Microcontrollers I

Introduction to Microcontrollers

Michael Chang • Devan Lai • Richard Lin • Adam Resnick

**Application**: Our entire car is based around microcontrollers called BRAINs. These BRAINS are based heavily on these Arduinos. Programming for the car is very similar to programming these BRAINs. Plus microcontrollers are cool!

Contents

[Revision History 3](#_Toc335864752)

[Parts List 4](#_Toc335864753)

[Introduction 5](#_Toc335864754)

[Microcontrollers - what are they? 5](#_Toc335864755)

[Where is it used? 5](#_Toc335864756)

[Lab Introduction 5](#_Toc335864757)

[Safety Briefing 5](#_Toc335864758)

[Installation Instructions 6](#_Toc335864759)

[Editor’s Note 6](#_Toc335864760)

[Lab 1: Hello, World! 7](#_Toc335864761)

[Objectives 7](#_Toc335864762)

[Lab Manual 7](#_Toc335864763)

[Lab 2: Blinky 10](#_Toc335864764)

[Objectives 10](#_Toc335864765)

[Blinking the LED 10](#_Toc335864766)

[A Slightly Different Implementation 12](#_Toc335864767)

[Lab 3: Controlling LEDs with Buttons 14](#_Toc335864768)

[Objectives 14](#_Toc335864769)

[Lab Manual 14](#_Toc335864770)

[Lab 4: Reading from an Analog Pin 16](#_Toc335864771)

[Lab 5: Cycling Colors on an RGB LED 17](#_Toc335864772)

[Objectives 17](#_Toc335864773)

[A Little Bit of Physics 17](#_Toc335864774)

[“while” and “for” Loops Refresher 17](#_Toc335864775)

[A Short Example 17](#_Toc335864776)

[A Shorter Example 18](#_Toc335864777)

[Less talk, more pretty lights! 19](#_Toc335864778)

[Extra for Experts: Cooperative Multitasking 22](#_Toc335864779)

[Objectives 22](#_Toc335864780)

[A First Attempt 22](#_Toc335864781)

[So, what happened? 24](#_Toc335864782)

[Cooperative Multitasking 24](#_Toc335864783)

[Multitasking LEDs 25](#_Toc335864784)

[Extra for Experts for Experts: Linearly Varying Brightness 27](#_Toc335864785)

[Other Microcontrollers 28](#_Toc335864786)

# Revision History

|  |  |  |
| --- | --- | --- |
| **Version** | **Date** | **Changelist** |
| 1.0 | Fall 2011 | Initial release |
| 1.1 | 20 Sept 2012 | Major reformatting  Fixed for / while flowcharts  Minor bugfixes |
| 1.2 | 23 Sept 2012 | Added instructions for Mac OS X  Added BRAIN notice  Added pinouts for potentiometer and switch |
| 1.3 | 26 Sept 2013 | Added git tutorial  Lab clarifications |

# Licensing

Creative Commons License

This document (including source elements, like diagrams), is licensed under the Creative Commons Attribution-ShareAlike 3.0 Unported License (<http://creativecommons.org/licenses/by-sa/3.0/deed.en_US>).

While commercial use is not prohibited under the terms of the license (the drawbacks outweigh the benefits, nor is a solar car team going to start suing people for obscene amounts over copyright infringement), you shouldn’t be shamelessly trying to turn a profit off someone else’s work. What exactly this means is completely up to you – but please behave like responsible adults. Keep in mind that you must license any derivative works under a similar license – which gives recipients redistribution rights.

# Parts List

|  |  |  |  |
| --- | --- | --- | --- |
| **Item** | **Image** | **Function** | **Notes** |
| **Breadboard** | C:\Users\Ducky\Desktop\calsol-training\ee1\images\breadboard.png | Prototyping work area |  |
| **Arduino** | Image from Wikipedia, public domain | Microcontroller | Cheap. This one is 35$, but other microcontrollers can be anywhere from 50 cents and up. You also need the Arduino Environment or related drivers to program it with a USB A->B cable |
| **Resistor**  **330 ohm** |  | Current limiting |  |
| **Resistor**  **180 ohm**  **x3** |  | Current limiting for RGB LEDs |  |
| **Switch**  **x2** |  | Digital input |  |
| **Potentiometer** |  | Analog input |  |
| **LED** |  | Pretty lights! |  |
| **RGB LED** |  | Make the rainbow! |  |

# Introduction

## Microcontrollers - what are they?

1. Small programmable computer in a very small chip. The chip to the right is as small as a typical thumb.
2. Enables you to take in signals about how the car is doing and react to them.
3. Easy to reprogram.
4. Like many chips, they are also easy to break – ESD (electrostatic discharge) can damage internal transistors, overcurrent faults can cause thermal damage, and the good old sledgehammer can completely pulverize it…

## Where is it used?

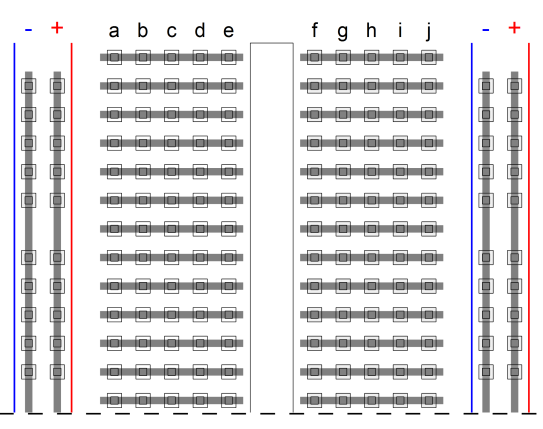
An example of a tiny chip.

(Image from Wikipedia, public domain)

1. Pretty much anywhere.
2. Even in mundane stuff: flashlights, toys, cameras, printers, routers, even vacuums, (and I could go on and on…)
3. Cars typically have over 80 microcontrollers on them – though ours only has around 10.
4. And of course, the typical smart devices: cellphones and even their accessories.

# Lab Introduction

## Safety Briefing

1. You are now dealing with more exposed wires which can be much more dangerous
   1. Never ever let the power and ground connections touch (this is known as a short circuit, and can damage components or even burn things).
   2. Never let wires hang around all over the place where they can accidentally touch
      1. Consequences include shorting the Arduino.
      2. Luckily, your computer should be able to shut off the Arduino on an overcurrent condition, but you don’t want to risk it! Be careful!
2. Reminder: a breadboard is internally connected like this:  
   
   1. The row connections across the middle are intended for components, and the column connections on the edges are intended for power connections.
   2. **Neatness counts a lot**, especially when you need to debug your circuit – so arrange your connections logically!

## Editor’s Note

This document was intended to be understandable to those who have never programmed before, and therefore contains detailed information about basic C language constructs like loops. The parts of this document were also written by 4 different people, without a great deal of coordination. So if you’re wondering why we explain for loops and while loops 4 different times… this is it – please bear with us…

## BRAIN Users:

If you are using a BRAIN instead of an Arduino, see the BRAIN addendum document on the training site: <http://mallard.calsol.berkeley.edu/training/content/ee3/>

**Also, note that the code here is Arduino-specific, so if you are using a BRAIN, you must use the BRAIN code on the website.**

# Pre-Lab: Installing Required Software

1. Download git here: <http://git-scm.com/downloads>
   1. The tutorial will cover using the command-line version, though you can use a GUI if you are familiar with one. We will be pulling code from a public GitHub repository.
2. Download the Arduino software here: <http://arduino.cc/en/Main/Software>
   1. Disable Bluetooth if your computer is making the Arduino environment slow.
   2. Make sure to select the right board and right serial port.
   3. Quick tip to find the right serial port: see what ports are there while Arduino unplugged. Then, plug it in, and the new port on the list should be for the Arduino.
3. Download device drivers for your platform at <http://www.ftdichip.com/Drivers/VCP.htm>
   1. Windows users: you may have to run the executable twice to get it to install.
   2. Mac users: you will need to restart your computer before continuing.

# Lab 0: Getting the Code

## Objectives

* Learn basic version control concepts
* Learn basic git concepts including commits, branches, and pushing / pulling

## Introduction to Version Control

**Source code** is the human-readable text form of software, and is what you will be working on in this lab in terms of programming the microcontroller. Since it’s text, you can open and modify it with any editor. However, it’s often desirable to keep previous versions, either as a reference, or if the new version turned out to be a bad idea. This is the main concept behind **version control** – a tool that manages file changes and keeps revisions.

**git** is a popular and powerful version control tool. The main concepts are:

* You have a **repository** (or **repo**), a database that keeps current and past version of all tracked files. For git, an entire copy of the repository is cloned onto your local machine (in contrast to some other systems, which only copy your working set)
* You edit files on your local machine as you normally would. Once you feel like you’ve sufficient progress, you **commit** your changes into the repository. This creates a new revision with the current files.
  + In git, commits only modify the local repository. You need to **push** the changes to the remote server so others can see them. Similarly, you need to **pull** changes from the remote server periodically so you can see changes others have made.
* If multiple people try to commit to the same file at once, the system will try to **merge** the changes intelligently. The most common algorithm is a **3-way merge**, where it takes the latest common version of the file, and tries to combine the changes you’ve made and the other person made.
  + A **merge conflict** results when the file cannot be automatically merged. You will need to edit the file and manually resolve the differences. The specifics of this process are outside the scope of this document.
* A **branch** is an independent line of development – it will not see the commits of other branches (unless explicitly merged), nor will its commits affect other branches. To branch is to fork the development stream at some point.
  + The **master** branch is usually the main branch.
  + One development methodology is **keeping master clean**. That is, the code in master is always working. Development of new features happens in separate branches, which get **merged** back into master when they’re complete.
  + When you’re on a branch, you can **pull** changes from upstream (merge new upstream changes since you’ve branched), preventing your branch from getting too out of sync with its parent.

## Lab Manual

1. Clone the training-code repository onto your local drive. In a terminal, enter the following command:This should create a new folder training-code with a copy of the training code repository, with the files on the master branch.

git clone https://github.com/CalSol/training-code.git

1. You can switch to another branch <branch> using the command:The relevant branches for the training repository are: master (Arduino code as provided in the lab manuals, so you don’t have to copypaste), brain (BRAIN version of the lab manual code), solutions (solutions to the Arduino code), and brain-solutions (self-explanatory).

git checkout <branch>

* 1. You should checkout the brain branch if you’re using the BRAIN.
  2. The solutions are provided if you get stuck, but please try to work through the problems first – you’ll learn more that way.
  3. To see a listing of all branches, you can do:

git branch

1. You can commit all your changes using:  
   The -a means “commit all modified files”. You can commit only some files using:  
   You should try doing this as you complete each lab section, just for practice. Note that this only affects your local repository. If you needed to push changes back into the CalSol GitHub account, you could do: This is here for completeness only – you do not have write permission to the CalSol repostiory.

git commit -a

git add <file 1>, <file 2>, …, <file *n*>

git commit

git push

1. If you want to see a list of previous revisions, use: To checkout the code *n* revisions ago, use: To get back to the latest version of the branch, use:

git log

git checkout HEAD~*n*

git checkout <branch>

**Note that this is not a comprehensive tutorial to git – in fact, this barely scratches the surface.**

For more information on using git, you can either search on the Internet, ask one of the TAs, or:

git help

# Lab 1: Hello, World!

## Objectives

* Understand the basic structure of an Arduino program
* Introduce serial communication

## Lab Manual

We will go through a very basic program line by line (so that you understand what is going on), then we will have your Arduino actually run it. Here is the code for this part of the lab:

(If you have ever worked with C before it should look pretty familiar! And if you haven’t, don’t worry, it’s straight forward - don’t let the text scare you!)

**Code: ee2\_lab1.pde**

/\*CalSol

Introduction to Microcontrollers Lab

Adam Resnick

\*/

void setup**()** **{**

Serial**.**begin**(**9600**);** //Initializes serial communication

**}**

void loop**()** **{**

Serial**.**println**(**"Hello World!"**);** //Print Hello World

delay**(**1000**);** //Wait 1 second

**}**

The first block of text is bounded by a /\* at the beginning and a \*/ at the end. This is a long comment. You can write whatever you want here, and your Arduino will ignore it. This is where you usually give a general description of what your code does.

Below the comment block, there are two other blocks of code. The first block is just some initializing things that you will need. The second block is what is where the meat of the program will go.

So, let’s start with the first line of the setup procedure:

void setup**()** **{**

This is the beginning of the setup procedure. Every Arduino sketch (program) that you write will have this line in it, even if nothing is going in the setup procedure. Don’t worry about why it says void for now. The syntax must read exactly like this (i.e. you need that empty parenthesis there, etc.). Any code that will be executed during the setup procedure must go between the open and close curly brackets {}.

The next line reads:

Serial**.**begin**(**9600**);** //Initializes serial communication

The double slash // denotes a one-line comment. Everything that comes after the double slash in that line will be ignored when you actually run the code. Comments are used so that other people can read and understand your code.

This line of code initializes what is known as serial communication. Serial communication is just a way of making two electrical devices talk to each other by transmitting bits. This initialization is done by making a call to a certain procedure in the Serial library. A library simply refers to a collection of functions that are all related - so all the functions in the Serial library are somehow related to serial communication. In this case, we call the begin function to start up serial communication. The format of Library.procedure() is the normal way of invoking functions, although some functions are so common that they aren’t in a library. The argument to Serial.begin is 9600. This may seem like a random number, but it actually indicates how fast bits will be sent and received. Before serial communication can be successfully implemented, both parties have to agree on a **baud rate** - speed to talk and listen at to ensure successful data transmission.

The final thing on this line before the comment is a semicolon. Semicolons come at the end of every line to tell the Arduino that that is the end of that line of code. The curly bracket on the next line signifies the end of the setup procedure.

We now move on to the main body of the program, which is in the loop. It is called the loop procedure for good reason – it runs the code inside of it over and over and over. The first line is:

void loop**()** **{**

This looks much like the first line of the setup procedure. This time though, instead of running setup code during device initialization, it will run code inside the loop() function over, and over, and over, and over.

The next line:

Serial**.**println**(**"Hello World!"**);** //Print Hello World

This is another call to a function in the Serial library. This time, it is a call to a function called println, which prints its argument (the data passed to the function inside the parentheses) on a new line (hence the ln in the name). The text is enclosed in quotes because it is a string. A **string** is just a bunch of characters (letters, numbers, and more) grouped together. Without those quotes, the text will be interpreted as executable code, and probably seriously confuse the Arduino.

The final line of the loop is:

This line is pretty self-explanatory. It makes the program wait for 1000ms (1 second). The delay function is so common that it isn’t in a library. The curly bracket after this ends the loop.

delay**(**1000**);** //Wait 1 second

So, overall, this loop prints out some text, waits for a second and then repeats until you either reset your Arduino, cut the Arduino’s power or the world ends. To see this in action, do the following:

1. Open the Arduino environment and copy/paste the code into the editor. Some of the text will change colors. This is to help you read code easier.
2. To check to make sure that you copied everything correctly, hit the play button (or press ctrl+r) to compile the code.
   1. If there are errors, a bunch of angry red text will pop up, otherwise when it is done, it will just say “Done Compiling” at the bottom.
3. Make sure your Arduino is connected to your computer and the Arduino’s green power LED is on.
4. Make sure that the right Arduino board is selected under Board (also in the Tools drop-down menu).
5. Go to the Tools drop-down menu, and select the COM port that the Arduino is on.
6. Hit the upload button (or Ctrl-U) to upload the code to the board. During the upload, you will see the RX and TX LEDs on the board flashing quickly. It will tell you “Done uploading” at the bottom of the Arduino environment when it is done.
7. The code is now running on the board, but to see what the board is sending back to the computer, go to Tools -> Serial Monitor (or Ctrl+Shift+M) to open the Serial Monitor. This is where all serial messages will be displayed.
8. At the bottom right of the Serial Monitor, make sure that you have a baud rate of 9600 selected.
9. You should see the text “Hello World!” (without the quotes) appear every second.

Congratulations! You just made your first Arduino program.

# Lab 2: Blinky

## Objectives

* Understand digital inputs / outputs
* Understand C if-statements

## Blinking the LED

We are going to now use an Arduino to make an LED blink. Here is the code:

**Code: ee2\_lab2.pde**

int ledPin **=** 13**;**

void setup**()** **{**

pinMode**(**ledPin**,** OUTPUT**);**

**}**

void loop**()** **{**

digitalWrite**(**ledPin**,** HIGH**);**

delay **(**1000**);**

digitalWrite**(**ledPin**,** LOW**);**

delay**(**1000**);**

**}**

Arduinos have a whole bunch of digital input/output (I/O) pins (on an Arduino Duemilanove, these are digital pins 0-13). These can be configured so that the Arduino sees them as inputs (“listens” on them) or sees them as outputs (“talks” on them). Since they are digital pins, they have two possible input/outputs states: high voltage or low voltage.

Another useful thing to know is a bit about is how numbers are represented in software. In the C language, you must declare the type of a variable when you define it. There are many types of numbers, but the simplest one is just an integer, or int.

Let’s take a look at the first line:

This line declares (brings into existence) an integer variable named ledPin and associates the number 13 with that variable. Programming variables aren’t like algebra variables which are unknowns. Programming variables are more like a box with a value stored inside. Instead of constantly writing the stored value, you can just write the name of the box.

int ledPin **=** 13**;**

Note that this takes place outside of the setup loop. This is because I want to use the variable ledPin everywhere in my code, not just in the setup or the loop part of the code. This makes the variable visible everywhere in that file, and it is said to be a **global variable**, since it has global **scope** (visibility).

Additionally, instead of the Serial.begin call from the previous section, we now see:

This is setting the ledPin to output mode. So, whatever pin I previously chose as my ledPin will become capable of outputting either a high or a low voltage.

pinMode**(**ledPin**,** OUTPUT**);**

Moving on to the loop, we see:

This sets the ledPin to the high voltage value. Because we are working with a digital pin, remember that there are only two possible values – high and low. This will turn your LED on because the anode (positive) terminal of LED is now higher than the cathode (negative) terminal, and current begins to flow.

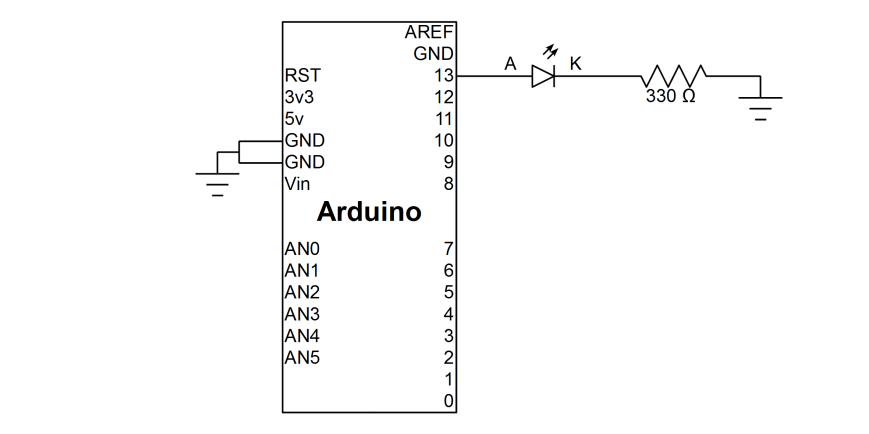
digitalWrite**(**ledPin**,** HIGH**);**

This line is followed by a delay, which you should already be familiar with. The line after the delay is:

This does the same as the other digitalWrite line, except that it now sets the pin to the low, or ground, voltage. This turns off the LED since there is no voltage difference across it, and thus, no current flowing through it. This is followed by another delay before the loop is repeated.

digitalWrite**(**ledPin**,** LOW**);**

To run this code, follow the same steps as before, but make sure that whatever digital pin you choose (it doesn’t have to be pin 13) is hooked up to one end of an a basic LED-resistor circuit. You should know how to do this from EE1, but if you forgot (or weren’t here for EE1), a circuit diagram is included:



Run the code, and make sure that your LED flashes and that you understand why it is flashing.

## A Slightly Different Implementation

The following code does the exact same thing as the previous program:

**Code: ee2\_lab2a.pde**

int ledPin **=** 13**;**

int ledOn **=** 1**;**

void setup**()** **{**

pinMode**(**ledPin**,** OUTPUT**);**

**}**

void loop**()** **{**

**if** **(**ledOn**)** **{**

digitalWrite**(**ledPin**,** HIGH**);**

delay **(**1000**);**

**}**

**else** **{**

digitalWrite**(**ledPin**,** LOW**);**

delay**(**1000**);**

**}**

ledOn **=** **!**ledOn**;**

**}**

Now at the beginning, I am initializing another variable called ledOn to have a value of 1. However, as you will soon see, this int is really being used as a Boolean variable, meaning that it only has two values (1 and 0, aka true and false). The C language does not natively support true Booleans.

In C, a variable with value 0 is considered false, while any non-zero number is considered true. However, we usually just think about it as 0 being false, and 1 being true. Thus, when the ledOn variable was initialized to 1, it was also being initialized to a Boolean value of true.

The new part of the code is in the loop:

**if** **(**ledOn**)** **{**

digitalWrite**(**ledPin**,** HIGH**);**

delay **(**1000**);**

**}**

**else** **{**

digitalWrite**(**ledPin**,** LOW**);**

delay**(**1000**);**

**}**

This is what is known as an if-statement. Let’s go through this line by line. The first line is:

**if** **(**ledOn**)** **{**

The part in the parenthesis after the if is known as the **condition**. The condition will either evaluate to a Boolean value of true or false. If the condition is true, the code in the block (inside the curly braces {}) after the if is executed. If the condition is false, the program skips to the block of code after the else (if there is one – generally it is not necessary to have an else block).

Thus, the first time this loop is run, ledOn has a value of true (since its 1). This means that the condition is true, and the code that writes ledPin high for a second is run. After doing this, the program then exits the if-statement and moves on to the next line of code in the loop, which is the following:

ledOn **=** **!**ledOn**;**

This line is read as “ledOn equals not ledOn.” The exclamation mark ! is the logical NOT operator. So, what this does is take whatever value ledOn has and assign it to be the opposite value.

In the first iteration of the loop when we reach this last line, ledOn has a value of 1, or true. This last line then switches the value of ledOn to false, or 0. The loop then executes again. This time though, when it gets to the if-statement, the block of code after the else is run, turning your LED off.

Run this code to verify that your LED blinks on and off just like with the previous code.

# Lab 3: Controlling LEDs with Buttons

## Objectives

* Learn how to read digital pins
* More if statements

## Lab Manual

Here is the code:

**Code: ee2\_lab3.pde**

#define LED 13

#define BUTTON 2

void setup**(){**

pinMode**(**LED**,** OUTPUT**);**

pinMode**(**BUTTON**,** INPUT**);**

digitalWrite**(**BUTTON**,** HIGH**);**

**}**

void loop**(){**

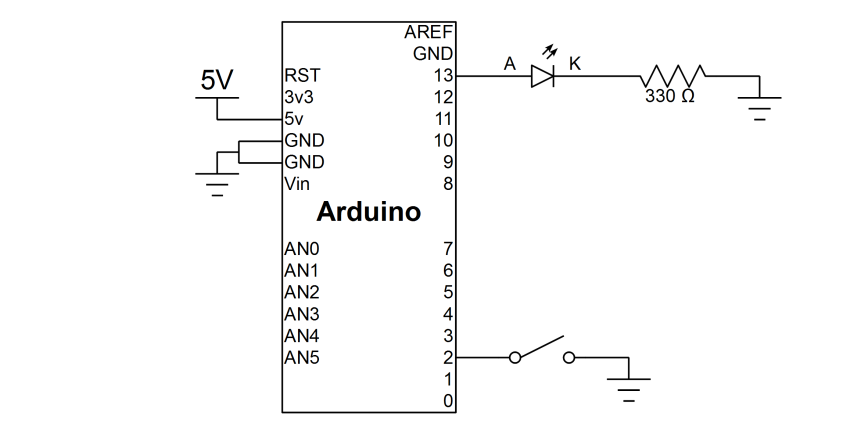
**while(**1**){**

digitalWrite**(**LED**,** digitalRead**(**BUTTON**));**

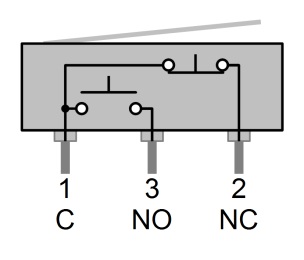
**}**

**}**

This code will work only if you hook up the switch we give you as indicated by the schematic below: one side to the board and one side to the digital pin labeled 2.



The switch pinout is:



**What should it do?**

When you hit the button, the LED turns on, and when you let go, the LED turns off.

**How does the code work to do this?**

Well, let’s first talk about some constructs that some of you probably have not seen before.

**The #define construct**

#define is a C/C++ construct that allows us to define constants that will never change. This is the complete opposite of variables which are designed to change. The format is as follows:

#define <name of constant> <definition of constant>

Something quirky is that no matter what you put as the definition of the constant, the compiler will automatically replace any occurrence of the name of the constant with the exact definition *without interpreting the definition*. Sloppy use of #defines can lead to subtle bugs involving order of operations, and it is recommended that more complex #defines have their definitions enclosed in parentheses.

Notice now that the pinMode is now for an input since we are using a switch to control the LED. Funny odd unpredictable behavior will occur if you do not initialize this pinMode correctly, so ensure you initialize *all* your inputs and outputs.

Most of you have probably never seen a while loop before. First of all, the curly braces after the while denote the block of code inside the statement. The parentheses right after the while loop contain the condition, and while this condition is true, then we will repeatedly run the block inside the curly braces (remember that in C/C++, 0 evaluates to false, and any other number evaluates to true). So in this case, we are telling the program to keep checking the LED and the switch over and over. Note that while statements do not automatically break in the middle if your statement becomes false in the middle. The statement is only checked at the beginning of each loop. It is possible for the loop to not execute at all if the condition is initially false.

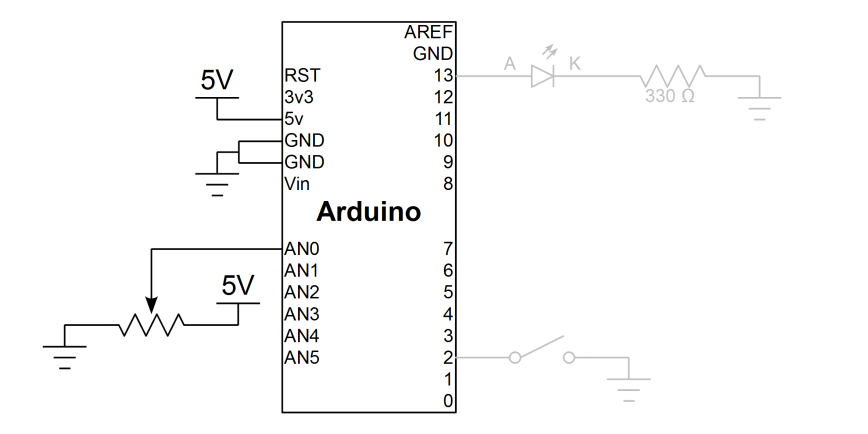
This line is a little bit unique. First of all, it introduces a concept you have probably never thought of: nested function calls. I can call digitalRead in the middle where I would usually put a number. digitalRead will execute and return a number that digitalWrite then uses. digitalRead does exactly what you think it should: it reads a digital I/O pin and returns either a 1 or 0 depending on if the pin senses a high or low voltage, respectively. The high and low voltages thresholds for a particular chip can be found in the device’s datasheet under Electrical Characteristics. For the Arduino, the maximum low voltage is and the minimum high voltage is (where is the supply voltage to the chip, 5 V in this case).

digitalWrite**(**LED**,** digitalRead**(**BUTTON**));**

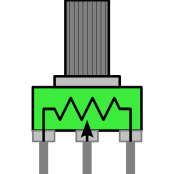
# Lab 4: Reading from an Analog Pin

This lab is going to be a bit more freeform. I'm not going to give you the exact code, but will instead give the tools for you to figure it out yourself, based on all that you have learned.

First off, you're going to need a potentiometer. Hook one end to ground, one end to 5v, and the middle pin to Analog In 0 (which should be labeled on the Arduino):



The potentiometer pinout is:



When using Analog pins, you need to initialize them using

pinMode(<analogPin>, INPUT);

Then all you need to do to read the pin is to call

analogRead(<analogPin>);

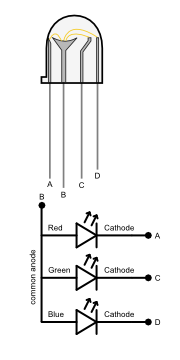
This function will read the voltage at the <analogPin>, and return it as a number from 0 to 1023, with 0 corresponding to 0 V and 1023 corresponding to (supply voltage). In the current setup, the Arduino’s supply is 5v, so it would give .

Now, here is your goal: setup an analog pin to read the potentiometer voltage value (in millivolts), then print it via Serial to your computer. You will have to do some math to convert the raw analogRead value into volts. You may consider the use of a float (floating-point number, can represent decimals) instead of an int. Because of the intricacies of C’s variable typing system, it is recommended that you only do one arithmetic operation per line.

Hint: you can do Serial.println(value) where value is an int or float.

# Lab 5: Cycling Colors on an RGB LED

## Objectives

* To learn about the basic idea of pulse-width modulation (PWM)
* To make pretty colors in the RGB color space
* To understand how to use while and for loops

## A Little Bit of Physics

* Pulse-width modulation is a way of encoding some signal in time using only binary (high and low) values.
* In our case, we want to vary brightness over time, but the Arduino can only output 0V or 5V.
* Solution: turn the LED on/off in short bursts so that the average power is something that we can vary continuously. The less time the LED is on, the less bright it appears.
* RGB LEDs are LEDs with a red, green, and blue LED in a single package.
* We’ll be using RGB LEDs with a common anode; they all have the same anode, but different cathodes

## “while” and “for” Loops Refresher

If you need a refresher on while and for loops:

Figure 1:A common-anode LED viewed from the side and its equivalent circuit.

* Often we’ll want to do something multiple times, or do something for every item in a list, or do something until some condition is met
* In those cases, you want some code that loops back and repeats itself until there’s nothing left to do.

### A Short Example

void setup**()** **{**

Serial**.**begin**(**9600**);**

int i **=** 1**;**

**while** **(**i **<** 10**)** **{**

Serial**.**println**(**i**);**

i **=** i **+** 1**;**

**}**

**}**

void loop**()** **{**

**;**//Don’t do anything here yet

**}**

The snippet above should print the numbers 1 through 9 and then stop.

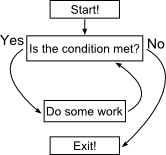
* The (i < 10) is the execution condition for this loop. When it no longer holds (i greater than or equal to 10), the loop will stop repeating.

Figure 2: The flow of a while loop.

* Everything in the curly braces {} after the condition is the body of the loop. This is executed if the condition still hold true, then it goes back and repeats. See the diagram on the right.
* Note that if the condition never becomes false, the while loop will never stop running!
* A similar structure is the “for” loop, which is a more compact way to write certain loops.

### A Shorter Example

void setup**()** **{**

Serial**.**begin**(**9600**);**

int i**;**

**for** **(**i**=**1**;** i **<** 10**;** i**=**i**+**1**)** **{**

Serial**.**println**(**i**);**

**}**

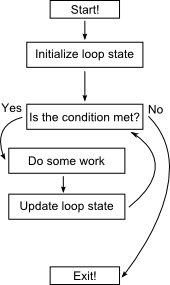
**}**

void loop**()** **{**

**;**//Don’t do anything here yet

**}**

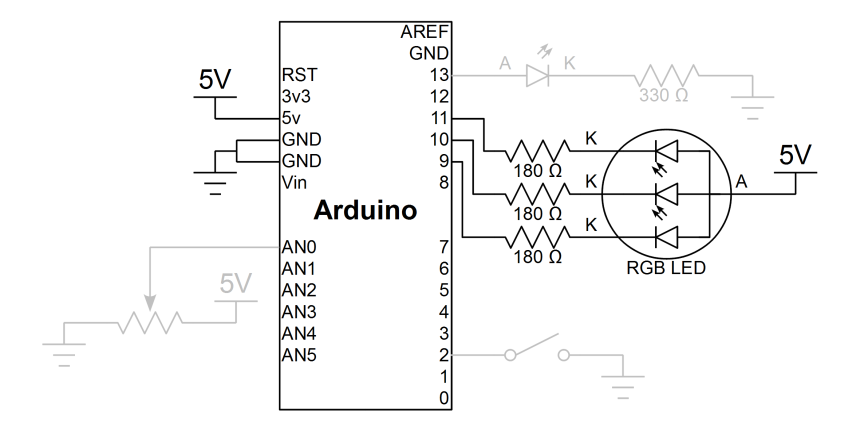
Figure 3: In the example, the loop state is the value of i.

* This is almost the same as the while loop, but we moved the initial value of i and the incrementing step into the beginning of the for loop. The end result is exactly the same, but we’ve put all of the expressions controlling the loop in one place.
* The three statements in the for(i=1; i < 10; i=i+1) statement are the initialization (setting up i), execution condition (checking if i is less than 10), and update statement (incrementing i).
* Comparing the diagram to the while loop diagram, we’ve made it clear that we update the variables that the condition depends on, rather than hiding it in the doing-work block.

## Less talk, more pretty lights!

1. Alright, enough theory, time to make things change color!
2. We’re going to use pins 9, 10, and 11 as current sinks. That is, where you would normally connect the LEDs through a resistor to ground, we’ll connect it to a PWM pin instead.
3. Connect the longest pin on the LED to the 5V power from the Arduino
4. Connect each of the red/green/blue cathodes to a PWM pin through a 180 ohm resistor

Your finished circuit should look like this:



1. Upload the following code:

**Code: ee2\_lab5.pde**

#define RED 9

#define GREEN 10

#define BLUE 11

float mod**(**float x**,** float d**)** **{**

**while** **(**x **>** d**)** **{**

x **=** x **-** d**;**

**}**

**return** x**;**

**}**

void setup**()** **{**

**;**

**}**

(continued on next page)

(continued from previous page)

/\* Maps a HSL color with L fixed at 0.5 and S fixed at 1

hue must be in the range [0.0, 360.0]

Returns a triple of floats in [0.0, 1.0] representing

the red, green, and blue values

\*/

void hue\_to\_rgb**(**float hue**,** float **\***dest**)** **{**

float h0 **=** hue**/**60.0**;**

float x **=** 1 **-** abs**(**mod**(**h0**,** 2**)** **-** 1**);**

**if** **(**h0 **<** 0.0 **or** h0 **>** 6.0**)** **{**

dest**[**0**]** **=** dest**[**1**]** **=** dest**[**2**]** **=** 0.0**;**

**}** **else** **if** **(**h0 **<** 1**)** **{**

dest**[**0**]** **=** 1.0**;**

dest**[**1**]** **=** x**;**

dest**[**2**]** **=** 0.0**;**

**}** **else** **if** **(**h0 **<** 2**)** **{**

dest**[**0**]** **=** x**;**

dest**[**1**]** **=** 1.0**;**

dest**[**2**]** **=** 0.0**;**

**}** **else** **if** **(**h0 **<** 3**)** **{**

dest**[**0**]** **=** 0.0**;**

dest**[**1**]** **=** 1.0**;**

dest**[**2**]** **=** x**;**

**}** **else** **if** **(**h0 **<** 4**)** **{**

dest**[**0**]** **=** 0.0**;**

dest**[**1**]** **=** x**;**

dest**[**2**]** **=** 1.0**;**

**}** **else** **if** **(**h0 **<** 5**)** **{**

dest**[**0**]** **=** x**;**

dest**[**1**]** **=** 0.0**;**

dest**[**2**]** **=** 1.0**;**

**}** **else** **{**

dest**[**0**]** **=** 1.0**;**

dest**[**1**]** **=** 0.0**;**

dest**[**2**]** **=** x**;**

**}**

**}**

void loop**()** **{**

float rgb**[**3**];**

**for** **(**int i**=**0**;** i **<** 1000**;** i**=**i**+**1**)** **{**

hue\_to\_rgb**(**i**\***360.0**/**1000**,** rgb**);**

analogWrite**(**RED**,** rgb**[**0**]\***255**);**

analogWrite**(**GREEN**,** rgb**[**1**]\***255**);**

analogWrite**(**BLUE**,** rgb**[**2**]\***255**);**

delay**(**20**);**

**}**

**}**

* The LED should cycle through a range of colors over the course of about 20 seconds.
* Here, we iterate through a thousand different hues in the HSL representation of color and convert it to the RGB representation
* We wait 20ms at each hue to give you enough time to perceive it.

Let’s modify this code so that it only cycles through colors as long as you hold down a button. We’ll use the more general looping control provided by a while loop to do so.

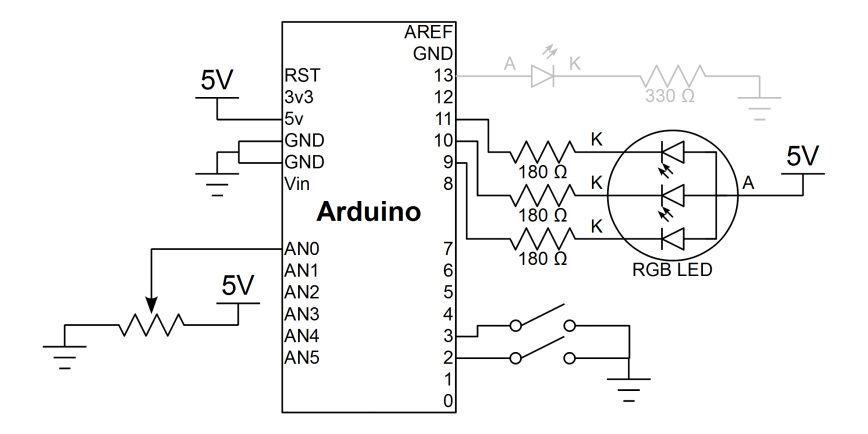
1. Rewrite the for loop as an equivalent while loop and re-upload the code. Make sure that it still acts the same.
2. Now attach a button as you did in the earlier labs. Modify the increment statement so that it only increments if the button is pressed.
3. Now you should be able to control when it cycles!

# Extra for Experts: Cooperative Multitasking

## Objectives

* To learn about multitasking, and the methods used to implement it
* To learn about cooperative multitasking, one simple method of multitasking commonly used on microcontroller systems
* To be able to implement a simple cooperatively multitasked system

## A First Attempt

1. First, setup your hardware for this lab.
   1. Ensure at least two LEDs are connected to PWM-capable pins on your Arduino. You can re-use the RGB LED from Lab 5.
   2. Ensure at least two switches are connected to your Arduino. If you retained your setup from Lab 3, you only need to connect one more switch to Pin 3.
   3. Ensure one potentiometer is connected to an analog-capable pin on your Arduino.
   4. One possible hardware setup is as follows:
   5. Ensure you already have code in your setup routine which initializes these components for use.
2. Write software that will detect a button press and turn on a LED for one second. See if you can come up with this yourself, otherwise copy and paste the code example below.

**Code: ee2\_extra.pde**

int led1Pin **=** 9**;**

int led2Pin **=** 10**;**

int switch1Pin **=** 2**;**

int switch2Pin **=** 3**;**

void setup**()** **{**

pinMode**(**led1Pin**,** OUTPUT**);**

pinMode**(**led2Pin**,** OUTPUT**);**

digitalWrite**(**led1Pin**,** LOW**);**

digitalWrite**(**led2Pin**,** LOW**);**

pinMode**(**switch1Pin**,** INPUT**);**

pinMode**(**switch2Pin**,** INPUT**);**

digitalWrite**(**switch1Pin**,** HIGH**);**

digitalWrite**(**switch2Pin**,** HIGH**);**

**}**

void loop**()** **{**

**if** **(**digitalRead**(**switch1Pin**)** **==** LOW**)** **{**

digitalWrite**(**led1Pin**,** HIGH**);**

delay**(**1000**);**

digitalWrite**(**led1Pin**,** LOW**);**

**}**

**}**

* 1. Most of the stuff happening in setup() is pretty straightforward and standard - it just tells whether a pin is an input or output.
     1. You may be wondering why we do a digitalWrite on the switch input pins. On the Arduino platform, a digitalWrite operation on an input pin enables the **pull-up resistors**. These resistors are internal to the microcontroller, and are connected between the pin and the power supply. They are typically very high resistance (on the order of tens of kiloohms), so that any external source could drive the pin to the desired logic level.
     2. This is a common technique for connecting switches - the switch is put between the pin and ground. Since the switch is normally open, there is no connection, and the pin is floating. With a pull-up resistor, the pin is weakly pulled up to a high level. However, when the switch is pressed, the circuit is closed and the pin is pulled down to ground. The low resistance switch circuit "overpowers" the pull-up resistor, leading to a low level on the pin.
  2. The loop is where the interesting stuff happens.
     1. The if statement checks if the level at the switch pin is low, which only happens if the switch is pressed.
     2. If the switch is not closed, then the contents of the if statement are skipped, and it goes to the end of the function. Eventually, another loop starts.
     3. If the switch is closed, it turns on the LED, waits one second, and then turns off the LED.

1. Now, expand the code to handle two LEDs. One possible implementation is as follows:

**Code: ee2\_extra\_wrong.pde**

// setup code remains the same

void loop**()** **{**

**if** **(**digitalRead**(**switch1Pin**)** **==** LOW**)** **{**

digitalWrite**(**led1Pin**,** HIGH**);**

delay**(**1000**);**

digitalWrite**(**led1Pin**,** LOW**);**

**}**

**if** **(**digitalRead**(**switch2Pin**)** **==** LOW**)** **{**

digitalWrite**(**led2Pin**,** HIGH**);**

delay**(**1000**);**

digitalWrite**(**led2Pin**,** LOW**);**

**}**

**}**

* 1. This code is very similar to the previous code involving one switch, except that it also tests another switch, and pulses on the second LED if that switch is closed.
  2. So, it works, right? Well, not as well as it could. Try pressing the other button while one LED is on - it won't work. Uh-oh!

## So, what happened?

When a switch is pressed, everything works nicely until you get to the delay statement. Since delay does just that - delay - it blocks the rest of the code, allowing nothing else (like the code that processes the second button press) to happen. The program just sits there, twiddling its thumbs, for one full second before moving on. Clearly, we have a problem.

## Cooperative Multitasking

What we would like to do is to make it appear as if both button checks were executing independently and at the same time - basically, multitask. There are multiple ways to accomplish this:

* Today, almost every PC you see is parallel - having **multiple cores**, each able to run separate programs. However, programming for parallel machines has its challenges (active research topics, actually), and you only get one core on most microcontrollers, including the Arduino.
* You may have also noticed that it was possible to run multiple programs at once before the introduction of dual-, quad-, and hexa- core processors. In this case, the **operating system** would essentially switch between programs very fast, giving the appearance that all the programs were executing at once, even though there is only one processor core. For microcontrollers, operating systems which accomplish this same task exist, but may be a bit overkill for our purposes. CalSol does not currently use those.
* You could also take the theory of how operating systems work and work it into your program structure. That is, for each different program, it does some work, then "yields" to another task. This is called **cooperative multitasking**. Note that our LED application is especially conducive to cooperative multitasking as the long delay is a perfect place to yield control to another task. However, doing so would require a bit of a radical change to our code. Let's see how we would implement this:

## Multitasking LEDs

1. Try the following version of the lab code - this controls one LED:

**Code: ee2\_extra\_new.pde**

void loop**()** **{**

// The "task" for Switch 1 and LED 1

static long led1OffTime **=** 0**;**

**if** **(**led1OffTime **<=** millis**())** **{**

digitalWrite**(**led1Pin**,** LOW**);**

**}**

**if** **(**digitalRead**(**switch1Pin**)** **==** LOW**)** **{**

digitalWrite**(**led1Pin**,** HIGH**);**

led1OffTime **=** millis**()** **+** 1000**;**

**}**

**}**

* 1. If you haven't programmed in C before, you're probably thoroughly confused. I'll explain some of the new syntax here:
     1. The static long led1OffTime = 0; creates a long integer (32-bit) variable called led1OffTime. The static qualifier means that this variable retains its value after loop ends - so that the next time loop is called, the previous value is kept. Note that the =0 assignment only occurs once at the beginning of the code, not every time the loop function runs.
     2. The millis() function is also new - this is just an Arduino function which returns the time, in milliseconds, since the board program started.
     3. Otherwise, the code should use statements you are already familiar with.
  2. However, the structure of the code is probably new, and here's how it works:
     1. We keep one counter, ledOffTime, which keeps track of the time to turn the LED off. Initially, it is set to 0, which means the LED starts off.
     2. Every time the loop runs, we compare ledOffTime to the current time. If the current time is larger, then we turn the LED off.
     3. When the button is pressed, we turn on the LED as usual. But, instead of delaying (and causing that nasty problem), we set the ledOffTime to be 1000 milliseconds (1 second) plus the current time. Then, on the loop call which occurs a second later, the LED turns off, but until then, the LED is left on.
     4. Note that the if statement ends there, and then the code for processing the next button executed without delay.
     5. So, the big idea is that instead of using a delay, we store the time at which an event will occur, then repeatedly check the current time against that time on future loop iterations. If the current time is detected to be larger than the stored time, then the event executes, but until then, other code is free to run, giving the appearance that the tasks are running in parallel.

1. Now, extend the code to control two LEDs.
   1. This should be a simple copy/paste operation of the LED 1 task, except with the pins and variable names modified for a second LED.
   2. The implementation is left as an exercise for the reader. See if you can get this to work - but if you're feeling frustrated, ask one of the instructors for help.
2. By now, you're probably thinking, "this really isn't that exciting." Sure, you've got two LEDs to be controlled by switches, but there's no *interactivity*. Let's change that. Instead of using a constant one second time LED on pulse, change that time to be the potentiometer output value.
   1. You can use the analogRead(<pin>) function to read the analog value of a particular analog-capable pin. If you set up the potentiometer such that one outer terminal is connected to ground, the other outer terminal to the positive voltage supply, and the center terminal to the analog input pin, then this function will return a number from 0 to 1023 depending on the position of the potentiometer dial. You can then use this number as the duration. Or, you can do something interesting with it, such as using twice or even thrice this number for the pulse duration.
   2. Again, this implementation is left as an exercise to the reader. But if you feel like you're banging your head against a brick wall, stop and ask for help.
3. Ok, that was cooler, but the LED is still just solid on and solid off. Let's change that too. Use PWM to allow the LED to vary brightness, and have the LED's light output linearly decay to 0 during the pulse duration.
   1. You can use the analogWrite(<pin>, <value>) function to adjust the brightness of a LED connected to a PWM-capable pin. The <pin> argument is straightforward, and the <value> argument takes on numbers between 0 to 255, with 0 being off and 255 being full-bright.
   2. **Refresher**: **PWM** stands for **pulse-width modulation**. Basically, it's a technique which shifts the pin between high and low really, really fast. The PWM **duty cycle** (which is controlled using the aforementioned analogWrite function) is the amount of time the pin is high divided by the period (amount of the time pin is high plus the amount of time the pin is low). Although the LED is truly flickering, it does so fast enough that you perceive it as a brightness change instead.

## Extra for Experts for Experts: Linearly Varying Brightness

1. You may notice that linearly varying the PWM duty cycle does not linearly vary the perceived LED brightness. This is, in part, because the eye does not perceive light linearly - see Steven's Power Law and the Webner-Fenchner law for more information. For now, just try these two different ways of modifying the PWM duty cycle:
   1. Try an exponential function, there the duty cycle is for some constant . If you know a little about how numbers are represented inside computers, then using the exponent base is often convenient as you can simply do a bit shift to adjust the value up or down.
   2. Try a square function, where the duty cycle is .
   3. Both should give results better than linearly varying the duty cycle.

# Other Microcontrollers

You had a chance to play with the Arduino today. It is a very popular hobbyist electronics platform, but by no means the only microcontroller platform available. In fact, most microcontrollers are sold as bare chips, and the Arduino just adds user-friendly features around that to make it more accessible to newbies.

Several other popular microcontrollers (with their manufacturers) are:

* AVR (Atmel) - this is the chip used on the Arduino. AVRs cores range from 8 bits to 32 bits and comes with a variety of peripheral features. In CalSol, these are the microcontrollers which powers our BRAINs.
* PIC (Microchip) - another popular line of microcontrollers. Similarly, there are 8-, 16-, and 32- bit cores, and a variety of peripheral features are available. In CalSol, a 16-bit PIC with DMA (and relatively complex code) powers the Datalogger board.
* MSP430 (Texas Instruments) - these are now being used by the EE40 course. There is a development kit build around this chip, called the LaunchPad, which is available for a dirt-cheap $4.30. However, it's not as user-friendly as the Arduino and thus has a steeper learning curve.
* ARM (various) - a 32-bit core. There exists both microcontrollers and microprocessors using ARM cores. ARM is really popular, and you can find these cores in devices like smartphones and tablets. We will be using these as the standard microcontroller platform for Zephyr.
* MIPS (various) - another 32-bit core. While less popular than ARM, this is the instruction-set architecture being taught in CS61C and CS150. These cores are common in routers, though.

One thing to note is that while the Arduino environment is easy to use, it does not really scale. For a large project, we need something more heavy duty, like a full IDE. If you're ever used Microsoft Visual Studio, NetBeans, or Eclipse, you'll know what we’re talking about.