1st assembler program in Arduino (A1a practice): Blink a led

1) Objectives (learning outcomes):

The main objectives of this first assembler programming practice are:

- Understanding the basic structure of an assembler program for Arduino (Atmel microprocessor)
- Learning how to use the development environment (Atmel Studio 6.1)
- Learning how to use Arduino's digital outputs (for example to **blink a LED once per second**)

This will be done using an **Arduino Mega2560** platform (based on the **Atmel ATMega2560 microprocessor**). The Arduino platform is little more than a board that allows interfacing with the processor, so the assembler programming will be very similar for other Arduino models (and also for other devices based on Atmel microprocessors). Atmel processors shares a common processor core (AVR) so most of the names and documents uses the name **AVR** instead of Atmel.

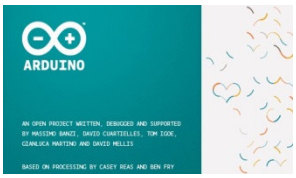
2) Installing the software

There are many different ways to write assembler programs and then upload them to the Arduino platform. The one requiring minimal software requires only any text editor (such as Notepad, Gedit, emacs, etc) and some of the *AVRtoolchain* or *WinAVR* utilities to translate the program to machine code (binary or hex program) and then uploading it to the Arduino's processor.

However, when writing assembler programs it can be difficult to find out why the code is not doing what it's supposed to do. There a ***debugger*** comes in handy. Also a ***simulator*** is a very useful tool for developing correct programs *before* uploading them to the Atmel processor. The **Atmel Studio 6.1** is a free IDE (*Integrated Development Environment*) that includes all those functionalities using the same interface. It also allows programming Atmel processors using C and C++ languages. Its major drawback is that only the Windows version is available (you'll need a virtual machine to run it under other platforms).

You can download Atmel Studio 6.1 (AvrStudio61sp2net.exe) from the manufacturer's webpage. Just install it (can ask to install some Visual Studio runtime components so the installation could take some time to complete).

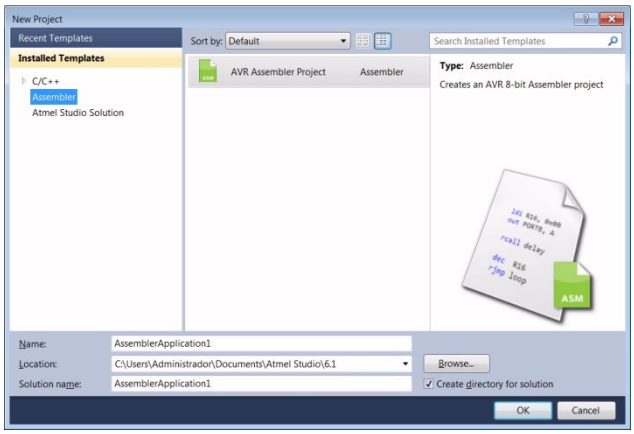
It's also recommended to install the **Arduino IDE** (open source). It's a [*too*] simple programming environment that allows writing and uploading C programs to the Arduino. It brings along several C examples showing how to use the different capabilities of the Atmel processor, and also includes **WinAVR** (that's why we need it; you can also forget of Arduino IDE and download just WinAVR from other source).



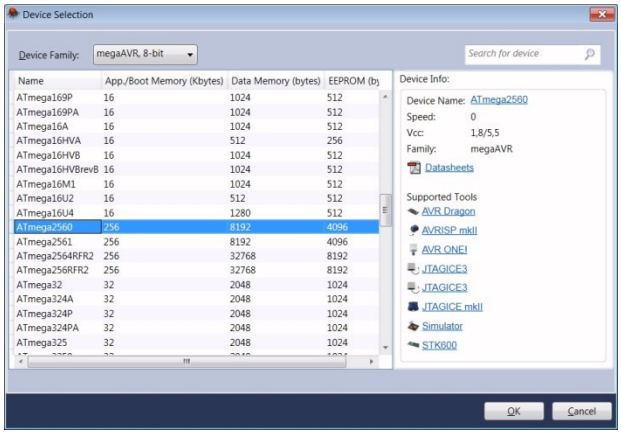
3) Starting a new assembler program in Atmel Studio

Once installed, Atmel Studio is ready to work.

1- Open it and select "New Project" from the menu.
2- Select the template "Assembler", then "AVR Assembler Project"
3- In the lower part of the window, give a name to the project (such as *blink_in_asm*) and choose the destination directory. It's very difficult to change it once the project is started so choose a suitable location from the beginning. Then click "ok".



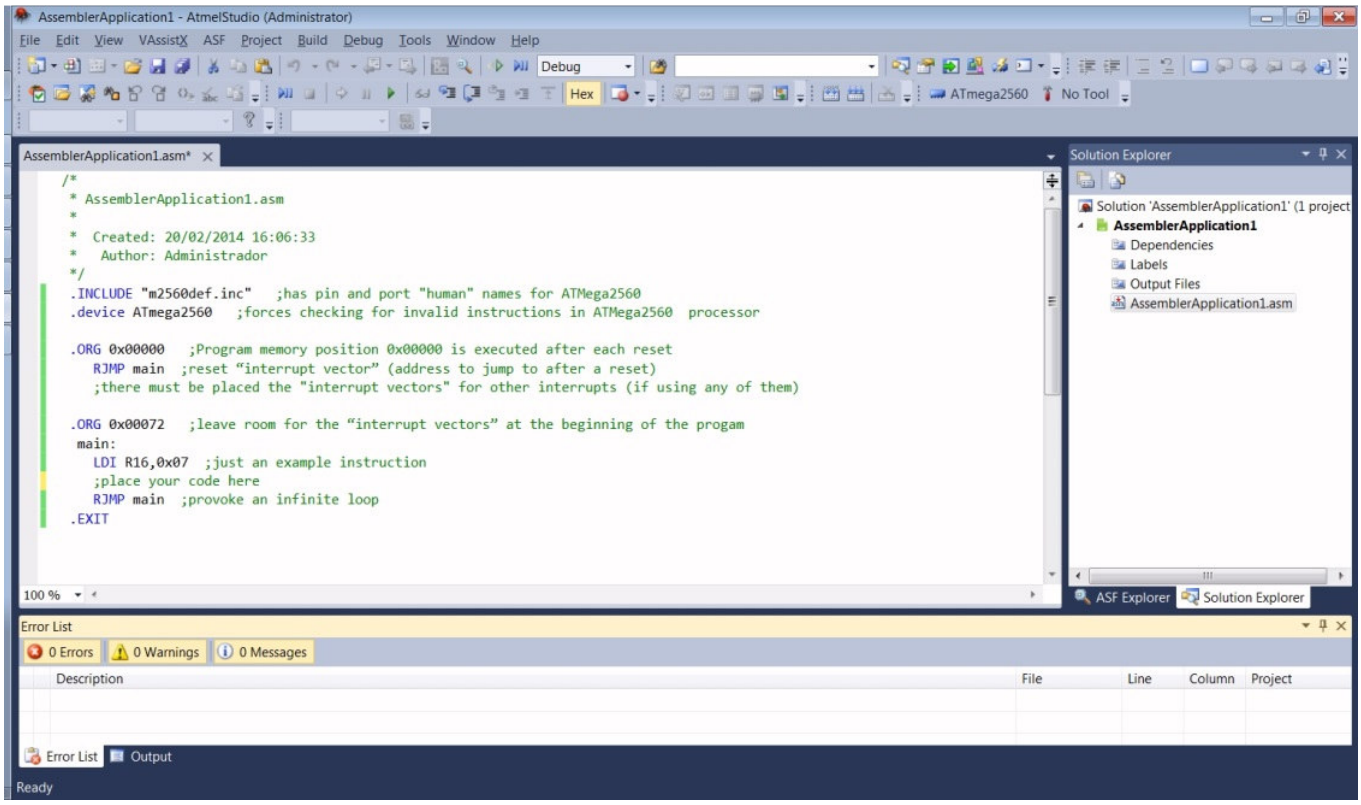
4- In the next window, select the processor we are going to use (the one into our Arduino, the ***ATmega2560*** model). Once selected you get the option of downloading the datasheets for this model (if you hadn't done it yet, that's the moment!). Once selected, click "ok".



5- Then a new project is created, containing a source file with the name you gave to the project (say *blink_in_asm*) and extension ***.asm***. Atmel Studio will automatically open any file with extension .asm (unless you change this setting).

6- Source file *blink_in_asm.asm* appears empty except for some comments. Comments in this editor can be written in two ways:
   a. For several-line comments, between **/*** and ***/** characters
   b. For single-line comments, after the **;** character

7- It's recommended (good programming practice) to start any assembler program always with the initial directives that you can find in the classroom slides. Write them, your source file should appear like this:



As you can see, the editor puts different colours for comments (green), instructions (blue) and operands (black), that helps a lot in reading the code. Once you save the program, a green vertical bar appears to the left; when any instruction is modified, the bar changes to yellow so changes can be quickly located.

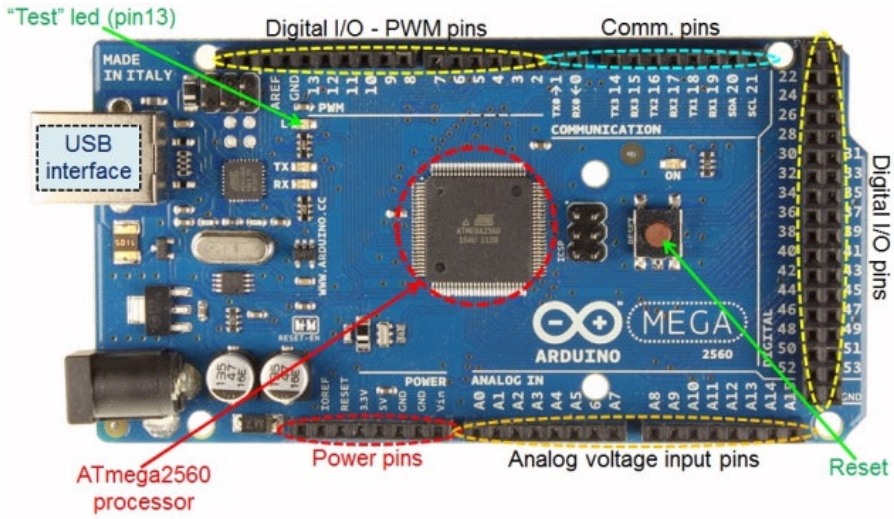
Note that the assembler translator **isn't case sensitive**: you can indistinctly type "LDI", "Ldi" or "ldi". That's also valid for labels and variable names: *my_label* is the same as *MY_Label* or *my_LABEL*

4) Understanding how to use an I/O PORT (for example to turn on or off a LED)

Do you remember from the classes that the ATmega2560 includes 11 input/output (I/O) ports to connect it to the world? Then you can use any bit from any of those ports to send a 0 or a 1 "to the world" by connecting other device to the pin corresponding to the selected port and pin.

Writing a 0 to an output bit in a port will put the corresponding pin to 0 volts. Writing a 1, will force 5 volts in that pin. And this can be used for any task, for example turning on or off a LED.

Although any port can be used for I/O tasks, PORT B has better driving capacity, in other words, it can give more current to the device connected to it, than other ports. Thus the Arduino platform includes an ***on-board "Test" LED*** (that programmers can use at our convenience) connected to one of PORT B's bits:

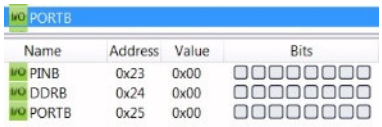


This **on-board LED is connected to <u>PORT B's bit 7</u>** in the Arduino Mega2560; the same bit is connected to Arduino's output pin 13 (so an external LED can be also connected). . Some words of warning: in older Arduino models, the on-board LED was traditionally connected to *PORT B's bit 5*, so check that for your Arduino model before using this LED.

You can find the full Arduino Mega2560 *pinout* in the classroom slides for Unit 1, page 24.

<u>4.1) I/O ports handling</u>

Before using any port, you need to understand how they work in the Atmel processors. Each I/O port '**p**' is handled using 3 registers: **PORTp, DDRp** and **PINp**. For example port B is handled using the PORTB, DDRB and PINB registers:



- **DDRp** (Data Direction Register *p*) is used to select the data **<u>direction</u>** (input or output) for each of the port *p* pins:
  - Writing a 0 value to bit *n* means that *port p, bit n* is going to be used as a <u>digital input</u> (so it will accept 0/5V voltages FROM "the world")
  - Writing a 1 value to bit *n* means that *port p, bit n* is going to be used as a <u>digital output</u> (so it will force 0/5V voltages TO "the world")
  - Example: *OUT DDRB,R17 ; say that R17=1001011b: then this* will configure port B pins 0, 1, 3 and 7 as outputs, pins 2,4,5 and 6 as inputs
- **PINp** (Port INput *p*) contains the bits which are ***physically connected to the I/O pins of the processor***. By **reading PINx**, the processor can read the present digital values at those pins:
  - For pins configured as *inputs*, the read values are the signals sent by "the world" to our processor
  - For pins configured as *outputs*, we'll just read back the values that our processor is sending to "the world" at that time
  - Example: *IN R16, PINB* ;now R16 contains the current values in the 8 pins of port B
- **PORTp** is the Data Register for port *p*. It can be used for two tasks:
  - For port pins configured as **outputs** in DDRp, writing a 1 (or a 0) to a bit will ***output*** a 1 (or a 0) through the corresponding pin (the value is transferred to the PINx bit).

- For port pins configured as inputs, writing a 1 (or a 0) allows us to select the "type of input" to use (with or without the internal *pull-up resistor*); don't worry about this now, we'll come back to this point in future practices.

Usually PINp is read to obtain the digital inputs' values (***IN Rd, PINp***), while PORTp is written for sending digital outputs' values (***OUT PORTp, Rr***).

Note that *IN* and *OUT* instructions are used to read/write the complete register (PORTp, DDRp or PINp) at a time. However you can use the SBI, CBI instructions to set or clear any individual bit (and thus a single pin).

In particular, in order to **manipulate the on-board LED**, firstly you have to configure PORT B's pin7 as **output** (by writing DDRB's bit 7 to 1). Then, writing a 1 to PORTB's bit 7 will output a 1 (5V) and the on-board LED will turn on, while writing a 0 will output a 0 (0V) and the on-board LED will turn off.

5) Timing the on-off switching using delays

The goal of the program is to turn on and off the on-board LED once per second (approximately). This will require doing some timing here. It can be done in two ways:

- Using one of the ***timers*** available in the Atmel processor. This allows the CPU execute other tasks while the time period expires, however it requires handling the timer interrupts, so we'll leave this for further practices
- Executing *delay* operations until the desired time has elapsed. This is a common approach for simple applications where the CPU does not need to do other tasks in the meanwhile.

The usual *delay loops* increase (or decrease) one or more registers, acting as *counters*, until a certain value is reached. To estimate how many INC/DEC operations we need to do, we must take into account that the ATmega2560 clock runs by default at a frequency of 16MHz (in other words, there are 16M clock ticks per second). A single delay loop with an 8-bit register that is increased from 0 to 256 (256=back to 0 again) will use 256 increments, 256 comparisons and 256 conditional jumps. Each increment, and comparison are executed in 1 clock tick each, plus 2 ticks to execute the jump, so the full loop will take 4*256=1024 ticks (requiring 0.064 milliseconds).

Thus, you'll need to use ***nested delays*** in order to get a 1 second delay. Each time that the first register reaches 256, a second register is increased in one unit. The second register will then take (1024+4)*256= 263168 clock ticks to run from 0 to 256 (taking into account the corresponding increment, comparison and jump operations), taking now 16.45 milliseconds. Finally a third register is needed, increasing it in one unit each time that the second register reaches the value 256. In order to complete 1 second, the third register needs to be increased $1/0.016 \approx 60$ times.

This calculation is not exact, because the loop's jump takes only 1 clock ticks to complete when the jump is not executed (that happens once out of 256 iterations). However we don't need to be very exact for this application, so it should do the work.

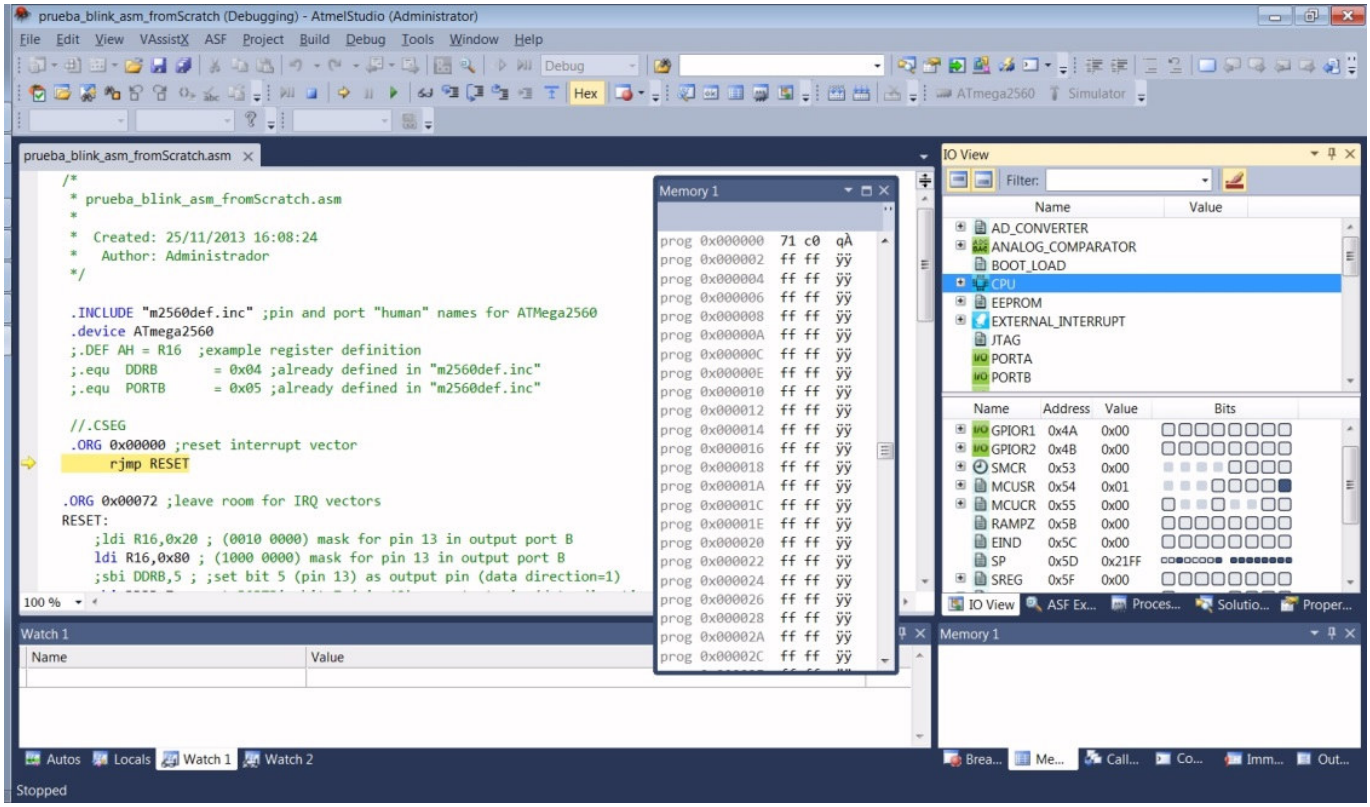
6) Writing, compiling and simulating/debugging our program

Once the program is written, run from the menu ***Build→Build solution*** (or hit F7). The assembler translator will show you any error in the program you're writing (typically trying to use an inexistent instruction or label). If everything is ok, the assembler translator will generate a **binary file** containing the machine code for your program. It will have the same name as your source file but ***.hex*** extension (so those files are commonly referred to as "hex files"). This file is ready to be uploaded to the Atmel processor inside the Arduino.

However the program can be correct from a syntactical point of view, and still operate in a wrong way. If you upload the program to the processor, run it, and don't get the expected results, how can you find what's wrong?

Before attempting to upload the program to the processor, it's recommended that you ***simulate*** its operation just to be sure it's working as you expect to and, if don't, find why and correct it. This can be done using the Atmel Studio ***debugger***.

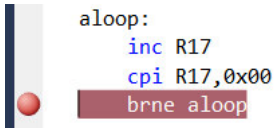
Once your .hex file is ready, start the debugger using the menu ***Debug→Start debugging and Break*** (or hitting the Alt+F5 keyboard shortcut). This will start simulating the execution of the program in the ATmega2560 CPU, stopping just before executing the first instruction (it must be a RJMP instruction at program address 0x000000).



The "I/O View" panel appears at the right side of the window, showing the contents of the different parts of the Atmel processor (ports, CPU registers, etc.). Another window allows exploring the contents of the different memories: FLASH program memory, SRAM data memory (R0 to R32 are the 32 first positions of it), EEPROM memory.

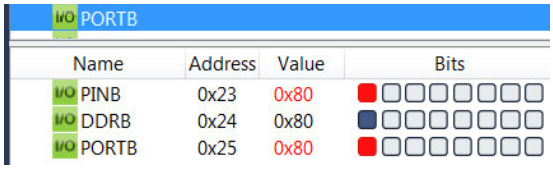
Once there you can control the debugging of the program using:

- F11 (**step into**) or F10 (**step over**) to execute the next instruction and stop again. If the instruction were a function CALL, F10 (step over) would execute it in a single step (stopping again after returning), while F11 (step into) would enter the function and stop in its first instruction.
- Clicking at the left of one instruction will set (or remove) a BREAKPOINT, indicated with a red circle. This will provoke the program to stop when it reaches the breakpoint. You can use F5 (**continue**) to execute the program without stopping until a breakpoint is reached.



- Selecting *Debug→Run to cursor* (Ctrl+F10) will execute the program until the line where the cursor is.

Once you have checked that the program is writing 1 and 0 the correct bit for the on-board LED (bit 7 in PORT B), it should work fine. You're ready to upload it to the Arduino!



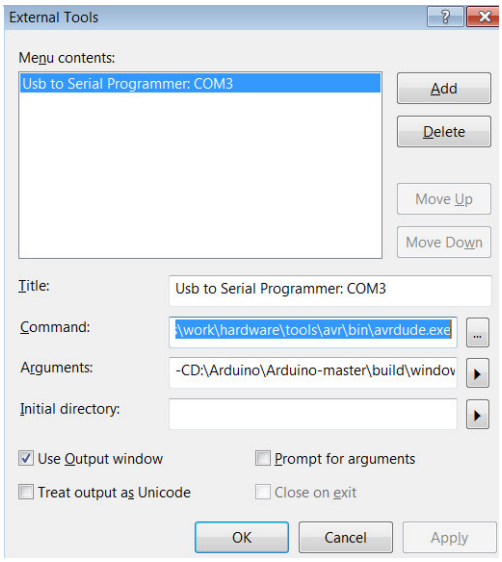
7) Uploading the assembled program to the Arduino

When you connect the Arduino board to any of your computer's USB ports (and after windows finishes doing its obscure things), a new serial port will appear in your computer representing this connection. Usually it will be named "**COM3**", you can review your hardware just to see if it's there. This will be the port you have to use to "talk" with the Arduino.

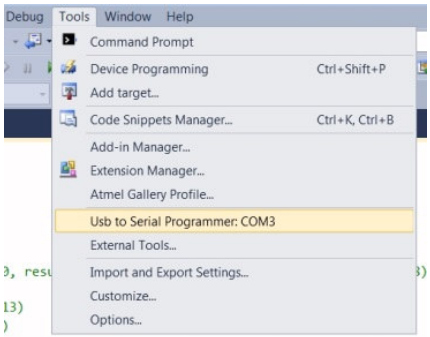
In order to upload your binary file (*blink_in_asm.hex*) to the program memory of the Atmel processor, you can use the **AVRdude.exe** utility. This is a command-line utility, however you can integrate it into the Atmel Studio IDE following those steps to set AVRdude as an external tool to upload de program to the Arduino board:

From the menu *Tools→External Tools*, in the window that will open:

- put a "Title" for the new tool (ex: Usb to Serial Programmer: COM3)
- In "Command", write the full path to the ***AVRdude.exe*** executable. You can search for that file, however it will be located in the folder where you installed Arduino IDE (or the AVR tools), something like this:
  [your_path_to_AVRdude.exe_here]\avrdude.exe
- In Arguments: write the following expression (substituting the full path to the file "***avrdude.conf***" with yours):
  -C***[your_path_to_AVRdude.conf_here]***\avrdude.conf -v -v -patmega2560 -cwiring -P\\.\COM3 -b115200 -D -V -Uflash:w:"$(ProjectDir)Debug\$(TargetName).hex":i



And that's all. You only need to do this once, from now on you can upload any program directly from the Tools menu; here a new entry with the name you gave to the tool will appear. By selecting it, your .hex program will be uploaded to the Arduino.



The program will start running automatically each time you power-up the Arduino, and it will be restarted whenever you click the Arduino's "reset" button. If everything is correct, you should see the on-board LED blinking on and off each second.