

1 Native C Code in C# Umgebung mit Visual Studio 2010

1.1 Ziel

Ein Embedded C Code soll innerhalb einer C# Umgebung lauf- und debuggfähig sein. Der C# Anteil stellt dem Embedded C Code einen Rahmen zur Verfügung, damit der Embedded C Code möglichst ohne Änderungen innerhalb der C# Visual Studio lauffähig ist.

Bei der Einbindung des Embedded Codes sollte beachtet werden, dass z.B. hardwarenahe Methoden oder Methoden mit Betriebssystemaufrufe nicht geeignet sind. Prädestiniert sind Module oder Methoden, welche eine klare und einfache Schnittstelle haben.

Die Einbindung ermöglicht u.a. Automatisierte Modultests und komfortables Debuggen.

Dieses Dokument beschreibt nicht eine komplett fertige Lösung. Vielmehr gibt es Hilfestellung wie eine Visual Studio Solution angelegt werden kann, welche grundlegend dafür geeignet ist aus einem C# Projekt Native C Code debuggbar aufzurufen. Die Implementierung der zum Ausführen des Embedded C-Codes notwendigen Umgebung ist nicht Bestandteil dieser Beschreibung.

1.2 Umgebung

Windows 7

Visual Studio 2010 Professional

1.3 Aufbau

Die Visual Studio 2010 Solution enthält ein C/C++ Projekt und ein C# Projekt.

Im C/C++ Projekt wird der Embedded C Code platziert, außerdem ein C++/CLI Wrapper um die Kommunikation zwischen C und C# zu ermöglichen.

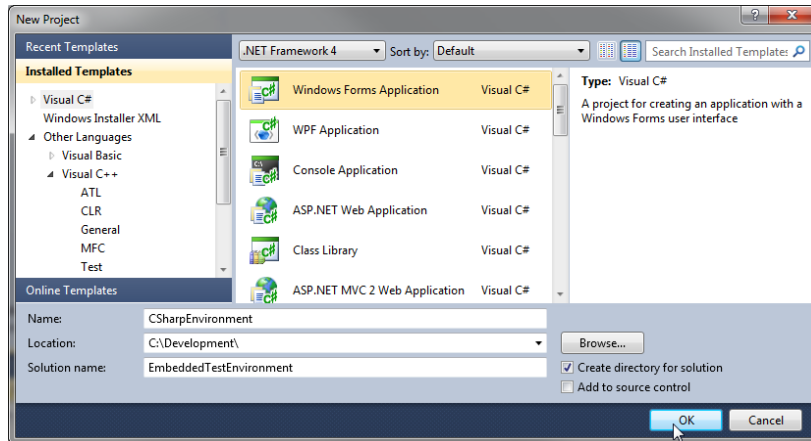
Das C# Projekt sorgt zum Einen für den Aufruf der Methoden im C Code. Zum Anderen kann mit dem C# Code die Ergebnisse entgegengenommen, bewertet und visualisiert werden. Auch eine einfache Protokollierung (TXT, XML, ...), ist über die Funktionen in C# einfacher als direkt im C Code.

1.4 Anlegen der Visual StudioSolution

Im Folgenden wird das Erstellen der Visual Studio Solution beschrieben.

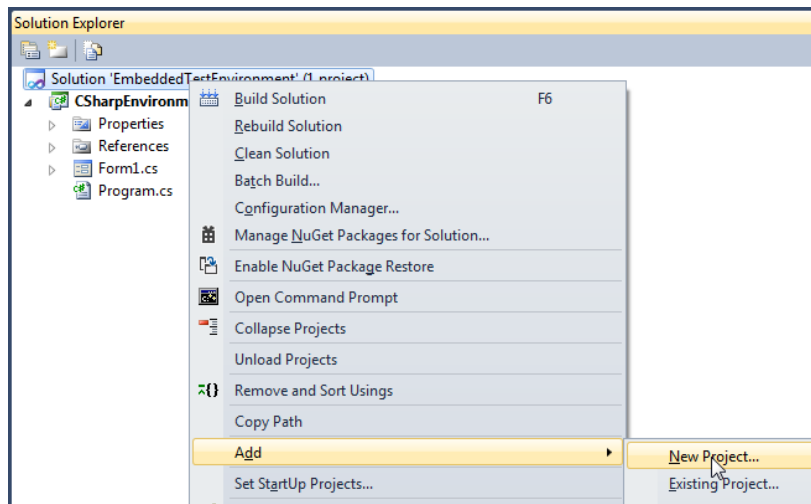
1.4.1 Projekte anlegen

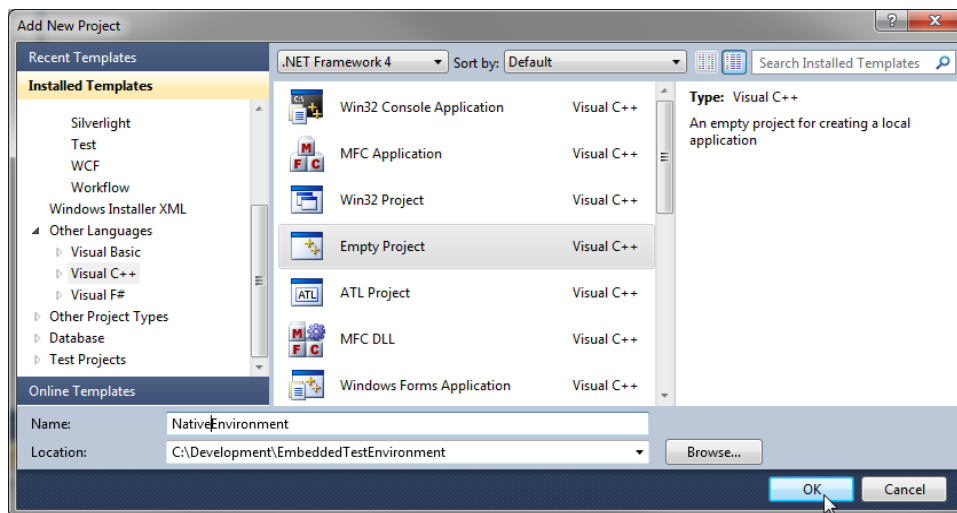
Visual Studio 2010 starten und neues Projekt mit dem Template „Visual C# - Windows Forms Application“ anlegen:



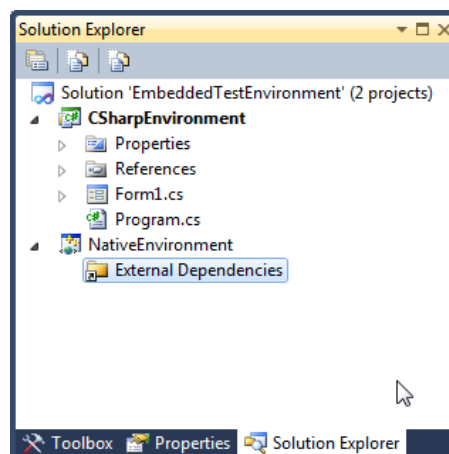
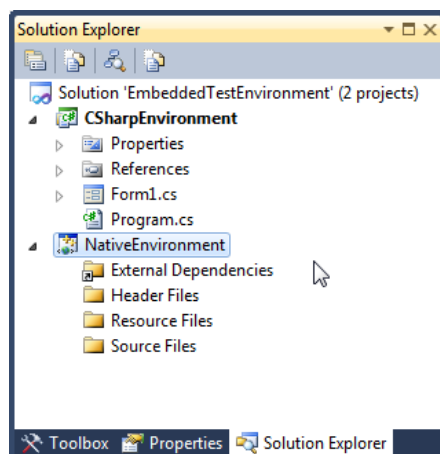
Im Solution Explorer ein weiteres Projekt anlegen.

In diesem Fall mit dem Template „Visual C++ - Empty Project“:





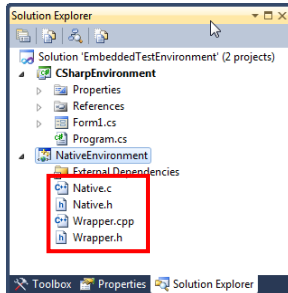
Optional: Da nur wenige Files ins NativeEnvironment Projekt gelegt werden, können im Solution Explorer die Ordner „Header Files“, „Resource Files“, „SourceFiles“ im C++ Projekt gelöscht werden:



1.4.2 C/C++ Source Code Files anlegen

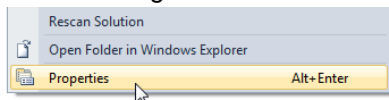
Im NativeEnvironment (C/C++) Projekt vier neue Dateien anlegen.

Darauf achten, dass die Dateierendung für die Native C Datei die Dateierendung „.c“ hat:

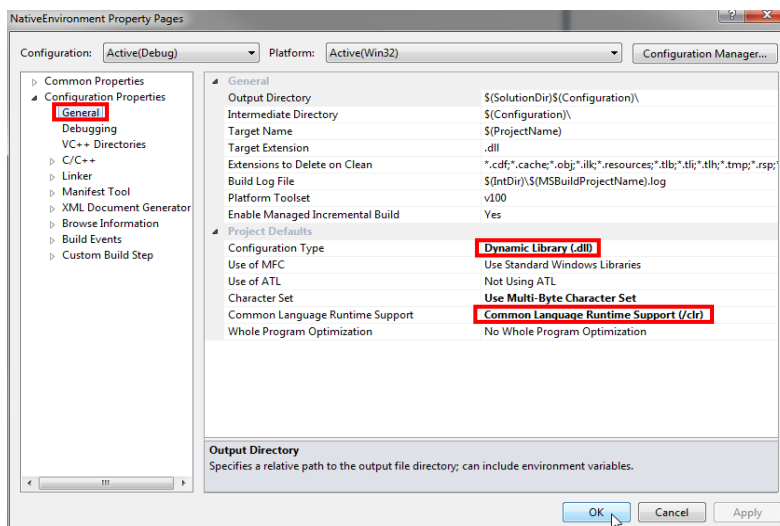


1.4.3 NativeEnvironment Projekt Eigenschaften einstellen

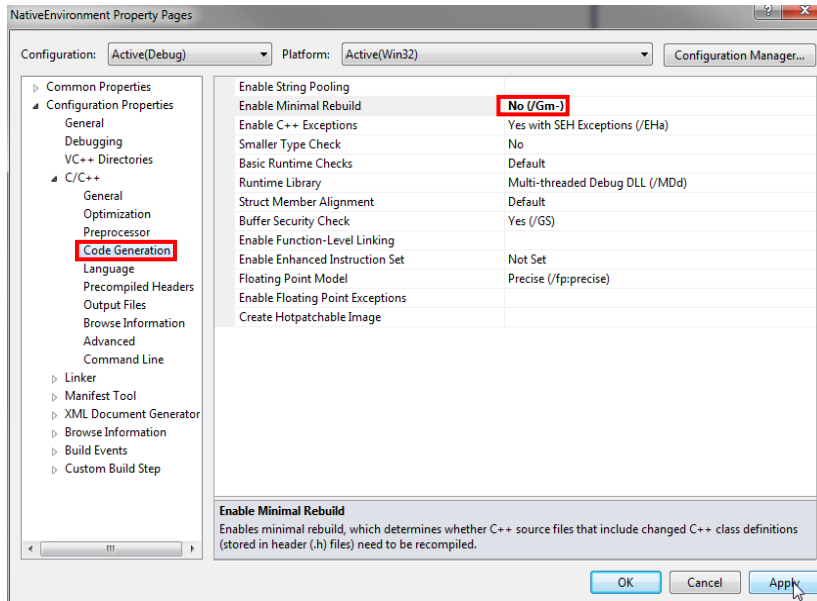
Die Eigenschaften (rechter Mausklick/Properties) des NativeEnvironment (C/C++) Projekts entsprechend der Abbildungen einstellen.



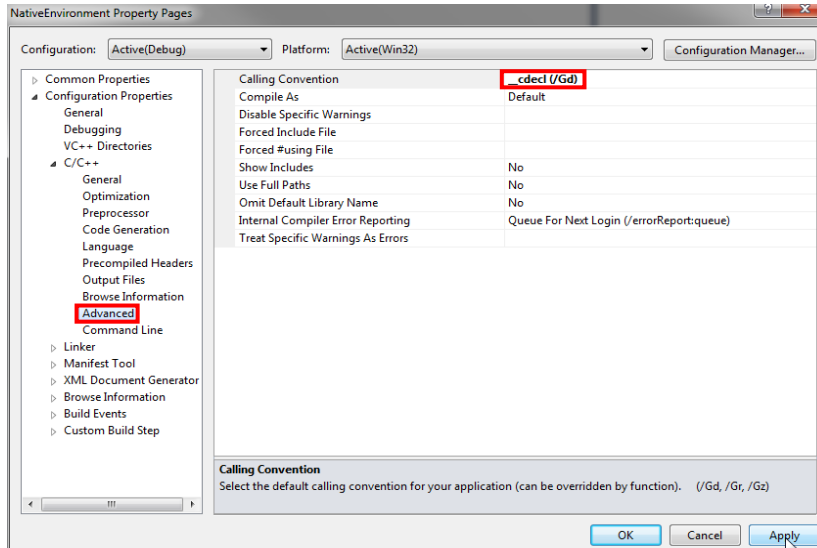
- **Dynamic Library (.dll)**
Aus dem Native C Code und dem C++ Wrapper wird eine DLL erzeugt.
- **Common Language Runtime Support (/clr)**
Da es möglich sein soll über den C++ Wrapper auch Methoden im C# Code aufzurufen muss die die Common Language Runtime gewählt werden.



- Option „Common Language Runtime Support“ und „Enable Minimal Rebuild“ ist nicht kompatibel.



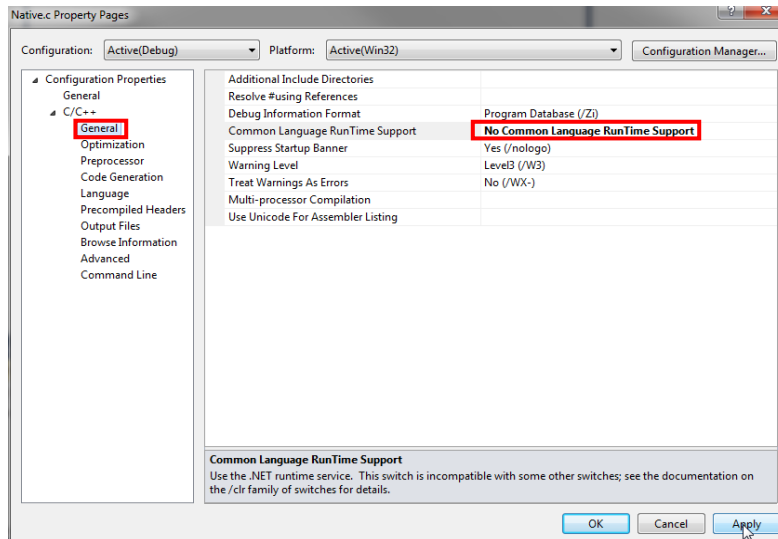
- Als „Calling Convention“ wird die Technologie bezeichnet, mit der ein Methodenaufwurf stattfindet. Diese muss mit der Dll-Import Deklaration im C# Code übereinstimmen. Siehe: <http://de.wikipedia.org/wiki/Aufrufkonvention>



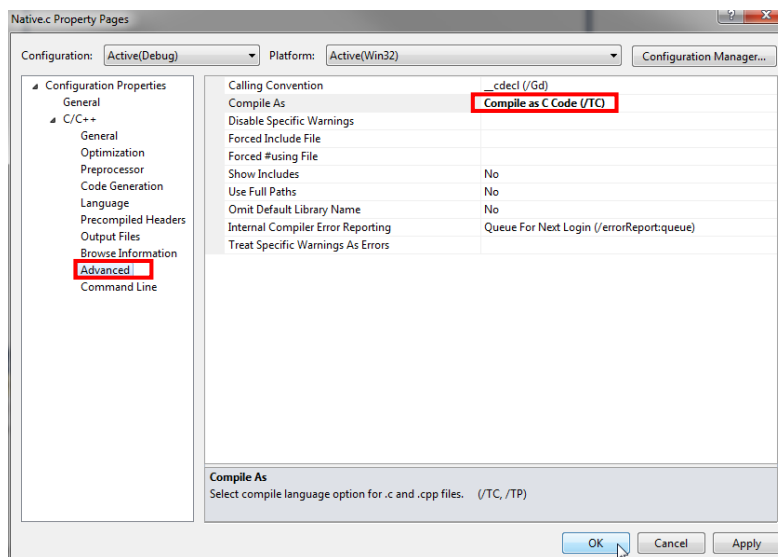
1.4.4 Native C-File Eigenschaften einstellen

Da im NativeEnvironment (C/C++) Projekt neben den C++ Wrapper Files auch das Native C File liegt müssen für dieses File die Eigenschaften getrennt entsprechend der folgenden Abbildungen angepasst werden.

- Der Native C Code soll natürlich ohne „Common Language Runtime“ kompiliert werden:



- Und natürlich auch als C und nicht als C++:



1.4.5 Native.c Datei Inhalt einfügen

In die leere Native.c Datei wird nun folgender Sourcecode eingefügt:

```
// Includes:
#include "Native.h"
#include "Wrapper.h"

// This Function will be called direct from CSharpEnvironment:
unsigned int DoSomethingInNativeC(unsigned int FooValue)
{
    return FooValue + 1;
}

// This Function will be called over C++/CLI Wrapper:
unsigned int DoSomeOthersInNativeC(unsigned int FooValue)
{
    CallManagedCode(FooValue + 1);
    return FooValue + 1;
}
```

1.4.6 Native.h Datei Inhalt einfügen

In die leere Native.h Datei wird nun folgender Sourcecode eingefügt:

```
// Methode declaration with CLL Export for direct call from CSharpEnvironment:
__declspec(dllexport) unsigned int DoSomethingInNativeC(unsigned int FooValue);

// Methode declaration without DLL Export but for use in C++ Wrapper:
#ifdef __cplusplus
extern "C"
#endif
unsigned int DoSomeOthersInNativeC(unsigned int FooValue);
```

1.4.7 Wrapper.cpp Datei Inhalt einfügen

In die leere Wrapper.cpp Datei wird nun folgender Sourcecode eingefügt:

```
// Usings:
using namespace CSharpEnvironment; // Needed to call methodes in CSharpEnvironment:

// Includes:
#include "Native.h"
#include "Wrapper.h"

// This methode is only called from CSharpEnvironment,
// it does not touch the Native C Code.
unsigned int DoSomethingInCpp(unsigned int FooValue)
{
    return FooValue + 1;
}

// This methode is called from CSharpEnvironment and call a methode in Native C.
// This is the typically wrapper C#->Native functionality.
unsigned int CallCSharpViaNative(unsigned int FooValue)
{
    unsigned int ReturnValue;
    ReturnValue = DoSomeOthersInNativeC(FooValue + 1);
    return ReturnValue + 1;
}

// This methode is called from Native C-Code and call a methode in CSharp Environment.
// This is the typically wrapper Native->C# functionality.
void CallManagedCode(unsigned int FooValue)
{
    CSharpClass::DoSomethingInCSharp(FooValue + 1);
}
```

1.4.8 Wrapper.h Datei Inhalt einfügen

In die leere Wrapper.h Datei wird nun folgender Sourcecode eingefügt:

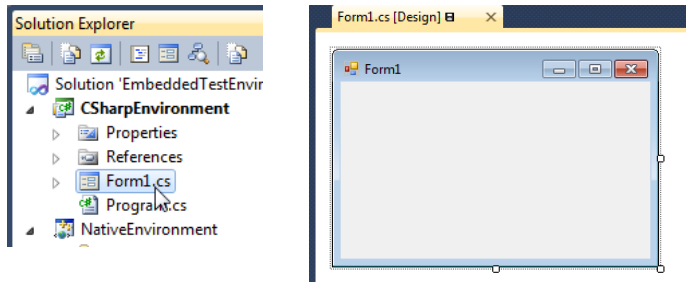
```
// Declaration without DLL Export:
#ifdef __cplusplus
extern "C"
#endif
void CallManagedCode(unsigned int FooValue);

// Declaration with DLL Export:
#ifdef __cplusplus
extern "C"
#endif
__declspec(dllexport) unsigned int DoSomethingInCpp(unsigned int FooValue);

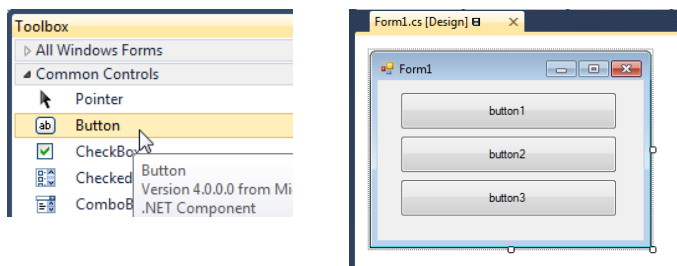
// Declaration with DLL Export:
#ifdef __cplusplus
extern "C"
#endif
__declspec(dllexport) unsigned int CallCSharpViaNative(unsigned int FooValue);
```


1.4.9 Buttons auf GUI platzieren

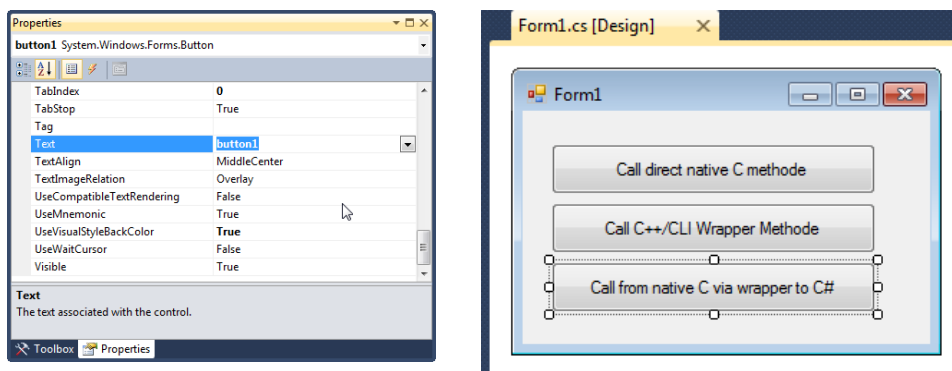
- Im Solution Explorer einen Doppelklick auf Form1.cs ausführen, es öffnet sich die leere Form:



- Drei Buttons auf der Form mit Hilfe der Toolbox platzieren:



- Buttonbeschriftung über die Properties anpassen:



- Idealerweise sollte auch der Titel der Form, der Name der Form und die Namen Buttons angepasst werden. In diesem Beispiel lassen wir es bei den vom Visual Studio vorgegeben nichtsagenden Namen.

1.4.10 Event Routinen für Button Klick anlegen

Jeweils einen Doppelklick auf jeden Button ausführen. Hiermit werden die Event Methoden in der Datei Form1.cs angelegt:

```
private void button1_Click(object sender, EventArgs e)
{
}

private void button2_Click(object sender, EventArgs e)
{
}

private void button3_Click(object sender, EventArgs e)
{
}
```

1.4.11 Inhalt der Event Routinen einfügen

In die drei Event Methoden wird folgender Inhalt eingefügt:

```
private void button1_Click(object sender, EventArgs e)
{
    UInt32 MyFooValue = 20;
    UInt32 MyReturnValue;
    MyReturnValue = DoSomethingInNativeC(MyFooValue);
    Debug.WriteLine("Return value of DoSomethingInNativeC = " + MyReturnValue.ToString());
}

private void button2_Click(object sender, EventArgs e)
{
    UInt32 MyFooValue = 10;
    UInt32 MyReturnValue;
    MyReturnValue = DoSomethingInCpp(MyFooValue);
    Debug.WriteLine("Return value of DoSomethingInCpp = " + MyReturnValue.ToString());
}

private void button3_Click(object sender, EventArgs e)
{
    UInt32 MyFooValue = 30;
    UInt32 MyReturnValue;
    MyReturnValue = CallCSharpViaNative(MyFooValue);
    Debug.WriteLine("Return value of DoSomethingInNativeC = " + MyReturnValue.ToString());
}
```

1.4.12 Vollständige Form1.cs Datei

In die Form1.cs Datei muss zusätzlich noch ein Using für die InteropServices und ein Using für Diagnostics eingefügt. Desweiteren wird die Deklaration der Methoden in der DLL benötigt. Außerdem fehlt noch die Klasse CSharpClass. Die noch hinzuzufügende Anteile sind im folgenden hervorgehoben:

```
using System;
using System.Windows.Forms;

using System.Runtime.InteropServices; // Needed for DllImport
using System.Diagnostics; // Used for Debug.WriteLine

namespace CSharpEnvironment
{
    public partial class Form1 : Form
    {
        [DllImport("NativeEnvironment.dll", EntryPoint = "DoSomethingInNativeC",
            CallingConvention = CallingConvention.Cdecl, CharSet = CharSet.Auto)]
        public static extern UInt32 DoSomethingInNativeC(UInt32 FooValue);

        [DllImport("NativeEnvironment.dll", EntryPoint = "DoSomethingInCpp",
            CallingConvention = CallingConvention.Cdecl, CharSet = CharSet.Auto)]
        public static extern UInt32 DoSomethingInCpp(UInt32 FooValue);

        [DllImport("NativeEnvironment.dll", EntryPoint = "CallCSharpViaNative",
            CallingConvention = CallingConvention.Cdecl, CharSet = CharSet.Auto)]
        public static extern UInt32 CallCSharpViaNative(UInt32 FooValue);

        public Form1()
        {
            InitializeComponent();
        }

        private void button1_Click(object sender, EventArgs e)
        {
            UInt32 MyFooValue = 20;
            UInt32 MyReturnValue;
            MyReturnValue = DoSomethingInNativeC(MyFooValue);
            Debug.WriteLine("Return value of DoSomethingInNativeC = " + MyReturnValue.ToString());
        }

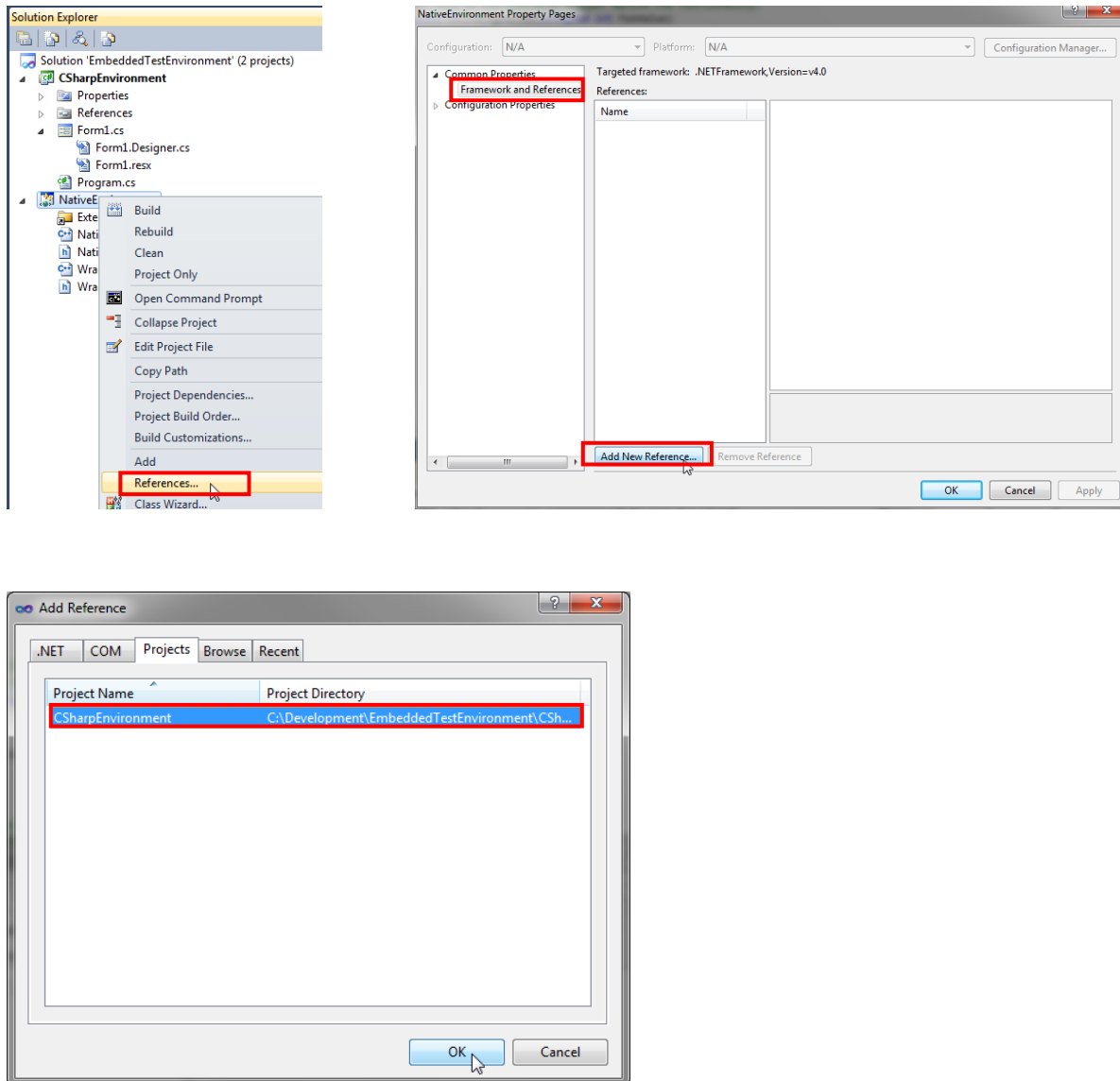
        private void button2_Click(object sender, EventArgs e)
        {
            UInt32 MyFooValue = 10;
            UInt32 MyReturnValue;
            MyReturnValue = DoSomethingInCpp(MyFooValue);
            Debug.WriteLine("Return value of DoSomethingInCpp = " + MyReturnValue.ToString());
        }

        private void button3_Click(object sender, EventArgs e)
        {
            UInt32 MyFooValue = 30;
            UInt32 MyReturnValue;
            MyReturnValue = CallCSharpViaNative(MyFooValue);
            Debug.WriteLine("Return value of DoSomethingInNativeC = " + MyReturnValue.ToString());
        }
    }

    public class CSharpClass
    {
        public static void DoSomethingInCSharp(UInt32 CSharpOutValue)
        {
            Debug.WriteLine("CSharpOutValue = " + CSharpOutValue.ToString());
        }
    }
}
```

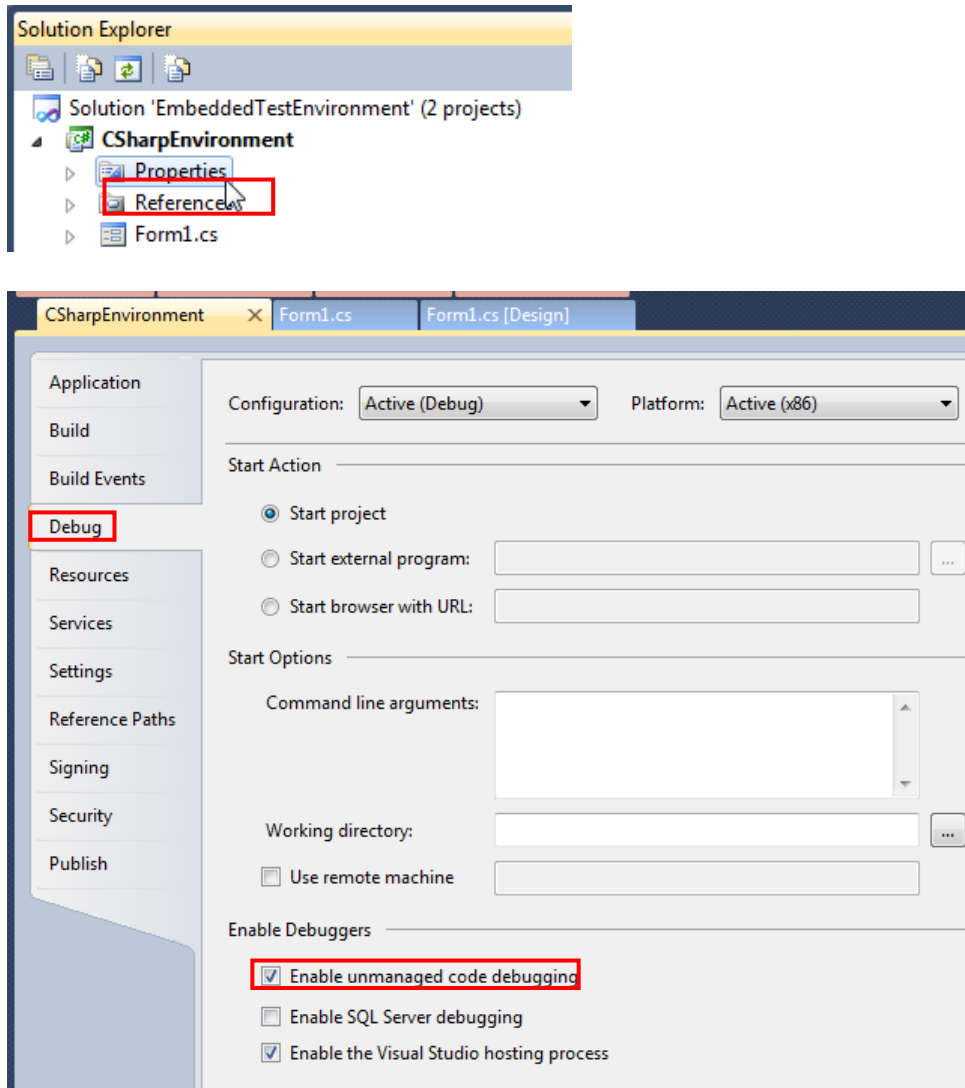
1.4.13 Referenz auf CSharpEnvironment Projekt

Damit das NativeEnvironment Projekt Zugriff auf die Funktion DoSomethingInCSharp bekommt muss eine Referenz erstellt werden:



1.4.14 Debuggen von unmanaged Code aktivieren:

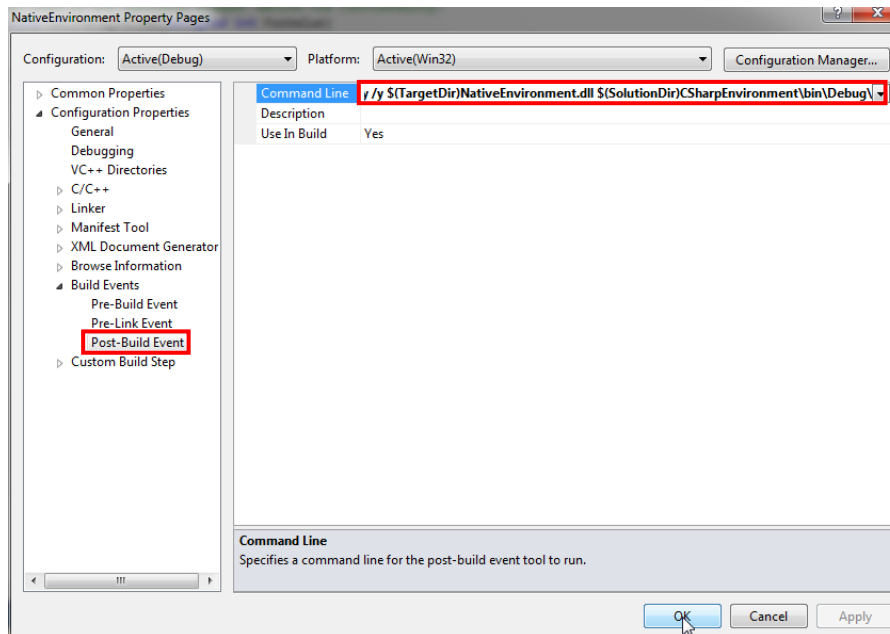
Um den Native C-Code debuggen zu können muss im CSharpEnvironment Projekt die Option „Enable unmanaged code debugging“ aktiviert werden:



1.4.16 Automatisches Kopieren der DLL

Um nicht nach jedem Rebuild die DLL von einem Projekt ins andere manuell kopieren zu müssen fügt man beim NativeEnvironment Projekt im Post-Build Event folgendes ein:

```
xcopy /y $(TargetDir)NativeEnvironment.dll $(SolutionDir)CSharpEnvironment\bin\Debug\
```




Aufgrund der Referenz vom NativeEnvironment Projekt zum CSharpEnvironment Projekt (für den Zugriff auf die C# Methoden) wird von Visual Studio die Buildorder festgelegt:

1. CSharpEnvironment
2. NativeEnvironment


Dies hat zur Folge, dass die DLL erst erstellt wird, wenn bereits das CSharpEnvironment Projekt erzeugt wurde. Daher kann man den Pre-Build Event des CSharpEnvironment Projekts nicht nutzen um die DLL ins CSharpEnvironment Projekt zu kopieren.

1.4.17 Debugging C# -> Native C-Code

Ein einfacher Klick auf den „Start Debugging“ Button  baut bei Bedarf das Projekt und startet es.


Setzt man vor dem ersten Aufruf einer NativeEnvironment.dll Methode einen Breakpoint im Native.c Code oder im Wrapper.c Code so ist dieser Breakpoint nicht aktiv:


```
7
8 // This methode is only called from CSharpEnvironment,
9 // it does not touch the Native C Code.
10 unsigned int DoSomethingInCpp(unsigned int FooValue)
11 {
12     return FooValue + 1;
13 }
14
```





Erst beim ersten Aufruf einer NativeEnvironment.dll Methode wird die DLL in den Speicher geladen und der Breakpoint wird aktiv:

```
9 // it does not touch the Native C Code.
10 unsigned int DoSomethingInCpp(unsigned int FooValue)
11 {
12     return FooValue + 1;
13 }
14
```





Setzt man in der ersten Button Event Methode einen Breakpoint und geht dann per Step Into  im Einzelschritt weiter wird man direkt nach dem Befehl:

```
25
26 private void button1_Click(object sender, EventArgs e)
27 {
28     UInt32 MyFooValue = 20;
29     UInt32 MyReturnValue;
30     MyReturnValue = DoSomethingInNativeC(MyFooValue);
31     Debug.WriteLine("Return value of DoSomethingInNativeC =
32 }
```



Im dazu gehörigen Native.c File landen:

```
4
5 // This Function will be called dir
6 unsigned int DoSomethingInNativeC(u
7 {
8     return FooValue + 1;
9 }
10
```

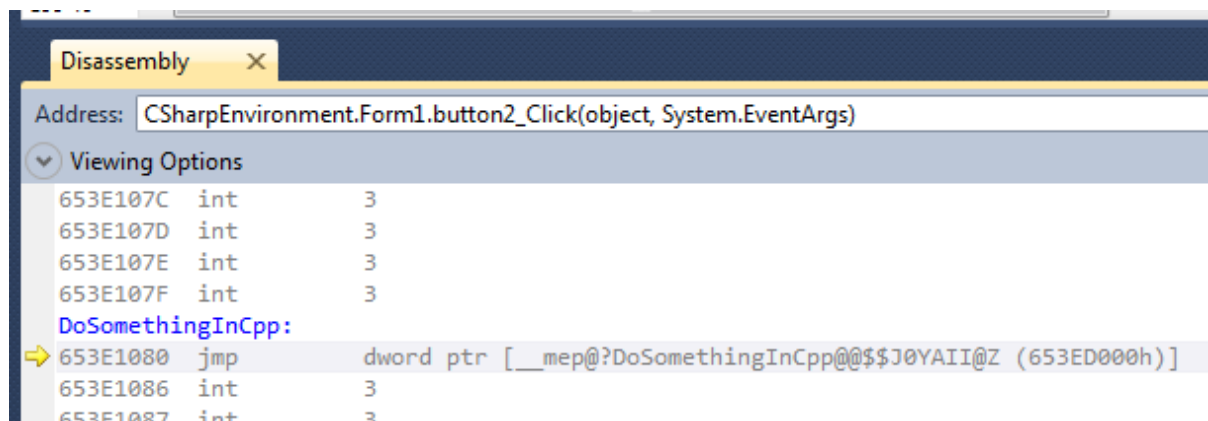


1.4.18 Debugging C# -> C++/CLI-Code

Setzt man hingegen im zweiten Button Event einen Breakpoint und geht dann per „Step Into“ im Einzelschritt weiter wird man direkt nach dem Befehl:


```
34 private void button2_Click(object sender, EventArgs e)
35 {
36     UInt32 MyFooValue = 11;
37     UInt32 MyReturnValue;
38     MyReturnValue = DoSomethingInCpp(MyFooValue);
39     Debug.WriteLine("Return value of DoSomethingInCpp =
```

Nicht direkt im C++ Code landen. Stattdessen öffnet sich das Disassembly Fenster:



Um die aktuelle Programmposition (gelber Pfeil) im Wrapper.cpp Fenster zu sehen, muss man das Fenster manuell öffnen und solange weitergehen, bis der gelbe Pfeil im Code erscheint:

```
9  // it does not touch the Native C Code.
10 unsigned int DoSomethingInCpp(unsigned int FooValue)
11 {
12     return FooValue + 1;
13 }
```

Alternativ setzt man im Wrapper.cpp Code einen Breakpoint und geht nicht im Einzelschritt weiter sondern führt die Codeausführung mit Continue  fort. Der Debugger hält dann automatisch im Wrapper.cpp Code an:

```
10 unsigned int DoSomethingInCpp(
11 {
12     return FooValue + 1;
13 }
```


1.4.19 Debugging C# -> C++/CLI-Code -> Native C und zurück

Der Code im dritten Button Event demonstriert den Aufruf einer Methode im Wrapper, der dann in den Native C Code weitergereicht wird. Hier wird eine Methode im Wrapper aufgerufen, welche dann wiederum eine Methode im C# Code aufruft:

Form1.cs – Button Event:

```
42 private void button3_Click(object sender, EventArgs e)
43 {
44     UInt32 MyFooValue = 30;
45     UInt32 MyReturnValue;
46     MyReturnValue = CallCSharpViaNative(MyFooValue);
47     Debug.WriteLine("Return value of DoSomethingInNativeC: " + MyReturnValue);
48 }
```

Wrapper.cpp:

```
14
15 // This methode is called from CSharpEnvironment and call a method
16 // This is the typically wrapper C#->Native functionality.
17 unsigned int CallCSharpViaNative(unsigned int FooValue)
18 {
19     unsigned int ReturnValue;
20     ReturnValue = DoSomeOthersInNativeC(FooValue + 1);
21     return ReturnValue + 1;
22 }
```

Native.c:

```
11 // This Function will be called over C++/CLI Wrapper:
12 unsigned int DoSomeOthersInNativeC(unsigned int FooValue)
13 {
14     CallManagedCode(FooValue + 1);
15     return FooValue + 1;
16 }
```

Wrapper.cpp:

```
24 // This methode is called from Native C-Code and call a method
25 // This is the typically wrapper Native->C# functionality.
26 void CallManagedCode(unsigned int FooValue)
27 {
28     CSharpClass::DoSomethingInCSharp(FooValue + 2);
29 }
30
```

Form1.cs – CsharpClass:

```
53
54 public class CSharpClass
55 {
56     public static void DoSomethingInCSharp(UInt32 CSharpOutValue)
57     {
58         Debug.WriteLine("CSharpOutValue = " + CSharpOutValue.ToString());
59     }
60 }
61
```

1.4.20 Und nun?

Jetzt darf jeder seine eigene Phantasie freien Lauf lassen. Angefangen vom kleinen automatisierten Test mit Reporting z.B. als XML oder HTML. Bis hin zur kompletten Simulation mit GUI usw. ist alles möglich.