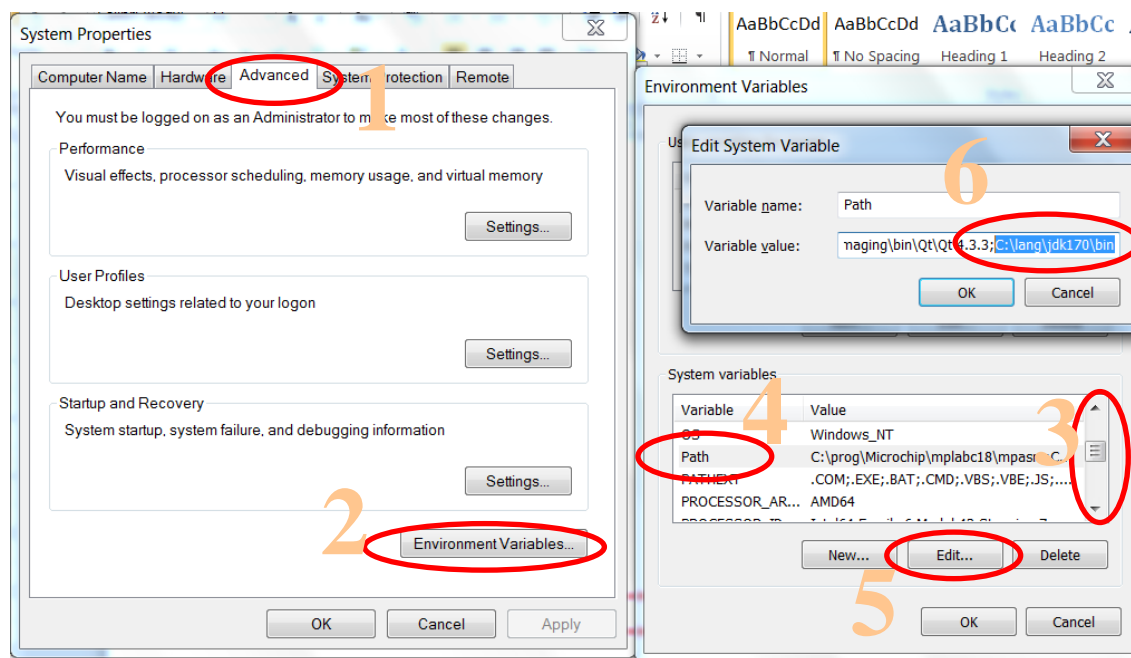


## Annex 2 – Accessing native code

In this document, we describe how to use the Java Native Interface, in order to let your java program access the low-level resources of the hardware or operating system, through Java Native Interface (JNI). As such, we describe the development of a Dynamic link library (DLL), composed of c++ functions, which will be called from your java program. During the steps below, if you copy and past source-code, some characters might not be well translated, yielding syntax errors. For a complementary description of using JNI, see <http://www.javaxt.com/Tutorials/Netbeans/How to Create a JNI with Netbeans>.

### Step 1 – Configuration of your system

Go to “Start menu” -> “Control panel” -> “system” -> “advanced system settings”. Find, select and edit the system variable “Path”, as described in the figure below. In step 6, you must insert a semicolon (;) and add the path to your installation of java, specifically the ‘c:\...jdk170\bin’ folder. For such, you should locate the Java SDK that was installed in your system.



If these steps went well, you should successfully invoke the java compiler as illustrated in the figure below.



```
Command Prompt
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\jrosas>javac
Usage: javac <options> <source files>
where possible options include:
  -g                  Generate all debugging info
  -g:none             Generate no debugging info
  -g:{lines,vars,source}  Generate only some debugging info
  -nowarn             Generate no warnings
  -verbose            Output messages about what the compiler is doing
  -deprecation        Output source locations where deprecated APIs are used
  -classpath <path>   Specify where to find user class files and annotation processors
  -cp <path>          Specify where to find user class files and annotation processors
  -sourcepath <path>  Specify where to find input source files
  -bootclasspath <path> Override location of bootstrap class files
  -extdirs <dirs>     Override location of installed extensions
  -endorseddirs <dirs> Override location of endorsed standards path
  -proc:{none,only}   Control whether annotation processing and/or compilation is done.
  -processor <class1>[,<class2>,<class3>...] Names of the annotation processors to run; bypasses default discovery process
```

Attention, sometimes, when you compile with javac.exe and then you try to execute with java.exe, a mismatch java versions error occurs. If this happens, you should open the folder c:\windows\system32\ and delete both “java.exe” and “javaw.exe”.

## Step 2 – Specifying native methods

In this step, we define the signature of the C++ native methods which will be called from java. We do this by declaring these methods in a java class. This is like declaring a normal (non-native) java method, but adding the native specifier. These methods have no body definition. Go to netbeans and create a new java class called Hardware, in the default package, as follows:

Ficheiro “Hardware.Java”
//if you see a line starting with the word “package”, as follows package trabalho2; //or a similar one, you should delete it and let netbeans perform refactoring.  public class Hardware { native public void create_di(int port); native public void create_do(int port); native public void write_port(int port, int value); native public int read_port(int port); }



Enter into a console window.

Go to the directory, which contains the file Hardware.java

Compile this class with javac: “javac Hardware.java”

The result should be a file “Hardware.class”

Next, invoke the program javah: “javah Hardware”

The result should be a file named “Hardware.h” This is illustrated in the figure below.

After clicking “Save ALL” in netbeans, open a “console window” and go to the “src” folder. Follow the path as illustrated in the figure below, towards your netbeans project folder.

```
Command Prompt
C:\Users\jrosas>cd\
C:\>cd str
C:\str>cd tttrab2
C:\str\tttrab2>cd trab2
C:\str\tttrab2\trab2>cd src
C:\str\tttrab2\trab2\src>dir
Volume in drive C is OS
Volume Serial Number is 008E-DBB0

Directory of C:\str\tttrab2\trab2\src

24-10-2012  18:10    <DIR>          .
24-10-2012  18:10    <DIR>          ..
23-10-2012  18:19             637 Hardware.java
23-10-2012  18:06             408 Hello.java
23-10-2012  16:36             446 Hello_args.java
               3 File(s)             1,491 bytes
               2 Dir(s)  89,468,227,584 bytes free

C:\str\tttrab2\trab2\src>
```

If you use the “dir” command, you should see the file “Hardware.java”. As mentioned before, you should have invoked the commands javac and javah, as described in the figure below.



```
Command Prompt
23-10-2012 18:06          408 Hello.java
23-10-2012 16:36          446 Hello_args.java
          3 File(s)          1,491 bytes
          2 Dir(s) 89,468,227,584 bytes free

C:\str\tttrab2\trab2\src>javac Hardware.java
C:\str\tttrab2\trab2\src>javah Hardware
C:\str\tttrab2\trab2\src>dir
Volume in drive C is OS
Volume Serial Number is 008E-DBB0

Directory of C:\str\tttrab2\trab2\src

24-10-2012 18:13    <DIR>          .
24-10-2012 18:13    <DIR>          ..
24-10-2012 18:13          642 Hardware.class
24-10-2012 18:13          890 Hardware.h
23-10-2012 18:19          637 Hardware.java
23-10-2012 18:06          408 Hello.java
23-10-2012 16:36          446 Hello_args.java
          5 File(s)          3,023 bytes
          2 Dir(s) 89,468,112,896 bytes free

C:\str\tttrab2\trab2\src>
```

The content of file “hardware.h” contain the “C” signatures of the native methods, which must be implemented in a DLL. The file Hardware.h is as follows:

File “Hardware.h”
<pre>/* DO NOT EDIT THIS FILE - it is machine generated */ #include &lt;jni.h&gt; /* Header for class Hardware */  #ifndef _Included_Hardware #define _Included_Hardware #ifdef __cplusplus extern "C" { #endif /*  * Class:   Hardware  * Method:  create_DI  * Signature: (I)V  */ JNIEXPORT void JNICALL Java_Hardware_create_1DI (JNIEnv *, jobject, jint);</pre>



```
/*
 * Class:   Hardware
 * Method:  create_DO
 * Signature: (I)V
 */
JNIEXPORT void JNICALL Java_Hardware_create_1DO
    (JNIEnv *, jobject, jint);

/*
 * Class:   Hardware
 * Method:  write_port
 * Signature: (II)V
 */
JNIEXPORT void JNICALL Java_Hardware_write_1port
    (JNIEnv *, jobject, jint, jint);

/*
 * Class:   Hardware
 * Method:  read_port
 * Signature: (I)I
 */
JNIEXPORT jint JNICALL Java_Hardware_read_1port
    (JNIEnv *, jobject, jint);

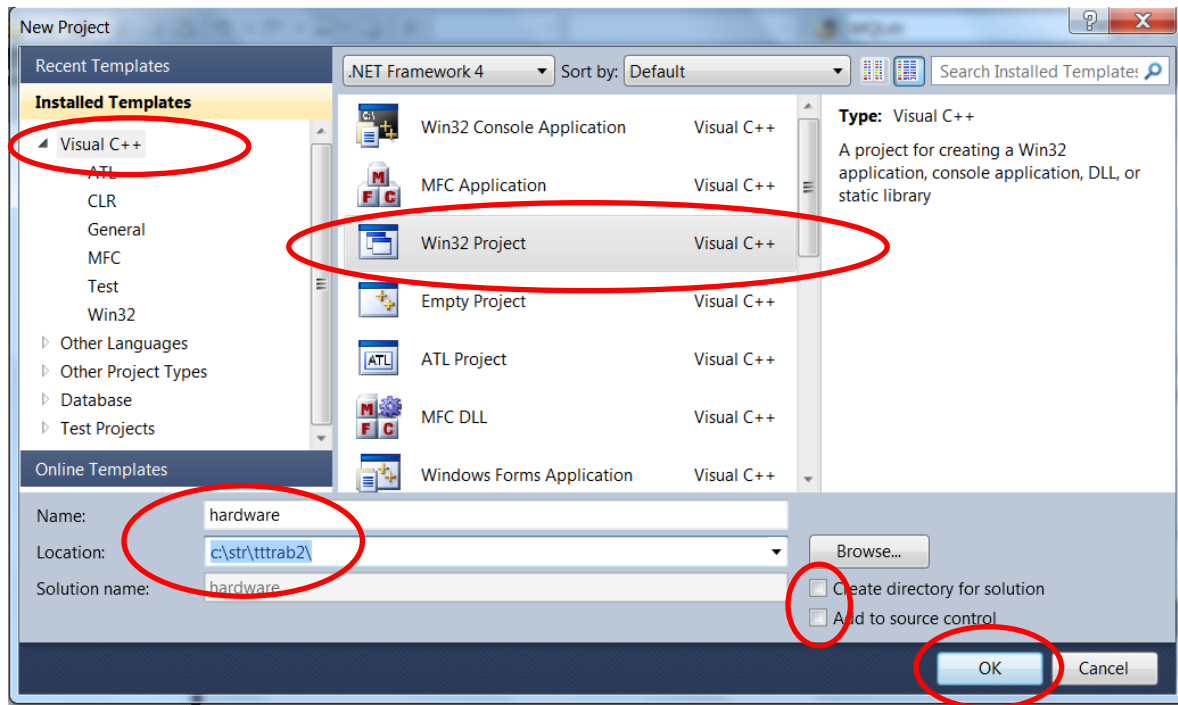
#ifdef __cplusplus
}
#endif
#endif
```

### Step 3 – Creation of a dynamic link library (DLL)

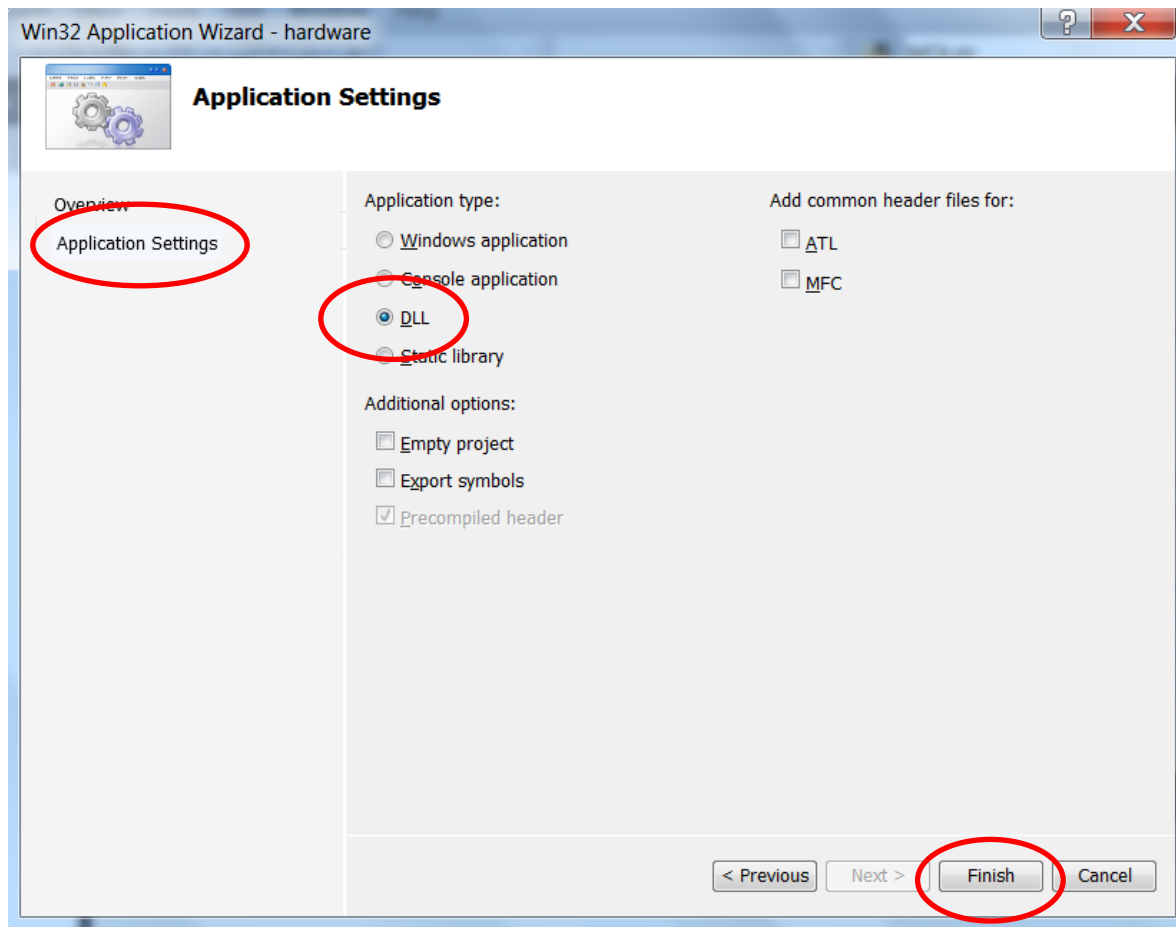
A DLL is a dynamic library (in opposition to a static library with extension lib), which contains a collection of compiled functions that are shared by multiple (and most of the) applications in windows, reducing their size in disk and the amount of RAM when application are running. Therefore, DLLs contribute to increase the performance of the (windows) system. In our case, we need a DLL to implement the native methods, which work at the operating system level. These functions can afterwards be invoked by Java methods, through the JNI.



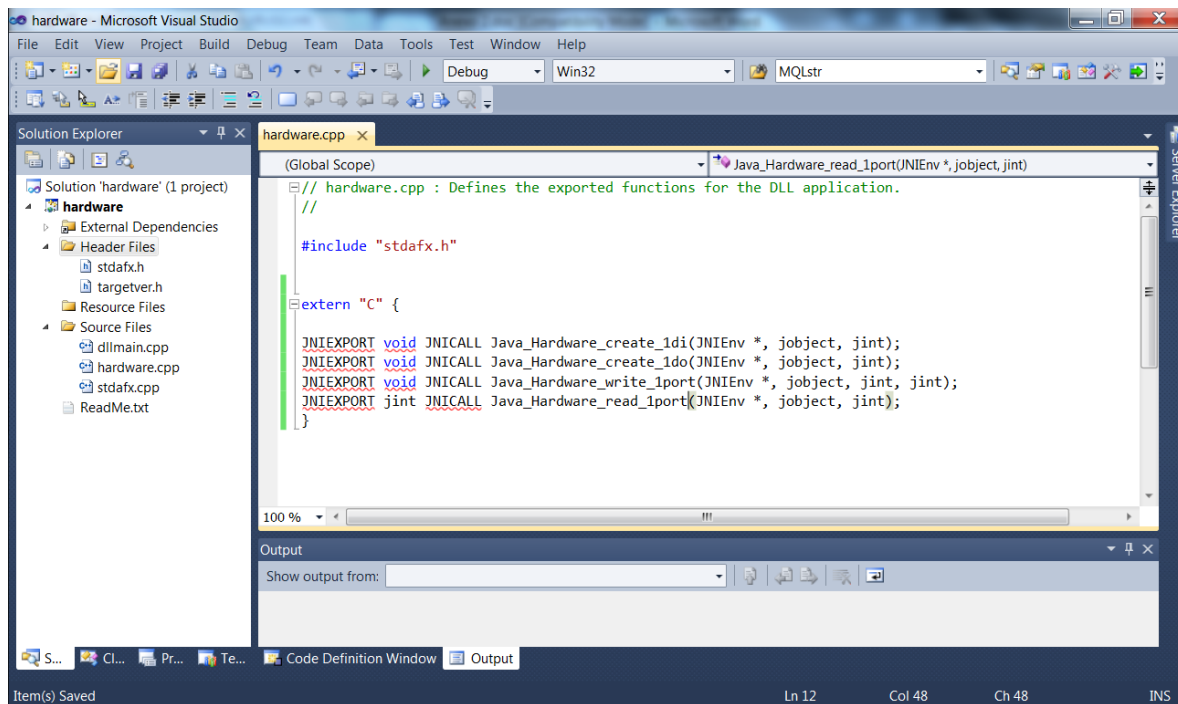
Open the Microsoft visual studio and select File->new->Project. Select “Visual C++” and Win32 Project Fill the fields as illustrated in the next figure:



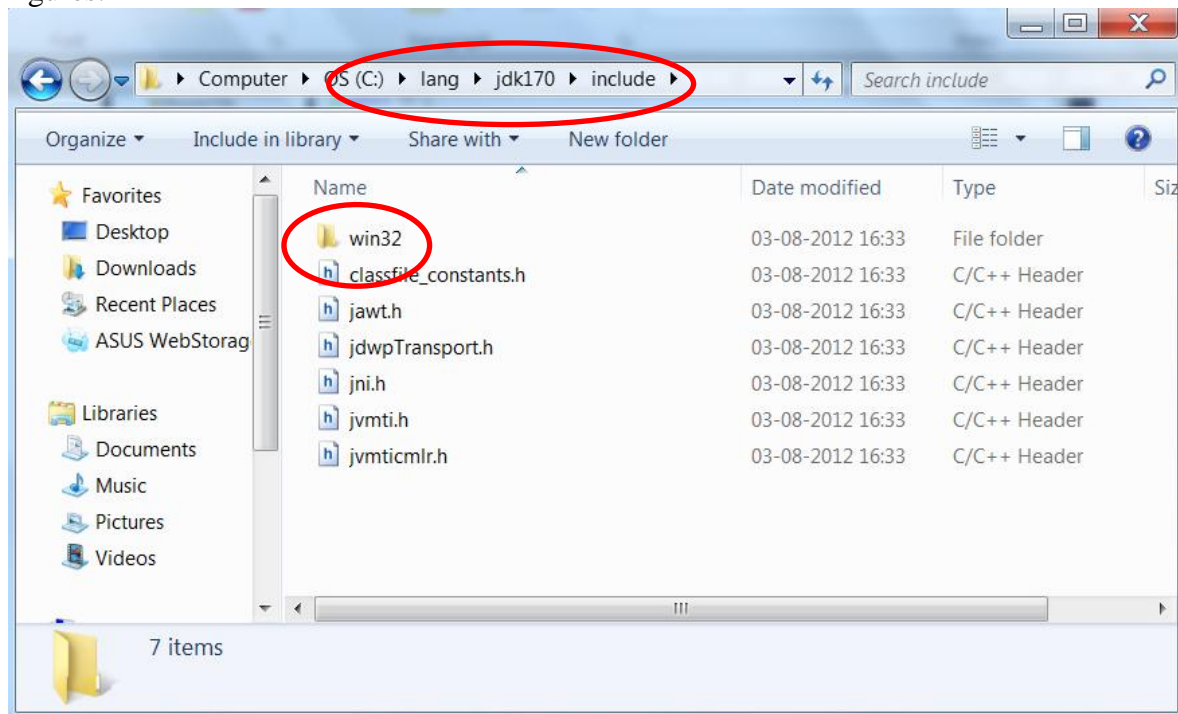
Click in “Application settings” and set the application type as DLL, as illustrated in the next figure:



Remember the file “hardware.h” developed in the previous step? Pick each function in that file and add to hardware.cpp in the visual studio project, as illustrated in the next figure (don’t forget the extern “C” clause):

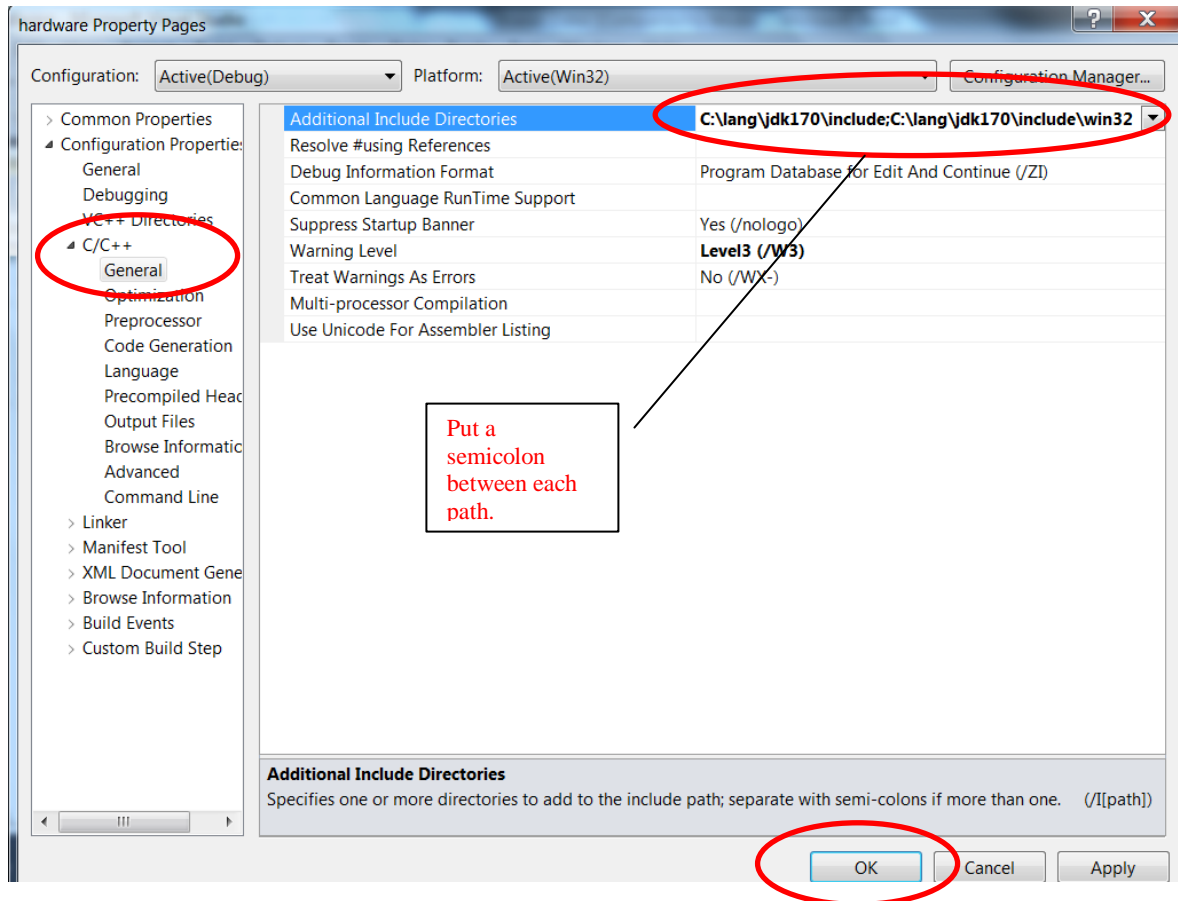


As you see in the previous figure, it is necessary to specify the missing definitions, which cause syntax errors (the red lines underlining java keywords). Go to the project properties and add the path for these definitions which are inside the installed java SDK, as illustrated in the next two figures:

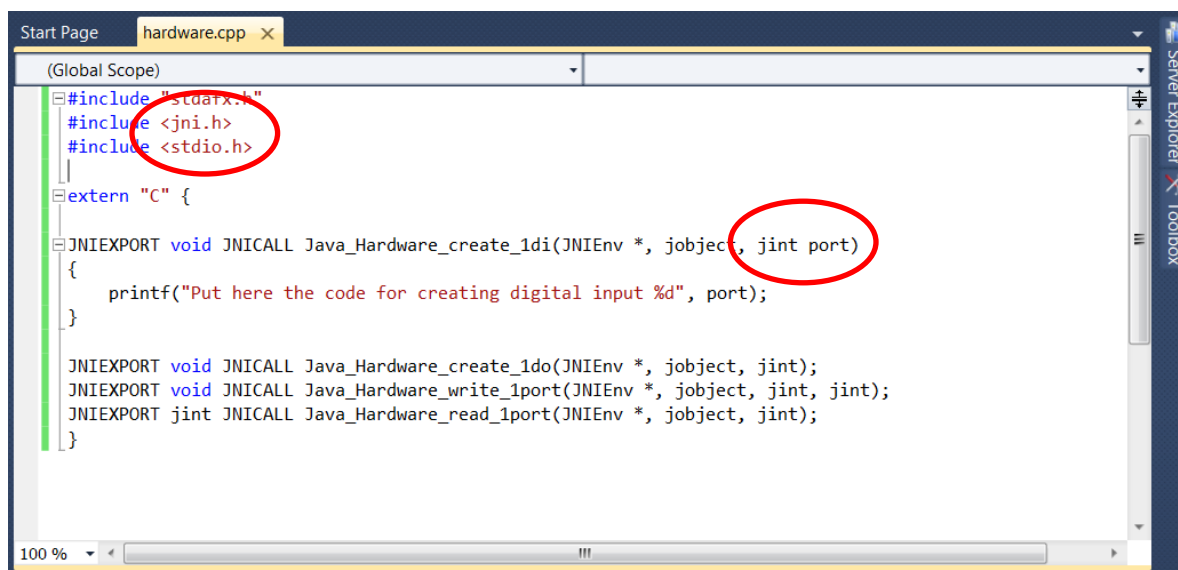


Add these two paths to the VStudio project properties:



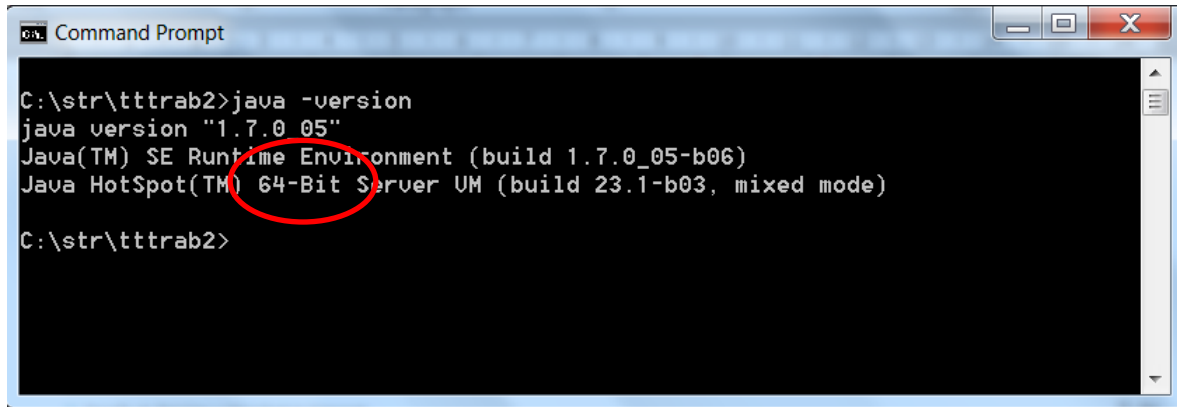


We can now edit our native functions in “hardware.cpp”. In this stage we are just interested in testing the connection between java and the native code. Firstly, add the “include <jni.h>” directive to the “hardware.cpp”. Proceed as illustrated in the next figure:





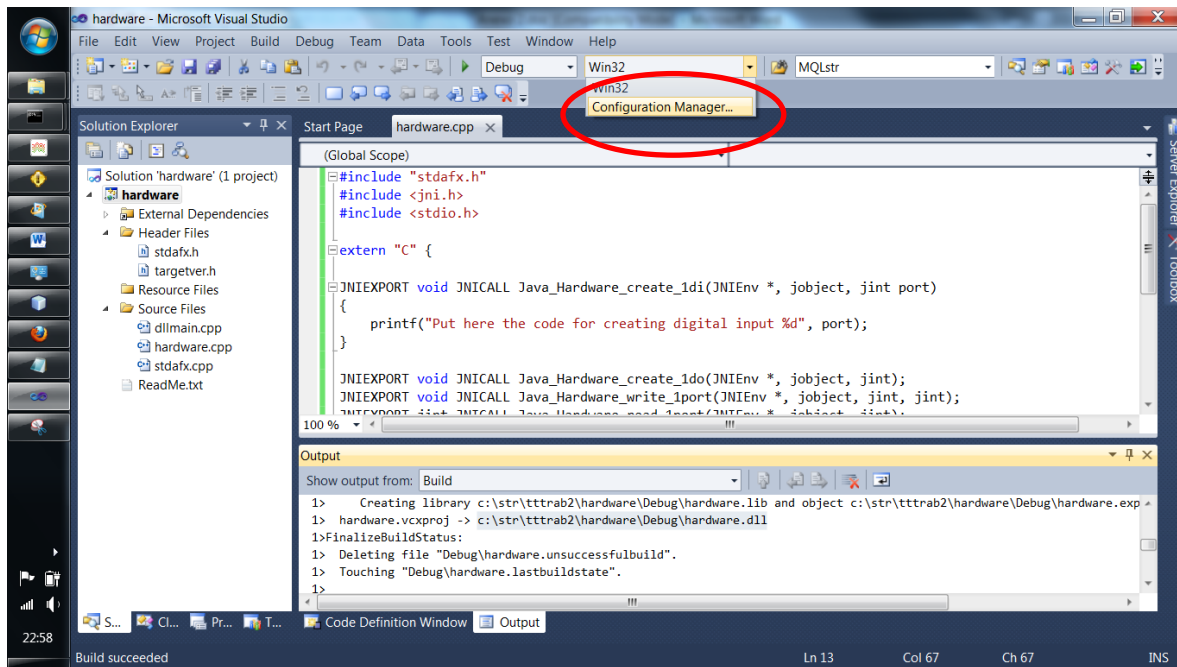
Another important detail is related to whether you have installed an 32 or 64 bit version of Java SDK. Go to a console and type “java –version”:



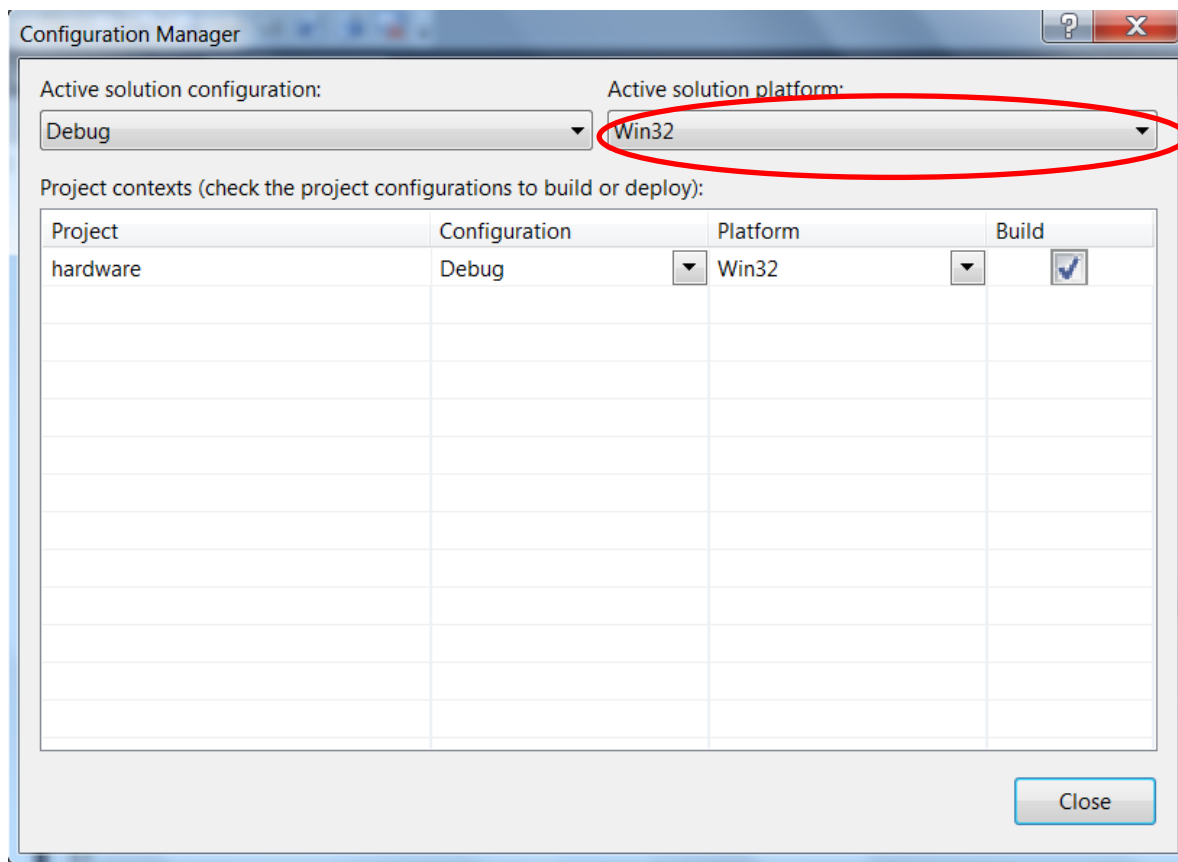
```
C:\str\tttrab2>java -version
java version "1.7.0_05"
Java(TM) SE Runtime Environment (build 1.7.0_05-b06)
Java HotSpot(TM) 64-Bit Server VM (build 23.1-b03, mixed mode)

C:\str\tttrab2>
```

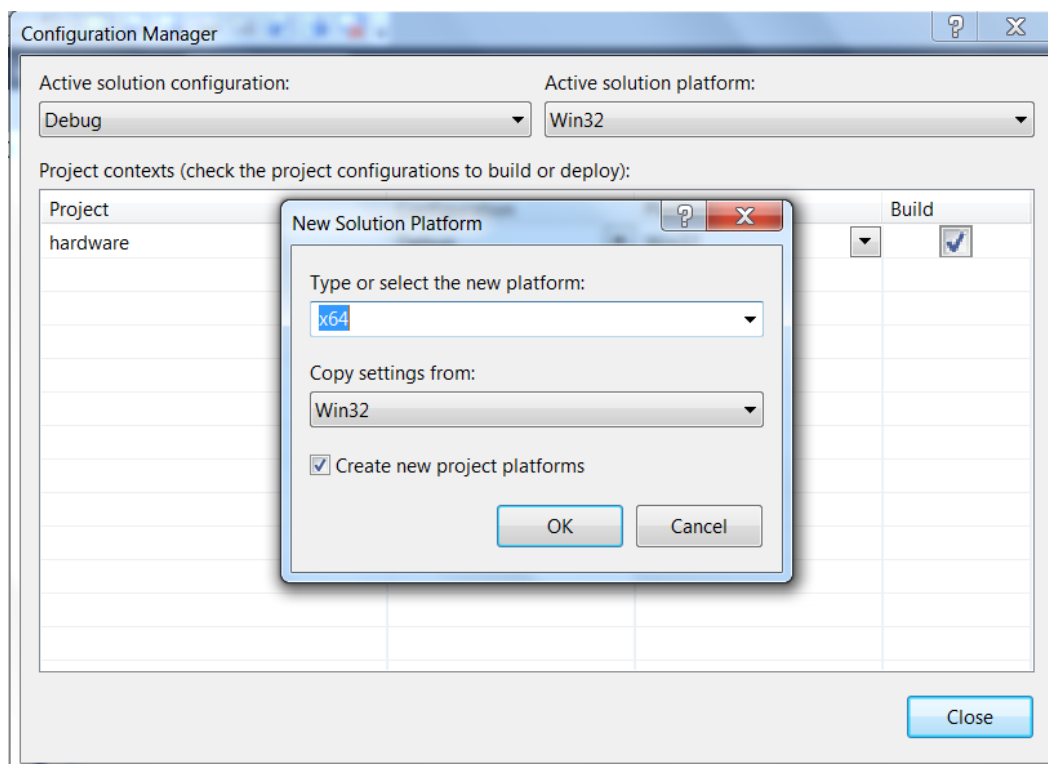
If Java Runtime environment is 64 bit, you need to go to the configuration manager, as described in the figure below. If it is 32 bit (not 64 bit), don't configure your project to 64 bit, and instead, proceed to step 3.1.



In the active solution platform of the next figure, select a new one and chose 64bit.

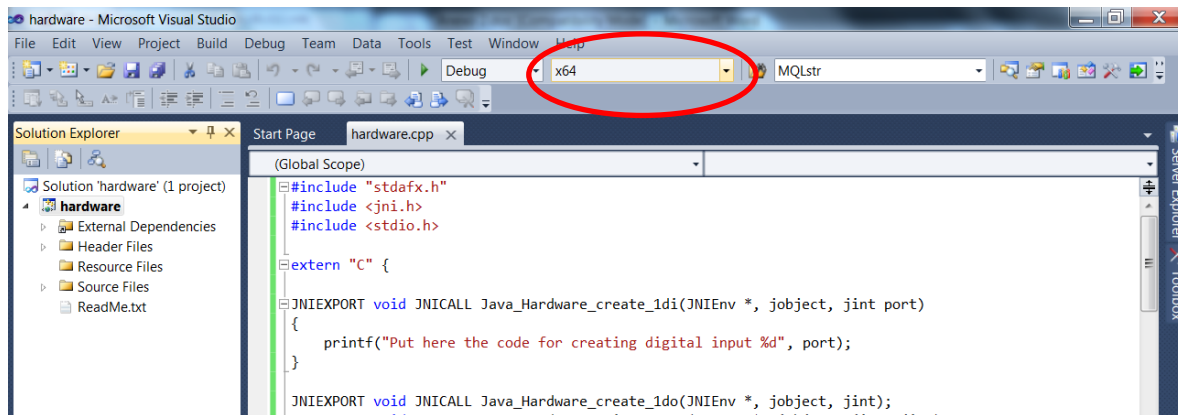


In the next figure, press ok and then close to proceed.



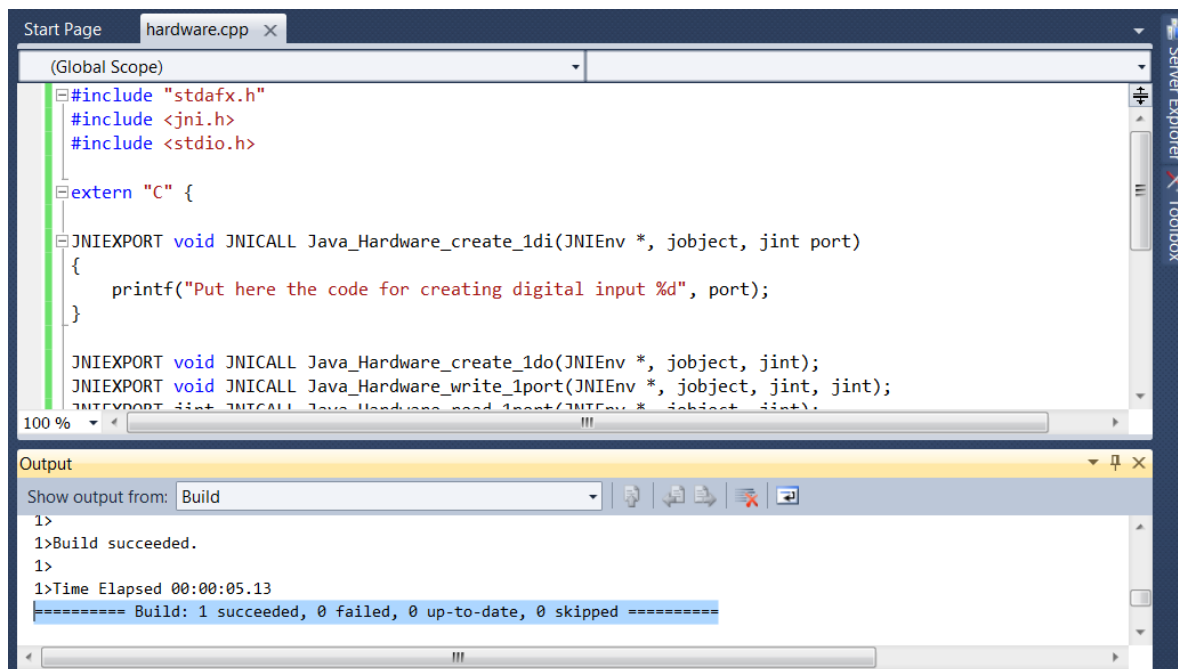


You should see now the project configured to x64.

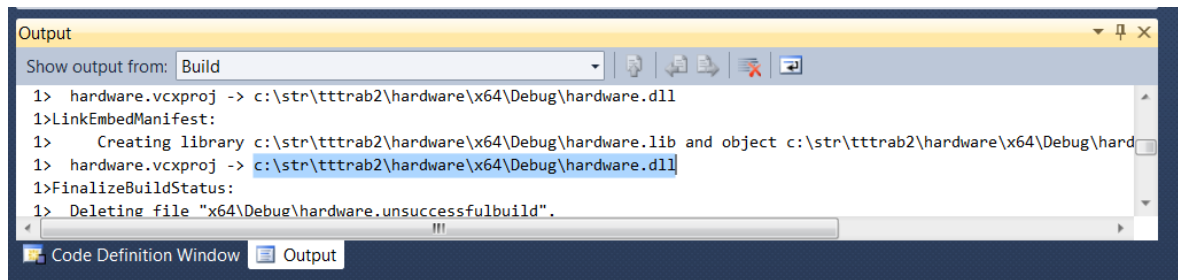


### Step 3.1 – Build the solution

Click in “Build->Build solution” in the menu. You should obtain a successful compilation, as shown in the output window:



In the output window, locate the information which tells us where the library “hardware.dll” has been created, as illustrated by the next figure:



Copy this line and add it to the java class in “Hardware.java”, by adding a load library in a static statement, as follows:

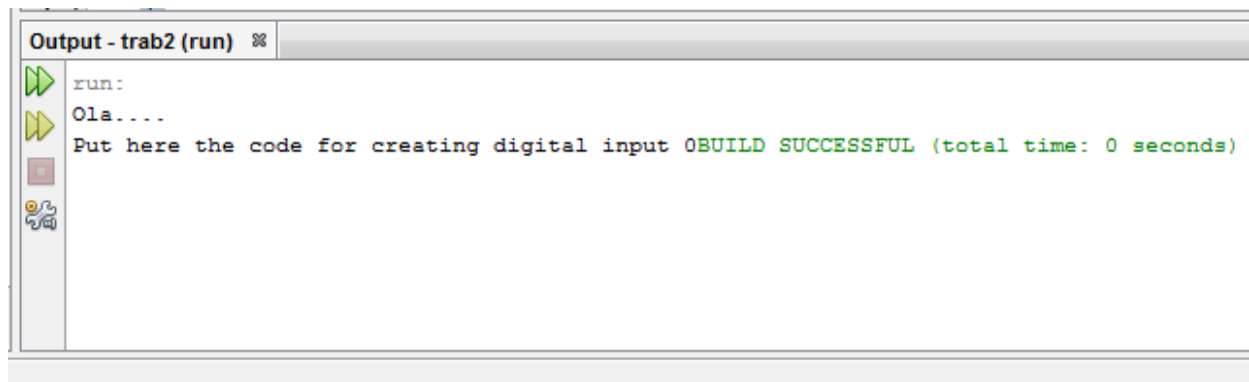
```
public class Hardware {
    static
    {
        //c:\str\tttrab2\hardware\x64\Debug\hardware.dll
        System.load("c:\\str\\tttrab2\\hardware\\x64\\Debug\\hardware.dll");
    }

    native public void create_di(int port);
    native public void create_do(int port);
    native public void write_port(int port, int value);
    native public int read_port(int port);
}
```

In order to test your code, go to the “Hello.java” file or other you have tried before, create an Hardware object and invoke the create\_di() method:

```
public class Hello {
    public static void main(String args[])
    {
        System.out.println("Ola....");
        Hardware h = new Hardware();
        h.create_di(0);
    }
}
```

Click on the project, with the right button of the mouse, and press “clean and build”. Running the class Hello should yield the following result, which means that Java JNI / DLL interaction has been achieved successfully:

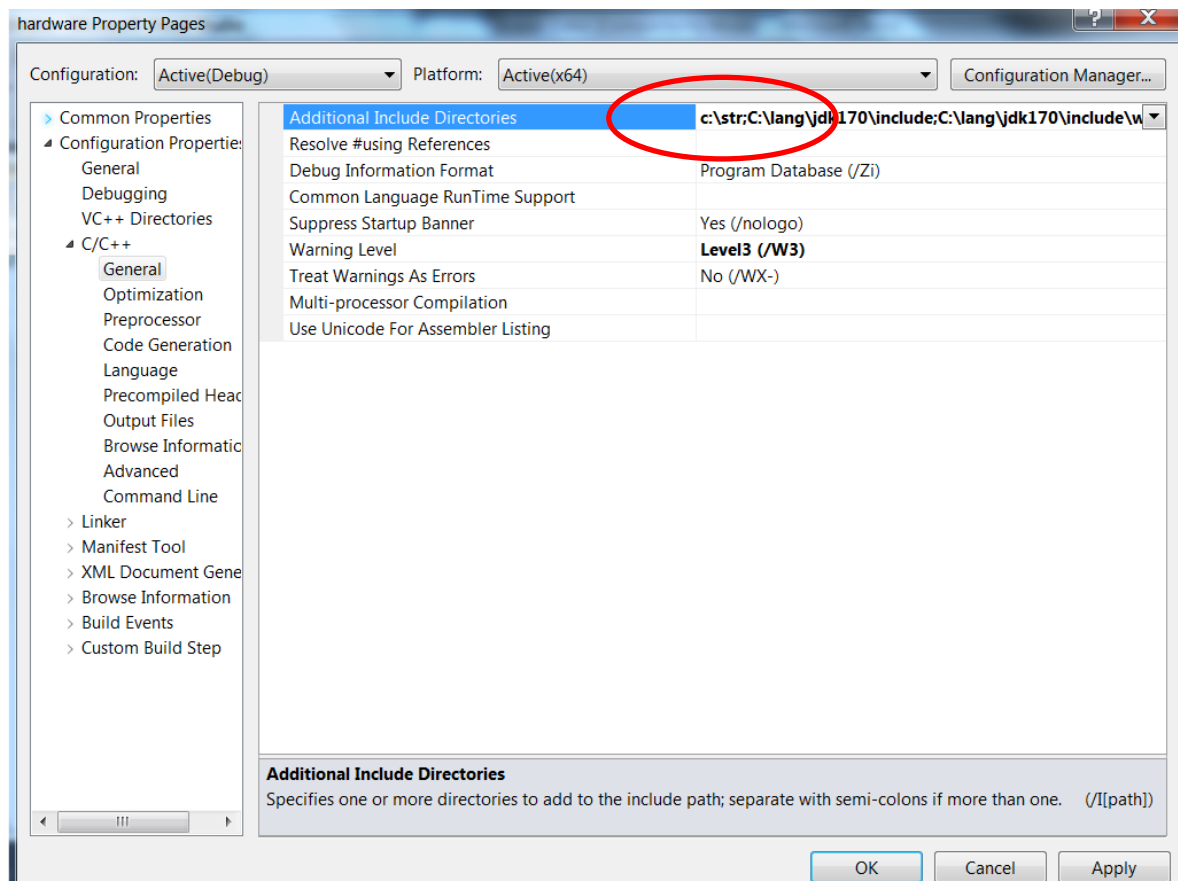


If the test did not succeed, delete any class file (e.g. 'Hardware.class'), which might appear in the default package of netbeans project, and click again "clean and build". If the test still does not work, try to perceive whether some aspects of the above steps were not undertaken.

#### Step 4 – Accessing the kit

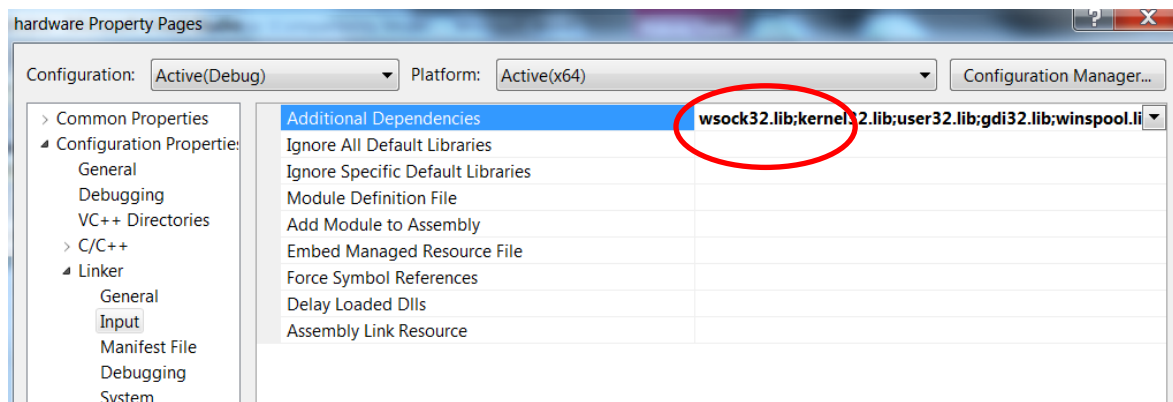
If you reached this step, it means you could develop a DLL and that your java class can invoke functions from your DLL. In this step we complete the development of your DLL, so that your java program can communicate with the lab. Kits.

Similarly to the first lab work, go to project properties of the DLL project and add "c:\str" to the additional paths (don't forget the semicolon):

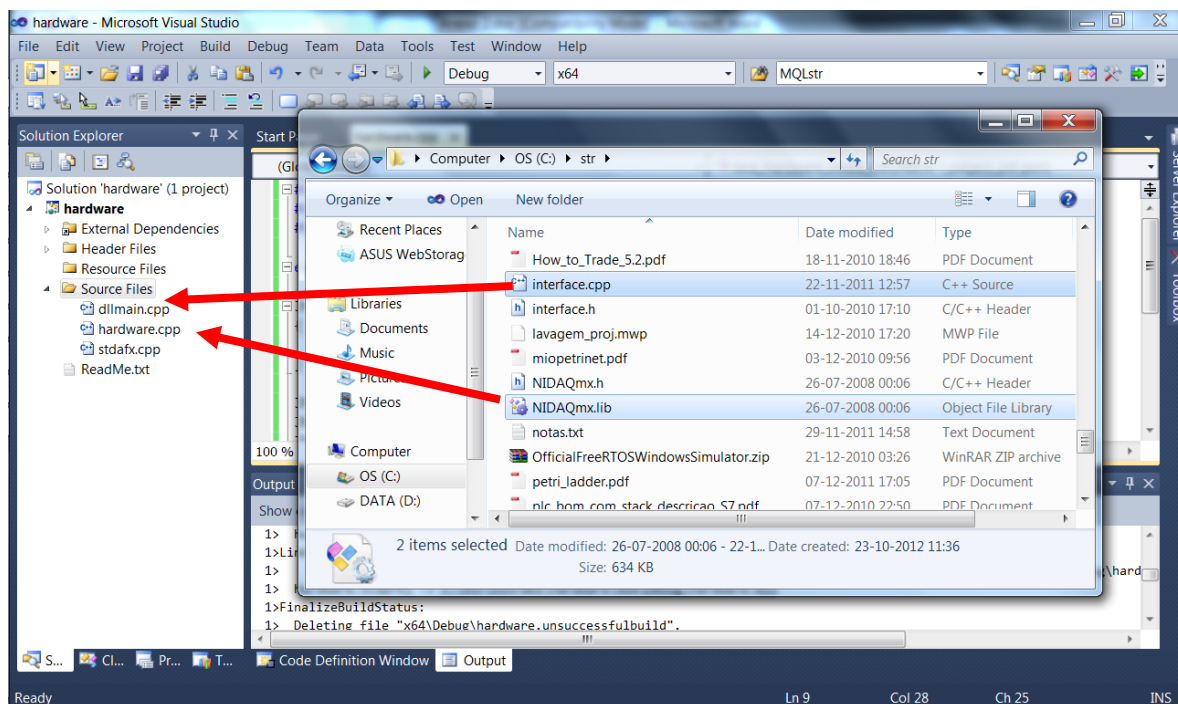




Add wsock32.lib to the additional dependencies (don't forget the semicolon):



Drag and drop “nidaqmx.lib” and “interface.cpp” to the source files:



Complete the definitions of the native functions inside your “hardware.cpp”. The complete list is as follows:

```
#include "stdafx.h"
#include <jni.h>
#include <stdio.h>
#include <interface.h>

extern "C" {

JNIEXPORT void JNICALL Java_Hardware_create_1di(JNIEnv *, jobject, jint port)
{
    printf("Put here the code for creating digital input %d", port);
    create_DI_channel(port);
}
```



```
}
JNIEXPORT void JNICALL Java_Hardware_create_1do(JNIEnv *, jobject, jint port)
{
    create_D0_channel(port);
}

JNIEXPORT void JNICALL Java_Hardware_write_1port(JNIEnv *, jobject, jint port , jint value)
{
    WriteDigitalU8(port, value);
}

JNIEXPORT jint JNICALL Java_Hardware_read_1port(JNIEnv *, jobject, jint port)
{
    int v = ReadDigitalU8(port);
    return(v);
}
}
```

Don't forget to rebuild the DLL. Now return to netbeans and edit your Hello, or other, class in order to use the ports:

File "Hello.java"
<pre>public class Hello {     public static void main(String args[])     {         System.out.println("Ola....");         Hardware h = new Hardware();         h.create_di(0);         h.write_port(2, 1&lt;&lt;0);     } }</pre>

Open a simulator, e.g the car\_wash. Click "clean and build the project" and run "Hello.java".

If you see the wash tower moving along the xx axis, then your java program can communicate with the kits successfully. ☺

Good work.