Introduction to Micro-controllers

Andres LaRosa and Bret Comnes

Portland State University

February 5, 2013

1 Introduction

Microcontrollers are small computers designed to go where desktop computers dare not go. They come in all shapes, sizes, and layouts. Usually, they are quite small and use less power than traditional computers. Microcontrollers are often deployed in an 'appliances' and serve an unmodifiable dedicated purpose, such as keeping track of what spin cycle your washing machine is on, or how much time is left before it should turn off your microwave oven. Make no mistake however, these are general purpose computers. The other major difference between a microcontroller and traditional computers is that they they come with an array of analog and digital input and outputs. These inputs and outputs can be used to read environmental data from sensors, talk to other computers or devices and electronically control other systems which provide environmental outputs such as a LCD screen, mechanical switches or servo motors etc. [8]

Getting started with microcontrollers can be tedious process, as they can require a number of supporting circuits, USB controllers, programmers, boot-loaders and power supplies just to load your first program onto the microcontroller chip. Often times you will start with a prototyping board, which puts all of the necessary components in a convenient ready to use package.

1.1 Arduino

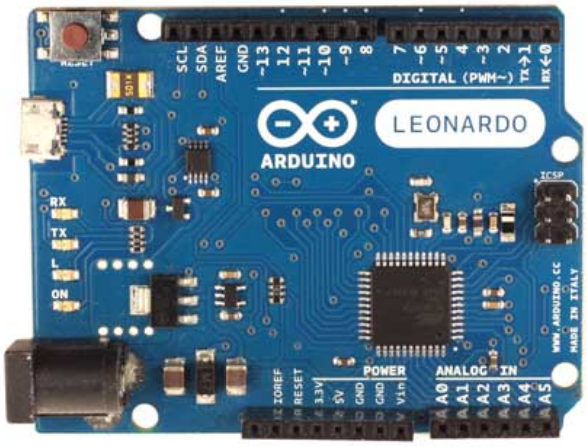

Figure 1: Arduino Leonardo[1]

> *"Arduino is a tool for making computers that can sense and control more of the physical world than your desktop computer. It's an open-source physical computing platform based on a simple microcontroller board, and a development environment for writing software for the board."* – Arduino.cc[3]

This lab will be using the Arduino Leonardo Microcontroller[1]. It is similar to the Arduino Uno[2], with the major difference being that it uses SMT[12] instead of the older "thru-hole"[13] technology in order to reduce cost.

Arduino drastically lowers the difficulty of getting started with a microcontroller (compared to to plain ATMEGA/PIC/ARM chips), as it provides all the necessary tools to start making the microcontroller do interesting things without nearly all the setup of just a plain microcontroller chip.

Arduino is based around an 8-bit Atmel AVR microcontroller, and has supporting systems like a boot loader for uploading programs, a USB controller as well as a barrel jack for external power.

It is programmed using a language that is based off of C++ and uses a fork of the Processing IDE used for writing, compiling and uploading your programs to the board. Many Arduino projects you will find will rely on programs running on your computer using Processing, but can interface with any serial enabled programs.[10]

2 Getting Started

2.1 Setting up your software

This lab is based off of the Arduino 1.0.3 software which can be downloaded for free from the Arduino website.[7] Unlike other embedded systems development environments, the Arduino software is quick to download and set up, and has zero cost associated with the software which makes it a convenient to work with when your primary goal is to come up with a working prototype quickly and cheaply.

It also has a large community and a massive pool of example programs and libraries.

2.2 Find a Computer

You are free to use your own laptop or one of the classroom computers. Plug your Arduino into the computer using the micro USB cable. **Please be careful with the delicate connectors.**

2.3 Download and Launch the Arduino Software

Visit `http://arduino.cc/en/Main/Software` and downloaded the latest Arduino software[7]. If you have decided to use your own Arduino, make sure you find the necessary USB drivers if you have a board that is older than the Uno.

2.4 Selecting the Board

Once the IDE is open, navigate to the toolbar and select the board you are working with from **Toolbar → Tools → Board → Arduino Leonardo**. If you are using a different board , select the one you have from this list instead. This step may vary from system to system as well, but pick the option most similar to this write up.

2.5 Selecting a Serial Port

This step varies from system to system. In this step, we tell the computer which serial port that the Arduino chip can be reached at, for programing the board, as well as talking to it during runtime.

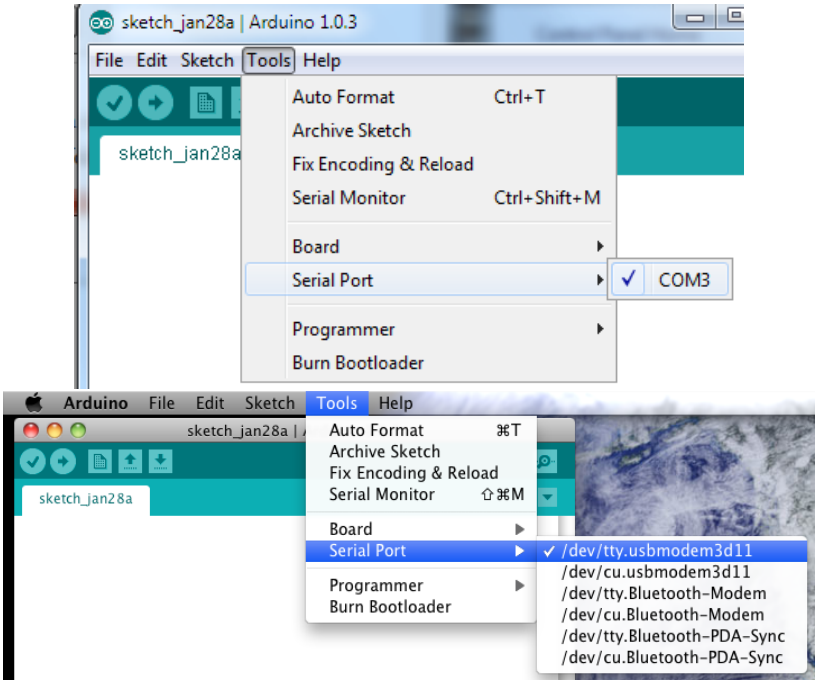

Figure 2: Port Selection in Windows and OS X

2.5.1 Windows

Select **Toolbar → Tools → Serial Port → COM3** where COM3 is the serial port that has been assigned to your Arduino by windows. You should only have one port available.

2.5.2 OS X

Select **Toolbar → Tools → Serial Port → /dev/tty.usbmodem3d11** where COM3 is the serial port that has been assigned to your Arduino by windows. You should only have one port available.

2.6 Uploading your first program

Next we will open an example program, verify that it compiles, then upload it to our board.

Navigate to **Toolbar → File → Examples → 01. Basics → Blink**. Press the verify button. It should compile the sketch and return a 'Done compiling' message. If you get an error, something went wrong.

If that completed, go ahead and upload the program to the board by pressing the upload button (the button right next to verify button) and it should provide a similar completion message after a few moments. The LEDs on the Arduino will blink during the upload, but should settle down after a few seconds. See Figure 3.

Once the program you uploaded is running, the LED labeled 'L' on your Arduino should be slowly blinking. This LED labeled 'L' is wired to Pin 13 on the Arduino, a digital pin with a

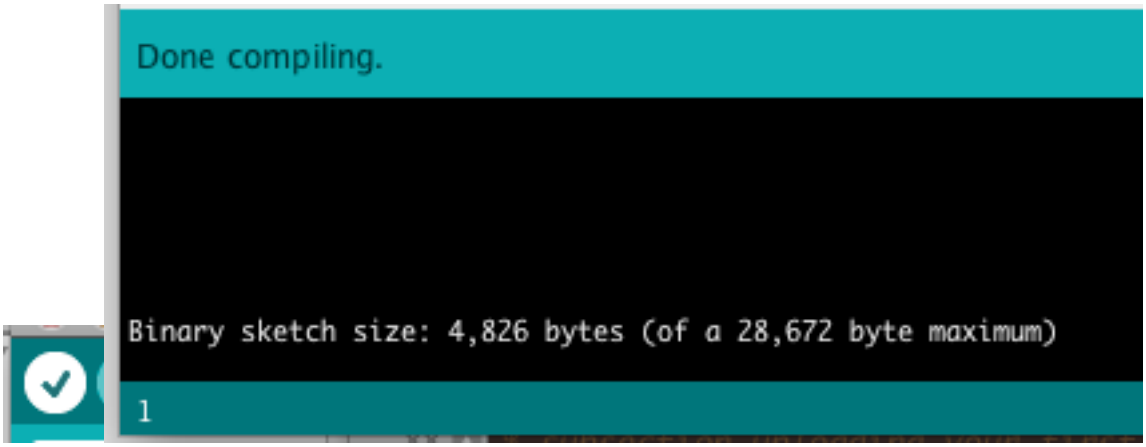

Figure 3: Verify Button (Left) Compile Success Message (Right)

resistor built in that so that LEDs can be added easily. **Go ahead and add an LED between Pin 13 and GND**. It should blink at the same rate as the 'L' LED on the board.

Congratulations! You now have a working Arduino that is talking to the Arduino IDE.

3 Programming Arduino

Arduino is based off the Processing[10] programming language, and has some similarities to C, however much of the language has been simplified from C. In this section we will go over the basics of the language, look at some simple examples of code and even write some of our own.

3.1 The Bare Minimum

The bare minimum code you need for an Arduino program is presented in Figure 4.

```
1 void setup() {
2 // put your setup code here, to run once:
3 }
4 void loop() {
5 // put your main code here, to run repeatedly:
6 }
```

Figure 4: The minimum Amount of code for an Arduino Program

There are two parts to this minimum program, the `void setup() {}` section and the `void loop() {}` section. When your program runs, It starts executing your code, line by line, starting in the `void setup()` section, inside the brackets, `{}`, that follow. Your program will execute any code that is in this section, and when it gets to the end, will begin executing the code inside `void loop() {}`, until it gets to the end of the available instructions, at which point, the program starts back over in at the beginning of `void loop() {}`, retaining any variables or settings from prior lines of code.

Lets look at a simple example, that you should already have pulled up, the `Blink` program, which is found in Figure 5. If you still need to open this, refer to Section 2.6.

4 Understanding the Blink Program

```
1  /*
2    Blink
3    Turns on an LED on for one second, then off for one second, repeatedly.
4
5    This example code is in the public domain.
6   */
7
8  // Pin 13 has an LED connected on most Arduino boards.
9  // give it a name:
10 int led = 13;
11
12 // the setup routine runs once when you press reset:
13 void setup() {
14   // initialize the digital pin as an output.
15   pinMode(led, OUTPUT);
16 }
17
18 // the loop routine runs over and over again forever:
19 void loop() {
20   digitalWrite(led, HIGH);   // turn the LED on (HIGH is the voltage level)
21   delay(1000);               // wait for a second
22   digitalWrite(led, LOW);    // turn the LED off by making the voltage LOW
23   delay(1000);               // wait for a second
24 }
```

Figure 5: The Blink Program in all its glory

If a block of text is wrapped in `/* */`, such as lines 1 thru 6 in Figure 5 or has a `//` in front of it on one line, such as line 12, means it's a comment. Comments are little notes you leave in your code. They are not executed or interpreted by your program in any way. It is good practice to add comments to your code. They can help you think about your program, and will also remind you, and others that see your code, what the program does or how it works.

Running down lines 1-9, we see that this is all comments describing the function of the program, how it works, as well as the license. The first piece of code we see is on line 10.

4.1 Variables

```
int led = 13;
```

The first thing to notice, is that this code is not inside `void setup()` or `void loop()`. That is because it is a variable and variables can be declared in the beginning of our program outside either loop.

Variables are incredibly useful tools. Variables store information that can be used later in the program as many times as you need. They can be updated during run-time and cab be used to store values temporarily for repeated use.

Any variable we decide to use, we have to describe to our program. This is referred to as declaring your variable. **Variables are declared before your `void setup()` or `void loop()` sections.**

Line 10 declares a variable named `led`, gives it a variable type of `int`, for integer, and then assigns it a value of 13. This variable is used to reference the physical pin we will be using in our program. It is used as an abstraction layer, so that if we ever go back and change which pin we want to use, we can update all the places in our program that reference this pin number simply by updating the initial variable value.

The basic syntax of a variable declaration is:

```
type variable_name = value;
```

The available variable types can be found on the Arduino website [6]

4.2 Pin Modes

The next piece of code we arrive at is our `void setup()`. Stepping inside the curly braces of this structure, we come to the following line:

```
pinMode(led, OUTPUT);
```

Digital pins can either read or write digital signals (0 or $5v$). Before you do either, you must tell your program what you will be using the digital pin for. This tells our program that the pin associated with the `led` variable will be a Digital output.

```
pinMode([PIN-NUMBER], [PIN-TYPE]);
```

Available pin modes can be found a the Arduino website.[5]

Each pin has a name assigned to it. Each pin on the Arduino has its name printed next to it, which can be seen in Figure 1. There are two primary types of pins: Digital and Analog. Digital pins simply have a number for a name, and analog pins have the letter A followed by a number for a name.

Digital pins can read or write digital signals varying from 0 to $5v$. Some can also output PWM signals, a way to emulate analog voltage signals with a digital pin PWM capable digital pins have a ~ printed next to their name on the board. See Section 6.1 for more information.

Analog Pins can read in analog voltages between 0v and 5v and converts them to a value that your program can use between 0 and 1023. They do not need a pin mode set before reading within your program.

4.3 Generating Output

Once the pin mode is set, we exit our `void setup()` and enter our `void loop()`, the part of the program that will run over and over in an infinite loop.

The first thing we do is execute:

```
digitalWrite(led, HIGH);
```

`digitalWrite([PIN], [VALUE])` lets us set the output value of a digital pin that has been set to an OUTPUT type. In this case, we write a value of `HIGH`, or $5v$, to the pin referred to by the `led` variable. A value of `HIGH` would refer to $0v$.

The next part of the program tells the Arduino to wait for a period, before executing the next line of code. **This period is equal to 1000ms, or 1 second.**

```
delay(1000);
```

After waiting for a second, we then write a value of `LOW` to our `led` pin, and then wait 1 more second.

```
digitalWrite(led, LOW);

delay(1000);
```

At this point, there is no more code left in our program, so it starts executing `void loop()` again.

Congratulations, you now should have some basic understanding of how an Arduino program is written.

5 Modifying the Blink Program

Now you will try your hand at modifying the blink program. What we are going to do is define a new variable called `wait`, give it a value, and then replace the delay time on the Arduino with our new variable.

5.0.1 Create a new variable

Right below the `led` variable declaration, add a new variable named `wait` of type `int` and give it a reasonable value different than 1000 (like 100). Also, add a comment describing what this variable is used for.

5.0.2 Use your new variable

We want to use this new variable to declare the time we wait in between turning our LED on and off. Go ahead and replace the old delay values with your new variable name.

5.0.3 Verify and Upload your modified program

The Arduino should still be set up from when you first uploaded the first blink program. Verify your new program to see if it compiles. If you get an error, check your work for syntax error. Did you forget a semicolon or a brace?

Once your program verifies, and you are able to upload it to your board, you should start to see your LED blink faster or slower, depending on the value you defined your variable.

5.0.4 Final modification

Once you make your modification, you will have code that looks similar to Figure 6.

6 Using Inputs to Control Outputs

6.1 Understanding PWM

Arduino cannot output true analog signals. It can, however, generate Pulse Width Modulation (PWM) signals, which are similar to analog signals and can sometimes work interchangeably.

> *"Pulse-width modulation uses a rectangular pulse wave whose pulse width is modulated resulting in the variation of the average value of the waveform"'* – Wikipedia[11]

Basically, if driving an LED with a PWM signal, you are able to vary the brightness of the LED by having it blink really fast while varying the duration of each blink, resulting a brighter or dimmer output. The result is that you can generate signals with similar properties as analog signals using pins only capable of switching between 0 and $5v$.

```
int led = 13;
int wait = 100; // Time to wait before blinking

void setup() {
  pinMode(led, OUTPUT);
}

void loop() {
  digitalWrite(led, HIGH);
  delay(wait);  //Wait for the ammount declaired in the wait variable
  digitalWrite(led, LOW);
  delay(wait);  //Wait for the ammount declaired in the wait variable
}
```

Figure 6: The modified Blink Program

6.1.1 Observing PWM

Load the **01.Basic** → **Fade** example program and upload it to your Arduino Board, and wire up the LED to the pin that is used in the program (this requires reading the program). If it is a digital pin other than pin 13, you will need to add a 330Ω resistor in series for good measure. If you increase the delay time to $\tilde{8}0$, you should be able to observe the PWM flicker at lower brightness levels.

Hook up your function generator to observe PWM output. You can leave the LED in place. Observe what the signal looks like on your oscilloscope.

6.2 Controlling output with a Potentiometer

Next, hook up a 10k potentiometer between the 5v and ground and middle leg to the A0 analog input on your Arduino as seen in Figure 7. Modify your **Fade** code to the code in Figure 8

6.2.1 Understanding the changes to Fade

We declared a few new variables to keep track of an analog input pin and variables used for storing values from our analog input readings and our PWM output values. See the comments in the code for more context.

We also used the `Serial.begin(9600);` line in our `void setup()` loop to tell the Arduino that we want to open a serial communication session with our computer (remember how we picked a serial port in Section 2.5?).

Outputting data to the serial line is a nice way to tell what is going on in your code, but remember that your program will run no faster than the speed of your serial line.

The next new piece of code is `analogRead(pot);`. This does what it sounds like. It reads the 0-5v input on pin `pot` and converts it to a value between 0 and 1023.

We can only output PWM values analogous to 0 and 5v by writing a value between 0 and 255 to our led pin. Lucky we have a useful command called `map ()` which handles the analog input range to PWM output range conversion for us. See the `map` info page for more details.[4]

Next we write the adjusted input value to our `led` pin using `analogWrite(led, outputValue);`.

Finally we print these input and output values the the serial line using the `Serial.print(value);` command. **Open the serial monitor now to view these values in real time. Go to *Tools* → *Serial Monitor***. You should see the input and output values similar to Figure 9.

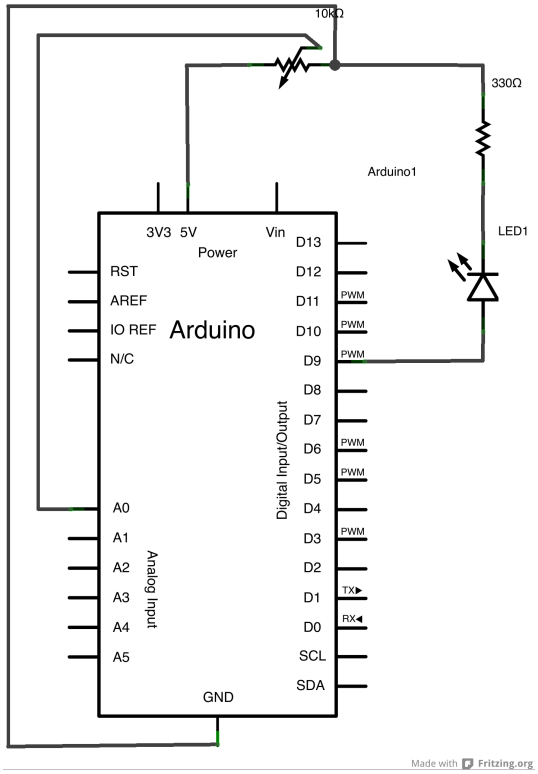

Figure 7: Potentiometer and LED wired up to the Arduino

```
const int led = 9;       // the pin that the LED is attached to
const int pot = A0;      // A0 will be the analog input channel
int sensorValue = 0;     // this will store the value read from the pot
int outputValue = 0;     // this is the value sent to our pwm pin

void setup()  {
  pinMode(led, OUTPUT); // declare pin 9 to be an output
  Serial.begin(9600); // Open a serial monitor at 9600 baud
}
void loop()  {
  sensorValue = analogRead(pot); // store pot value in sensorValue
  outputValue = map(sensorValue, 0, 1023, 0, 255); // map sensorValue to correct range
  analogWrite(led, outputValue); // write the analog out value to led:

  Serial.print("sensor = " ); // print the results to the serial monitor:
  Serial.print(sensorValue);
  Serial.print("\t output = ");
  Serial.println(outputValue);

  delay(2);  // wait 2 milliseconds for daq to settle
}
```

Figure 8: The blink program modified to control the LED with a Potentiometer

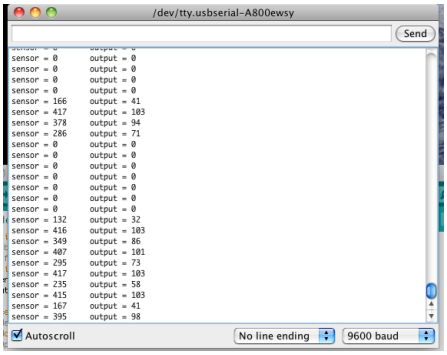

Figure 9: The serial monitor in action

6.3 Photo Resistor

Remove the potentiometer and wire in a photo-resistor as seen in Figure 10. A photo-resistor has a variable resistance depending on how much light is hitting it.

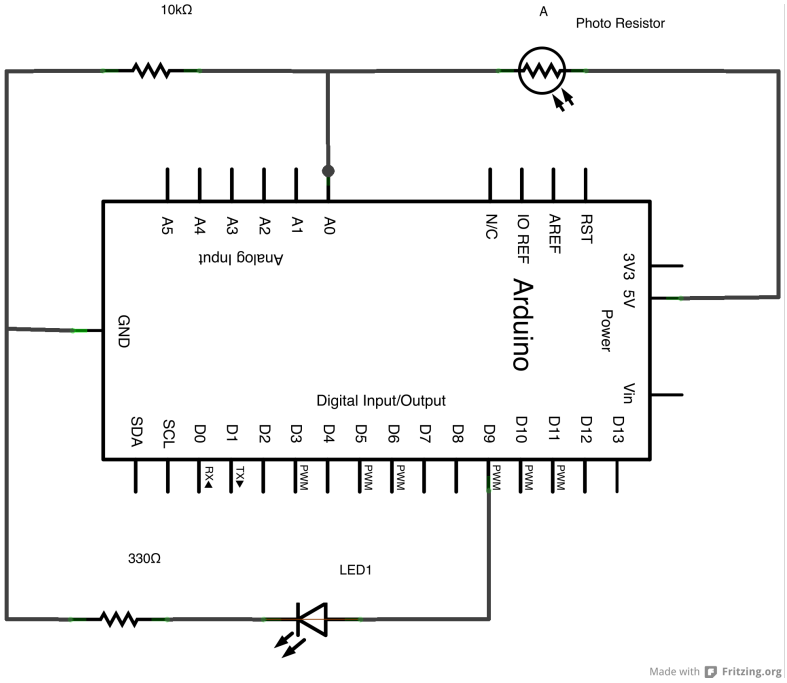

Figure 10: Photo-resistor wired to A0

Using the same code as in Section 6.2, observe the output of the photo resistor and the PWM output on pin 9 simultaneously on your oscilloscope. **Vary the amount of light hitting the photo-resistor and observe and compare the PWM output and voltage across the photo-resistor.**

7 PID and LEDs

PID (Proportional, Integral, Differential) is way to control a system and take into account any kind error in your control mechanism. There are there primary components to think about in a PID control loop. You have voltage corresponding to the current state of your system (position, temperature, etc) that is called your "Process Variable" or PV. You also have a setpoint (SP) voltage, corresponding to the state you wish your PV to reach. Third, you have a control

voltage, u, which corresponds to the instantaneous voltage value you are using to drive your system towards its SP voltage.

The PID algorithm is show in Equation 1.

$$u(t) = MV(t) = K_p e(t) + K_i \int_0^t e(\tau)d\tau + K_d \frac{d}{dt} e(t) \tag{1}$$

There is a proportional, integral and differential part to Equation 1. The constants K_p, K_i, and K_d are used to set the sign and contribution gain of each part of this equation. $e(t)$ is your proportional "error" corresponding to $SP - PV$. The variable t corresponds to the current time in our system, and τ is simply a variable of integration.

The proportional portion of the equation takes into account how far away our PV is from our SP. The differential part takes into account how fast we are moving (if we move to fast near our SP, we will over shoot), and can be used to reduce the proportional portion if we are moving to fast, or speed us up if we are experiencing resistance despite our proportional contribution.

The integral part of the equation takes into account how long we have been off of the set point, contributing more to our output the longer we are missing the SP. This is important because our P and D contributions will typically lead our PV to sag slightly above or below our SP variable.[9]

7.1 Installing Libraries

7.1.1 Windows

7.1.2 OS X

7.2 Using the PID Library

8 PID Temperature Control

Controlling the temperature using PID and a fan.

9 PID Extras

9.1 Implementing your own PID algorithm

9.2 Auto-tuning PID

10 Communicating With other Devices

Opening serial ports and talking to humans and computers.

References

[1] Arduino - arduinoboardleonardo. `http://arduino.cc/en/Main/ArduinoBoardLeonardo`.

[2] Arduino - arduinoboarduno. `http://arduino.cc/en/Main/ArduinoBoardUno`.

[3] Arduino - introduction. `http://arduino.cc/en/Guide/Introduction`.

[4] Arduino - map. `http://arduino.cc/en/Reference/Map`.

[5] Arduino - pinmode. `http://arduino.cc/en/Reference/PinMode`.

[6] Arduino - reference. `http://arduino.cc/en/Reference/HomePage`.

[7] Arduino - software. http://arduino.cc/en/Main/Software.

[8] Microcontroller - wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Microcontroller.

[9] Pid controller - wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/PID_controller.

[10] Processing.org. http://processing.org/.

[11] Pulse-width modulation - wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Pulse-width_modulation.

[12] Surface-mount technology - wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Surface-mount_technology.

[13] Through-hole technology - wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Through-hole_technology.

[14] Marshal Colville. Process control with a microcontroller pwm output, pid control, and hardware implementation. 2012.