C++ project using Arduino.cc core and AVR Studio

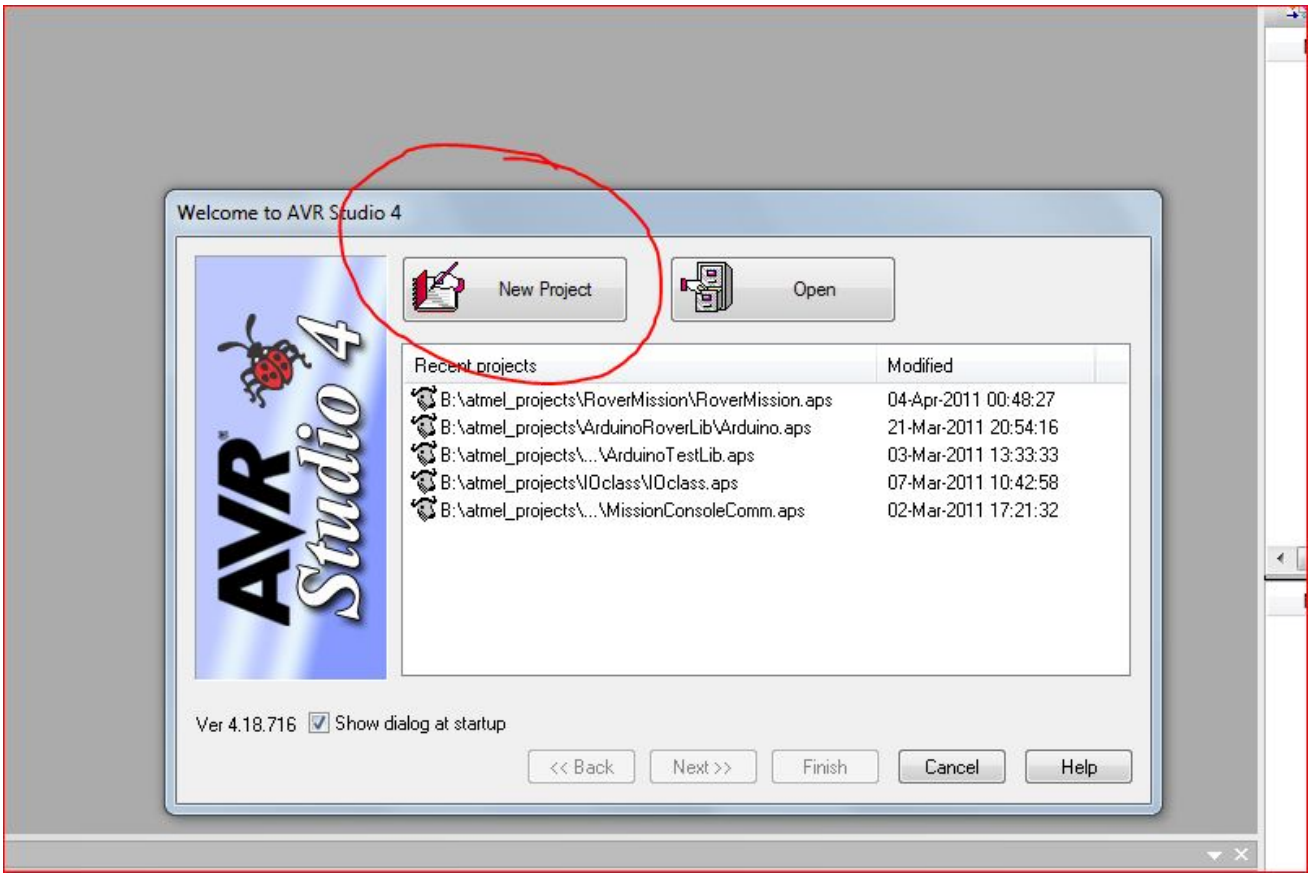
The following tutorial requires Arduino 022 (version 1.0 and higher are currently not supported) and AVR Studio 4 (AVR Studio 5 and higher are currently not supported).

This tutorial help you get a C++ project up and running that incorporates the Arduino.cc core functionality. This allows us to debug and simulate the Arduino core files and also other Arduino community library files.

Before we begin you must have Atmel GCC toolchain, the AVR Studio IDE installed on your computer and the Arduino IDE.

NOTE:
There will be heavy reference to “core files” and “library files”. Core files include functions and macros that are defined here. Library files are .cpp and .h files that are documented here.

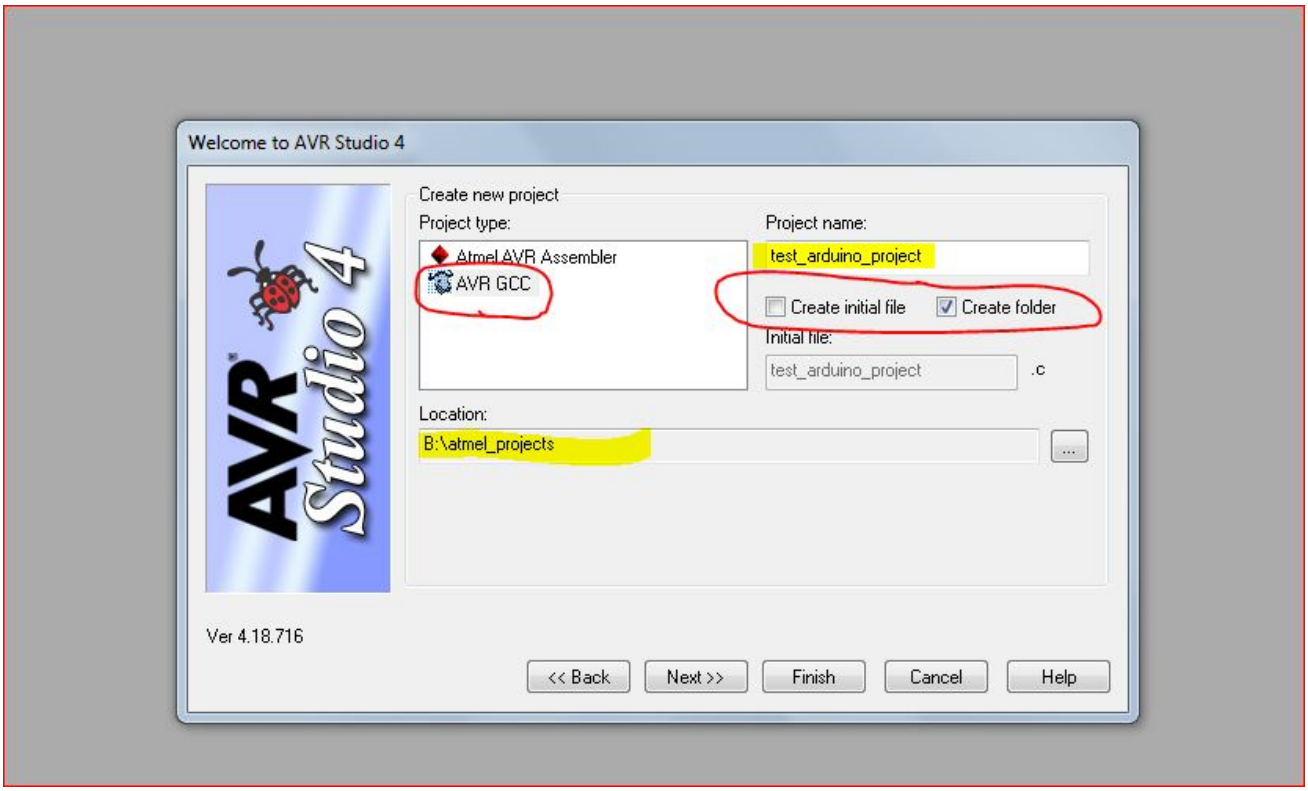
Step1: Open AVR Studio and choose new project (Project -> New Project)



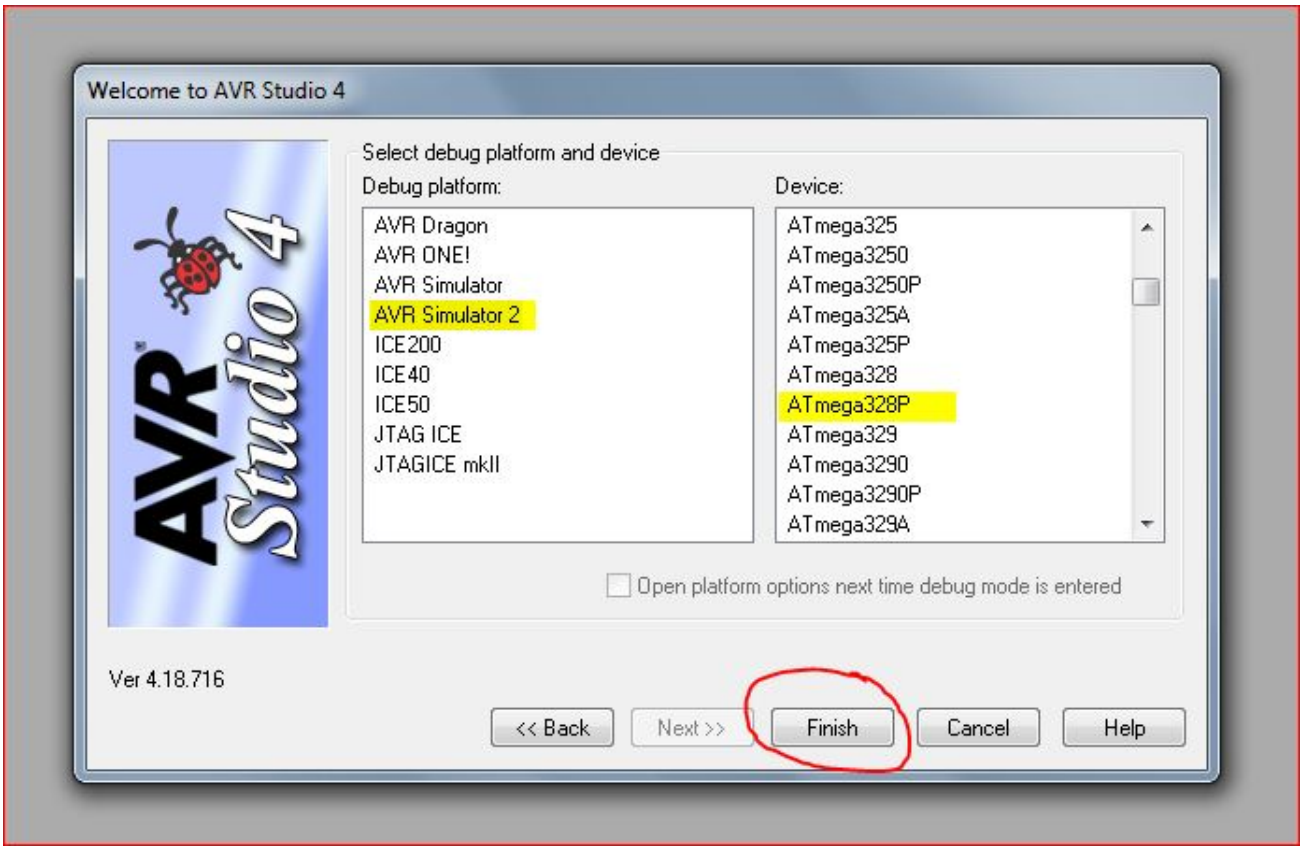
Step 2: Configure the new project

1. Choose AVR GCC as the project type
2. Give the project a name

3. Make sure that “Create initial file” **IS NOT CHECKED**. If it is you will configure a C project, not a C++ project.
4. Select “Create folder” and choose the location that you want this project to be stored

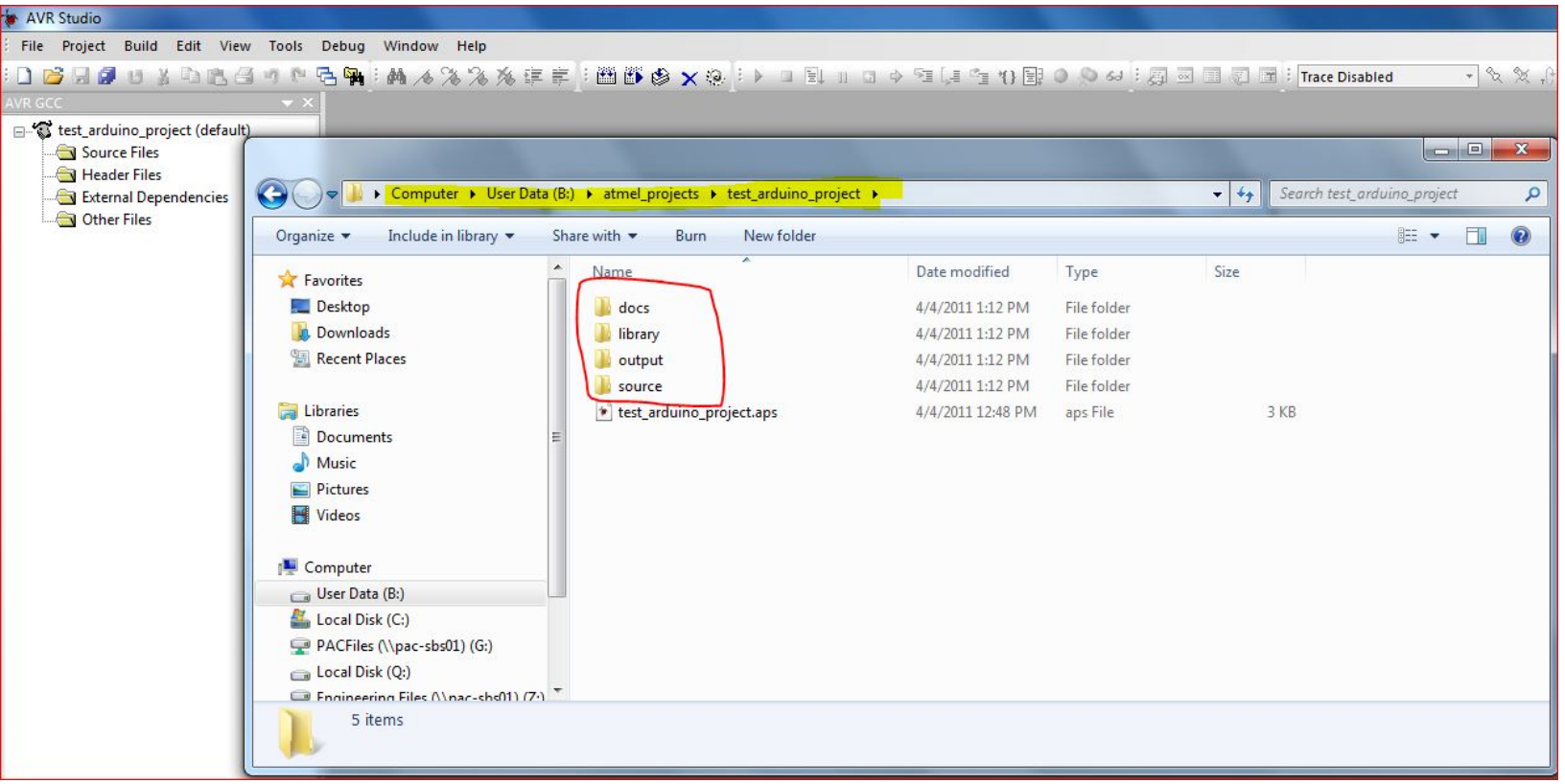


Step3: Select the Simulator and micro-controller that is on your Arduino, then click Finish

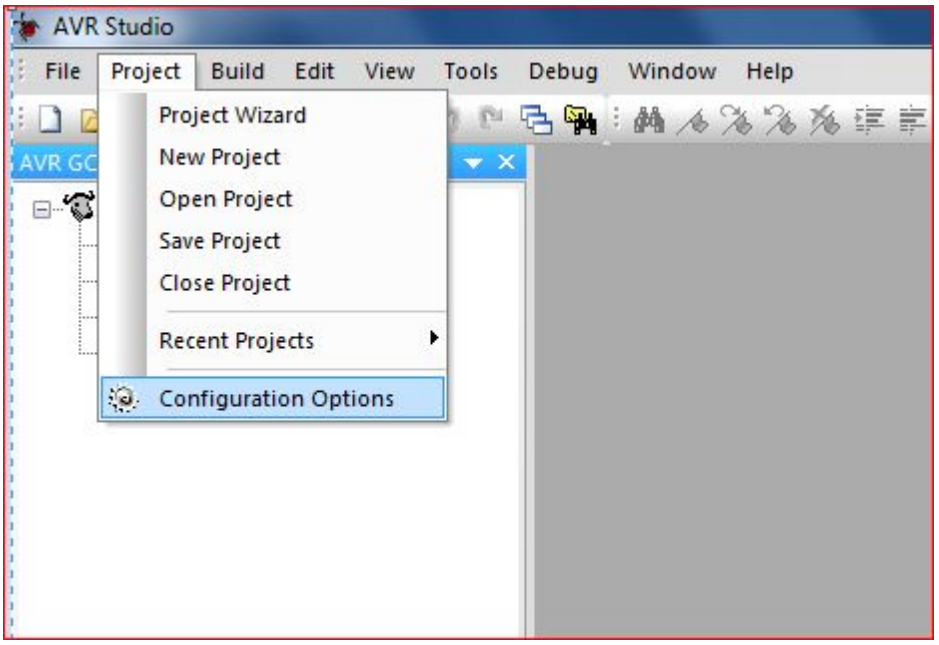


Step4: Open the directory where the project was made and create 4 new folders

1. source - this will contain the user generated source code files
2. output - this is where the compiler and linker will barf
3. library - this will contain the Arduino.cc core files
4. docs - this will contain the output generated by Doxygen if it is used



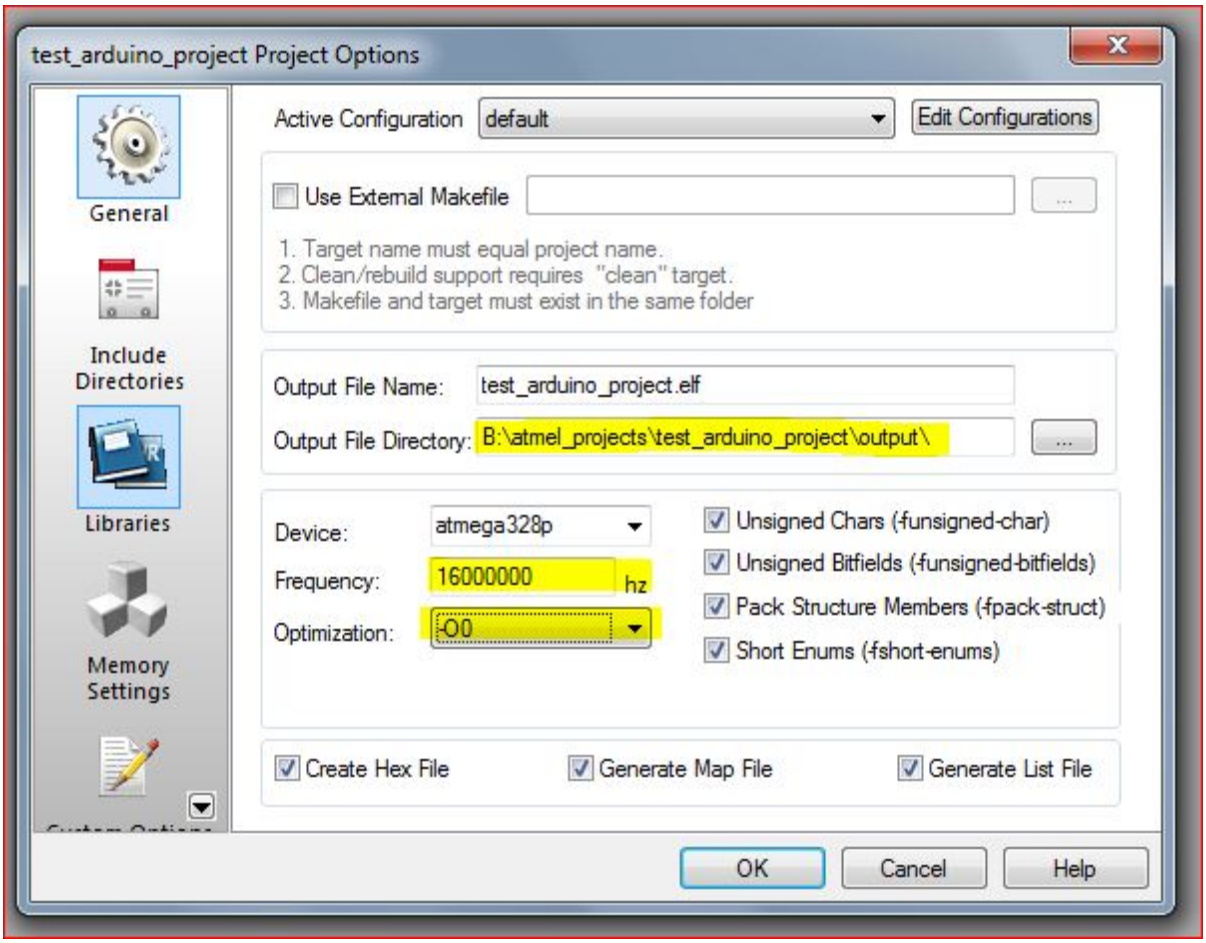
Step5: Configure the project



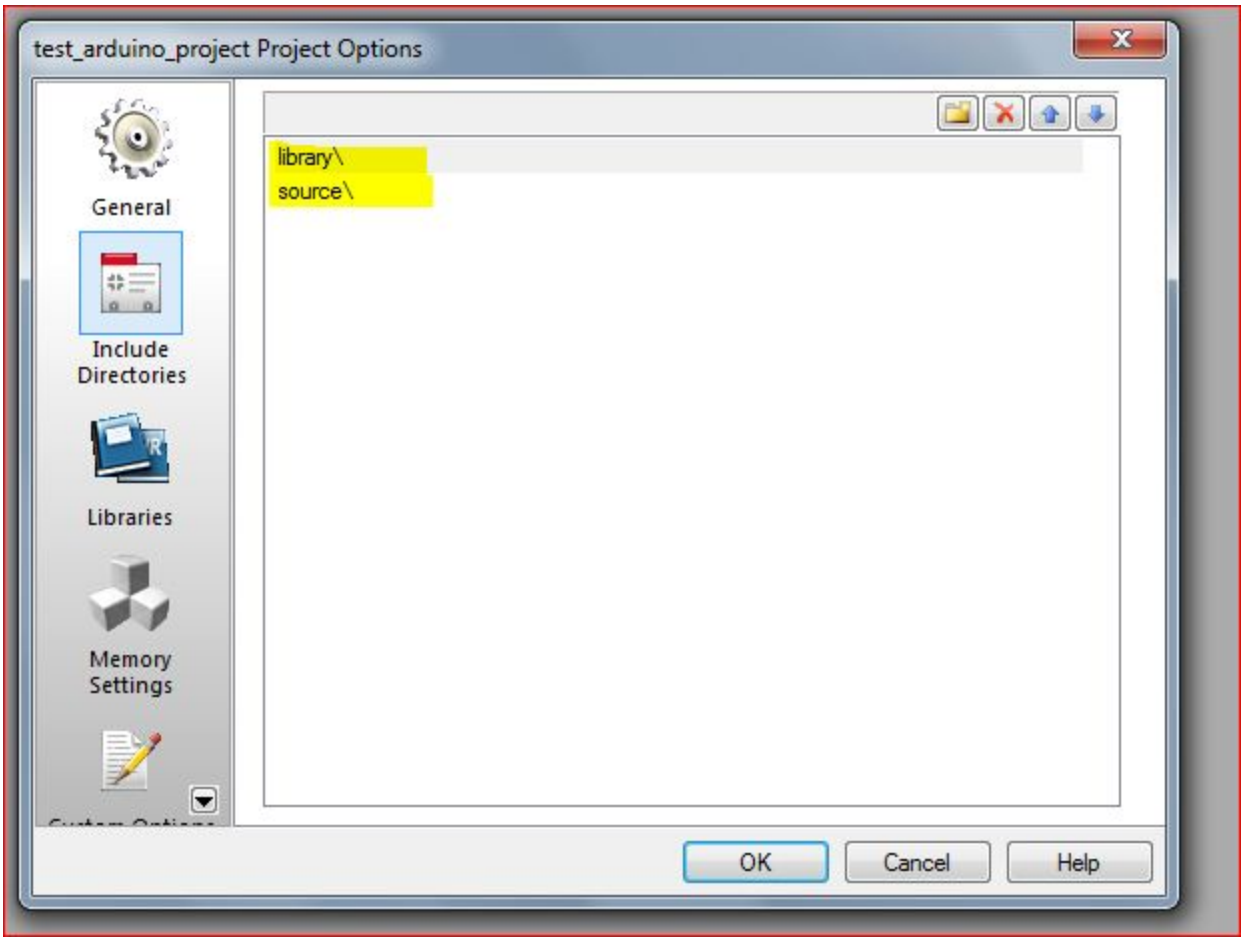
Step6: Change the general settings

1. The output file directory should be pointed to the output file that was created in step4
2. The oscillator frequency of the Arduino is 16MHz (16000000Hz)

3. Optimization for debugging needs to be -O0
    a. The code size will be large but this is the only way that the debugger can be used to produce appropriate steps when jogging through the code.

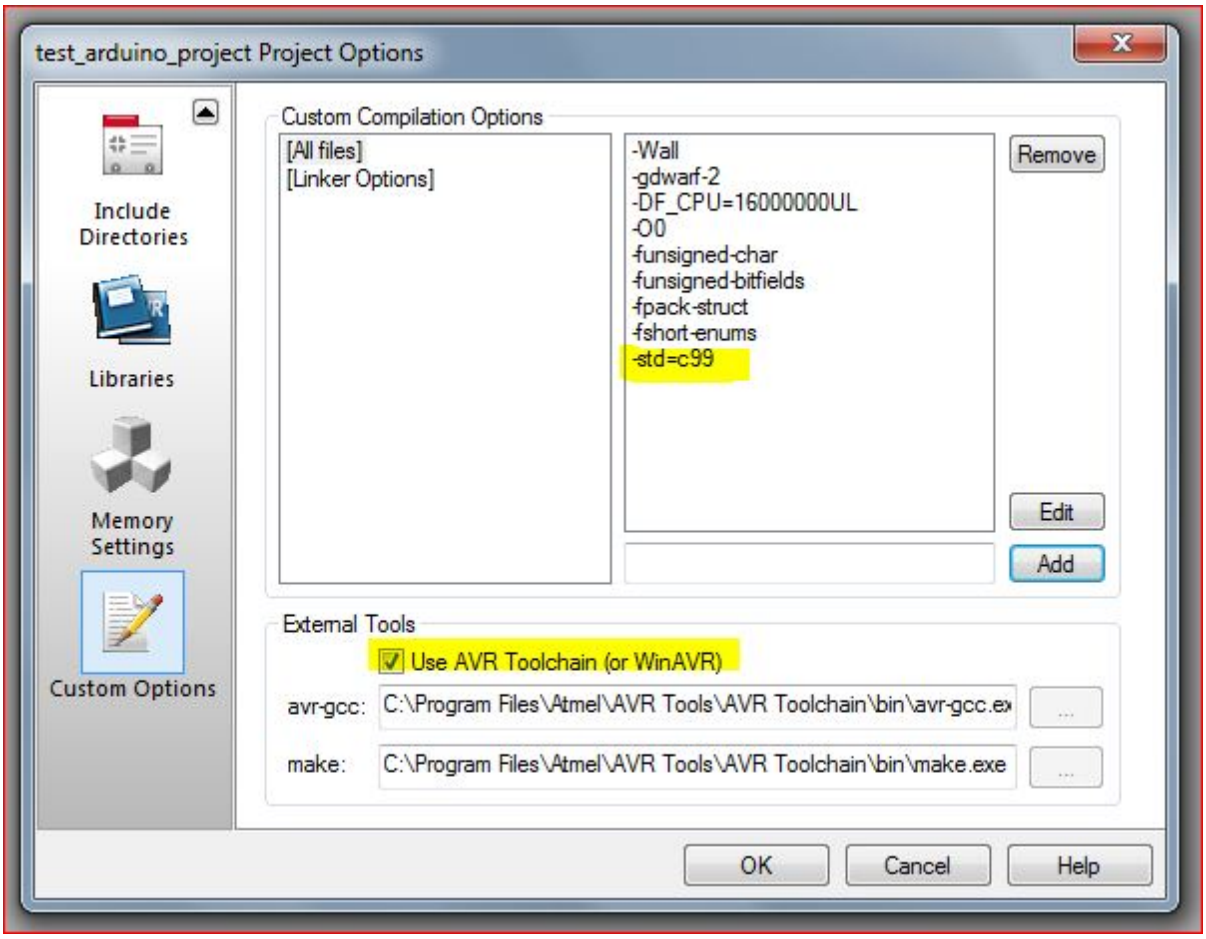


Step7: Include Directories - point to the library and source folders that were created in step4.



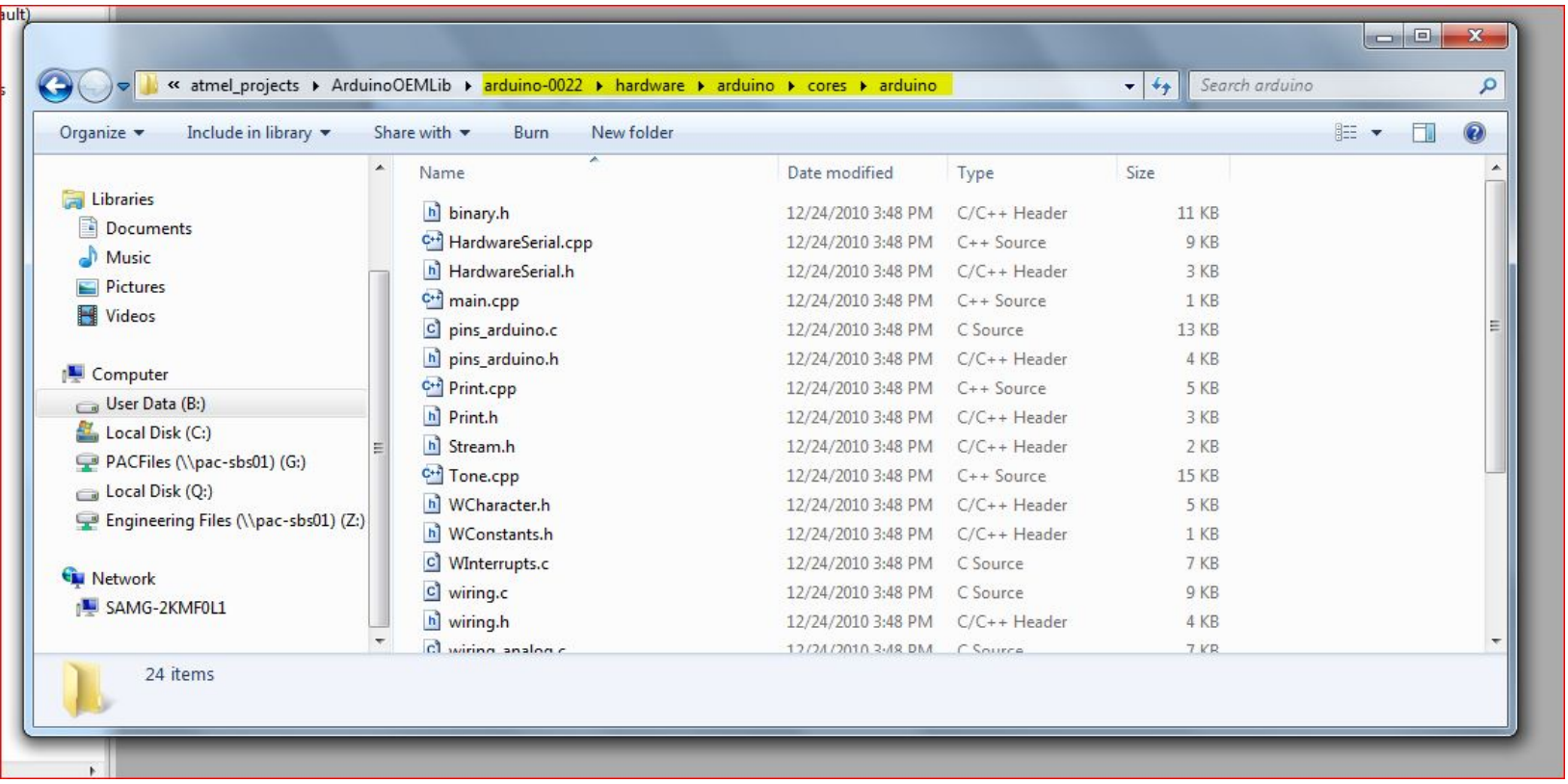
Step8: Custom Options

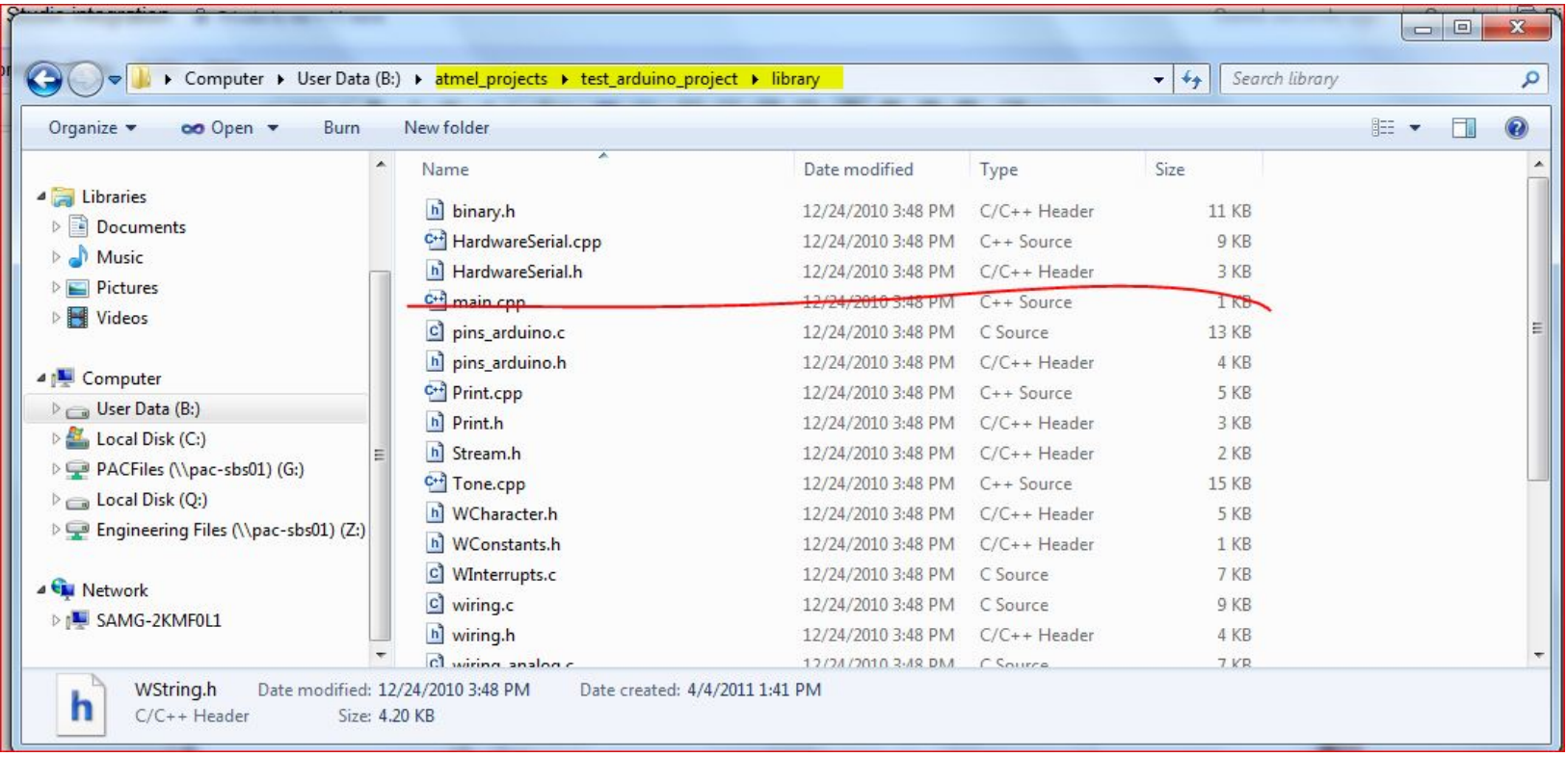
1. Click on -std=gnu99 and choose edit
2. Change gnu99 to c99
3. Click Add
4. Make sure that “Use AVR Toolchain” is selected and the path points to the installed tools
5. Click OK



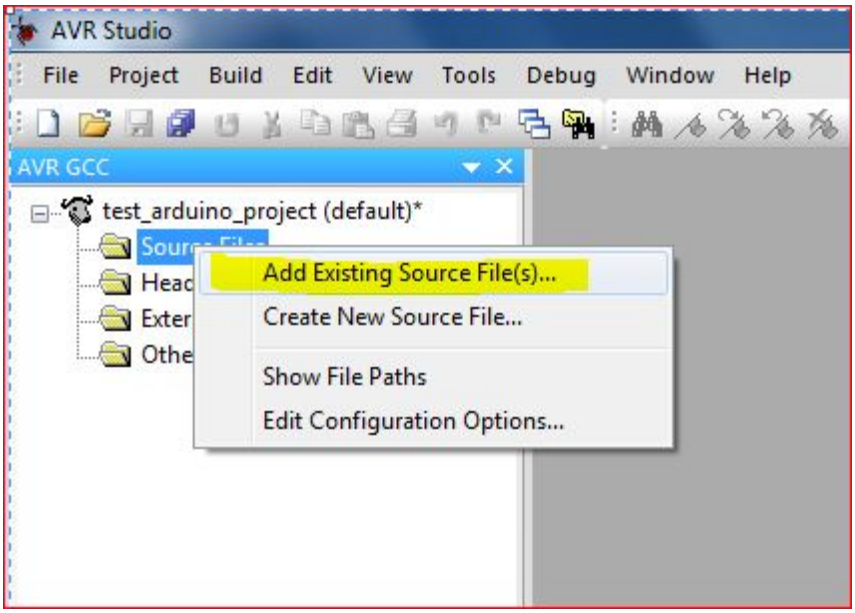
Step9: Importing the Arduino.cc core files

1. browse to the location .\arduino-0xx\hardware\arduino\cores\arduino
2. select all files from this location and copy them
3. paste all files in the location t.\est_arduino_project\library
    a. Find file main.cpp and delete it. We will later create a main.cpp file with our code



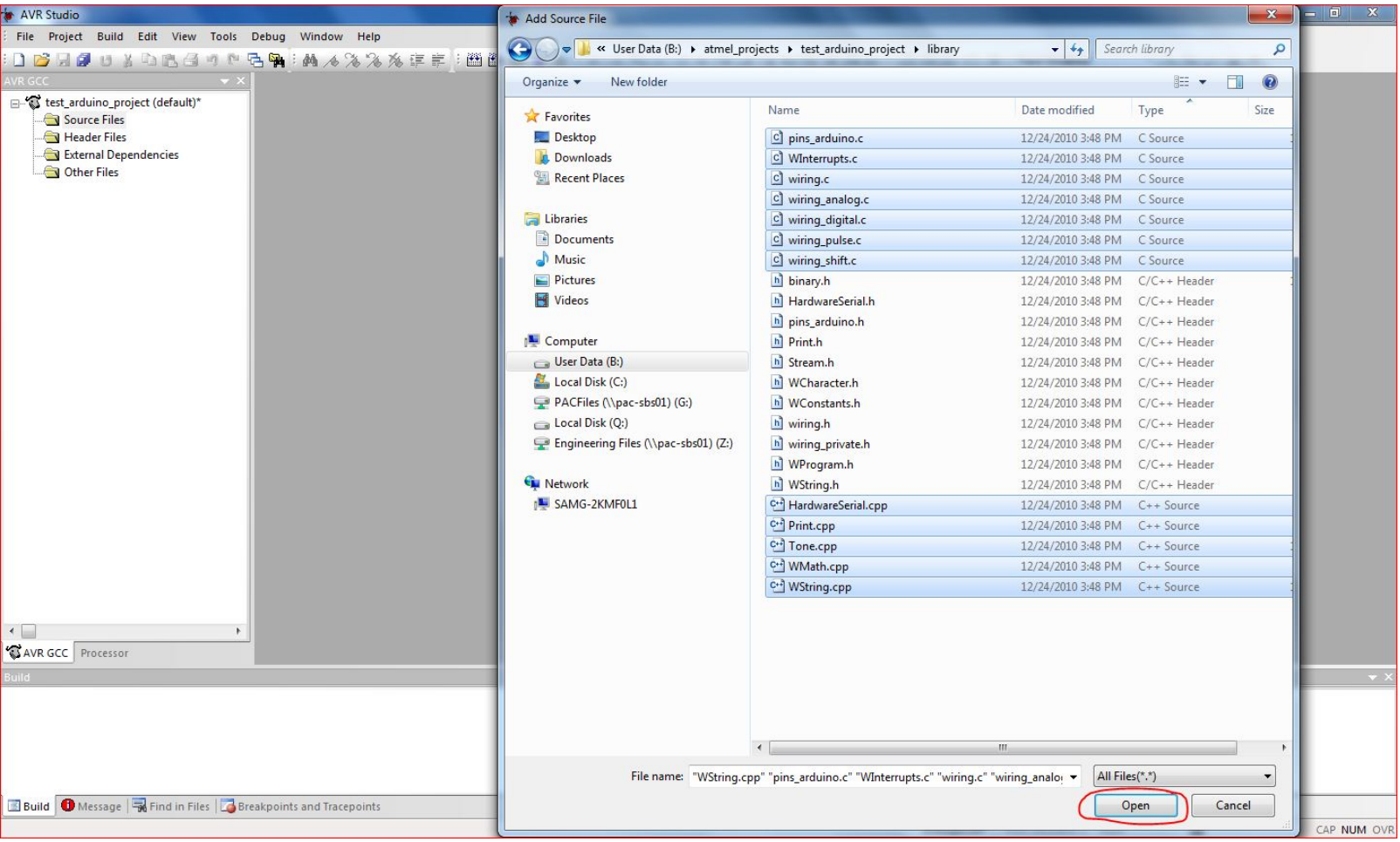


Step10: Add files to the project - Right click on Source Files and select “Add Existing Source File(s)”



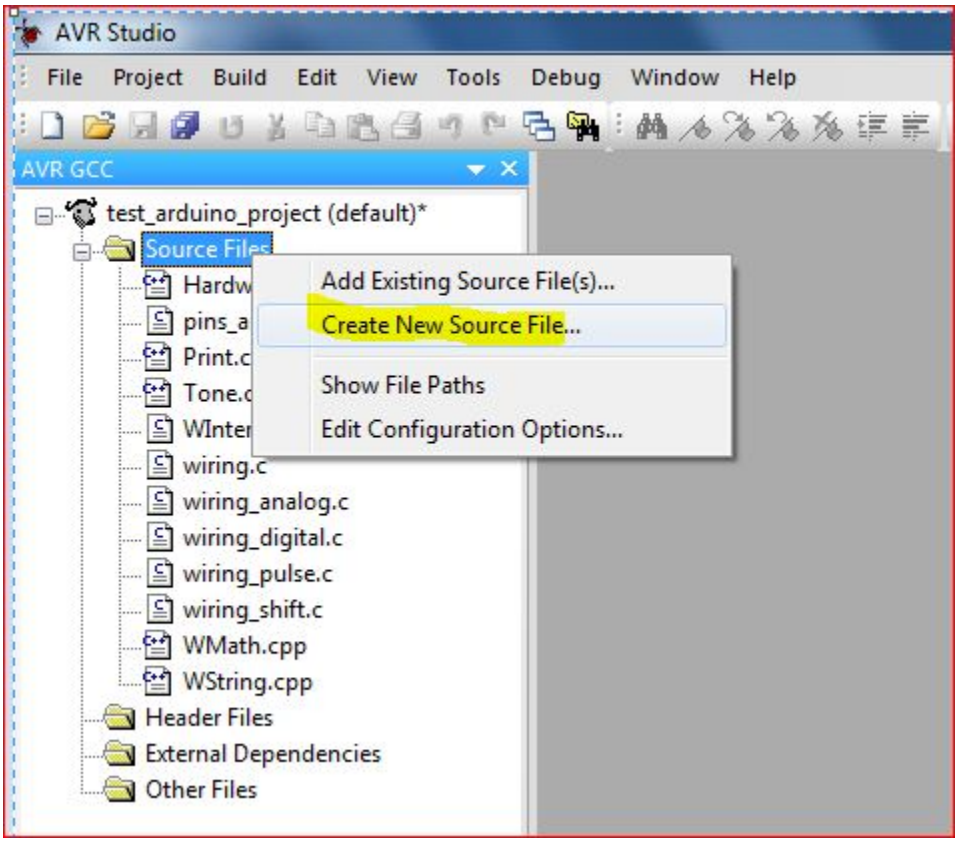
Step11: Adding the necessary files - navigate into the library folder that contains all the files copied from step9 (.\test_arduino_project\library)

1. Change the file filter from *.c, *.s to *.*
2. Select every .c and .cpp file
3. Click “Open”

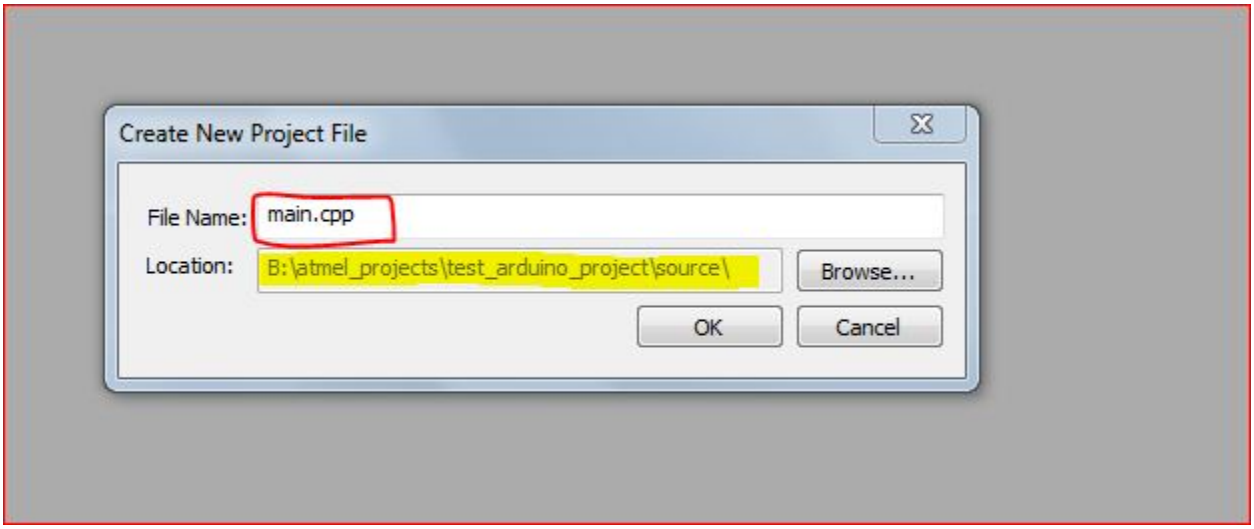


Step12: Now all the core source files from the Arduino.cc project are part of our project and we can create our main source file.

1. Right click on “Source Files” and choose “Create New Source File...”



1. Name the file main.cpp. The most important thing is that the file extension is .cpp
2. “Browse” to the “source” folder created in step4.
3. Click OK



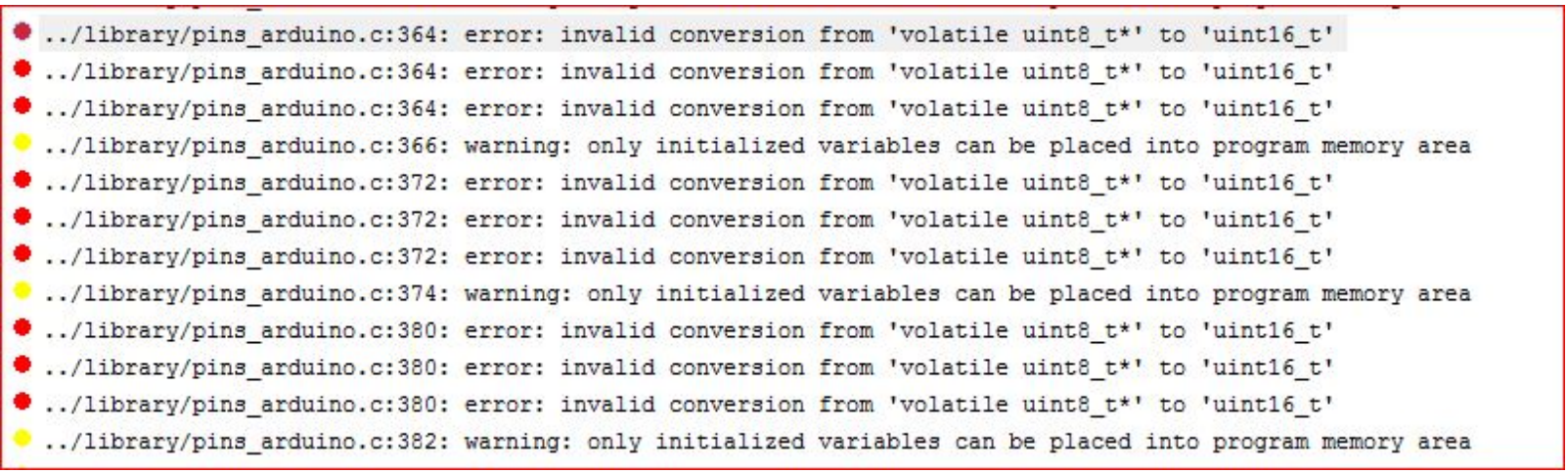
Step14: Include the main resource file and setup the program entrance

1. The main header that must be included is WProgram.h. This will gather all core macros and functions.

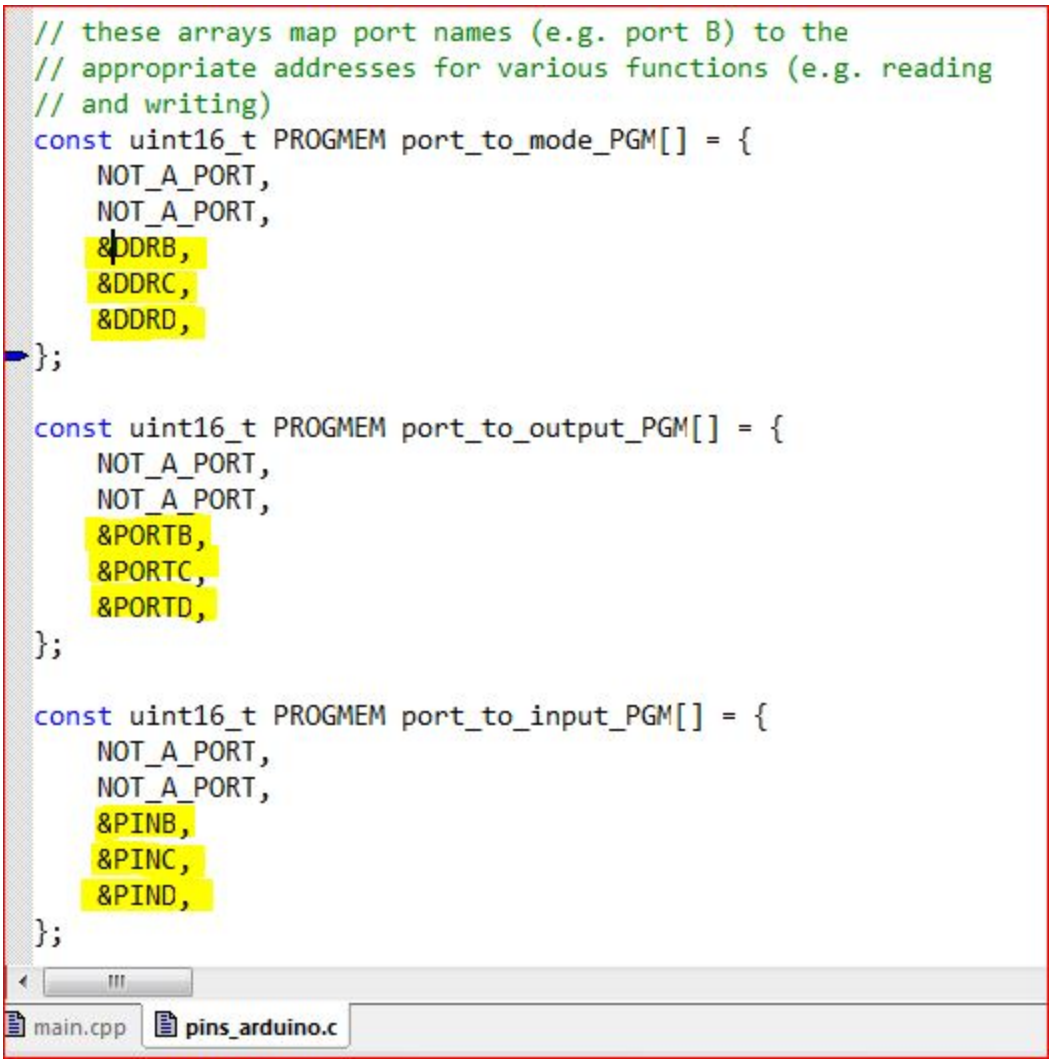
2. init() - this is an Arduino function that is used to configure timing routines
3. setup() - this can be omitted or implemented. If it is used enter all setup code that is to be called once otherwise just list the function calls in its place.
4. loop() - Again this can be omitted or implemented. If it is used - all repetitious code is to be placed in the the function below. Otherwise just list the code withing the while(1) brackets

```
#include "WProgram.h"

int main (void)
{
    //! This must be called first to configure
    //! routines that the core code relies on
    init();

    //! This is can be any standard setup function
    //! from a previous Arduino.cc project
    setup();

    while(1){

        //! This is the same loop from a previous
        //! Arduino.cc project
        loop();

    }

    //! This will never get called but must be present
    //! in any C or C++ program.
    return 0;
}

void setup (void)
{

}

void loop (void)
{

}
```

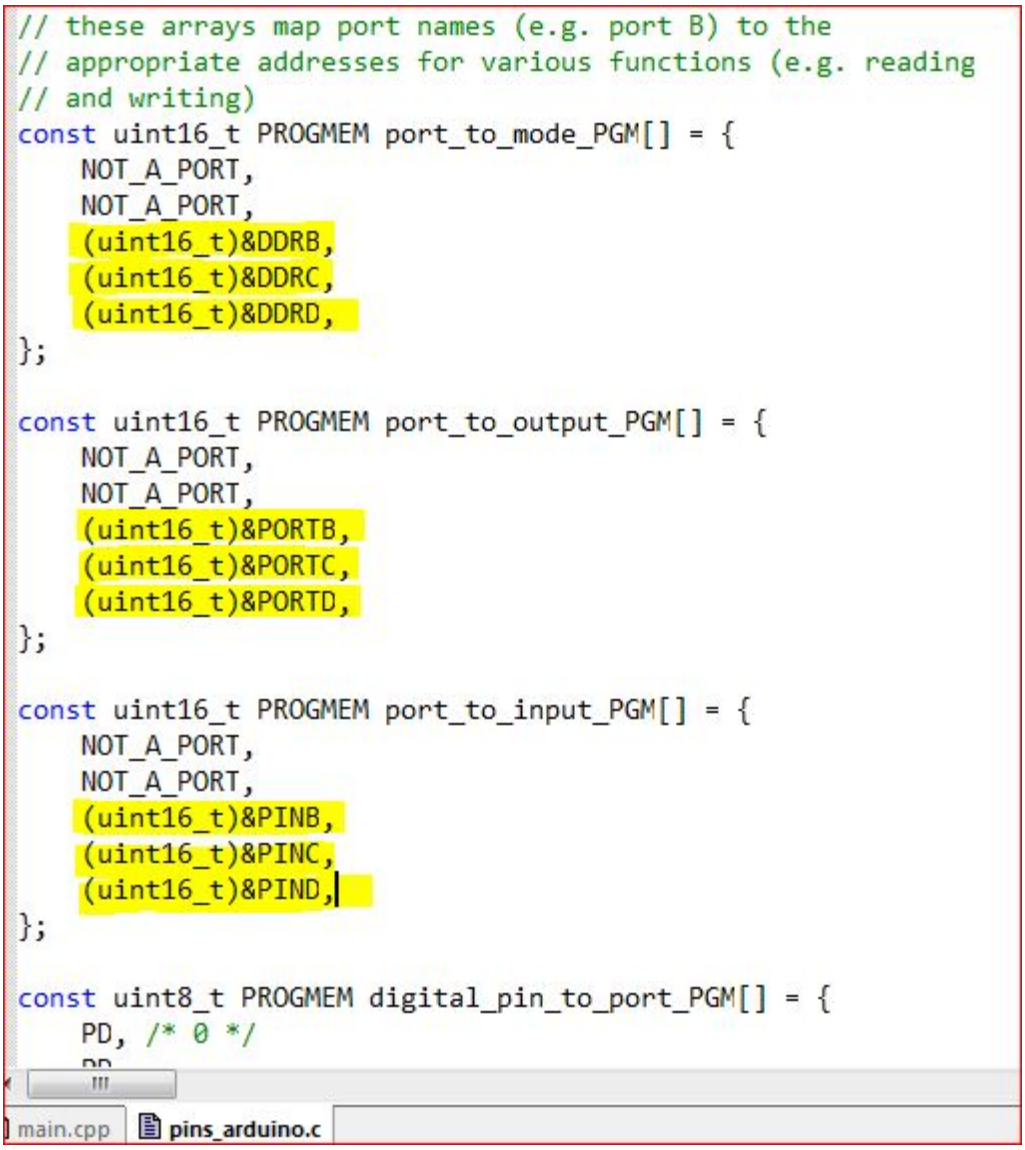
Step15: Compile the code. There will be 9 errors that need to be rectified. The easiest way to find these to to click on the errors and let the program take you to them.



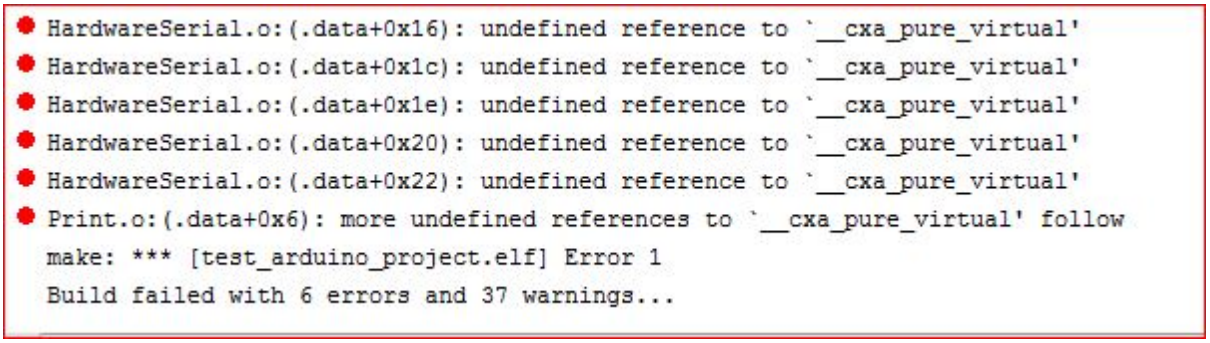
The errors are due to a type-mismatch. This can be rectified by typecasting the variables.



Before each ampersand type “(uint16_t)”



When we compile again all errors are gone and we can correct a linking error caused by using virtual class members.



To correct this we need to create a dummy cxa_pure_virutal function for the linker to work with. This needs to be done as a traditional C function call so make sure the definition is properly attributed.

```
#include "WProgram.h"

//! Give the linker a place to start when trying
//! to realize the virutal functions used in
//! HardwareSerial and Print classes
extern "C" void __cxa_pure_virtual(void);
void __cxa_pure_virtual(void){}

int main (void)
{
    //! This must be called first to configure
    //! routines that the core code relies on
    init();

    //! This is can be any standard setup function
    //! from a previous Arduino.cc project
    setup();

    while(1){

        //! This is the same loop from a previous
        //! Arduino.cc project
        loop();

    }
```

The final results.

```
cc1plus.exe: warning: command line option "-std=c99" is valid for C/ObjC but not for C++
avr-g++ -mmcu=atmega328p -Wl,-Map=test_arduino_project.map pins_arduino.o WInterrupts.o wir
avr-objcopy -O ihex -R .eeprom -R .fuse -R .lock -R .signature  test_arduino_project.elf te
avr-objcopy -j .eeprom --set-section-flags=.eeprom="alloc,load" --change-section-lma .eeprc
avr-objdump -h -S test_arduino_project.elf > test_arduino_project.lss
ECHO is off.
AVR Memory Usage
----------------
Device: atmega328p

Program:   25756 bytes (78.6% Full)
(.text + .data + .bootloader)

Data:        552 bytes (27.0% Full)
(.data + .bss + .noinit)


Build succeeded with 1 Warnings...
```

There amount of flash memory used is high because of compiling the project with the -O0 option. I cannot put enough emphasis that ***DURING ANY DEBUGGING THE -O OPTION MUST BE 0 IN ORDER TO STEP THROUGH AND TRACE CODE. ANY OTHER OPTION WILL NOT DEBUG PROPERLY DUE TO OPTIMIZATIONS.*** For the release code this can be changed to -O2 or -Os.

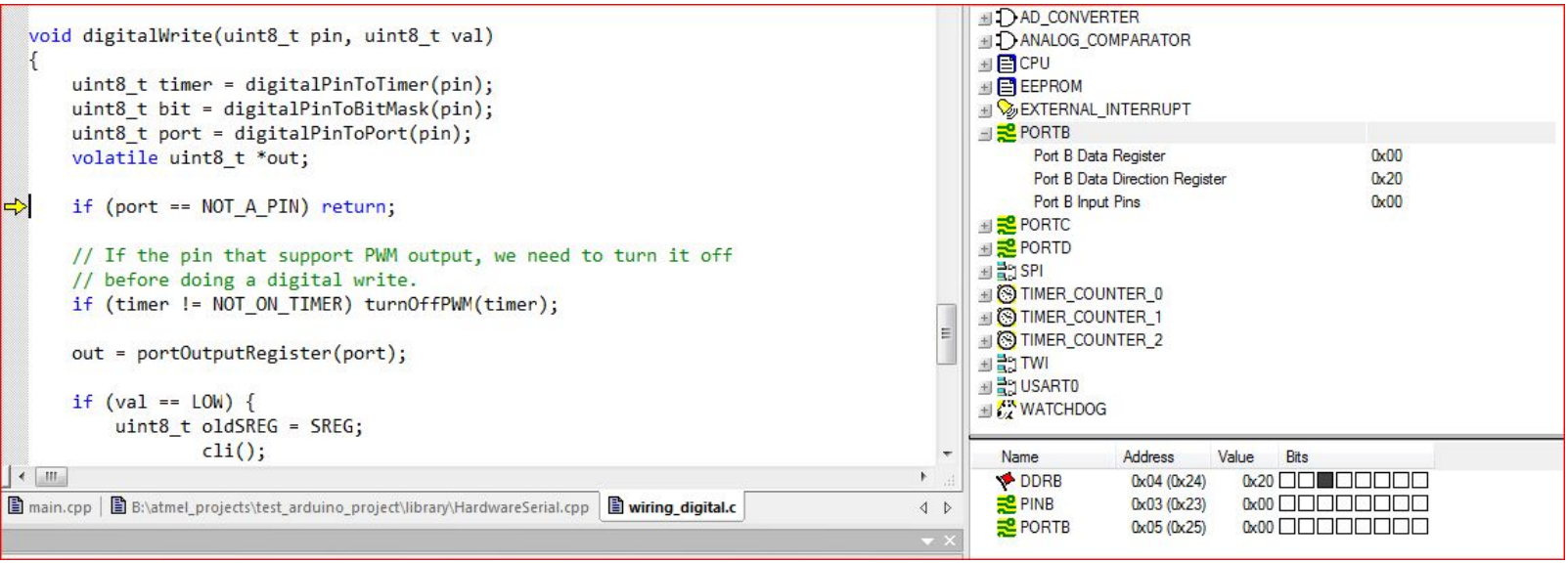
Step16: Make an IO toggle. Since we used the standard setup() and loop() in step14 we can just enter the code in the methods below the main routine. The main routine will call us here. wiring.h contains the definition of setup() and loop() so we don't need to do it again.

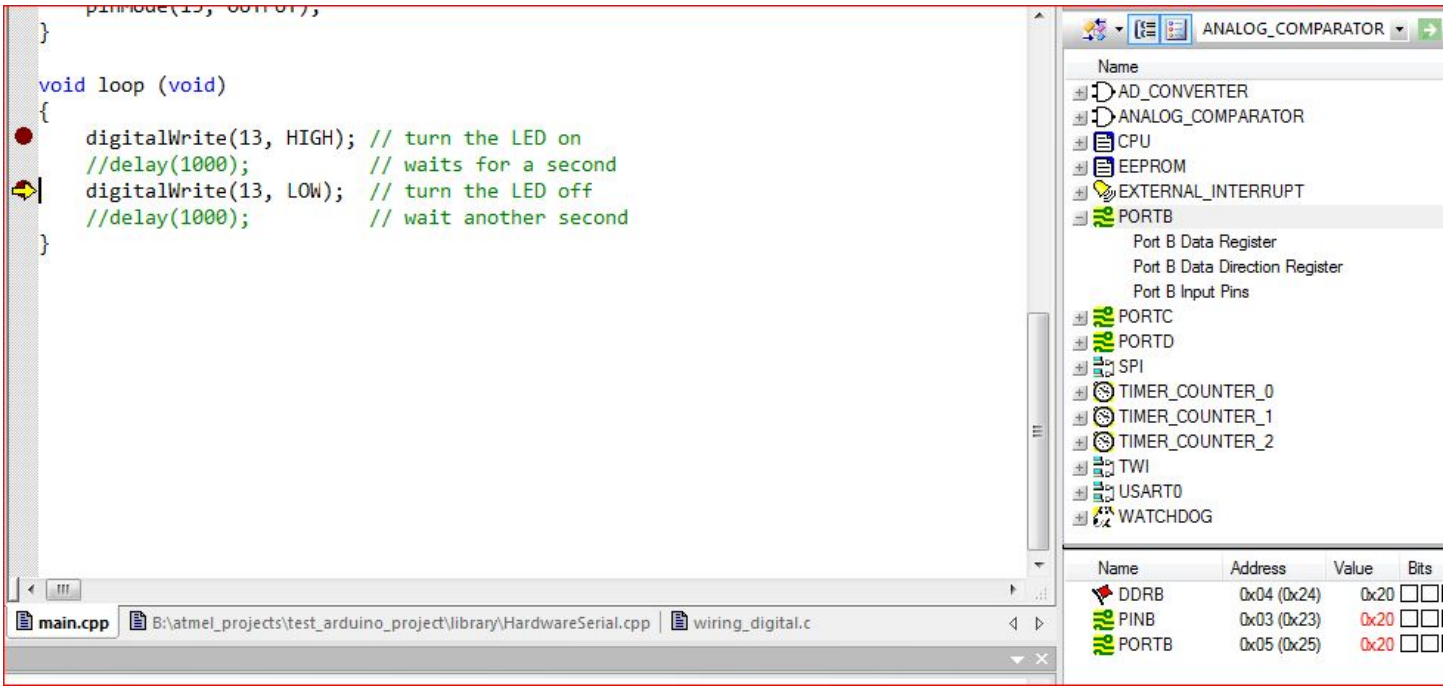
```
void setup (void)
{
    //! sets the digital pin as output.  This needs to
    //! be done only once
    pinMode(13, OUTPUT);
}

void loop (void)
{
    digitalWrite(13, HIGH); // turn the LED on
    delay(1000);            // waits for a second
    digitalWrite(13, LOW);  // turn the LED off
    delay(1000);            // wait another second
}
```

You can compile this code and try to run it through the simulator. You may get some “Browse for folder” requests but just ignore them. Place a break point at the init() function call and tell the simulator to just run!!

```
int main (void)
{
    //! This must be called first to configure
    //! routines that the core code relies on
    init();

    //! This is can be any standard setup function
    //! from a previous Arduino.cc project
    setup();

    while(1){

        //! This is the same loop from a previous
        //! Arduino.cc project
        loop();

    }
```

Now the simulator will act a bit funky and this is due to interrupts running while in simulation. Interrupts cannot be easily simulated - especially for timers. To see the IO toggle comment out the delay(1000) function calls and re-build and enter simulation. You should see the IO View change and also be able to step into the digitalWrite function.





At this point AVR Studio can be used to build and compile C++ projects for the Arduino. To download the generated binary file see some more IDE configurations are needed. A sample project with all the included Arduino core files and also the Rover mission dependency files can be found here.

Go to the AVR Studio integrated Arduino programming tutorial.

Go to the gcc-ar turorial.