JITHook.Install().

1.1.2 Install the hook and run the code injection

To call the `TestHelloWorld` method, the main program has to initialize NetAsm:

```
using System;
using NetAsm;

namespace NetAsmDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            // Install the JIT Hook
            JITHook.Install();
            // Run TestHelloWorld Method
            TestHelloWorld.NetAsmReturn();
            // Remove the JIT Hook
            JITHook.Remove();
        }
    }
}
```

If you run this program, it will return without any exception. While executing this code, the CLR Virtual Machine use our native code "RET" command instead of the IL code inside the method. It means that the original IL code is not compiled by the default JIT compiler.

Now that we know how to do a simple native code injection, we are going to see other code injection techniques available in NetAsm.

1.2 Usage recommendations and restrictions

1.2.1 Recommendations

- Before using NetAsm, **make sure that it is worth to use native code** instead of the default JIT. Managed code can be already compiled quite efficiently by the default JIT.
- **Keep your native code as small as possible**: don't make any call from your native code to any operating systems functions.
- You should always (when possible) try to **implement first the method in managed code**, and leave it in the method body. This way, if you disable NetAsm, your code should be able to run in full managed mode.
- **Always benchmark your native code** versus your managed code.

1.2.2 Restrictions

- NetAsm only supports currently 32bit Microsoft OS.
- Debugging is limited to static and dynamic code injection.

2 CODE INJECTION TECHNIQUES

NetAsm provides three native code injection techniques:

- **Static code injection**: We have seen this technique in the TestHelloWorld. The native code is stored in an attribute of the method.
- **Dll code injection** : this method is similar to the DllImport mechanism but CLR methods are directly bind to the DLL function, without going through the interop layers.
- **Dynamic code injection**: you can generate native code dynamically with a callback interface that is called by the JIT when compilation of a method is occurring. It means that you can compile a method "on the fly". You have also access to the IL code of the method being compiled.

2.1 General

2.1.1 Code injection mechanism

NetAsm provides a hook just before the default JIT compiler of the CLR VM is called. With NetAsm, you can replace the native code of a method with your own optimized assembler code.

The general mechanism and restrictions of NetAsm are:

- NetAsm is working on any **public/protected/internal/private methods** (but not on constructors):
    - **Instance methods**: The native method expect the first parameter to be the this pointer of the instance of the object.
    - **Static methods**
- **NetAsm is a JIT native code injector and consequently not a JIT compiler**: it doesn't compile the original IL Code to native code : this is the work of the default JIT compiler. Although, it is almost possible to do your own compiler with NetAsm dynamic native code injection technique, but this is out of the scope of NetAsm.
- **NetAsm doesn't inline native assembler code inside a part of a method** but **replace the whole code of the method** with your own native code: it means that when you use NetAsm, the IL code inside the method is not executed – even partially.
- Once the native code is generated for a method, it cannot be re-generated again.
- **NetAsm cannot inline the native code of a method into another**. Only the default JIT compiler supports this feature: You need to be very carefully when choosing to use NetAsm for small methods and to bypass the inline capabilities of the JIT: you may have a slight performance impact. Use the attribute `MethodImpl(MethodImplOptions.NoInlining)` to force the default JIT compiler to not inline the method.
- **The generated native code is considered as CLR code and not as unmanaged code**. It means that there are some restrictions: it is for example impossible to debug DLL injected native code. See Debug section for more information.
- Native code injection is called by the CLR VM, when you try to access at runtime a method that is not yet compiled. NetAsm doesn't compile methods at startup: **native code for methods is therefore JIT injected**.

2.1.2 Code injection level

Native code injection can take place at 3 different levels:

- **Method level**: the code injection is specified at the method level using [`CodeInjection`] attribute.
- **Class level**: the code injection is specified at the class level using [`CodeInjection`] attriute.
- **Application level**: the code injection is specified at the application level, using global code injector in the `JITHook` class.

Code injection techniques do not apply to each level. The following table shows code injection techniques allowed for each level:

Technique	Method level	Class level	Application level
Static code injection	🟢	🔴	🔴
Dll code injection	🟢	🟢	🔴
Dynamic code injection	🟢	🟢	🟢

To determine which code injection to use for a method, NetAsm tries to find a code injection at the method level first, then at the class level, and if it is configured, at the application level.

So the order to get the code injection is:

If defined, take **Method Level** →, else take **Class Level** →, else take **Application Level**

For example, if you define a static code injection on a method1 of Class A and a dynamic code injection on Class A, the method1 will be injected with its static code injection defined in the attribute and all other methods will be dynamically compiled by the ICodeInjector provided at the class level.

2.1.3 Calling conventions

2.1.3.1 SUPPORTED CALLING CONVENTIONS

The **calling convention of the CLR is ClrCall**. This is a variant of the FastCall calling convention but arguments are passed from left to right.

In order to achieve optimal speed and avoid any conversion, **NetAsm use by default the ClrCall convention**. When you declare a code injection technique, the default Calling Convention is then ClrCall. You can change the Calling Convention technique using the optional parameter CallingConvention in the [`CodeInjection`] attribute. The behavior of the ClrCall calling convention is:

- First parameter is stored in ECX register, Second Parameter is stored in EDX. Other parameters are pushed on the stack, from left to right parameters. We will see that there are some exceptions to this rule. The callee cleans the stack.

NetAsm also supports other calling conventions:

- **FastCall** : Similar to **ClrCall**, but method's parameter are passed from right to left. The callee cleans the stack.
- **StdCall** : All parameters of a method are pushed on the stack, from right to left. The callee cleans the stack.

- **ThisCall** : Used to communicate with c++ object (no example yet with NetAsm). First parameter is ECX (and is the This pointer of an instance of the object). Other parameters are passed on the stack from right to left. The callee clean the stack.
- **Cdecl** : Similar to **StdCall**, but the caller cleans the stack.

To use other calling conventions than the default ClrCall, use the optional parameter CallingConvention in the CodeInjection attribute.

Be aware that these calling conventions are partially supported by NetAsm. For example, parameters using structure larger than 8 bytes are not supported (will be fixed in next version of NetAsm)

See `TestCallingConventions.cs` in NetAsmDemo for a set of examples of calling conventions.

2.1.3.2 THE CLRCALL CALLING CONVENTION

ClrCall (as FastCall) can be a bit confusing when parameters are mixed with 4 bytes and 8 bytes size parameters (64Bit). We are going to see through some examples how to declare a C prototype equivalent of a C# method ClrCall.

All the following examples are extracted from `TestCallingConventions.cs` in NetAsmDemo.

Example 1: 3 integer parameters:

The C# declaration code is:

```
[CodeInjection("NetAsmDemoCLib.dll"), MethodImpl(MethodImplOptions.NoInlining)]
public static int Test3ArgClrCall(int x, int y, int z) {return 0;}
```

- x will be in ECX register
- y will be in EDX register
- z will be push on the stack

The C declaration code of the function is:

```
extern "C" int __fastcall Test3ArgClrCall(int x, int y, int z) {return x + y * 3 + z * 5;}
```

As you can see, in C/C++, you can't use the __clrcall calling convention, but you have to **use the fastcall convention**. This is because when using __clrcall convention in C/C++, the compiler expect the solution to be compiled with the /clr flag. We don't want to use any CLR method, because we are using native code! So, we have to fake a ClrCall with a FastCall convention.

Up to 3 parameters, ClrCall and FastCall convention are similar between C# and C/C++.

Example 2: 4 integer parameters:

The C# declaration is:

```
[CodeInjection("NetAsmDemoCLib.dll"), MethodImpl(MethodImplOptions.NoInlining)]
public static int Test4ArgClrCall(int x, int y, int z, int w) {return 0;}
```

- x will be in ECX register
- y will be in EDX register
- z will be push on the stack
- w will be push on the stack

The equivalent ClrCall declaration in C code of the function is:

```
int __fastcall Test4ArgClrCall(int x, int y, int w, int z) {return x + y*3 + z*5 + w*7;}
```

As you can see. The W and Z parameters are reversed. This is because FastCall convention is pushing parameters on the stack from right to left.

Example 4: 4 integer parameters and a double 64bit parameter. This example shows the main difference between the clrcall and fastcall convention.

The C# declaration is:

```
[CodeInjection("NetAsmDemoCLib.dll"), MethodImpl(MethodImplOptions.NoInlining)]

                                                       ECX             EDX
public static double Test4ArgWith1DoubleClrCall(int x, double y, int z, int w, int ww) {
return 0;}
```

With ClrCall, Arguments are push on the stack from left to right (y first, w, and ww)

The behavior is different here, because 64 bit parameters are not stored in registers but are pushed on the stack:

- x will be in ECX register
- y will be pushed on the stack
- z will be in EDX register
- w will be push on the stack
- ww will be push on the stack

The equivalent declaration in C of this ClrCall is thus a FastCall:

```
                                                    ECX    EDX
double __fastcall Test4ArgWith1DoubleClrCall(int x, int z, int ww, int w, double y) {

    return x + y * 3 + z * 5 + w * 7;
}
```

The equivalent ClrCall in C is a fastcall with reversed arguments (pushed from right to left)

Notice the order of the parameters: Because double (but also Int64 UInt64, structure) are pushed on the stack, they are not used by a register. Still, the compiler allocates a register for the first available parameter (z) from left to right that has a maximum size of 4 bytes (and is not a structure as well).

This behavior needs to be fully understood before playing with ClrCall conventions!

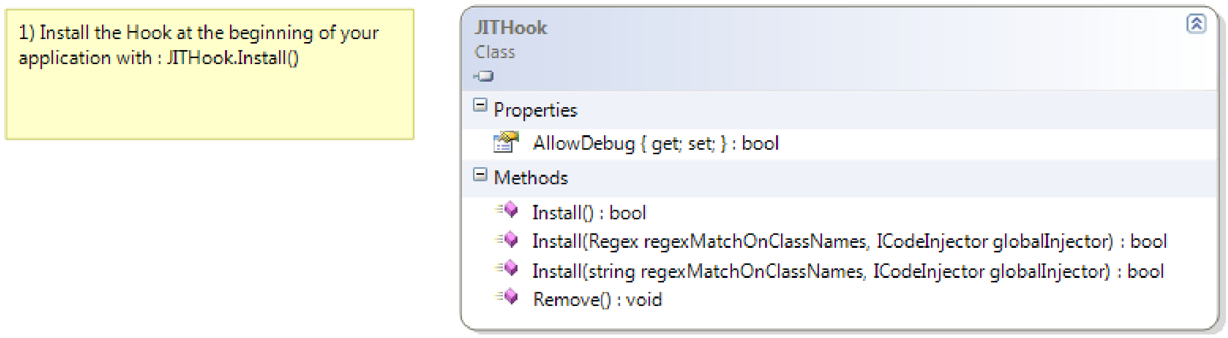
All the methods in the previous examples were declared static: be aware that **if a method is not static (instance), the first parameter of the C equivalent method is the this pointer** of the instance of the object. The **this pointer is always stored in the ECX register**.

2.1.4 NetAsm class diagram

The NetAsm public API is small and thus very simple to use and understand.

2.1.4.1 JITHOOK CLASS

This is the main class that should be called at the beginning of your application.

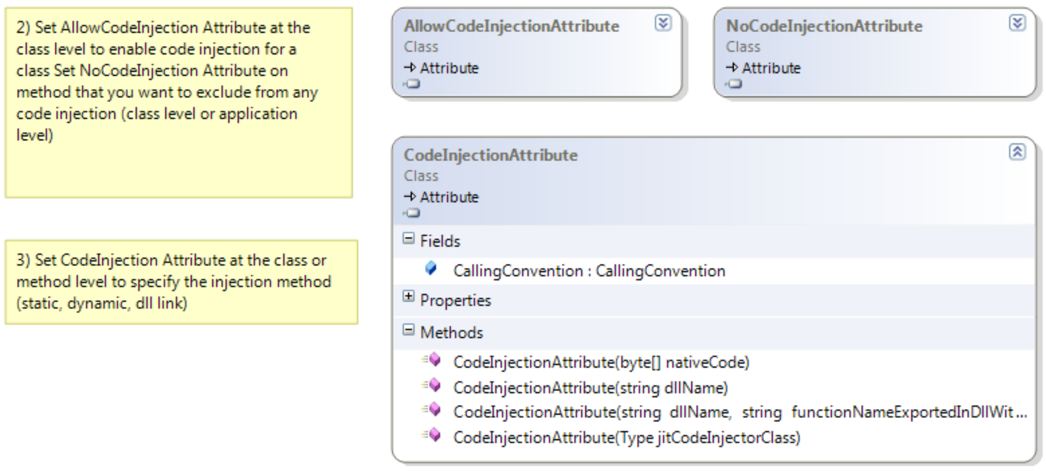


1) Install the Hook at the beginning of your application with : JITHook.Install()

```
JITHook
Class

Properties
    AllowDebug { get; set; } : bool
Methods
    Install() : bool
    Install(Regex regexMatchOnClassNames, ICodeInjector globalInjector) : bool
    Install(string regexMatchOnClassNames, ICodeInjector globalInjector) : bool
    Remove() : void
```

2.1.4.2 ATTRIBUTES CLASS

To identify the code injection, NetAsm uses three classes:



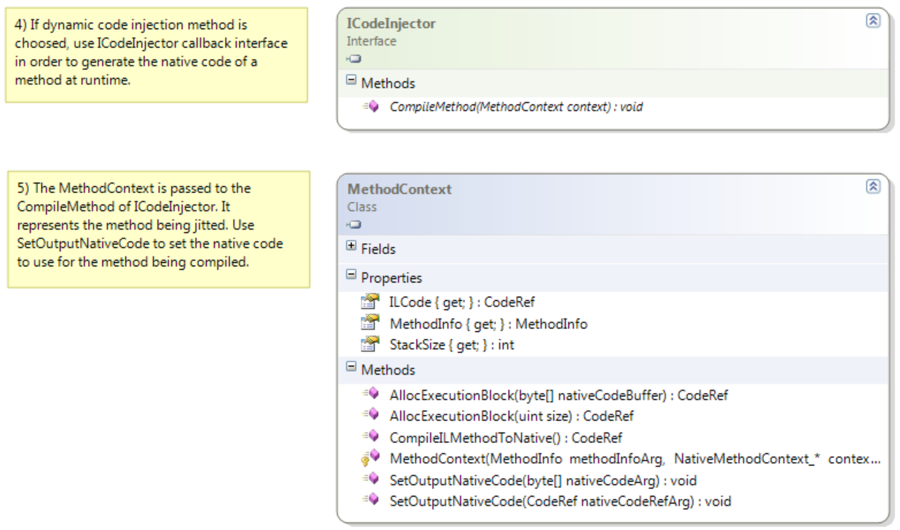
2) Set AllowCodeInjection Attribute at the class level to enable code injection for a class Set NoCodeInjection Attribute on method that you want to exclude from any code injection (class level or application level)

3) Set CodeInjection Attribute at the class or method level to specify the injection method (static, dynamic, dll link)

```
AllowCodeInjectionAttribute
Class
→ Attribute

NoCodeInjectionAttribute
Class
→ Attribute

CodeInjectionAttribute
Class
→ Attribute

Fields
    CallingConvention : CallingConvention
Properties
Methods
    CodeInjectionAttribute(byte[] nativeCode)
    CodeInjectionAttribute(string dllName)
    CodeInjectionAttribute(string dllName, string functionNameExportedInDllWit...
    CodeInjectionAttribute(Type jitCodeInjectorClass)
```

2.1.4.3 DYNAMIC CODE INJECTION CLASS

To use dynamic native code injection, Netasm provides one interface and one class:



4) If dynamic code injection method is choosed, use ICodeInjector callback interface in order to generate the native code of a method at runtime.

5) The MethodContext is passed to the CompileMethod of ICodeInjector. It represents the method being jitted. Use SetOutputNativeCode to set the native code to use for the method being compiled.

```
ICodeInjector
Interface

Methods
    CompileMethod(MethodContext context) : void

MethodContext
Class

Fields
Properties
    ILCode { get; } : CodeRef
    MethodInfo { get; } : MethodInfo
    StackSize { get; } : int
Methods
    AllocExecutionBlock(byte[] nativeCodeBuffer) : CodeRef
    AllocExecutionBlock(uint size) : CodeRef
    CompileILMethodToNative() : CodeRef
    MethodContext(MethodInfo methodInfoArg, NativeMethodContext_* contex...
    SetOutputNativeCode(byte[] nativeCodeArg) : void
    SetOutputNativeCode(CodeRef nativeCodeRefArg) : void
```

2.2 Static code injection (SCI)

Static native code injection is the most simple code injection technique in NetAsm although you need to know how to generate your assembler native code. If you want to inject your own native code without depending on an external DLL, you should use native code injection.

To setup a static native code injection, we just have to initialize a [CodeInjection] attribute with an array of native code bytes (assembler x86):

```
        [CodeInjection(new byte[] { 0xC3 }), MethodImpl(MethodImplOptions.NoInlining)]
        static public void NetAsmReturn()
```

NetAsm doesn't provide currently any assembler to facilitate translation of textual assembler instruction to binary assembler code. To import your own code, you should use different techniques:

- Use an external assembler. FASM for example provides a way to output a simple binary format (without any COFF's like structure) from an assembler source code.
- Use the Microsoft visual C++ inline assembler, configure your project to generate assembler listing and copy back the data from the *.cod generated files.

Due to its static nature, **Static code injection only works on method** and is not available at class or application levels:

Technique	Method level	Class level	Application level
Static code injection	🟢	🔴	🔴

NetAsm may provide in the future a simplified assembler to allow the use of textual assembler code inside the [CodeInjection] attribute.

See TestStaticCodeInjection.cs in NetAsmDemo for a set of examples using static code.

2.3 Dll code injection (DLLCI)

NetAsm Dll native code injection is the equivalent of the interop DllImport mechanism. Although, NetAsm DLLCI differs in many ways:

- **DLLCI use ClrCall** as the default calling convention. You can still use other calling conventions (StdCall, FastCall, ThisCall, Cdecl), even if these implementations are still limited (structure on the stack are not supported).
- The DLL function is directly associated to the method implementation: there is **no additional layer between the CLR and your code**. The Dll function is executed as it was a pure CLR native code.
- You can bind a dll function on an **instance and static method** (DllImport enable only on static methods). Be careful that the first parameter of the native method will be the **this** pointer on the instance object.

To setup a DLL native code injection, we just have to initialize a [CodeInjection] attribute at the method level or at the class level :

Technique	Method level	Class level	Application level
Dll code injection	🟢	🟢	🔴

At the **method level**, with an **explicit name for the dll function** :

```
[CodeInjection("NetAsmDemoCLib.dll", "NetAsmAddInC"), MethodImpl(MethodImplOptions.NoInlining)]
public int NetAsmAddDll(int x, int y)
```

Or by using the **name of the method to resolve the dll function**. In this example, the C# method is binded to the `NetAsmAddInC` function exported by the `NetAsmDemoCLib.dll`

```
[CodeInjection("NetAsmDemoCLib.dll"), MethodImpl(MethodImplOptions.NoInlining)]
public int NetAsmAddInC(int x, int y)
```

At the **class level, without any name function**. All the methods in the class will be bind to the equivalent exported dll function (using the name of the method):

```
[AllowCodeInjection, CodeInjection("NetAsmDemoCLib.dll")]
class TestDllCodeInjection {    }
```

When you set a code injection at the class level, all the methods will be resolved to the external DLL. If you want to force a method not to be code injected, you have to set the attribute `[NoCodeInjection]` on the method. If you set both `[NoCodeInjection]` and `[CodeInjection]` `on a same method`, the code injection attribute won't be used.

See `TestDllCodeInjection.cs` in NetAsmDemo for a set of examples using dll code.

2.4 Dynamic code injection (DCI)

Dynamic native code injection is the most versatile code injection technique in NetAsm. You can dynamically generate the native code of methods at runtime, with the use of a callback interface that is called every time a method needs to be compiled.

With this technique, you can:

- Make a static like native code injection.
- Make a Dll like native code injection.
- Combine any native code injection techniques.
- Generate a native code based on the IL Code (DCI gives access to the IL Code of the method being compiled).
- And a lot more…

DCI can be applied at **method level**, **class level** and **application level** (also called Global Injector):

Technique	Method level	Class level	Application level
Dynamic code injection	🟢	🟢	🟢

DCI is based on a simple callback interface `ICodeInjector` that provides an easy way to generate dynamic native code.

2.4.1 Use of ICodeInjector interface

The use of `ICodeInjector` is straightforward. In order to understand DCI, we are going to analyze the `NopCodeInjector` example class provided in NetAsmDemo.

This class injects a simple native code that makes the method returning immediately. The specificity of this code injector is that:

- It prints information about the method being compiled.
- It generates a different native code, if there is a stacksize or not on the method being JITed
  - If no stacksize, it generates the simple code we have already seen in TestHelloWorld : The .x86 RET assembler opcode (0xC3).
  - If stacksize > 0, it generates a RET + StackSize .x86 assembler opcode (0xC2 + Short stack size)

The code of the `NopCodeInjector`:

```
/// <summary>
/// Simple NopCodeInjector used by <see cref="TestDynamicCodeInjection"/>.
/// This CodeInjector generate a Nop method for any methods (with eventual parameters).
/// </summary>
public class NopCodeInjector : ICodeInjector
{
    // .X86 instruction : RET + stacksize
    private static byte[] ReturnWithStackSize = new byte[] {0xC2, 0x00, 0x00};
    private static byte[] ReturnWithNoStack = new byte[] { 0xC3};

    public void CompileMethod(MethodContext context)
    {
        // 1) Print information about the method being compiled using reflection classes.
        MethodInfo methodInfo = context.MethodInfo;
        Console.Out.WriteLine("> JIT NopCodeInjector is called on method [{0}] with a StackSize {1}",
methodInfo.Name, context.StackSize);

        // 2) Generate a native code with a Ret + StackSize of the method
        CodeRef nativeCodeRef;
        if (context.StackSize > 0)
        {
            // If there is a stack, then, generate the RET + Stacksize opcode
            nativeCodeRef = context.AllocExecutionBlock(ReturnWithStackSize);
            Marshal.WriteInt16(nativeCodeRef.Pointer, 1, (short) context.StackSize);
        } else
        {
            // If there is no stack used, then RET
            nativeCodeRef = context.AllocExecutionBlock(ReturnWithNoStack);
        }

        // 3) Inform NetAsm with the native code to use for the method
        context.SetOutputNativeCode(nativeCodeRef);
    }
}
```

As we can see, we need to implement the CompileMethod. `MethodContext` is the only parameter of this method and gives access to:

- The reflection `MethodInfo` being JITed.
- The IL code of the method (A CodeRef is returned with a pointer to the IL Code)
- Compile the IL code to a native code and have access to the compiled code.
- Method to effectively output the native code back to the JIT : `SetOutputNativeCode` . This method has two signatures:
  - `void SetOutputNativeCode(byte[] nativeCodeArray);`
  - `void SetOutputNativeCode(CodeRef nativeCodePointer);`

CodeRef is a small structure that contains a pointer to a code and the size of the code in bytes.

In the NopCodeInjector example, we use the method context.AllocExecutionBlock to allocate a CodeRef with a native code buffer. This method allocates a block of memory that can be executed by the processor and copy the native code buffer to this newly allocated block of execution memory. The CodeRef returned through SetOutputNativeCode method should be instantiate within the context and using the AllocExecutionBlock from the context. This is necessary to enable debugging for native code. Although, it is possible to allocate other external execution block (see CodeRef.AllocExecutionBlock).

To use the NopCodeInjector we only have to set the code injection technique on a method (or a class):

```
[CodeInjection(typeof(NopCodeInjector)), MethodImpl(MethodImplOptions.NoInlining)]
public static void NetAsmNopMethod()   {...}
```

2.4.2 Global Native Code Injection

We have seen that Dynamic Code Injection (DCI), is able to provide a code injection at the application level : this is also called Global Code Injection (GCI). NetAsm provide a code injection based on pattern regular expression matching on class names.

The CodeInjector is setup while initializing the JITHook. We need to install the code injector here, and associate it with a regular expression. The following example extracted from NetAsmDemo force the JIT to use the GlobalCodeInjector for all classes inside the NetAsmDemo.* namespace.

```
// ---------------------------------------------------------------------------------
// Install the JITHook.
// Test a Global Code Injector with pattern regex on classes names. Perform a Global
// Code Injection only on NetAsmDemo..* namespace
JITHook.Install("NetAsmDemo\\..*",new GlobalCodeInjector());
...Run_your_code_here...
// Remove it
JITHook.Remove();
// The JITHook with the Global Injector is removed here
// ---------------------------------------------------------------------------------
```

Because all the classes inside the NetAsmDemo namespace will be compiled, we have placed the GlobalCodeInjector in a different namespace. Otherwise, NetAsm would have end up in an infinite recursive loop, trying to compile the compiler with itself…

As a consequence, you should always verify that the GlobalCodeInjector is not matched by the regular expression passed to JITHook.Install method!

```
namespace NetAsmDemoSafeGarden
{
    public class GlobalCodeInjector : ICodeInjector
    {
        public void CompileMethod(MethodContext context)
        {
            // Just display that the Global hook is called
            PerfManager.Instance.WriteLine("> Global JIT Code Injector is called : Compile Method
{0}.{1}", context.MethodInfo.DeclaringType.Name, context.MethodInfo.ToString());
            // Nothing to do. Let the default JIT compile the method.
        }
    }
}
```

3 BENCHMARKS

NetAsmDemo provides two micro-benchmarks to explicit performance gains using NetAsm:

- **Simple Add Benchmark** is a benchmark that demonstrates the overhead of calling methods using different managed, interop techniques against NetAsm.
- **Matrix Multiplication Benchmark SSE2** is a benchmark that demonstrates the use of SSE2 for optimized methods and compare different calling techniques (interop, mixed, managed and NetAsm).

The **main results are:**

- **For calling overhead, NetAsm can perform as fast as pure managed calls** and **is two times faster than fast interop** (no security checks).
- In the case of using optimized instructions not available in .NET (like SSE2 in the matrix benchmark), **NetAsm was able to be 50-60% faster than managed code**.

Although, be aware that micro-benchmarks are always subject to issues and should not be considered as a proof for any other benchmarks. All the benchmarks here use a warm-up loop to avoid the cost of any compilation effects at startup and run the test several times.

3.1 Simple Add benchmark

This benchmark consists in measuring the processing time for adding 2 integers. This benchmark measure the performance between several implementations:

- **Managed** and **Managed Inline** : we test both managed with NoInline (with the attribute `[MethodImpl(MethodImplOptions.NoInlining)])` and the default managed code inline.
- **Interop** and **Interop NoSecurity** : we test default Interop technique and Interop with no security attribute flag (`[SuppressUnmanagedCodeSecurity]`)
- **Mixed C++/CLI** : we test the use of an external Mixed C++/CLI using native code.
- **NetAsm** : we use NetAsm with a static native code injection.

For example, the implementation of the Managed Inline is like:

```
        public int ManagedAddInlined(int x, int y)
        {
            return x + y;
        }
```

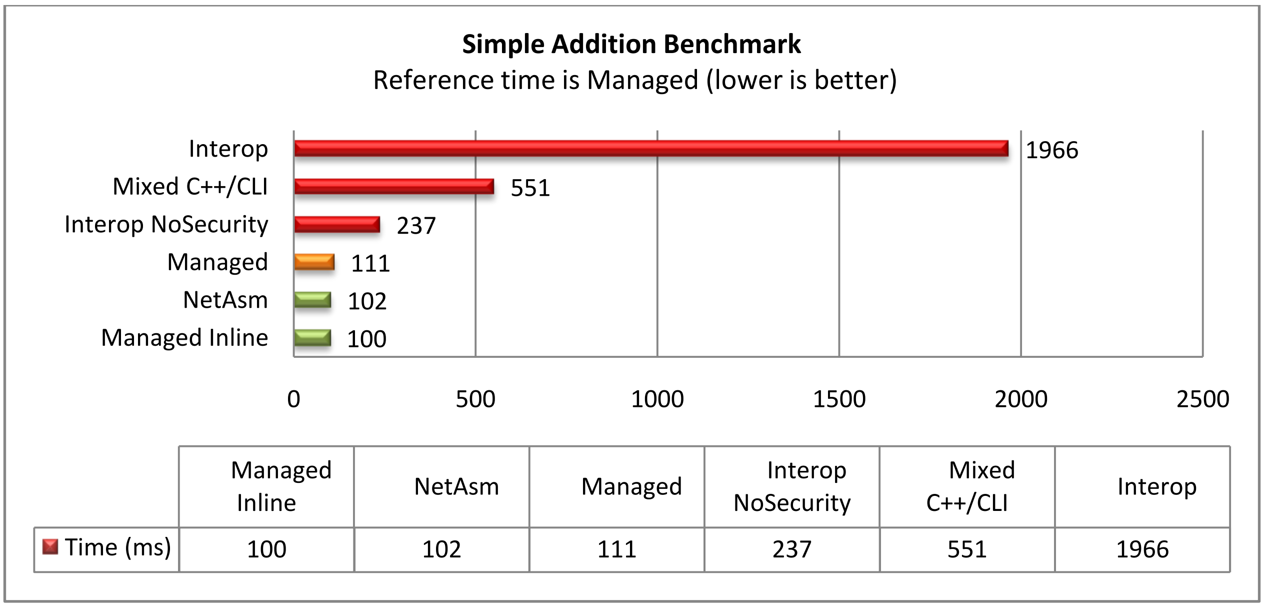
The NetAsm code of the Add method is:

```
[CodeInjection(new byte[] { 0x8b, 0x44, 0x24, 0x04, 0x03, 0xc2, 0xc2, 0x04, 0x00 }),
MethodImpl(MethodImplOptions.NoInlining)]
public int NetAsmAdd(int x, int y)
{
    // This method is compiled using the native code from the CodeInjection Attribute
    // The IL code of this method is never compiled by the JIT
    // ADD +1 to check that this method is not used.
    return x + y + 1;
}
```

It was generated from the C method and the assembler code was copied from the *.cod output assembler listing files generated by Microsoft visual C++ :

```
extern "C" int __fastcall NetAsmAddInC(void* pThis, int x, int y) {
       return x + y;
}
// The generated native code of this method is :
  00000 8b 44 24 04      mov     eax, DWORD PTR _y$[esp-4]
  00004 03 c2            add     eax, edx
  00006 c2 04 00         ret     4
```

Results are indexed on a 100 time based. The default base is managed inline. Lower is better.



	Managed Inline	NetAsm	Managed	Interop NoSecurity	Mixed C++/CLI	Interop
🟥 Time (ms)	100	102	111	237	551	1966

As we can see, **NetAsm performs almost as fast as Managed Inline** and is **twice faster than the Interop** (with no security checks).

This result was expected, as NetAsm native code is considered by the CLR VM as pure managed code compiled by the JIT. Therefore, the performance should be the same than managed code.

See `TestSimpleAddBenchmark.cs` in NetAsmDemo for the code of this benchmark.

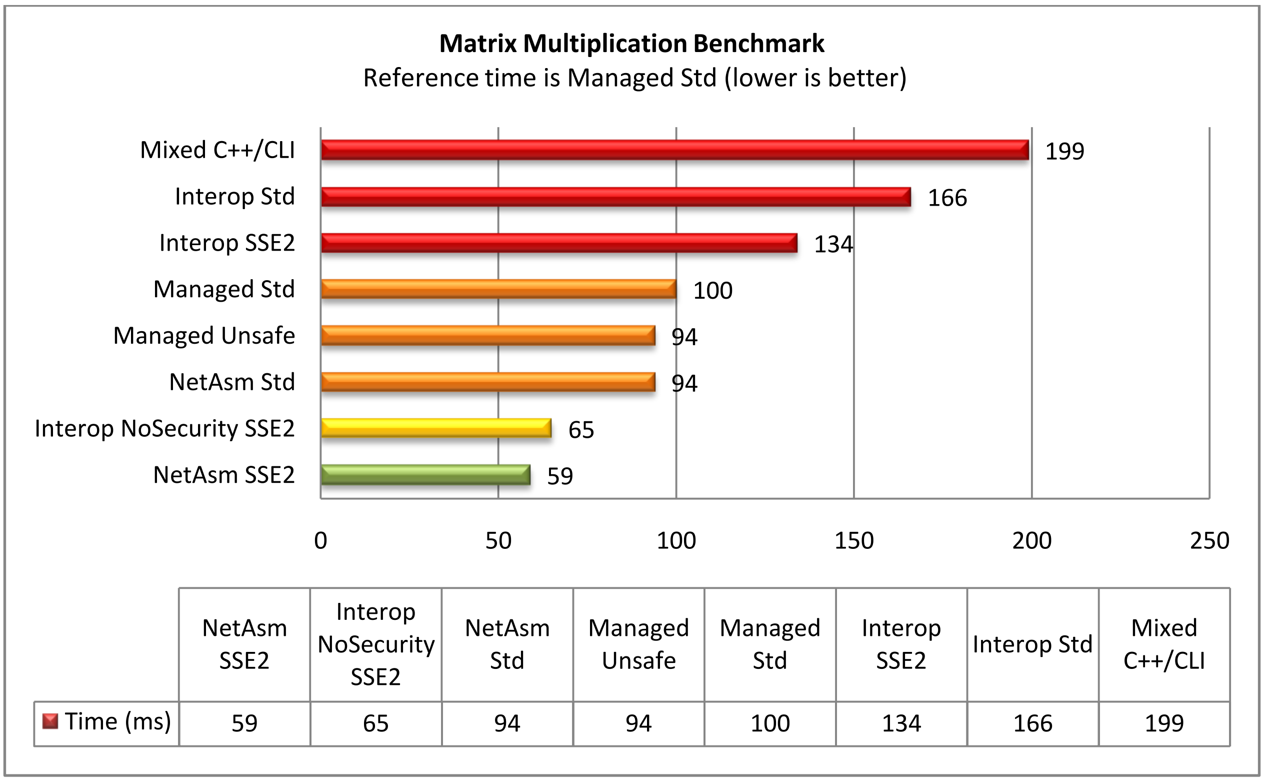
3.2 Matrix Multiplication benchmark using SSE2

This benchmark is based on the work of scapecode in the article Playing with the .NET JIT Part3 : it measures the performance between different matrix multiplications implementations using standard CLR, standard C and SSE2 instructions.

As for SimpleAddBenchmark, this benchmark compares different implementations:

- **Managed Std** and **Managed Unsafe**: we test both standard managed code and unsafe managed code (using pointers on matrix array instead of CLR arrays).
- **Interop Std** and **Interop SSE2, and Interop SSE2 NoSecurity** : we test default Interop technique with a C implementation (without using any SSE2 instructions), and two interop using SSE2 (with one using the no security attribute flag `[SuppressUnmanagedCodeSecurity]`)
- **Mixed SSE2 C++/CLI** : we test the use of an external Mixed C++/CLI using native code.
- **NetAsm Std and NetAsm SSE2**: we use NetAsm with a standard C matrix multiplication and a SSE2 implementations.

Results are indexed on a 100 time based. The default base is managed inline. Lower is better.



	NetAsm SSE2	Interop NoSecurity SSE2	NetAsm Std	Managed Unsafe	Managed Std	Interop SSE2	Interop Std	Mixed C++/CLI
🟥 Time (ms)	59	65	94	94	100	134	166	199

There are several remarks concerning the results:

- First, **NetAsm SSE2 outperforms any other implementations**, ranging from 10% to 300% in speed gain (70% faster than the default managed code).
- **Managed Unsafe and NetAsm unsafe are equivalent**: it means that the default JIT compiler is performing very well in optimizing the C# code (as fast as C code). Consequently, you should always test if it is relevant to replace managed code with c code, as the JIT compiler can generate a fast native code.
- **NetAsm SSE2 is only 10% faster than Interop SSE2**: this result is different from SimpleAddBenchmark. The reason is the cost of calling the interop code is negligible compare to the time used for computing the result.

See `TestMatrixMulBenchmark.cs`, NetAsmDemoCLib.dll and NetAsmDemoMixedLib.dll in NetAsmDemo for the code of this benchmark.

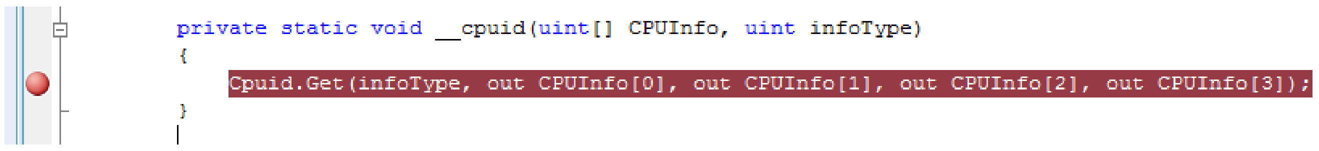
4 DEBUGGING

NetAsm supports debugging of native code injection with CLRCall conventions, although static and dynamic code injections are only supported. DLL code injection is not supported: this is due to the fact that to be able to debug injected code, native code should run from an execution block allocated by the CLR (the DLL function is not allocated by the CLR but is loaded via the LoadLibrary functions from the operating system).

Moreover, the code injected is considered as CLR code, and cannot be debugged with standard interrupt breakpoints (in C DebugBreak… etc.). The only way to achieve the debugging of external DLL should be to leave the CLR runtime and inform the transition to an unmanaged environment, but this is currently impossible due to an API unavailable inside the jit (IHostTaskManager EnterRuntime/LeaveRuntime). For debugging external DLLs, using interop is recommended.

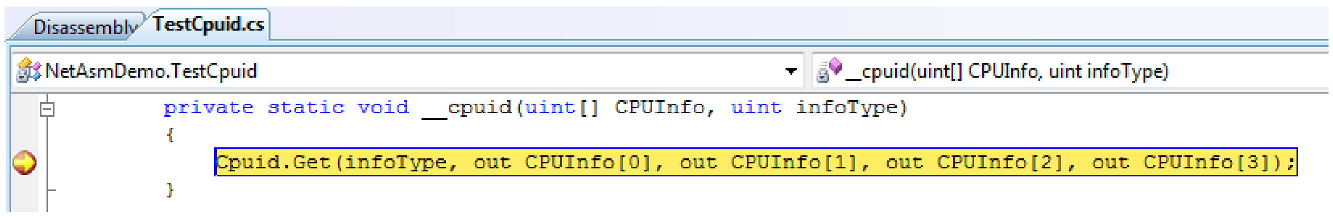
To debug NetAsm native code, you need to set a breakpoint while calling the method to debug into. Breakpoints should not be inside the NetAsm method code injected (this is not working).

If we look at the TestGuid example in NetAsmDemo, to debug the Guid.Get method, we need to set a breakpoint on the calling site:



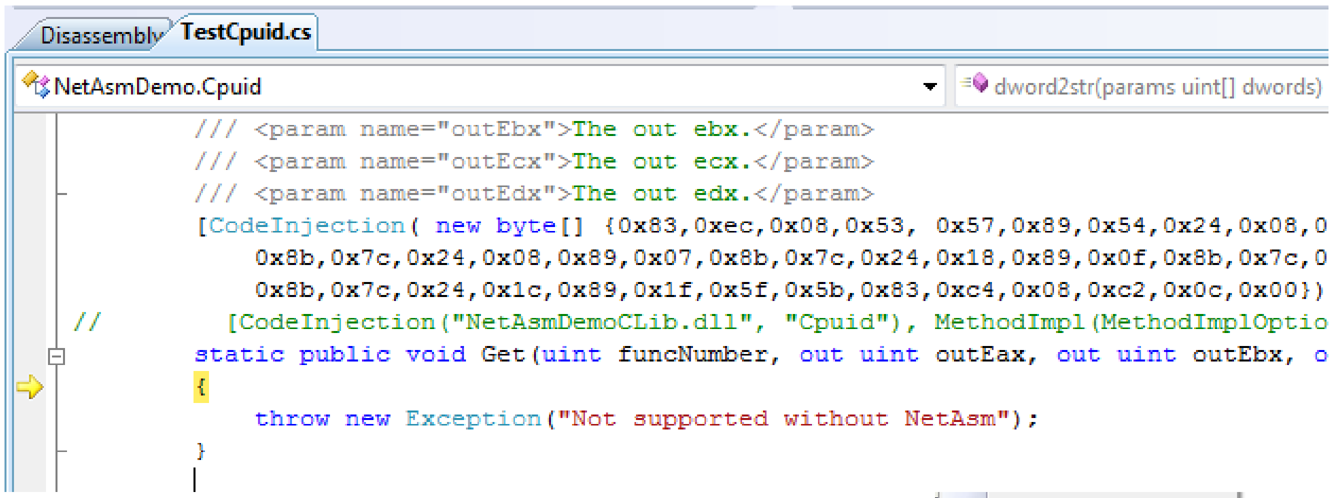
```
        private static void __cpuid(uint[] CPUInfo, uint infoType)
        {
            Cpuid.Get(infoType, out CPUInfo[0], out CPUInfo[1], out CPUInfo[2], out CPUInfo[3]);
        }
```

When you run the program in debug mode, you should have a hit on your breakpoint:



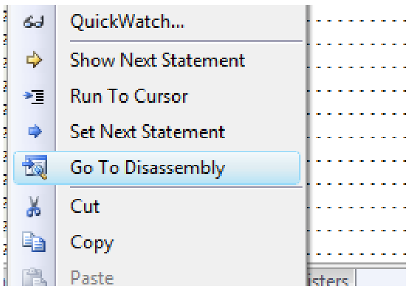
```
        private static void __cpuid(uint[] CPUInfo, uint infoType)
        {
            Cpuid.Get(infoType, out CPUInfo[0], out CPUInfo[1], out CPUInfo[2], out CPUInfo[3]);
        }
```

You go inside the Cpuid.Get method, you should then **press Step+Into [F11] :**

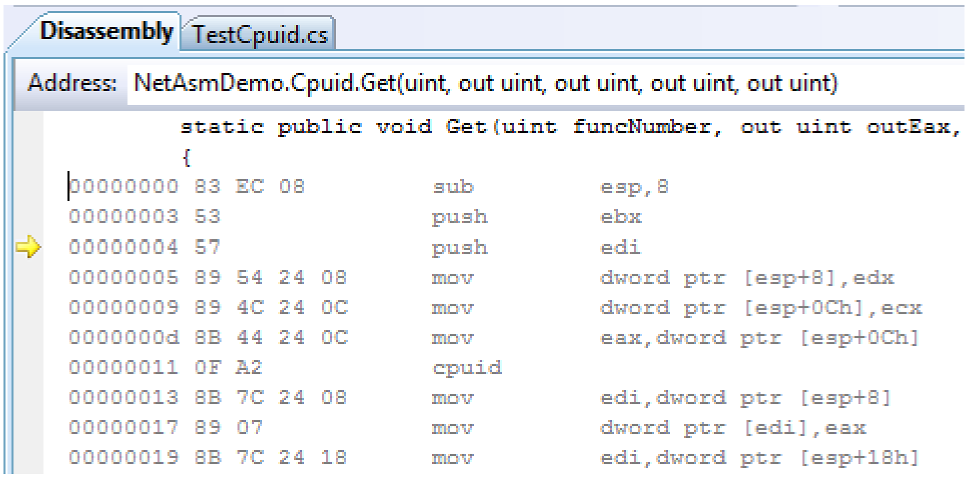


```
        /// <param name="outEbx">The out ebx.</param>
        /// <param name="outEcx">The out ecx.</param>
        /// <param name="outEdx">The out edx.</param>
        [CodeInjection( new byte[] {0x83,0xec,0x08,0x53, 0x57,0x89,0x54,0x24,0x08,0
            0x8b,0x7c,0x24,0x08,0x89,0x07,0x8b,0x7c,0x24,0x18,0x89,0x0f,0x8b,0x7c,0
            0x8b,0x7c,0x24,0x1c,0x89,0x1f,0x5f,0x5b,0x83,0xc4,0x08,0xc2,0x0c,0x00})
  //        [CodeInjection("NetAsmDemoCLib.dll", "Cpuid"), MethodImpl(MethodImplOptio
        static public void Get(uint funcNumber, out uint outEax, out uint outEbx, o
        {
            throw new Exception("Not supported without NetAsm");
        }
```

As you can see, we are now inside the NetAsm native code method, BUT the CLR code is not used. We need then to **switch to the Disassembly Window** to get the native assembler code. Use the popup menu (right button click on the method to force to go to the disassembly window and update the assembler code).



You don't have access to parameter's variable but you can look at the register windows.


Disassembly
TestCpuid.cs
Address: NetAsmDemo.Cpuid.Get(uint, out uint, out uint, out uint, out uint)

```
            static public void Get(uint funcNumber, out uint outEax,
            {
00000000 83 EC 08          sub         esp,8
00000003 53                push        ebx
00000004 57                push        edi
00000005 89 54 24 08       mov         dword ptr [esp+8],edx
00000009 89 4C 24 0C       mov         dword ptr [esp+0Ch],ecx
0000000d 8B 44 24 0C       mov         eax,dword ptr [esp+0Ch]
00000011 0F A2             cpuid
00000013 8B 7C 24 08       mov         edi,dword ptr [esp+8]
00000017 89 07             mov         dword ptr [edi],eax
00000019 8B 7C 24 18       mov         edi,dword ptr [esp+18h]
```

As you can see, the native code is not related to any IL code and is directly disassembled from the native code provided by the NetAsm static native code injection (see previous snapshot : 0x83, 0xEC, 0x08….etc.).