Techshop EEE-201: Basic programming

In this class, you will learn how to use an Arduino in simple projects. We will cover programming and the simple electronic circuits needed to connect an Arduino to LEDs, switches and dials in order to control LED brightness and on/off state. THe three hours will switch between instruction, hands-on and some free time. A PDF of the course material can be downloaded in order to get yourself familiar with all the course content.

This is a fast track class for people who want to add simple automation to their projects. Example programs will be provided. You can cut and paste from the examples to start writing your own programs.

The class fee includes a breadboard, switch, LEDs and various electronic components.

Prerequisites: You must have an Arduino or Arudino compatible board

This is Part 2 of the Arduino Electronics Series. Part 1 is building your first Arduino board although you can skip that class and just buy one if you like. Please bring the Arduino board you built in the Arduino Part 1 class and a laptop with the Arduino software and drivers pre-installed. The software may be downloaded online.

If you bring your own laptop, you **MUST** pre-download and install the Arduino software and drivers.

You must make arrangements for a computer before signing up for the class - either be able to bring a laptop, or call the shop for assistance.

Course Suppplies

- 1 x Educato board (or any Arduino compatible board)
- 1 x small breadboard
- 1 x RGB LED
- 3 x 330 ohm resistors
- 1 x normally open (NO) switch button
- 1 x 10K potentiometer

Chapter 1

Let's get started with Arduino! In this chapter we are going to cover the most basic parts of getting a sketch running; we'll be covering the following topics:

- `setup()` function
- `loop()` function
- The serial monitor and the `println()` function
- Using comments

Chapter 2

Our Arduino examples haven't actually done much yet, let's change that! In this chapter we will learn how to control the blinking of a small light emitting diode (LED); we'll be covering the following topics:

- blinking an LED with `digitalWrite()`
- variable type `byte`
- using a normally open (NO) switch
- using a potentiometer to adjust the brightness of the LED

Chapter 3

- Analog input
- Fading a LED with PWM
- variable type `int`
- `if` control structure

Appendix

This appendix will cover the basics of how a breadboards work, will get you familiar with using them and will cover some of the basics of circuit diagrams.

Techshop EEE-201: Basic programming

Chapter 1

Welcome to chapter 1! Let's get started learning the basics of how to use an Arduino. In this chapter, we will cover the following topics:

- `setup()` function
- `loop()` function
- The serial monitor and the `println()` function
- Using comments

Table of contents

- Introduction
- Part 1 - bare minimum
- Part 2 - serial monitor

Introduction

In this class, we will be working with the Educato board - an Arduino compatible microcontroller board. For all the material that follows, I'll simply be referring to it as Arduino since that's the platform & software we'll be using. So what is a microcontroller and what is Arduino?

You can think of a microcontroller as a simple, mini-computer; let's see what wikipedia has to say about it:

> A microcontroller (or MCU, short for microcontroller unit) is a small computer (SoC) on a single integrated circuit containing a processor core, memory, and programmable input/output peripherals.

So, basically, it's an all in one computer package packed into a single

chip. Understand that I'm using the word computer pretty loosely here - it by no means compares to standard desktop or laptop computers. Microcontrollers are designed to do pretty basic tasks, have very little memory (comparatively) and can be slow. But for what we will be accomplishing in this class, it's a perfect platform!

So what about Arduino? The following quote is taken from the arduino.cc site:

> Arduino is an open-source electronics platform based on easy-to-use hardware and software. Arduino boards are able to read inputs - light on a sensor, a finger on a button, or a Twitter message - and turn it into an output - activating a motor, turning on an LED, publishing something online. You can tell your board what to do by sending a set of instructions to the microcontroller on the board. To do so you use the Arduino programming language (based on Wiring), and the Arduino Software (IDE), based on Processing.

You can think of Arduino as a hardware/software package that we can use for all our DIY projects. We will be writing code, or a set of instructions, that will tell the microcontroller what to do. This set of instructions is also known as a sketch and is named so because the Arduino platform was designed for artists. The sketches are written on a regular desktop or laptop and then "uploaded" to the Arduino through their software IDE.

Part 1 - bare minimum

Let's get started! Open the Arduino IDE and copy/paste the BareMinimum.ino sketch. You should see two sections of code, the first looks like:

```
void setup() {
  // put your setup code here, to run once:

}
```

The `setup()` function is called when a sketch starts. Use it to initialize variables, pin modes, start using libraries, etc. The setup function will only run once, after each powerup or reset of the board.

Next we see the `loop()` section of the sketch:

```
void loop() {
  // put your main code here, to run repeatedly:

}
```

The `loop()` function does precisely what its name suggests, and loops consecutively, allowing your program to change and respond as it runs. Code in the `loop()` section of your sketch is used to actively control the board.

Any line that starts with two slashes (`//`) will not be read by the Arduino, so you can write anything you want after it. The two slashes may be put after functional code to keep comments on the same line. Commenting your code like this can be particularly helpful in explaining, both to yourself and others, how your program functions step by step.

Part 2 - serial monitor

So far we haven't actually done anything with the Arduino, let's fix that! We begin with the quintessential *Hello, World!* example; copy/paste

HelloWorld.ino into your IDE. Next we will want to upload our sketch to the Arduino so that it knows what we want it to do; here's how:

1. Open the Arduino software (IDE)
2. Copy and paste your sketch into the window
3. Save a copy somewhere
4. Click the *Upload* button which looks like a right-arrow or choose from the menu *Sketch -> Upload*

If you have any problems uploading your sketch, check these two things:

- You have the proper board selected under *Tools -> Board* (it should be `Arduino/Genuino Uno`)
- You have the proper port selected

After uploading has finished, open the serial monitor (*Tools -> Serial Monitor*) and you should see something like this:

> Hello, World!
>
> Hello, World!
>
> Hello, World!
>
> Hello, World!
>
> Hello, World!
>
> Hello, World!
>
> Hello, World!
>
> Hello, World!
>
> Hello, World!

> Hello, World!
>
> Hello, World!

Let's walk through how this works; the first part of our sketch looks like this:

```
void setup() {
  // put your setup code here, to run once:
  Serial.begin(9600); // open the serial port at 9600 bps
}
```

All we've done is add the line `Serial.begin(9600)` - what this means is that we are telling the Arduino we want to use the serial monitor and that we are going to use it at a speed (also known as baud of 9600 bits per symbol (bps). The serial monitor is our way of communicating between our desktop/laptop and the Arduino; we're going to use it to let the Arduino "talk" to us. The idea of baud can get pretty confusing, so for now just think of it as how fast we are sending data.

Now that we've got the serial monitor setup, let's start using it!

```
void loop() {
  // put your main code here, to run repeatedly:
  Serial.println("Hello, World!"); // print our message
}
```

Remember, the `loop()` just repeats *Ad nauseam*; in this case, we use the `println()` function to print text to it so that we can see it on our screen. Since that's the only line of code we've added, and since the execution of the code will simply repeat itself, the Arduino keeps interpreting that print statement over and over.

Explore

- what happens if you change the line of code to `Serial.print("Hello, World!")`?

Techshop EEE-201: Basic programming

Chapter 2

In this chapter, we are going to start building circuits and using our Arduino to control those circuits; we will be covering the following topics in this chapter:

- blinking an LED with `digitalWrite()`
- variable type `byte`
- using a normally open (NO) switch
- using a potentiometer to adjust the brightness of the LED

Table of Contents

- Introduction
  - Current limiting resistors
  - Header pins
- Part 1 - simple blink
  - Circuit
  - Sketch
  - Named variables

- Part 2 - adding in a switch
- Part 3 - adding in a potentiometer

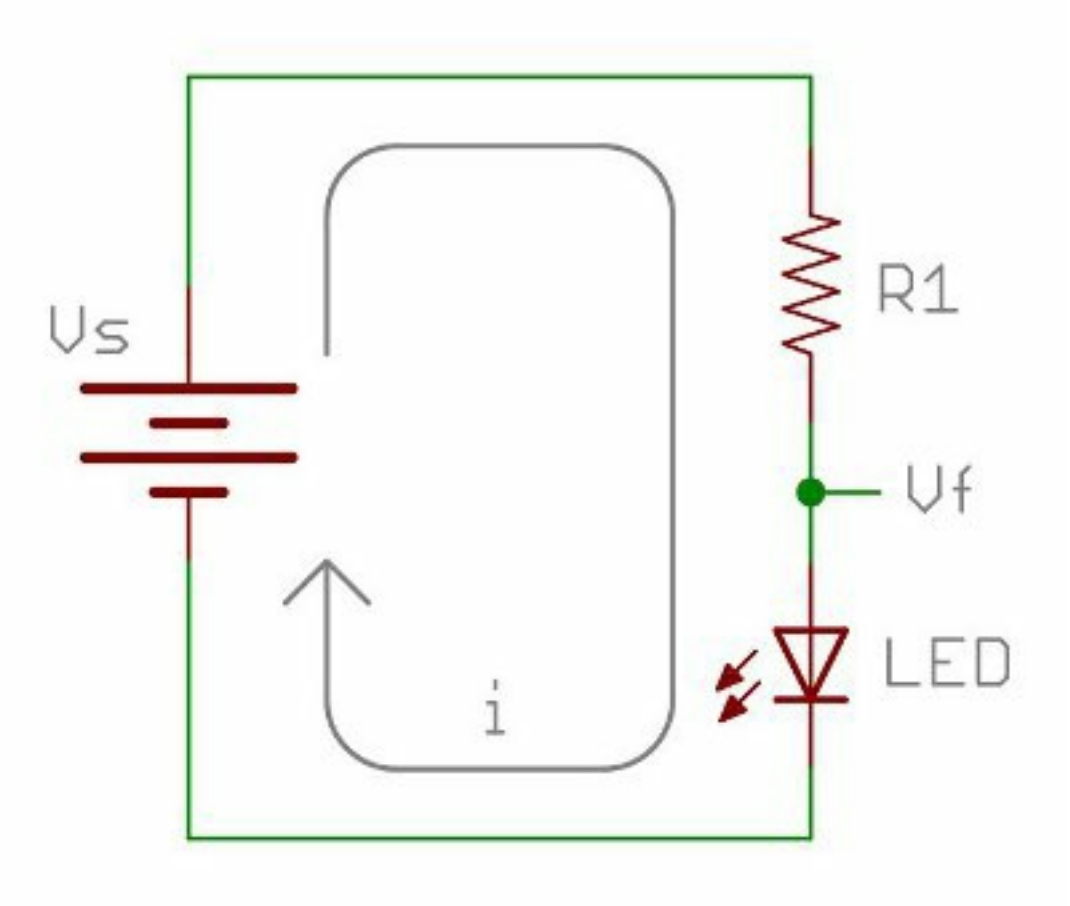
Introduction

This chapter assumes you are some knowledge of how to read circuit diagrams; if you have never seen one before, head over to the circuit diagram tutorial to read all about 'em.

Current limiting resistors

Limiting current into an LED is very important. An LED behaves very differently to a resistor in circuit. Resistors behave linearly according to Ohm's law: V = IR. For example, increase the voltage across a resistor, the current will increase proportionally, as long as the resistor's value stays the same. Simple enough. LEDs do not behave in this way. They behave as a diode with a characteristic I-V curve that is different than a resistor.

For example, there is a specification for diodes called the characteristic (or recommended) forward voltage (usually between 1.5-4V for LEDs). You must reach the characteristic forward voltage to turn 'on' the diode or LED, but as you exceed the characteristic forward voltage, the LED's resistance quickly drops off. Therefore, the LED will begin to draw a bunch of current and in some cases, burn out. A resistor is used in series with the LED to keep the current at a specific level called the characteristic (or recommended) forward current.



Using the circuit above, you will need to know three values in order to determine the current limiting resistor value.

`i` = LED forward current in Amps (found in the LED datasheet) `Vf` = LED forward voltage drop in Volts (found in the LED datasheet) `Vs` = supply voltage

Once you have obtained these three values, plug them into this equation to determine the current limiting resistor:

$$R = \frac{V_s - V_f}{i}$$

Also, keep in mind these two concepts when referring to the circuit above:

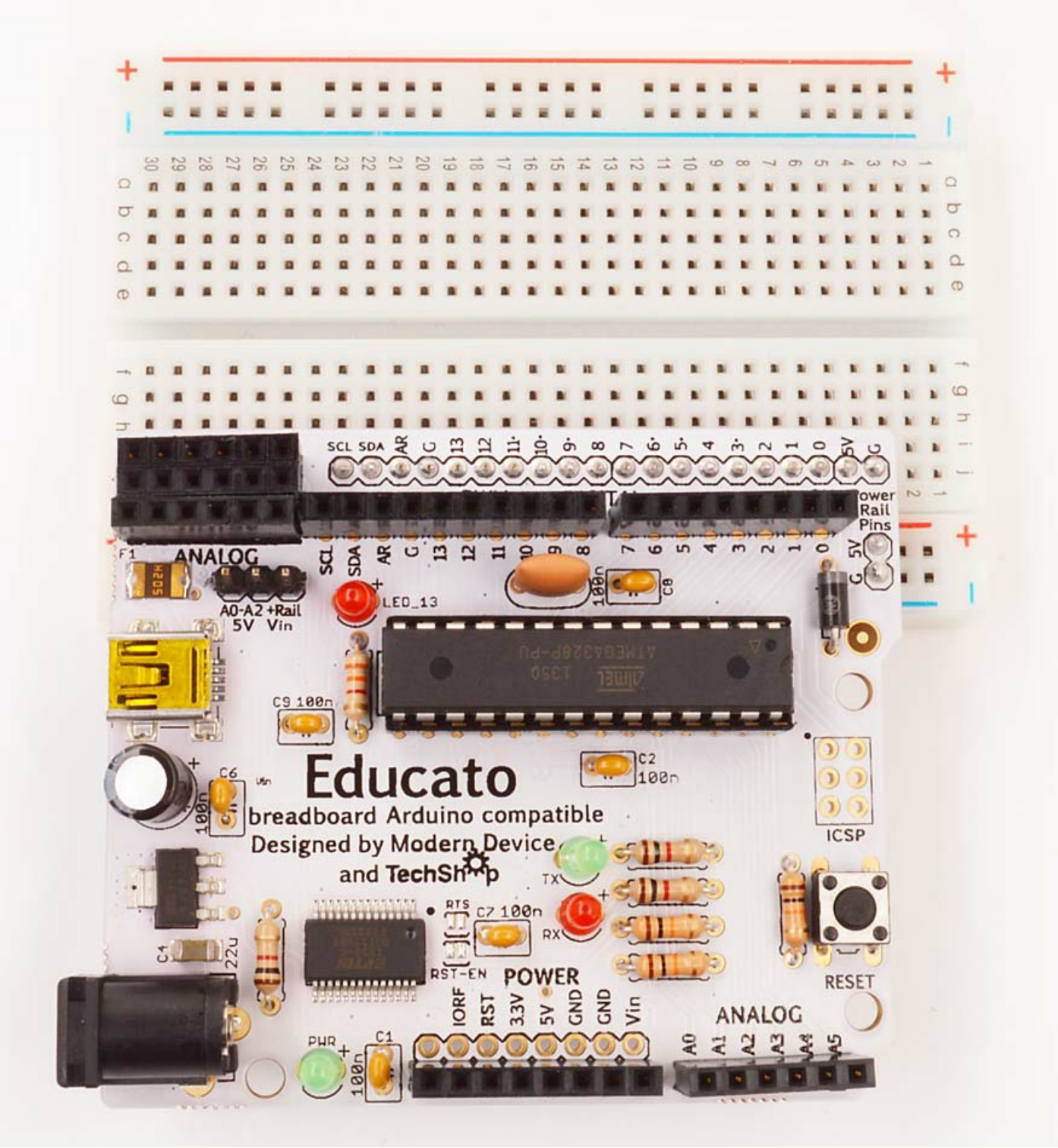
1. The current, `i`, coming out of the power source, through the resistor and LED, and back to ground is the same KCL
2. The voltage drop across the resistor, in addition to the forward

voltage drop of the LED equals the supply voltage KVL

You can find a handy resistor calculator online which will tell you exactly which resistor you should be using for your ciruit! When in doubt, always use a larger resistor than necessary because if you don't, you risk burning up the LED. In general, using a very large resistor will dim your LED; as you reduce the resistance, your LED will become brighter.

Header pins

Before we being using our Arduino, let's get oriented with the board:



On the board, we see multiple integraded circuits (ICs) such as resistors, capacitors, LEDs and so on; however most importantly we also find two rows of black, female headers. These headers, grouped into three main sections (POWER, ANALOG and DIGITAL) will be the primary way we will interface with the arduino. They are made up of numerous pins that we will be able to control individually with our sketch.

POWER

This is where we can get our voltage from, both as 5V or 3.3V. We can also get access to *Vin* which will be the same voltage as your power supply (which can be up to 12V).

ANALOG

In the analog section we find 6 pins labeled *A0* to *A5* - these are analog inputs which will be described in Chapter 3

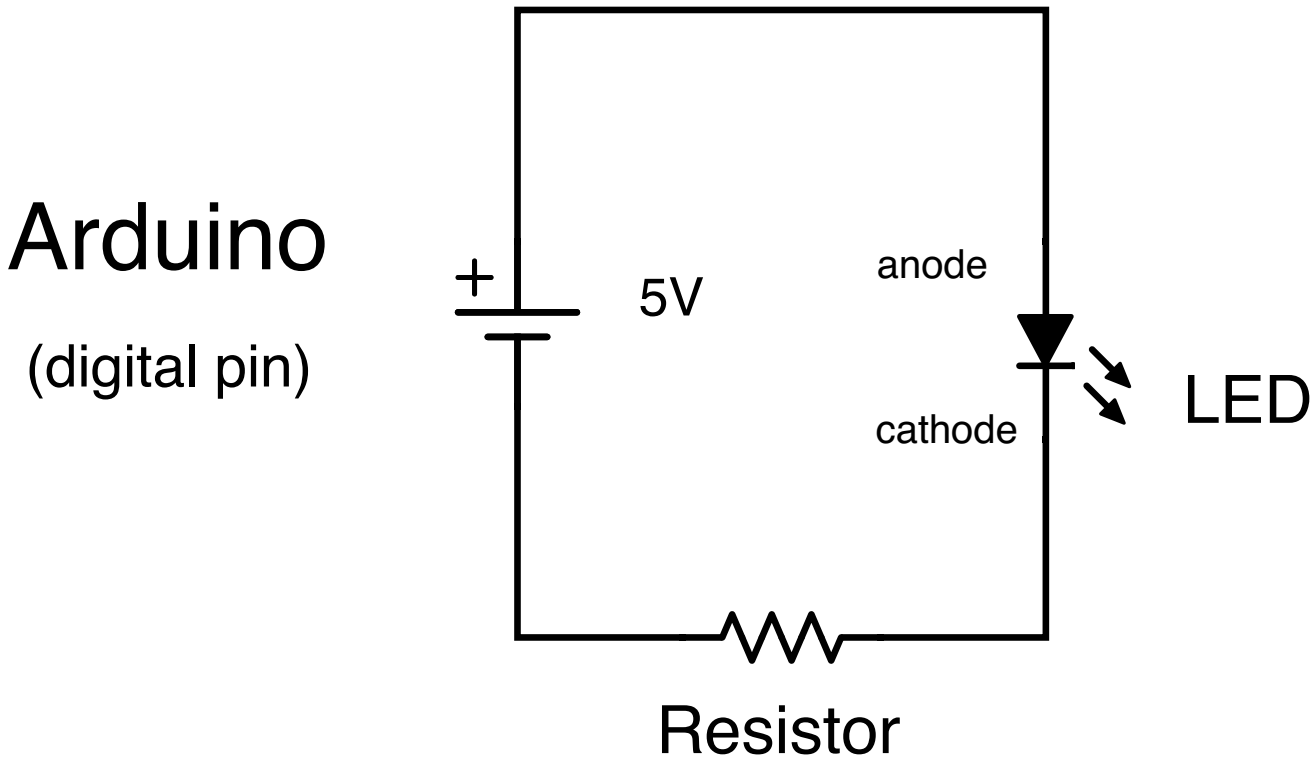
DIGITAL

In the digital section, we have 14 pins labeled *0* to *13* (there are others but these pins are outside the scope of this class) - these are simple digital pins that we can turn either `HIGH` or `LOW` and will be covered in this chapter. Notice also that some of the pins have an asterisk by the number; this identifies these pins as supporting pulse width modulation (PWM), whic will be covered in Chapter 3

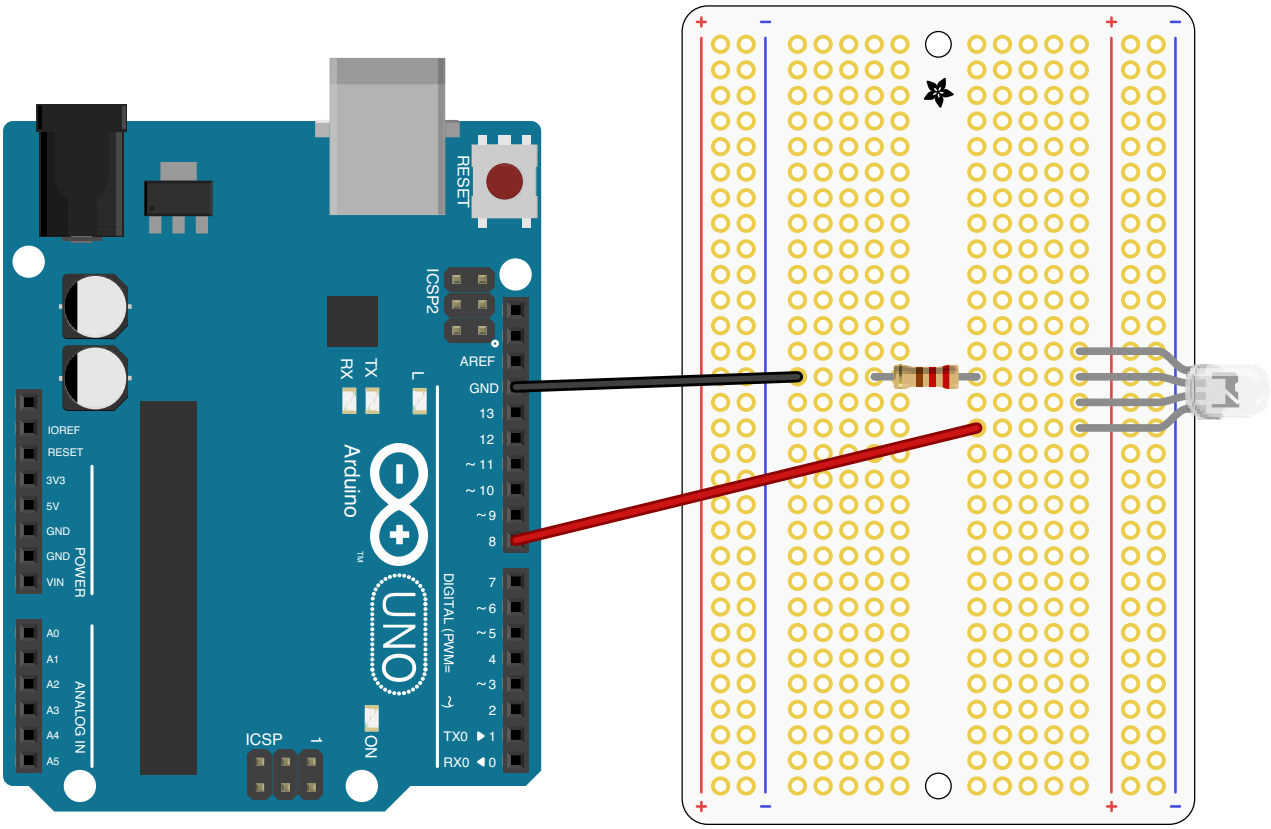
Part 1 - simple blink

Circuit

We start by putting together a simple LED circuit which we will blink on/off with our Arduino. The circuit you'll put together on your breadboard can be found below; note that the sketch we'll be using assumes you've hooked your LED up to digital pin **8**.



There isn't *one* correct way of putting the circuit together, but many different ways will work! To help you out, in the image below, we've shown you one way that you might put your circuit together.



You should be able to notice a few things with when comparing the

image above to the diagram:

- We are using an RGB LED instead of a simple LED; you can think of these RGB LEDs as 3 LEDs put together in a single package. In the diagram, I've only hooked up the blue (B) part of the LED.
- In the image, current will flow in a counter-clockwise direction: starting from pin 8, going through the LED, then the resistor, and back to GND; in the circuit diagram, current is shown to flow in a clock-wise direction.
- We are taking advantage of the breadboard row #14 to connect the resistor to the black wire.

A few important things to remember:

- The resistor does not have polarity; that is, you can put it in any direction.
- The LED **does** have polarity: hook the anode up to pin 8 (5V) and the cathode to resistor (which then goes to GND).

Sketch

You'll be using the Blink sketch on the arduino. Note that the sketch assumes you've hooked your LED up to digital pin **8**.

Just like we saw previously in Chapter 1, our sketch has two parts to is: `setup()` and `loop()`; let's walk through the first part:

```
// the setup function runs once when you press reset or power the
void setup() {
  // initialize digital pin 8 as an output.
  pinMode(8, OUTPUT);
}
```

In the `setup()` function, we need to tell the Arduino that we are going to use pin **8** and that we are going to use it as an `OUTPUT`; this is done through the `pinMode()` function. Setting it as an output means that the Arduino will be turning that pin `HIGH` (setting it to 5V) or `LOW` (setting it to 0V) based on the code in the sketch. Which is what we see in the `loop()` function:

```
// the loop function runs over and over again forever
void loop() {
  digitalWrite(8, HIGH);   // turn the LED on (HIGH is the voltage
  delay(1000);                       // wait for a second
  digitalWrite(8, LOW);    // turn the LED off by making the volta
  delay(1000);                       // wait for a second
}
```

If you remember what we learned in Chapter 1, you should know that whatever is written inside the `loop()` function is, as the name implies, looped or repeated as long as the Arduino is on. In this case, we use the `digitalWrite()` to turn the pin 8 *on* or `HIGH` - what this means is that the voltage on this pin is turned on to 5V so that current can flow from the pin - this will turn your LED on! Next, the Arduino interprets the `delay()` function; this simply pauses the sketch for the duration defined in the function, in this case it's 1,000 miliseconds (or 1 second). The next two lines within the `loop()` function are exactly the same except that instead of writing `HIGH` to pin 8, we tell the Arduino to write `LOW` - think of this as turning off the current flow in the pin, setting the voltage to 0V - this will turn your LED off.

And that's it! This sketch will turn your LED on for 1 second, then turn it off for 1 second; this gets repeated forever!

Explore

- how would you have to change your sketch so that the LED blinks at 2 Hz (once every 0.5 seconds)?

Named variables

What happens if you'd want to change which pin controls your blinking LED? That's easy enough, just change all the 8s in your sketch to say, 9. That means we'd have to change 3 lines of code; that doesn't sound very practical. We can get around this by using variable names - you can think of these as a place holder for a specific piece of information. Let's start by an example, change your sketch to match the following and upload it:

```
// the setup function runs once when you press reset or power the

byte ledPin = 8; // declare a new 'byte' type variable and set it

void setup() {
  // initialize digital pin 8 as an output.
  pinMode(ledPin, OUTPUT);
}

// the loop function runs over and over again forever
void loop() {
  digitalWrite(ledPin, HIGH);   // turn the LED on (HIGH is the vo
  delay(1000);                       // wait for a second
  digitalWrite(ledPin, LOW);    // turn the LED off by making the
  delay(1000);                       // wait for a second
}
```

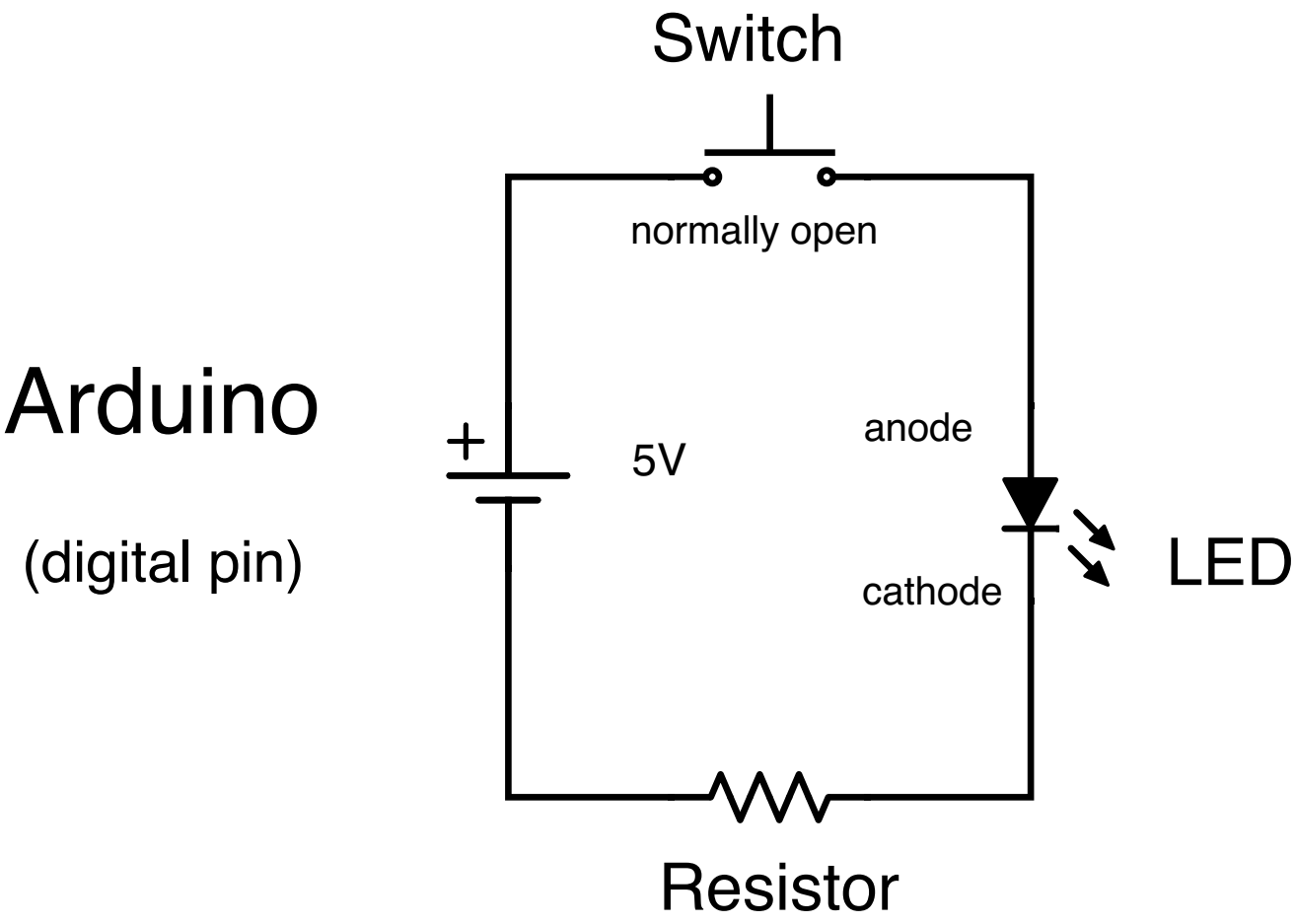
The LED should blink just like it did before! All we've done is create a

placeholder called `ledPin` and stored the value 8 to it. Then, whenever we have to refer to the value 8, we can just use the variable `ledPin`. But what is all this `byte` business about? Well that's the variable type. In the programming language that Arduino uses, we have to specify how much memory we want to allocate to memory; in this case, we are allocating one variable with the size of a `byte`. It's like telling the Arduino how much memory we expect to be using; this way the Arduino can check to see if it has enough. Remember the Arduino does not have a lot of memoery.

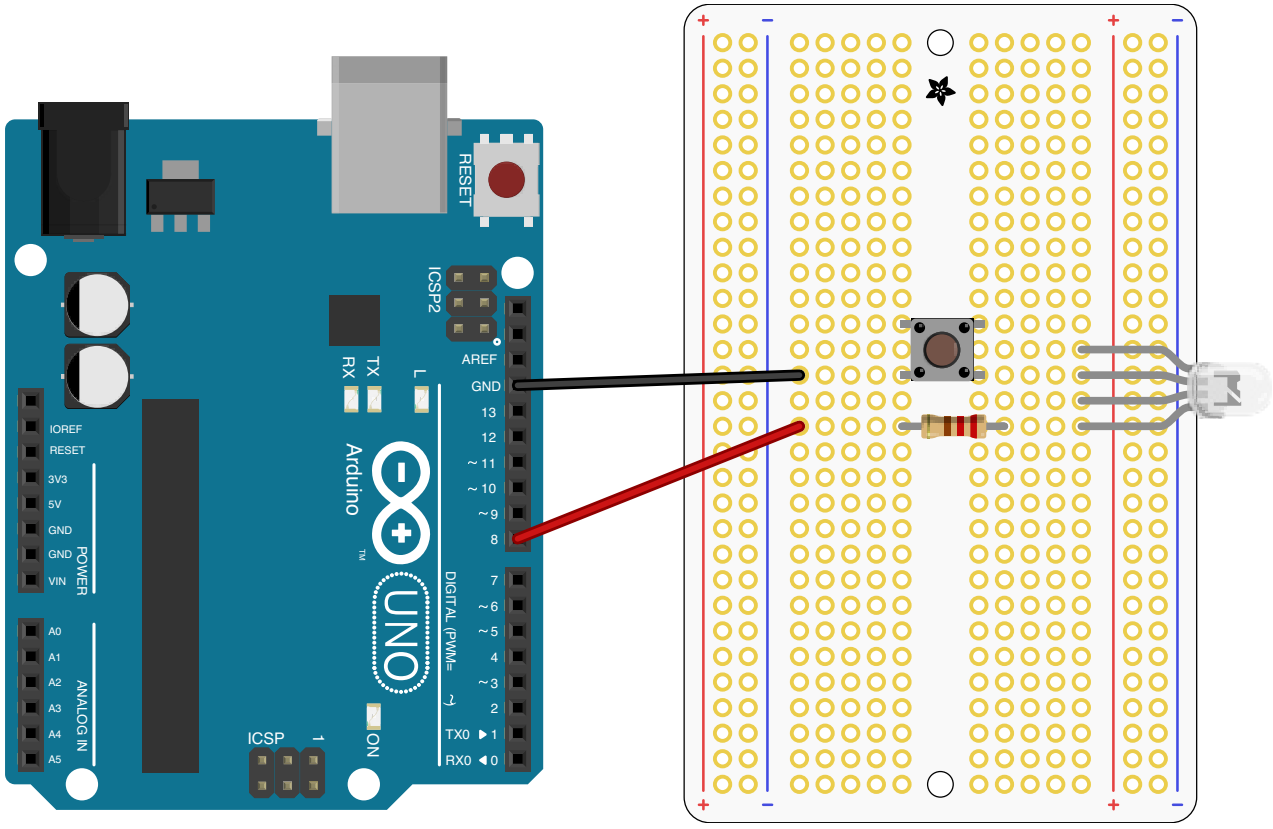
A `byte` stores an 8-bit unsigned number and can range from 0 to 255. Unsigned simply means that it can't be negative; and 8-bit is the same as saying 2 to the power of 8 (which is 255). You might be wondering, what if I want to store the value 438? Well, that wouldn't "fit" into a byte, and you'd have to use an `int` instead. We'll be covering the other variable types in the future, but for now the `byte` should be all that we need.

Part 2 - adding in a switch

Let's make a slight modification to our circuit and add a momentary normally open switch. The way this switch or button works is that unless you press on it, no connection is made accross the terminals (this is known as normally open or NO)



I've updated the layout by adding a switch in the image below and moving the resistor around a bit.



Let's also modify our sketch a little bit so that the LED always stays on; change the `setup()` and `loop()` to look like this:

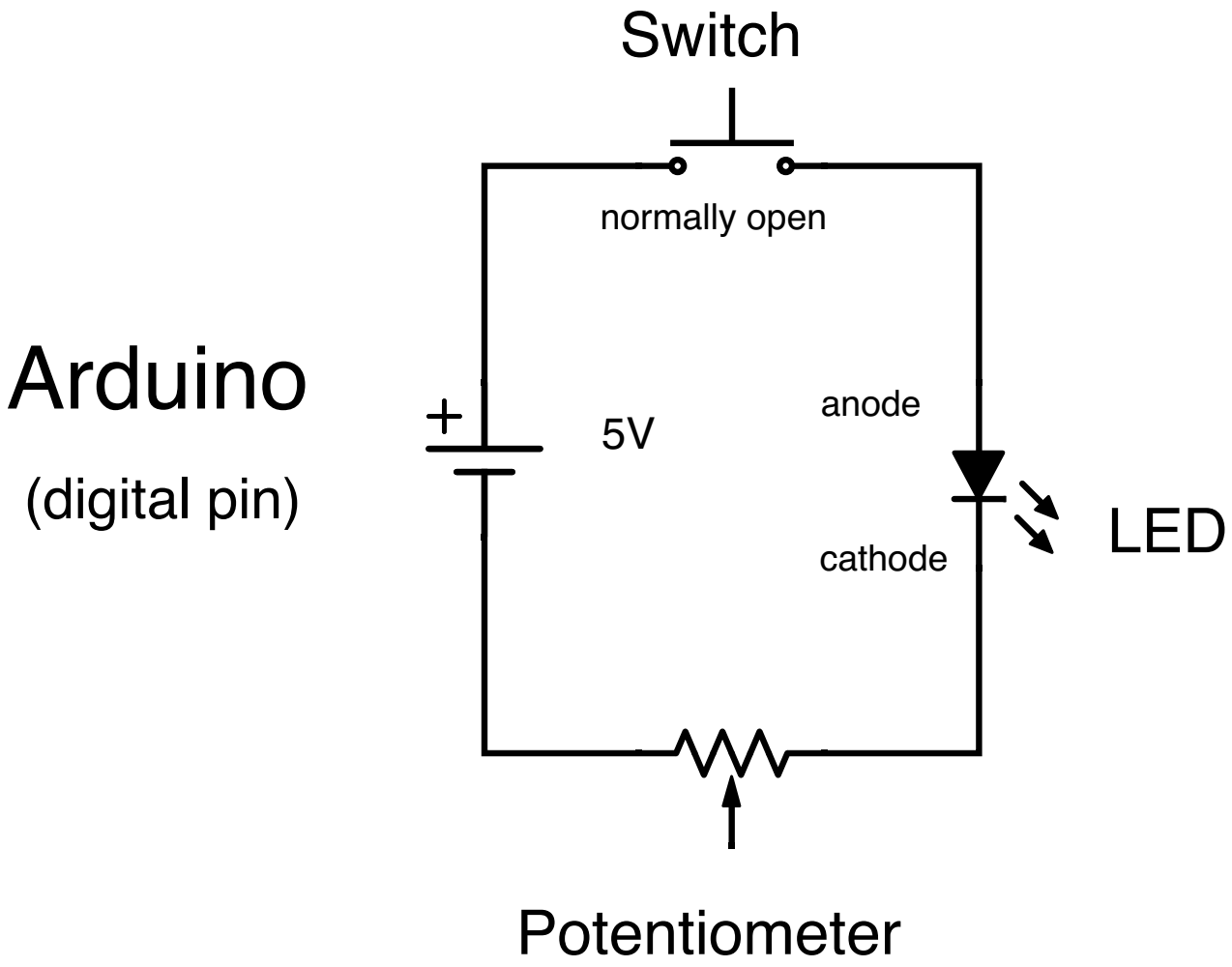
```
// the setup function runs once when you press reset or power the
void setup() {
  // initialize digital pin 8 as an output.
  pinMode(8, OUTPUT);
  digitalWrite(8, HIGH);   // turn the LED on (HIGH is the voltage
}

// the loop function runs over and over again forever
void loop() {
}
```

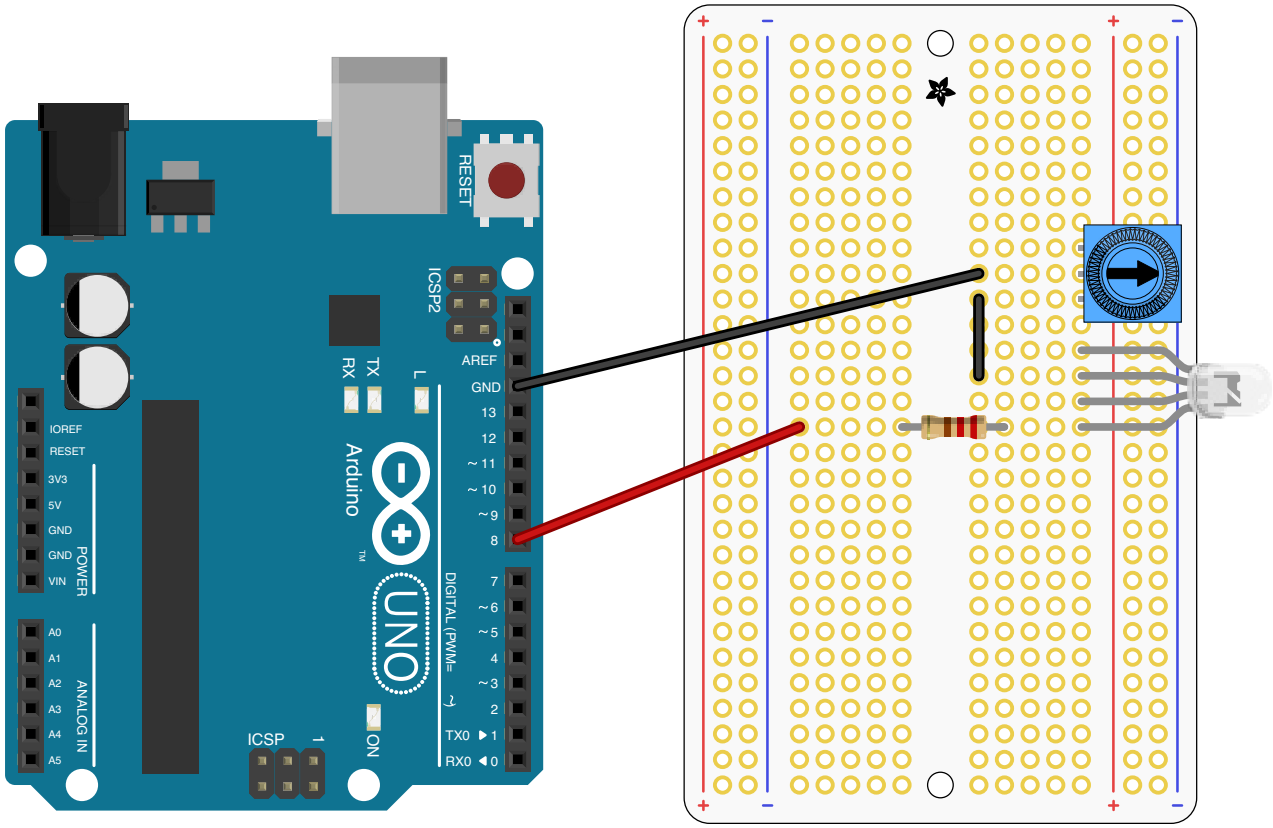
All we've done here is made the `loop()` empty (so that it doesn't do anything) and moved the `digitalWrite()` (which turns the LED on) to the `setup()` function, since we are only interested in turning the LED on once. After uploading the sketch to your Arduino, you should notice that the LED is off unless you press and hold the button!

Part 3 - adding in a potentiometer

Remember in the Introduction where we introduced the concept of a current limiting resistor and how we mentioned that the size of the resistor will impact the brightness of the LED? You might want to re-read that section if you don't remember; in any case, we are going to use a potentiometer (also known as a trimpot) to adjust the brightness of our LED. You can think of a trimpot as a variable resistance resistor; in our case we are using a 10K trimpot which means its resistance varies from 0 to 10,000 ohms.

To use the trimpot, adjust your circuit to match the following:





You can use either sketch that we've talked about (the Blink sketch or the version where the LED always stays on). Update your sketch (if needed), and try twisting the trimpot to adjust the brightness of the LED.

A few important points to remember:

- Adding the trimpot doesn't mean we can take out the other resistor! If we did remove it, and you twisted the trimpot all the way to 0 ohms, you'd blow up your little LED (not literally, but it would break!).
- We only need to use 2 of the 3 pins of the trimpot. The two outside pins will change in resistance, but in opposite directions. So when the left pin is 0 ohms, the right pin will be 10k ohms (and vice a versa).

Techshop EEE-201: Basic programming

Chapter 3

Up to this point, we've seen how to use the serial monitor and how to control the output of a single digital pin. There isn't a whole lot we can do with just that, so let's learn some more cool stuff. In this chapter we're going to read inputs, as well as cover the idea of an analog value; we'll cover the following topics:

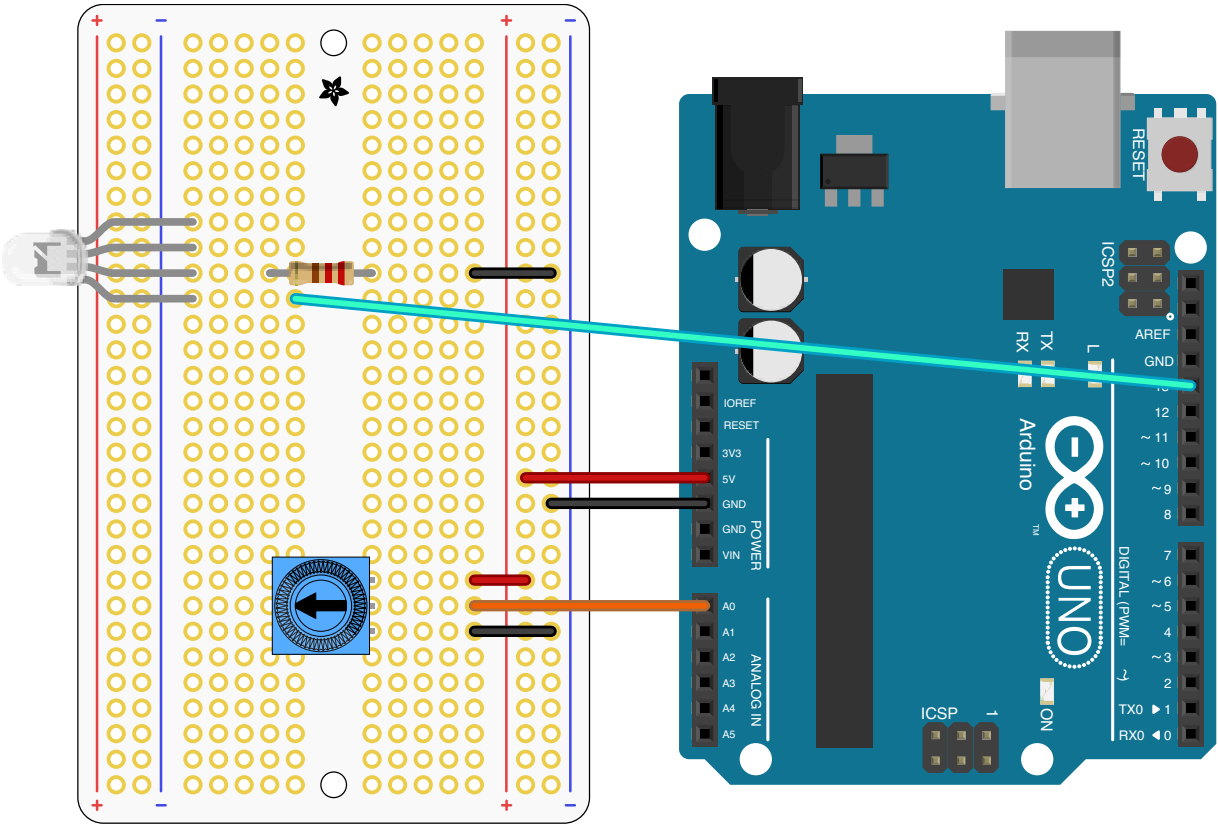
- Analog input
- Fading a LED with PWM
- variable type `int`
- `if` control structure

Table of contents

- Part 1 - analog input
- Part 2 - PWM digital output
- Part 3 - Putting it all together

Part 1 - analog input

In Chapter 2 we used the **output** of a digital pin to control the blinking of a LED; in this first part we are going to be looking at inputs. But not just any type of input, we're going to be using an analog input. Previously, we see that we could set the digital pin to either `HIGH` or `LOW`; that is, it could be set to only two values. An analog pin is different in that it can have many values, in our case, 1024 to be exact! Let's begin with an example; the circuit we want is shown below, and the sketch your after is AnalogInput.ino.



Let's break the sketch down by looking at the top part first.

```
byte sensorPin = A0;    // select the input pin for the potentiome
byte ledPin = 13;       // select the pin for the LED
int sensorValue = 0;  // variable to store the value coming from t

void setup() {
  // declare the ledPin as an OUTPUT:
  pinMode(ledPin, OUTPUT);
  // because the analog pins are always INPUT, we don't have to de
}
```

In the first three lines, we declare three variables, two `byte` types and one `int` type. Previously, we saw that a `byte` is a number that can be any value between 0 and 255 (inclusive); `int` (which stands for integer) is another type of number, but it's much bigger; it's a 16-bit number which equates to 2^16.

Bust out the calculator and you'll see that equals 65,536! However, an `int` is special in that it can hold negative values, in fact an `int` can hold the values -32,768 to 32,767; for those that are observant you can see that 32,767 + 32,768 = 65,535 (can anyone tell me what that isn't exactly equal to 2^16?). The reason we want to declare the variable `sensorValue` as an `int` is because the range we expect it to be reading from is 0 to 1023 (the range of the analog pins).

In the `setup()` function we see that we are setting the `ledPin` (pin 13) as an `OUTPUT`. Note that we don't have to define the analog pin as an input because it can only act asn an input. Let's move on to the interesting parts:

```
void loop() {
```

```
  // read the value from the sensor:
  sensorValue = analogRead(sensorPin);

  // turn the ledPin on
  digitalWrite(ledPin, HIGH);

  // stop the program for <sensorValue> milliseconds:
  delay(sensorValue);

  // turn the ledPin off:
  digitalWrite(ledPin, LOW);

  // stop the program for for <sensorValue> milliseconds:
  delay(sensorValue);
}
```

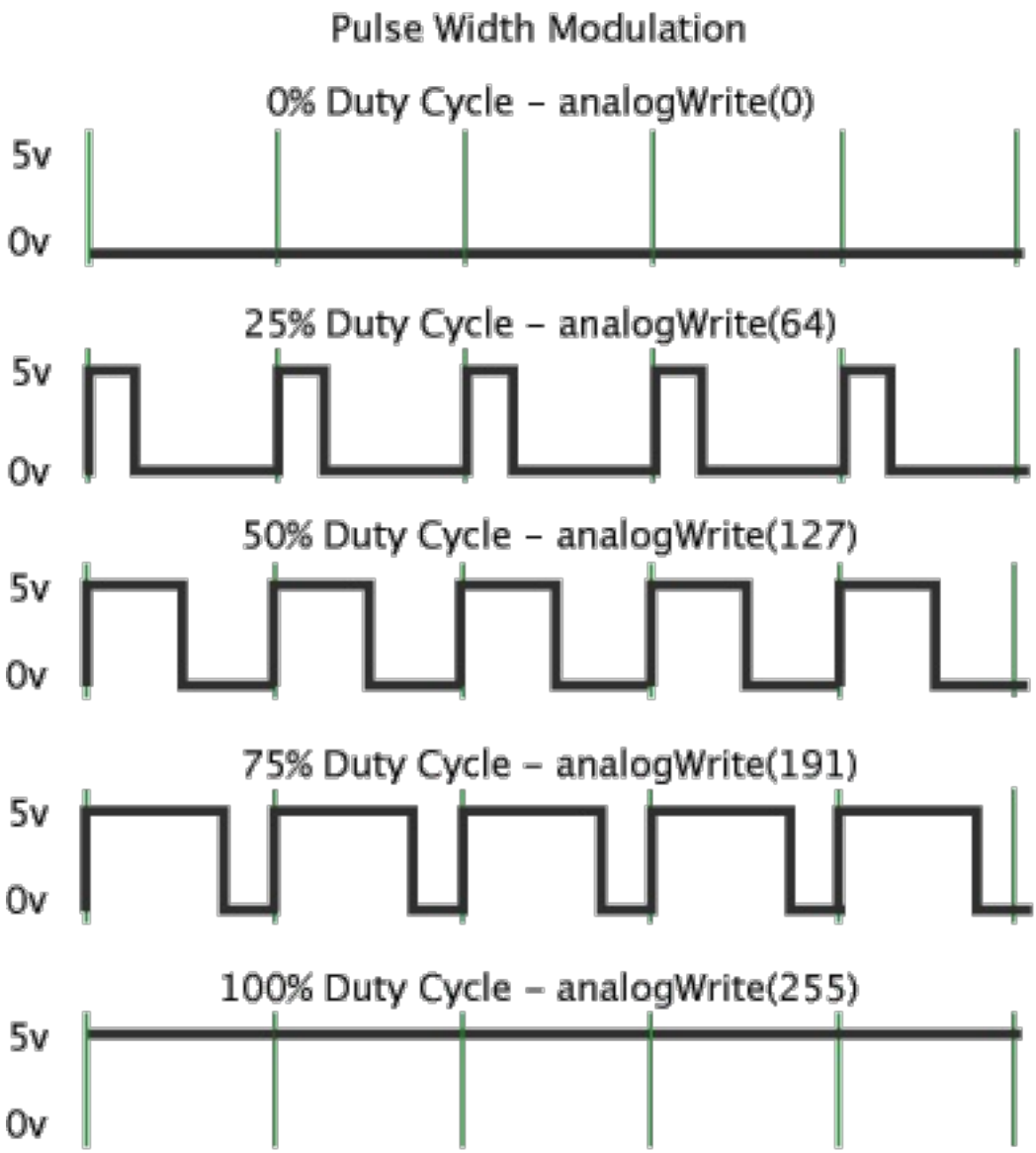
In the `loop()`, we see a function we haven't seen before `analogRead()` - this is the function which will be reading the input values on our analog pin (pin A0). As we mentioned previously, we expect this value to be anywhere between 0 and 1023 (inclusive). The remaining lines of code we've seen in Chapter 2: we turn our LED on with `digitalWrite()`, then we pause the sketch by a value of `sensorValue` number of milliseconds, then we turn off our LED with `digitalWrite()` and then we pause again. The interesting part here is that, instead of pausing with a fixed amount, we can control the length of pause with the value that is read by `analogRead()` which we control with the trimpot. That means, we can directly control how fast our LED blinks just by twisting the trimpot to different values!

Explore: how would we change our sketch to see what value our variable `sensorValue` is set to?

Part 2 - PWM digital output

In part 1 we covered analog inputs, but what about analog outpus? Unfortunately the Arduino can't directly do analog outputs, however we can fake it through the use of the Arudino software through a technique called pulse width modulation (PWM):

> Pulse Width Modulation, or PWM, is a technique for getting analog results with digital means. Digital control is used to create a square wave, a signal switched between on and off. This on-off pattern can simulate voltages in between full on (5 Volts) and off (0 Volts) by changing the portion of the time the signal spends on versus the time that the signal spends off. The duration of "on time" is called the pulse width. To get varying analog values, you change, or modulate, that pulse width. If you repeat this on-off pattern fast enough with an LED for example, the result is as if the signal is a steady voltage between 0 and 5v controlling the brightness of the LED.
>
> In the graphic below, the green lines represent a regular time period. This duration or period is the inverse of the PWM frequency. In other words, with Arduino's PWM frequency at about 500Hz, the green lines would measure 2 milliseconds each. A call to analogWrite() is on a scale of 0 - 255, such that analogWrite(255) requests a 100% duty cycle (always on), and analogWrite(127) is a 50% duty cycle (on half the time) for example.



Enough with the theory, let's take a look with a hands on example, upload the Fade sketch to your Aruino.

The only new part in this sketch is the use of the `analogWrite()` function:

```
void loop() {
  analogWrite(ledPin, 0); // update analog value on ledPin
  delay(500); // delay 500 miliseconds

  analogWrite(ledPin, 50); // update analog value on ledPin
  delay(500); // delay 500 miliseconds

  analogWrite(ledPin, 100); // update analog value on ledPin
  delay(500); // delay 500 miliseconds
```

```
  analogWrite(ledPin, 150); // update analog value on ledPin
  delay(500); // delay 500 miliseconds

  analogWrite(ledPin, 200); // update analog value on ledPin
  delay(500); // delay 500 miliseconds

  analogWrite(ledPin, 255); // update analog value on ledPin
  delay(500); // delay 500 miliseconds
}
```

All we're doing is generating an analog output through PWM at various frequencies. We start it at 0% duty cycle (LED is off), and then increase the frequency slowly, pausing for half a second at each frequency change. What you should be seeing is that the LED changes in brightness as the PWM frequency is increased.

Part 3 - Putting it all together

So far we've completely ignored the fact that the LED we've been using is actually 3 LEDs built into one. So, for this last part of this class, we are going to put everything we learned together in order to control all three colors of the LED. But before we do that, we need to learn one more thing: the `if` control statement.

An `if` statement allows us to run certain parts of code only *if* a condition, or multiple conditions, are true. An example might be: *turn LED on, if button 1 is pressed down*; the general format looks like this:

```
if (condition is true)
{
  // do something
}
```

The condition statment uses *comparison operators* and *boolean operators* to evalute whether a condition is true. Let's take a look at those.

Comparison operators

These allow us to check a single condition in multiple ways; the following operators are available in Arduino:

> x == y (x is equal to y)
>
> x != y (x is not equal to y)
>
> x < y (x is less than y)
>
> x > y (x is greater than y)
>
> x <= y (x is less than or equal to y)
>
> x >= y (x is greater than or equal to y)

So, for example, if we were reading an analog value and storing it to the variable `analogVal`, and we wanted to turn a LED on only if the `analogVal` was larger than 500, we'd write something like:

```
if (analogVal >= 500)
{
    digitalWrite(ledPin, HIGH); // turn on LED
}
```

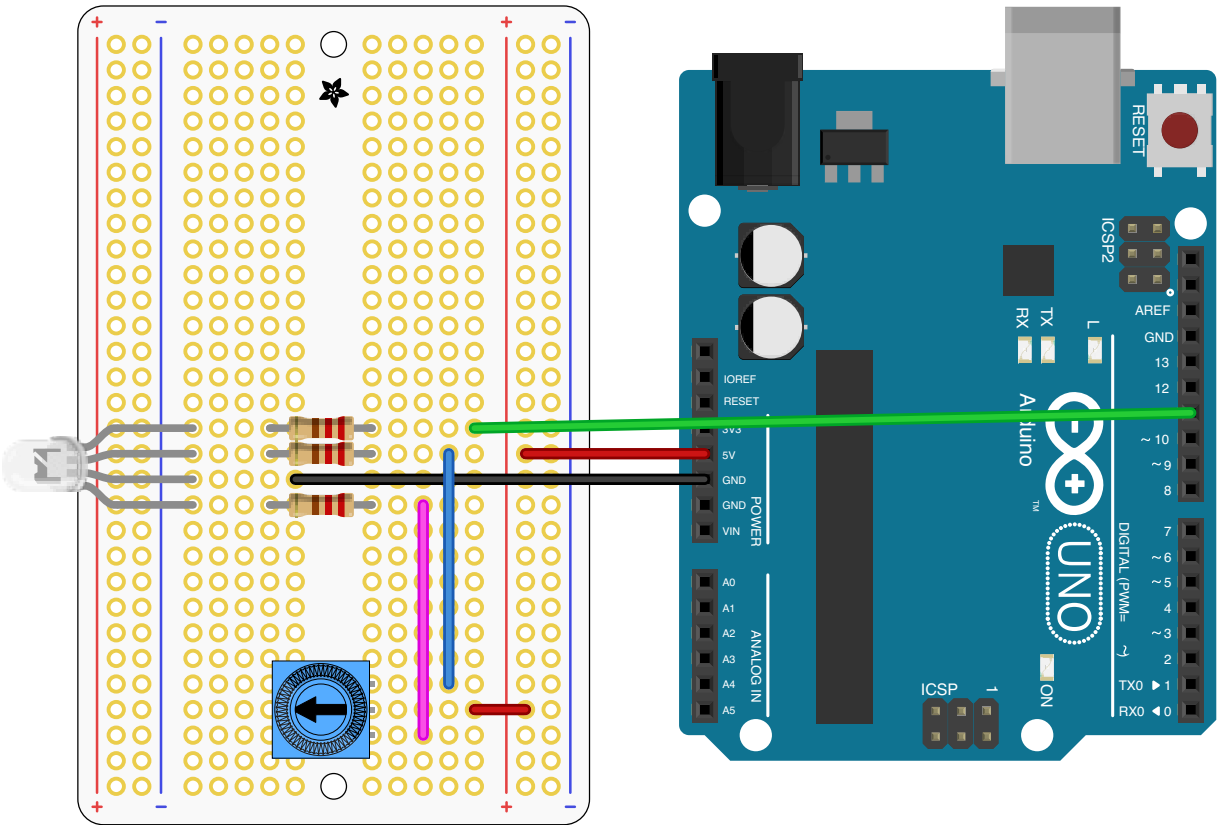
Boolean operators

These allow us to chain multiple comparison operators together through the use of a boolean `AND`, `OR`, or `NOT`. To exend the above example,

let's say we only want to turn the LED on when the `analogVal` is between 500 and 700, we'd write:

```
if (analogVal >= 500 && analogVal <= 700)
{
    digitalWrite(ledPin, HIGH); // turn on LED
}
```

Let's get to it! The circuit you need to build is seen below and the sketch we're going to use is the Fadelf.ino sketch.



As you can see in the circuit, we've now hooked up all three color components of the LED:

- green (G) is connected to a digital pin (13)
- blue (B) is connected to the trimpot
- red (R) is connected to a digital PWM pin (5)

TODO explain circuit/sketch

Explore:

- How can we see what voltage is being set by the trimpot?
- Right now the trimpot controls two of the RGB channels, how can we change our circuit so that one channel automatically blinks with a frequency of 1Hz (once a second)?
- How can we make the LED show a white color? How about purple?
- How could we turn all three colors on/off with a single pin?

Techshop EEE-201: Basic programming

Appendix

In this appendix we will quickly cover how to use a breadboard; knowing how to use one correctly will be really important in the coming chapters.

An electronics breadboard (as opposed to the type on which sandwiches are made) is actually referring to a solderless breadboard. These are great units for making temporary circuits and prototyping, and they require absolutely no soldering.

Prototyping is the process of testing out an idea by creating a preliminary model from which other forms are developed or copied, and it is one of the most common uses for breadboards. If you aren't sure how a circuit will react under a given set of parameters, it's best to build a prototype and test it out.

For those new to electronics and circuits, breadboards are often the best place to start. That is the real beauty of breadboards–they can house

both the simplest circuit as well as very complex circuits. As you'll see later in this tutorial, if your circuit outgrows its current breadboard, others can be be attached to accommodate circuits of all sizes and complexities.

We will be using breadboards to build circuits; in this class, we will often use circuit diagrams as a reference for building those circuits. The following document will guide you through the basics of circuit diagrams; it is meant as a starting guide for orienting yourself in some of the diagrams you'll expect to see in this course.

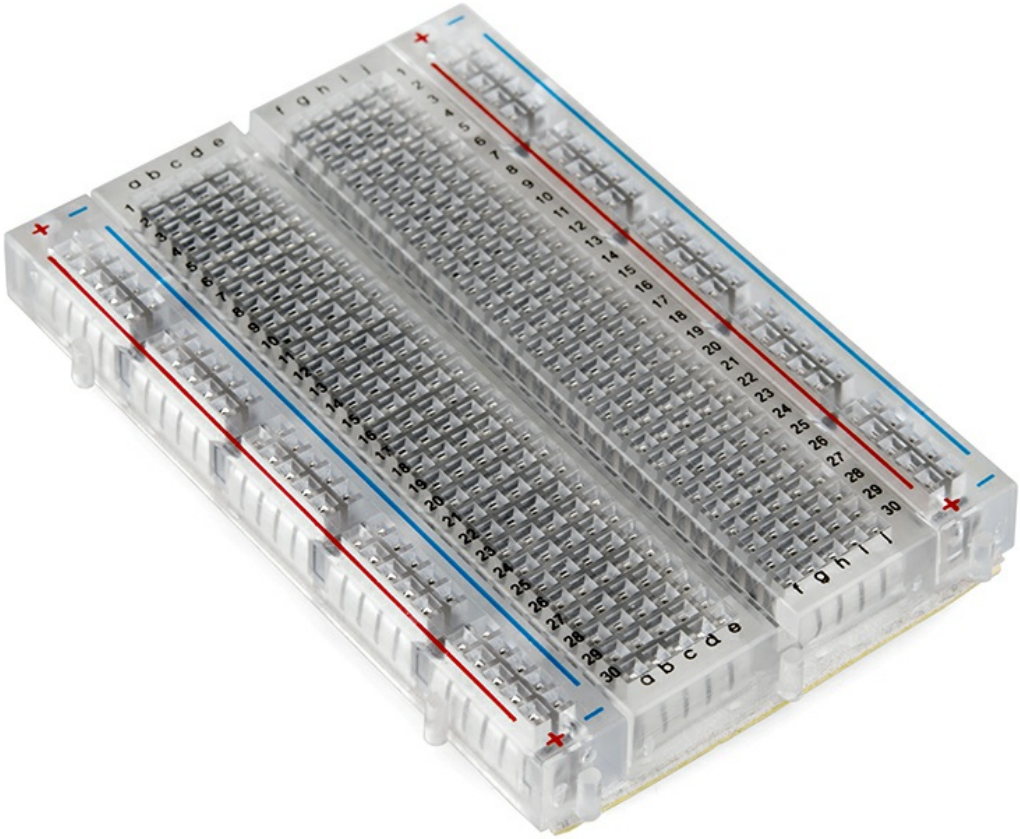
The content is adapted from Sparkfun: How to Read a Schematic

Table of Contents

- How breadboards work
- Circuit Diagram Introduction
- Schematic Symbols (Part 1)
    - Resistors
        - Potentiometers and Variable Resistors
    - Capacitors
    - Inductors
    - Switches
    - Power Sources
        - DC or AC Voltage Sources
        - Batteries
        - Voltage Nodes
- Schematic Symbols (Part 2)
    - Diodes
    - Transistors
        - Bipolar Junction Transistors (BJTs)

        - Metal Oxide Field-Effect Transistors (MOSFETs)
    - Digital Logic Gates
    - Integrated Circuits
    - Unique ICs: Op Amps, Voltage Regulators
    - Miscellany
        - Crystals and Resonators
        - Headers and Connectors
        - Motors, Transformers, Speakers, and Relays
        - Fuses and PTCs
- Reading Schematics
    - Nets, Nodes and Labels
        - Junctions and Nodes
        - Net Names
- Schematic Reading Tips
    - Identify Blocks
    - Recognize Voltage Nodes
    - Reference Component Datasheets

How breadboards work



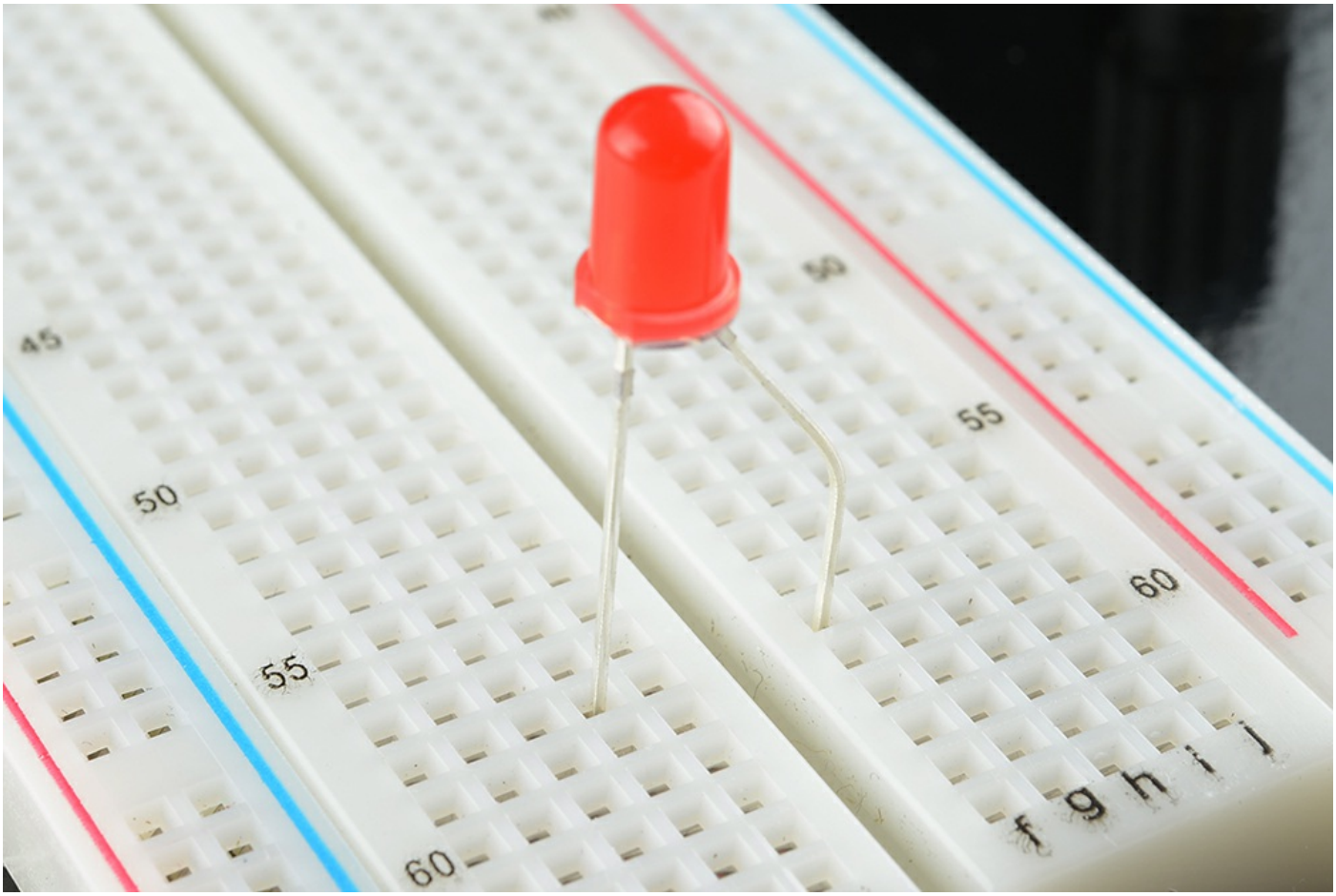
Here we have a breadboard where the adhesive backing has been removed. You can see lots of horizontal rows of metal strips on the bottom of the breadboard.

The tops of the metal rows have little clips that hide under the plastic holes. These clips allow you to stick a wire or the leg of a component into the exposed holes on a breadboard, which then hold it in place.

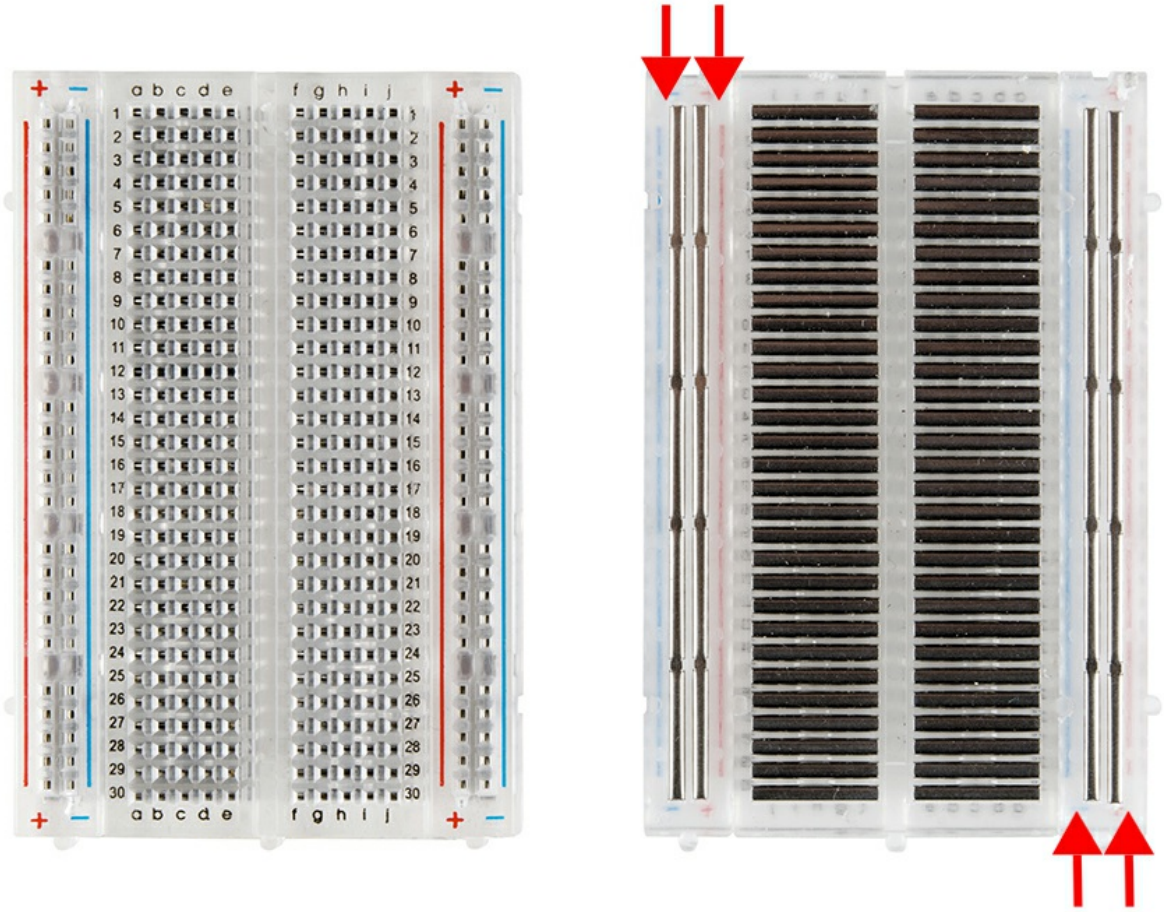
Once inserted that component will be electrically connected to anything else placed in that row. This is because the metal rows are conductive and allow current to flow from any point in that strip.

Notice that there are only five clips on this strip. This is typical on almost all breadboards. Thus, you can only have up to five components connected in one particular section of the breadboard. The row has ten holes, so why can you only connect five components? You'll also notice that each horizontal row is separated by a ravine, or crevasse, in the

middle of the breadboard. This ravine isolates both sides of a given row from one another, and they are not electrically connected. We won't discuss the purpose of this just yet, but, for now, just know that each side of a given row is disconnected from the other, leaving you with five spots for components on either side.



Aside from horizontal rows, breadboards usually have what are called power rails that run vertically along the sides.



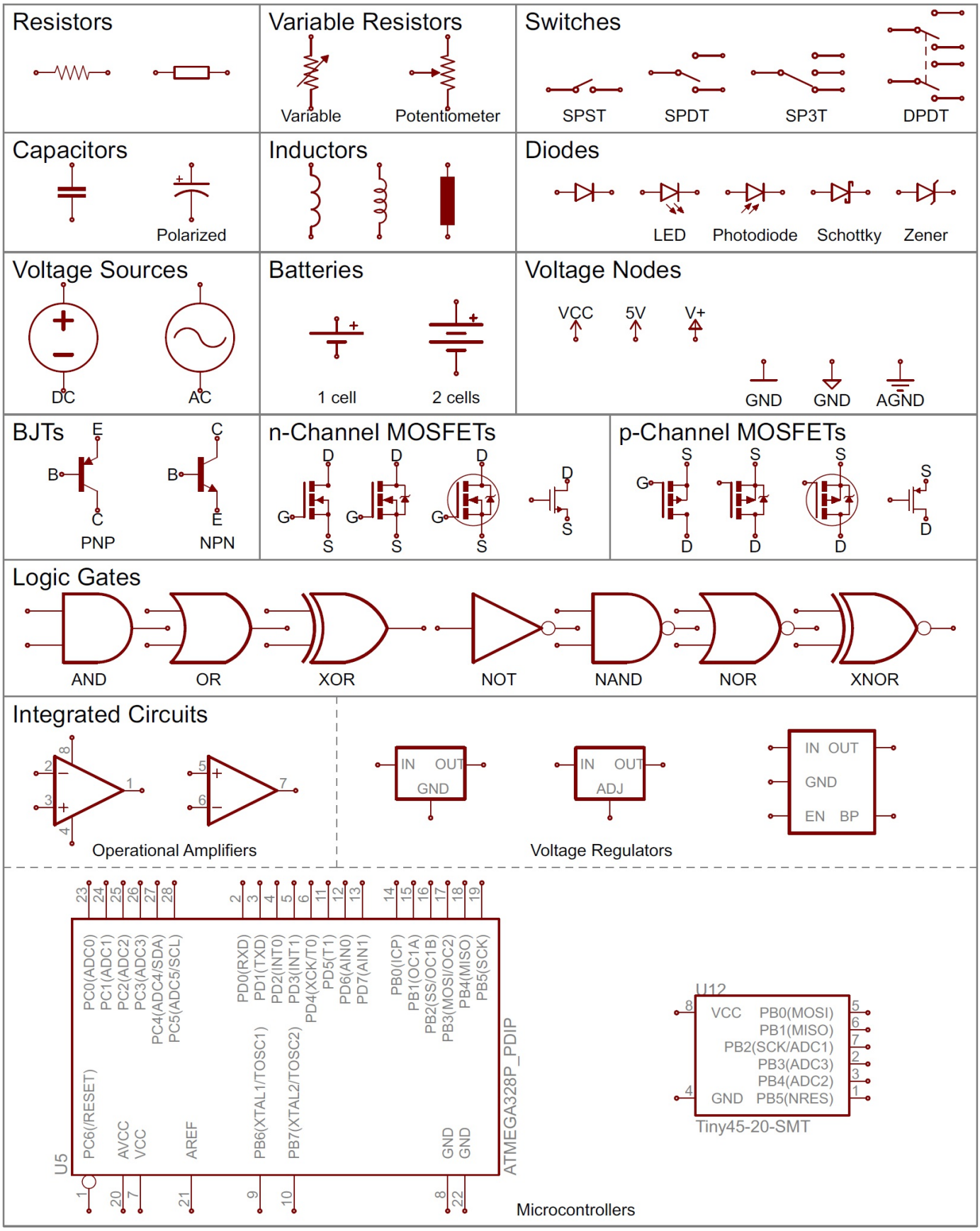
These power rails are metal strips that are identical to the ones that run horizontally, except they are, typically*, all connected. When building a circuit, you tend to need power in lots of different places. The power rails give you lots of easy access to power wherever you need it in your circuit. Usually they will be labeled with a ‘+’ and a ‘-’ and have a red and blue or black stripe, to indicate the positive and negative side.

It is important to be aware that the power rails on either side are not connected, so if you want the same power source on both sides, you will need to connect the two sides with some jumper wires. Keep in mind that the markings are there just as a reference. There is no rule that says you have to plug power into the ‘+’ rail and ground into the ‘-'rail, though it’s good practice to keep everything in order.

Circuit Diagram Introduction

Schematics are our map to designing, building, and troubleshooting circuits. Understanding how to read and follow schematics is an important skill for any electronics engineer.

This tutorial should turn you into a fully literate schematic reader! We'll go over all of the fundamental schematic symbols:



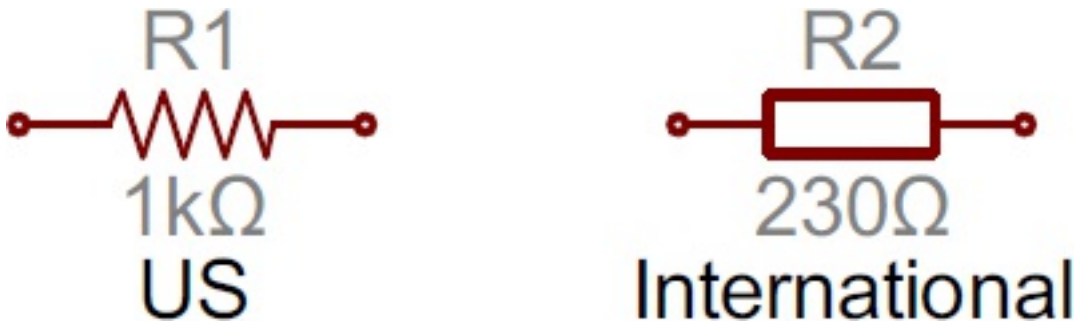
Then we'll talk about how those symbols are connected on schematics to create a model of a circuit. We'll also go over a few tips and tricks to watch out for.

Schematic Symbols (Part 1)

Are you ready for a barrage of circuit components? Here are some of the standardized, basic schematic symbols for various components.

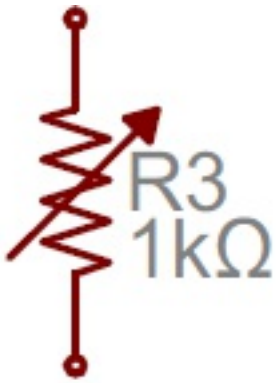
Resistors

The most fundamental of circuit components and symbols! Resistors on a schematic are usually represented by a few zig-zag lines, with two terminals extending outward. Schematics using international symbols may instead use a featureless rectangle, instead of the squiggles.

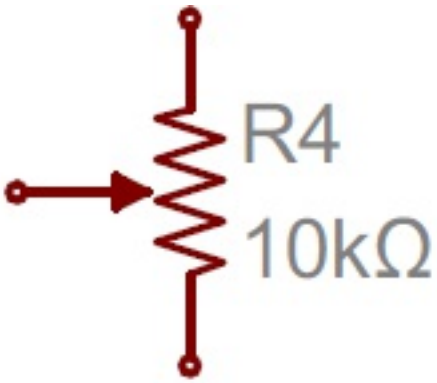


Potentiometers and Variable Resistors

Variable resistors and potentiometers each augment the standard resistor symbol with an arrow. The variable resistor remains a two-terminal device, so the arrow is just laid diagonally across the middle. A potentiometer is a three-terminal device, so the arrow becomes the third terminal (the wiper).



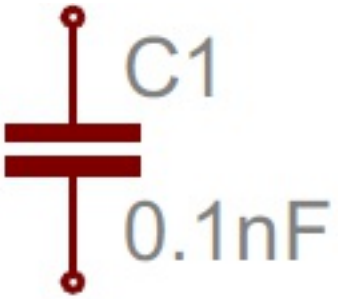
Variable



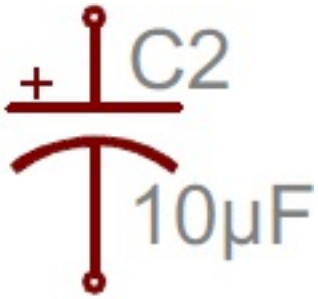
Potentiometer

Capacitors

There are two commonly used capacitor symbols. One symbol represents a polarized (usually electrolytic or tantalum) capacitor, and the other is for non-polarized caps. In each case there are two terminals, running perpendicularly into plates.



Non-polarized



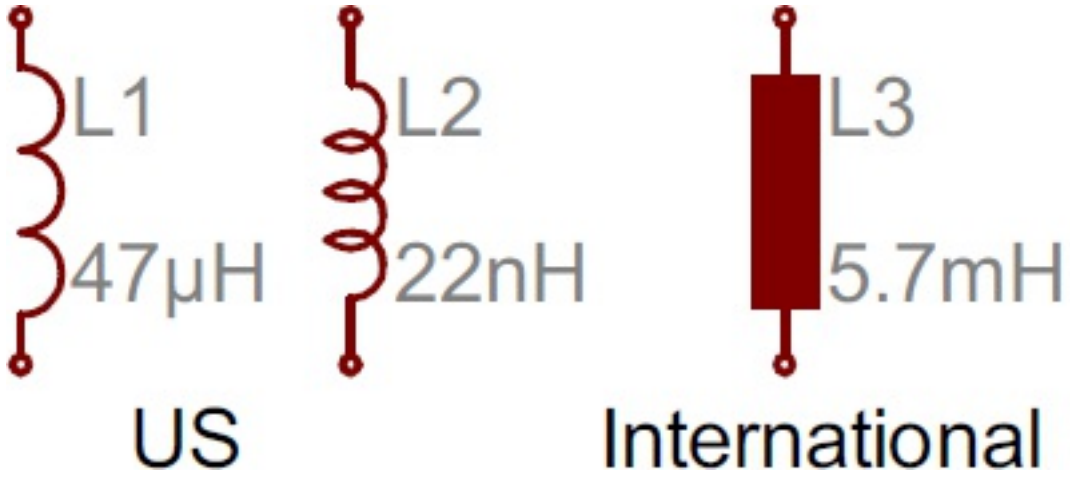
Polarized

The symbol with one curved plate indicates that the capacitor is polarized. The curved plate represents the cathode of the capacitor, which should be at a lower voltage than the positive, anode pin. A plus sign might also be added to the positive pin of the polarized capacitor symbol.

Inductors

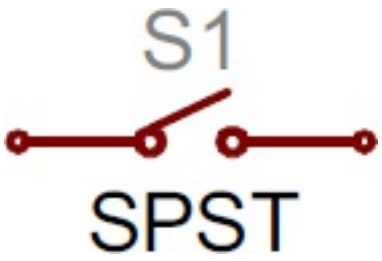
Inductors are usually represented by either a series of curved bumps, or loopy coils. International symbols may just define an inductor as a filled-

in rectangle.

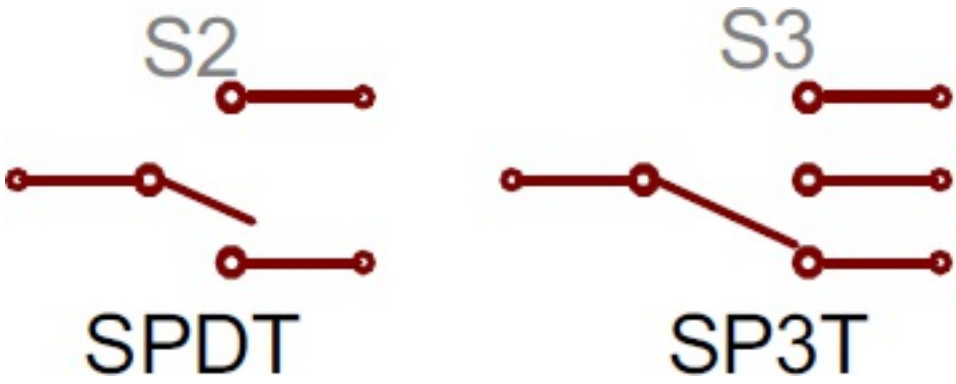


Switches

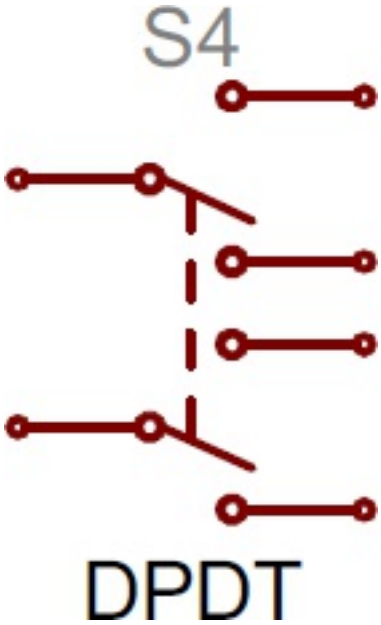
Switches exist in many different forms. The most basic switch, a single-pole/single-throw (SPST), is two terminals with a half-connected line representing the actuator (the part that connects the terminals together).



Switches with more than one throw, like the SPDT and SP3T below, add more landing spots for the the actuator.



Switches with multiple poles, usually have multiple, alike switches with a dotted line intersecting the middle actuator.

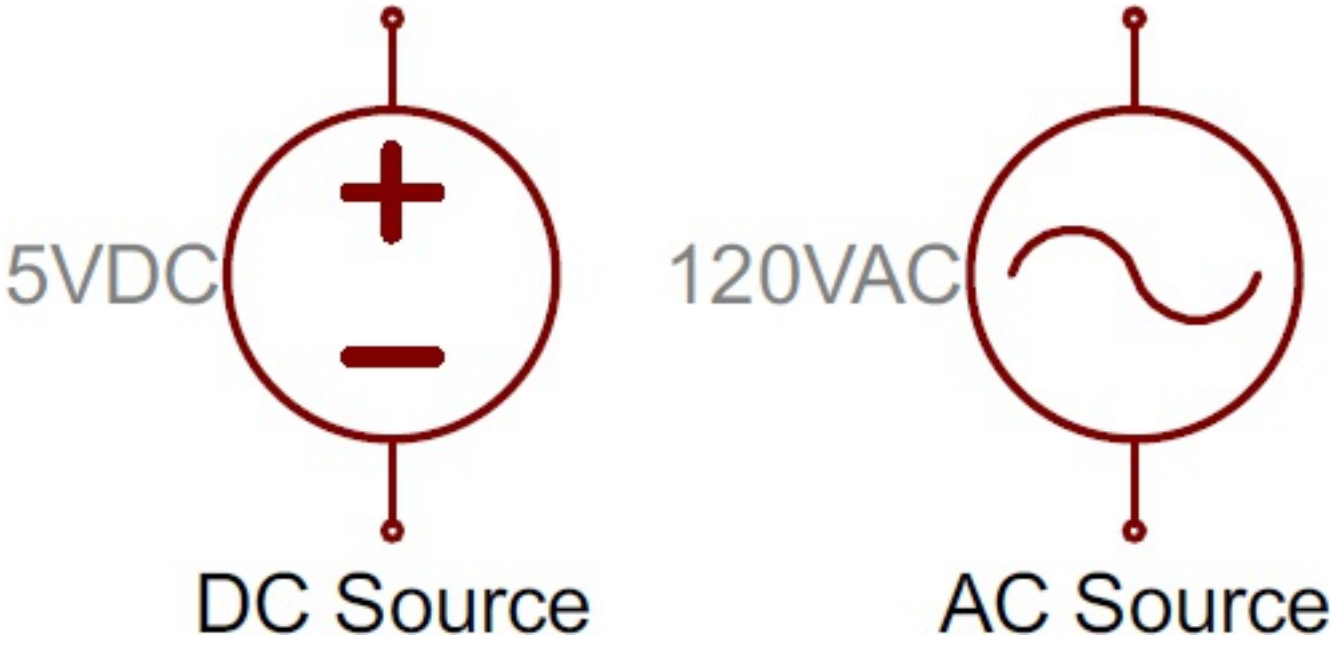


Power Sources

Just as there are many options out there for powering your project, there are a wide variety of power source circuit symbols to help specify the power source.

DC or AC Voltage Sources

Most of the time when working with electronics, you'll be using constant voltage sources. We can use either of these two symbols to define whether the source is supplying direct current (DC) or alternating current (AC):



Batteries

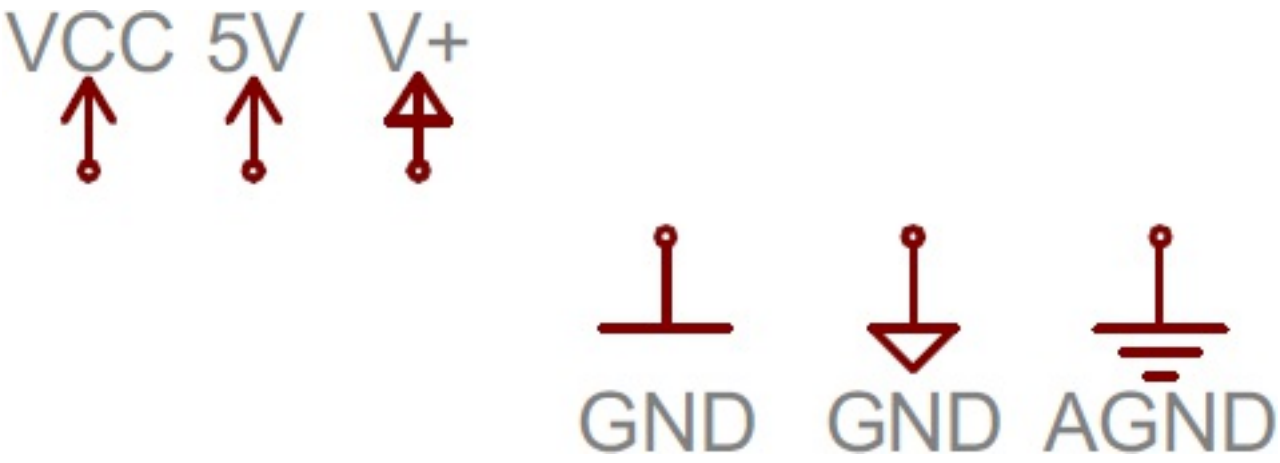
Batteries, whether they're those cylindrical, alkaline AA's or rechargeable lithium-polymers, usually look like a pair of disproportionate, parallel lines:

More pairs of lines usually indicates more series cells in the battery. Also, the longer line is usually used to represent the positive terminal, while the shorter line connects to the negative terminal.

Voltage Nodes

Sometimes – on really busy schematics especially – you can assign special symbols to node voltages. You can connect devices to these one-terminal symbols, and it'll be tied directly to 5V, 3.3V, VCC, or GND

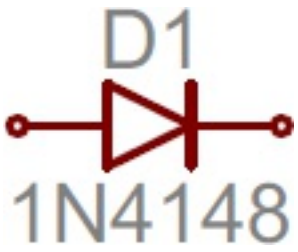
(ground). Positive voltage nodes are usually indicated by an arrow pointing up, while ground nodes usually involve one to three flat lines (or sometimes a down-pointing arrow or triangle).



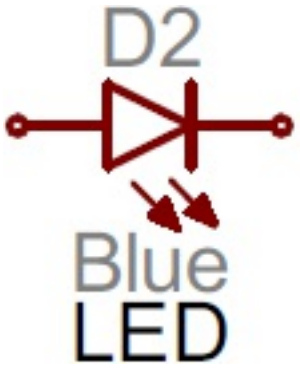
Schematic Symbols (Part 2)

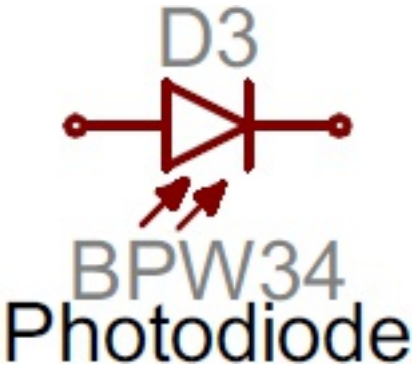
Diodes

Basic diodes are usually represented with a triangle pressed up against a line. Diodes are also polarized, so each of the two terminals require distinguishing identifiers. The positive, anode is the terminal running into the flat edge of the triangle. The negative, cathode extends out of the line in the symbol (think of it as a - sign).



There are a all sorts of different types of diodes, each of which has a special riff on the standard diode symbol. Light-emitting diodes (LEDs) augment the diode symbol with a couple lines pointing away. Photodiodes, which generate energy from light (basically, tiny solar cells), flip the arrows around and point them toward the diode.





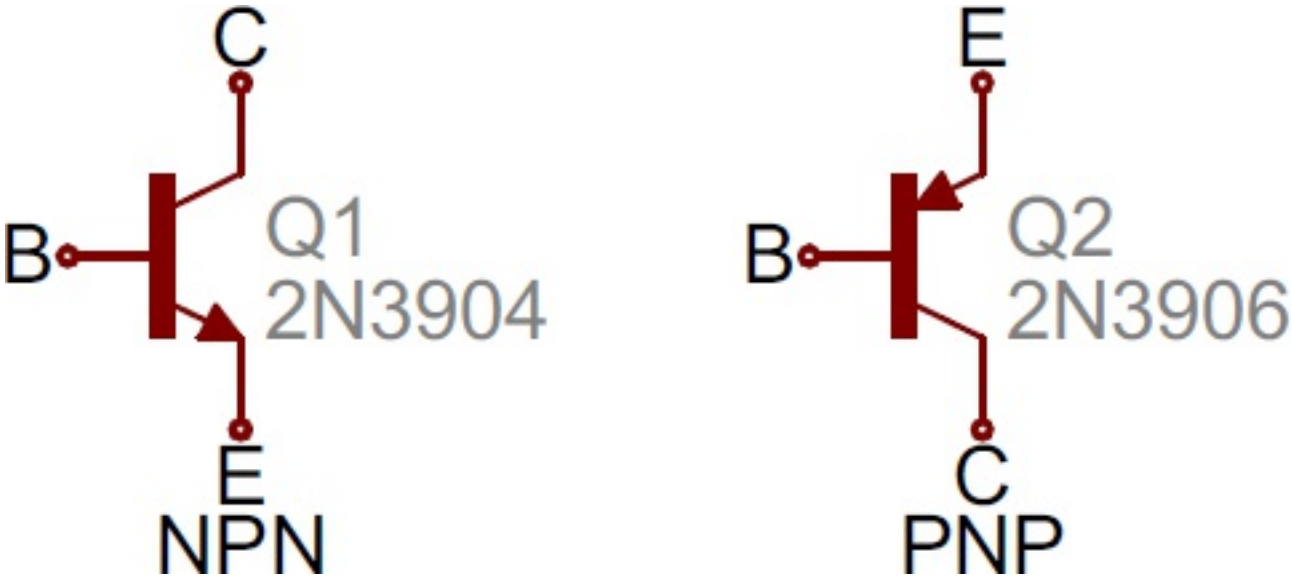
Other special types of diodes, like Schottky's or zeners, have their own symbols, with slight variations on the bar part of the symbol.

Transistors

Transistors, whether they're BJTs or MOSFETs, can exist in two configurations: positively doped, or negatively doped. So for each of these types of transistor, there are at least two ways to draw it.

Bipolar Junction Transistors (BJTs)

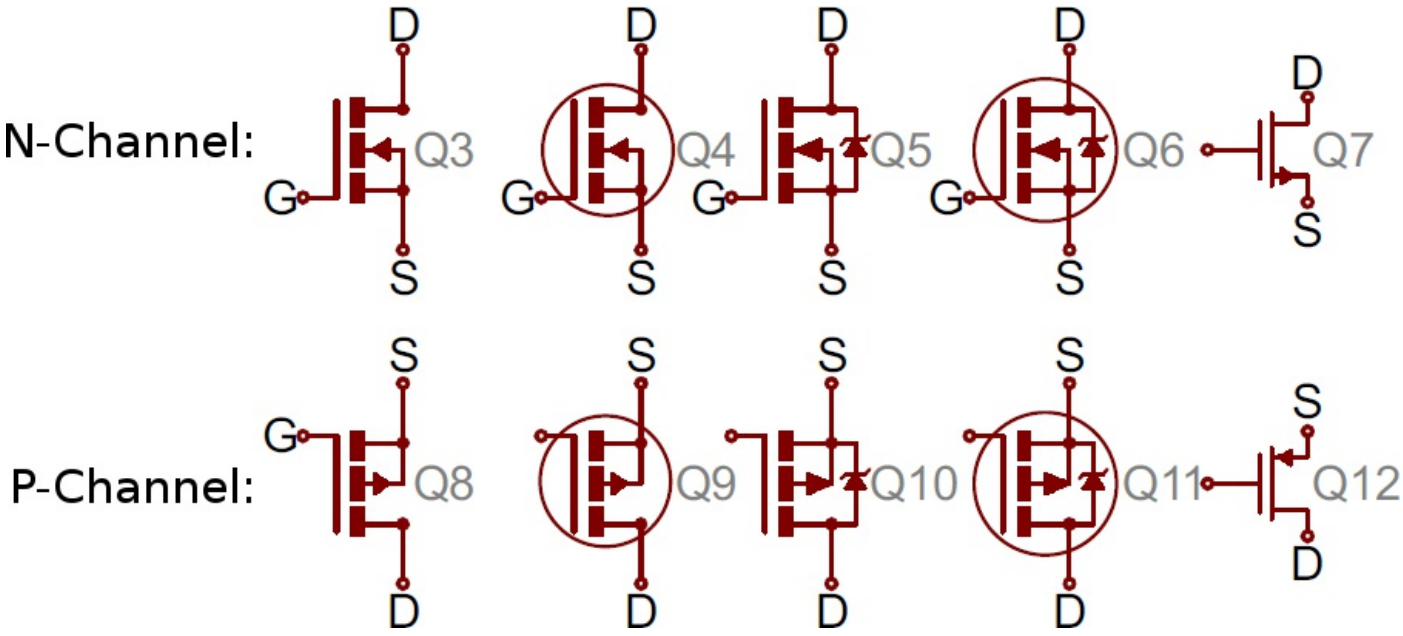
BJTs are three-terminal devices; they have a collector (C), emitter (E), and a base (B). There are two types of BJTs – NPNs and PNPs – and each has its own unique symbol.



The collector (C) and emitter (E) pins are both in-line with each other, but the emitter should always have an arrow on it. If the arrow is pointing inward, it's a PNP, and, if the arrow is pointing outward, it's an NPN. A mnemonic for remembering which is which is "NPN: not pointing in."

Metal Oxide Field-Effect Transistors (MOSFETs)

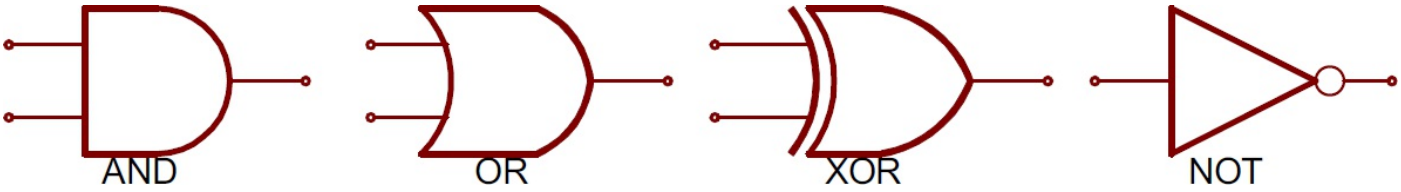
Like BJTs, MOSFETs have three terminals, but this time they're named source (S), drain (D), and gate (G). And again, there are two different versions of the symbol, depending on whether you've got an n-channel or p-channel MOSFET. There are a number of commonly used symbols for each of the MOSFET types:



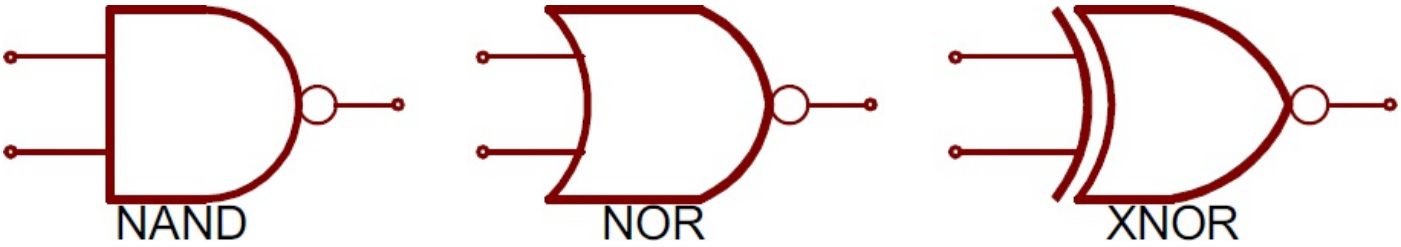
The arrow in the middle of the symbol (called the bulk) defines whether the MOSFET is n-channel or p-channel. If the arrow is pointing in means it's a n-channel MOSFET, and if it's pointing out it's a p-channel. Remember: "n is in" (kind of the opposite of the NPN mnemonic).

Digital Logic Gates

Our standard logic functions – AND, OR, NOT, and XOR – all have unique schematic symbols:



Adding a bubble to the output negates the function, creating NANDs, NORs, and XNORs:

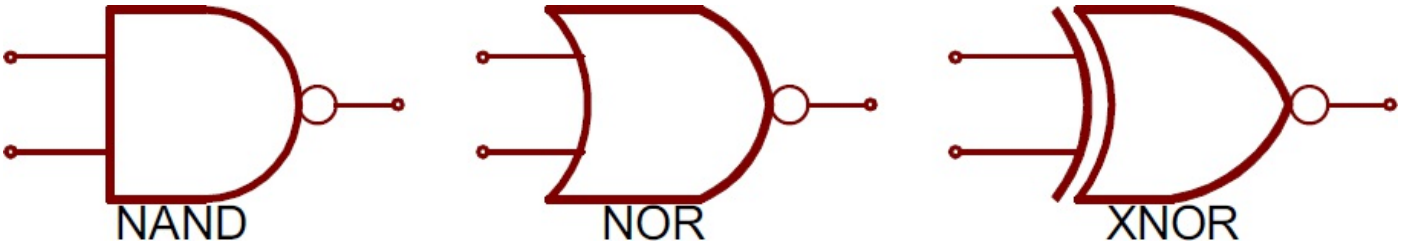


Negated logic gates

They may have more than two inputs, but the shapes should remain the same (well, maybe a bit bigger), and there should still only be one output.

Integrated Circuits

Integrated circuits accomplish such unique tasks, and are so numerous, that they don't really get a unique circuit symbol. Usually, an integrated circuit is represented by a rectangle, with pins extending out of the sides. Each pin should be labeled with both a number, and a function.

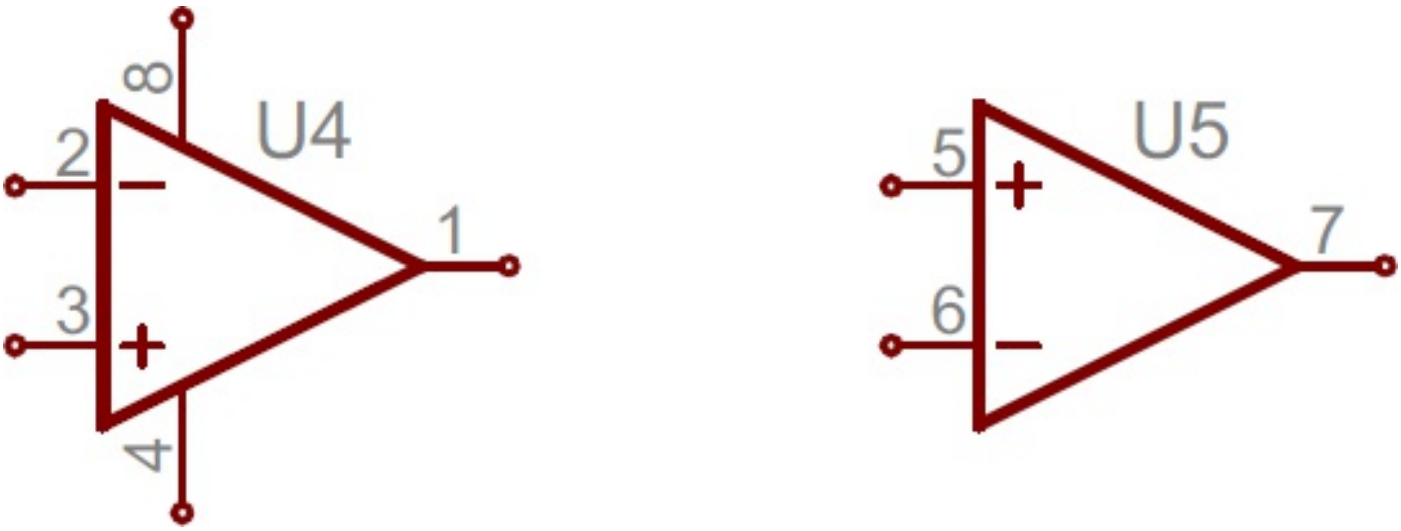


Schematic symbols for an ATmega328 microcontroller (commonly found on Arduinos), an ATSHA204 encryption IC, and an ATtiny45 MCU. As you can see, these components greatly vary in size and pin-counts.

Because ICs have such a generic circuit symbol, the names, values and labels become very important. Each IC should have a value precisely identifying the name of the chip.

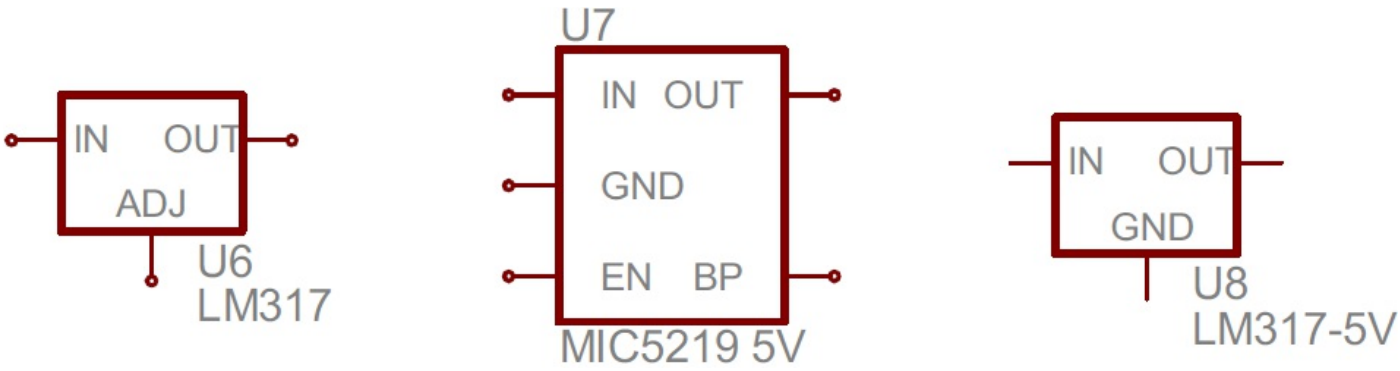
Unique ICs: Op Amps, Voltage Regulators

Some of the more common integrated circuits do get a unique circuit symbol. You'll usually see operation amplifiers laid out like below, with 5 total terminals: a non-inverting input (+), inverting input (-), output, and two power inputs.



Often, there will be two op amps built into one IC package requiring only one pin for power and one for ground, which is why the one on the right only has three pins.

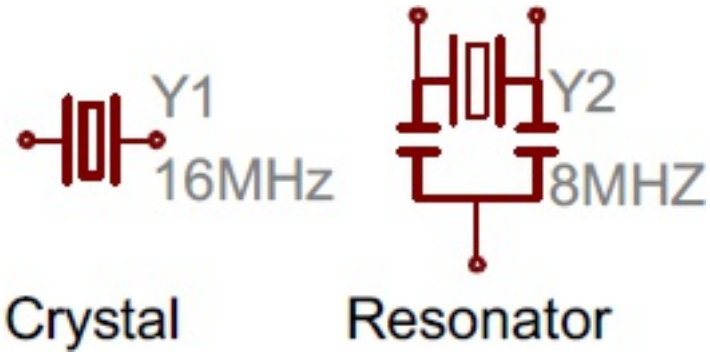
Simple voltage regulators are usually three-terminal components with input, output and ground (or adjust) pins. These usually take the shape of a rectangle with pins on the left (input), right (output) and bottom (ground/adjust).



Miscellany

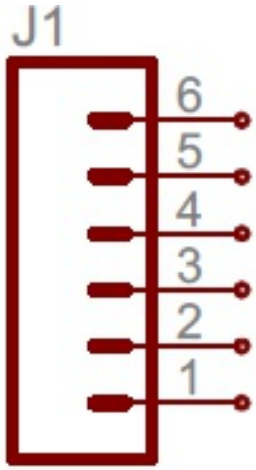
Crystals and Resonators

Crystals or resonators are usually a critical part of microcontroller circuits. They help provide a clock signal. Crystal symbols usually have two terminals, while resonators, which add two capacitors to the crystal, usually have three terminals.

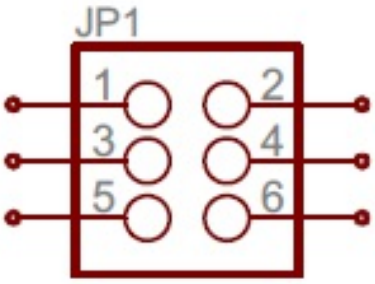


Headers and Connectors

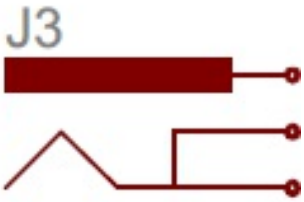
Whether it's for providing power, or sending out information, connectors are a requirement on most circuits. These symbols vary depending on what the connector looks like, here's a sampling:



6-Pin Connector



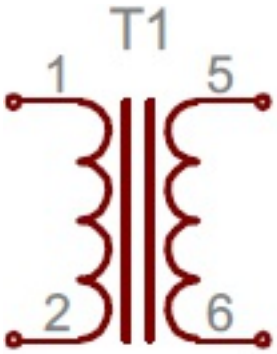
2x3-Pin connector

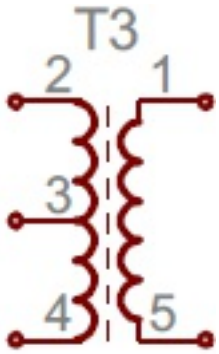


Barrel Jack

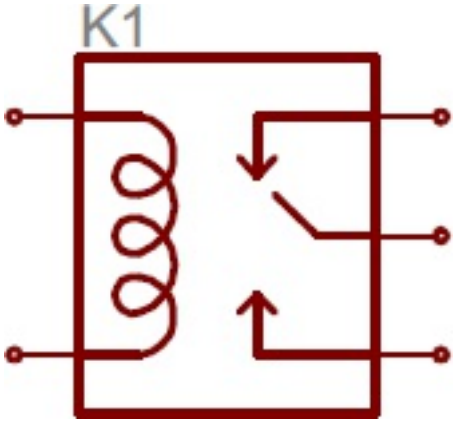
Motors, Transformers, Speakers, and Relays

We'll lump these together, since they (mostly) all make use of coils in some way. Transformers (not the more-than-meets-the-eye kind) usually involve two coils, butted up against each other, with a couple lines separating them:

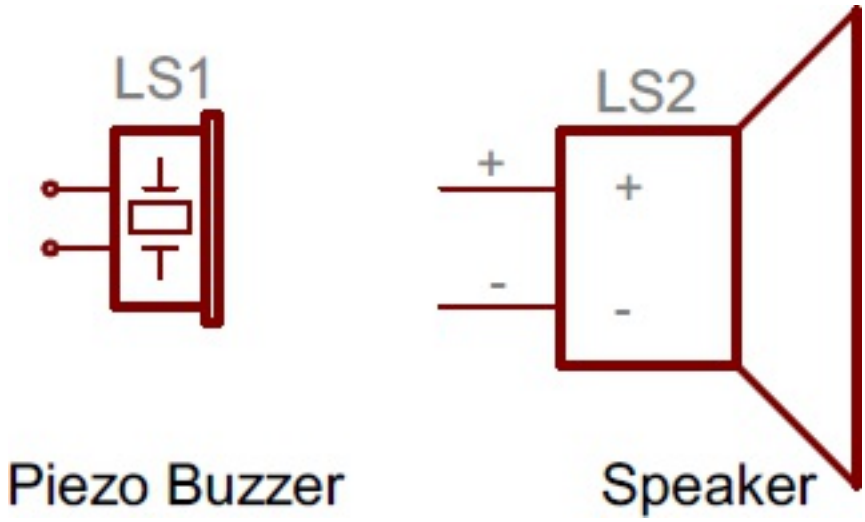




Relays usually pair a coil with a switch:



Speakers and buzzers usually take a form similar to their real-life counterparts:



And motors generally involve an encircled “M”, sometimes with a bit more embellishment around the terminals:

Fuses and PTCs

Fuses and PTCs – devices which are generally used to limit large inrushes of current – each have their own unique symbol:

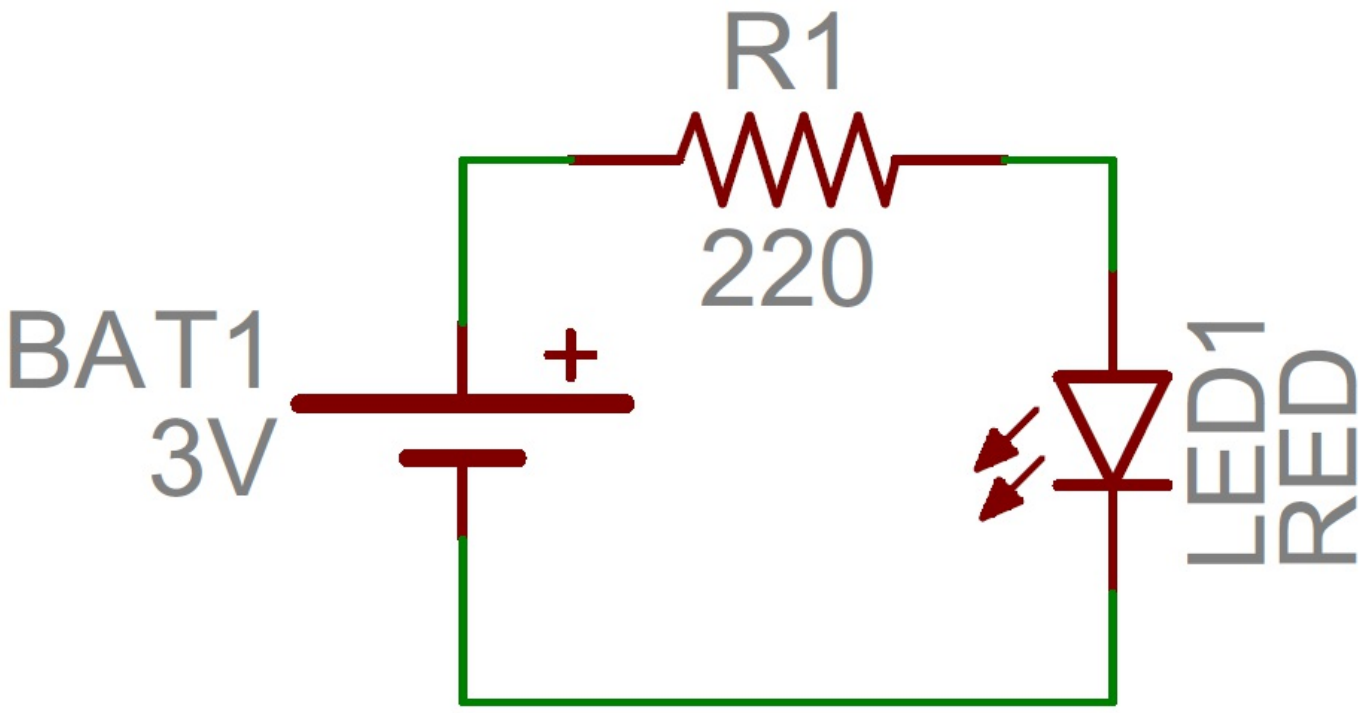
The PTC symbol is actually the generic symbol for a thermistor, a temperature-dependent resistor (notice the international resistor symbol in there?).

Reading Schematics

Understanding which components are which on a schematic is more than half the battle towards comprehending it. Now all that remains is identifying how all of the symbols are connected together.

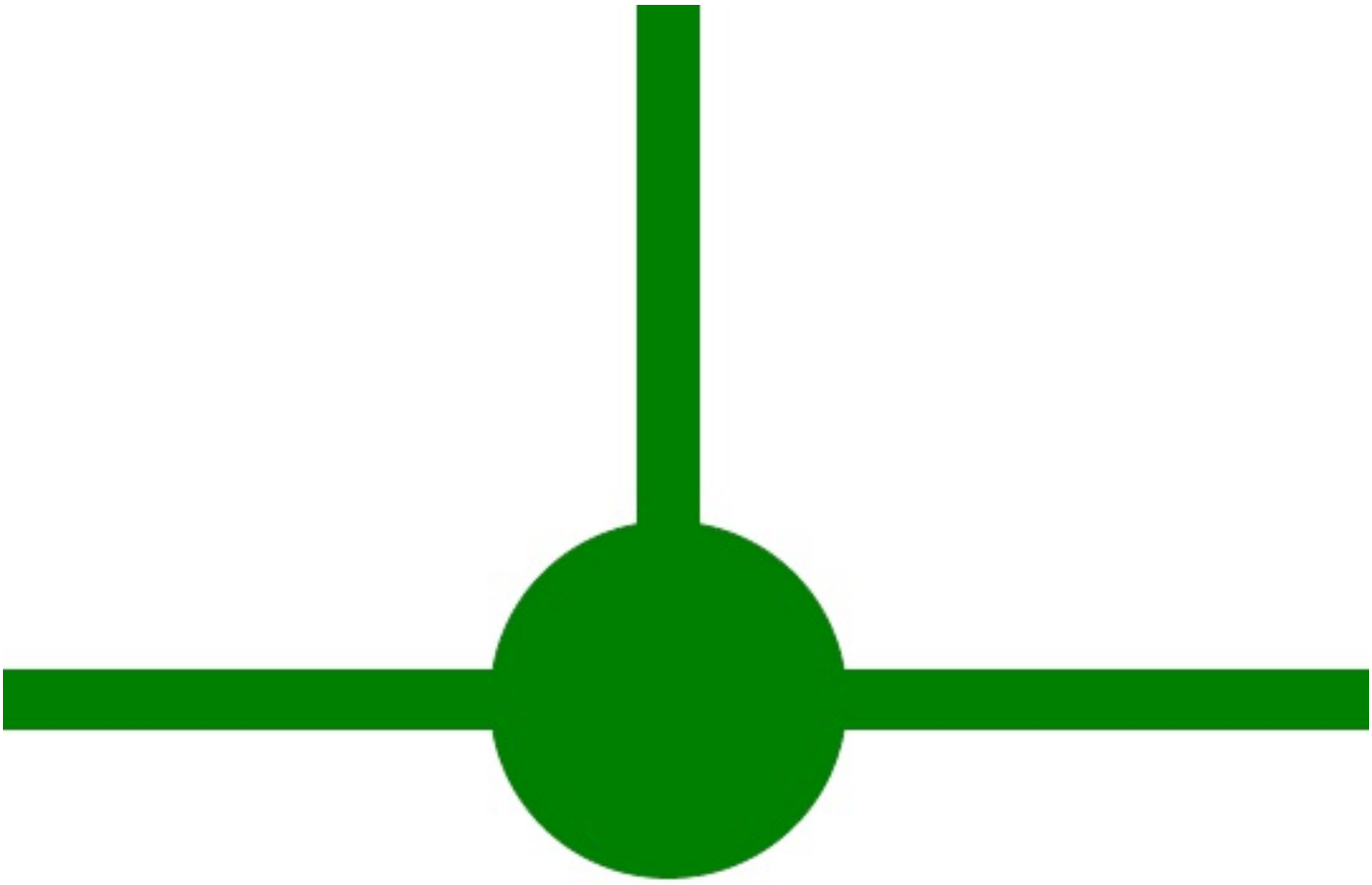
Nets, Nodes and Labels

Schematic nets tell you how components are wired together in a circuit. Nets are represented as lines between component terminals. Sometimes (but not always) they're a unique color, like the green lines in this schematic:

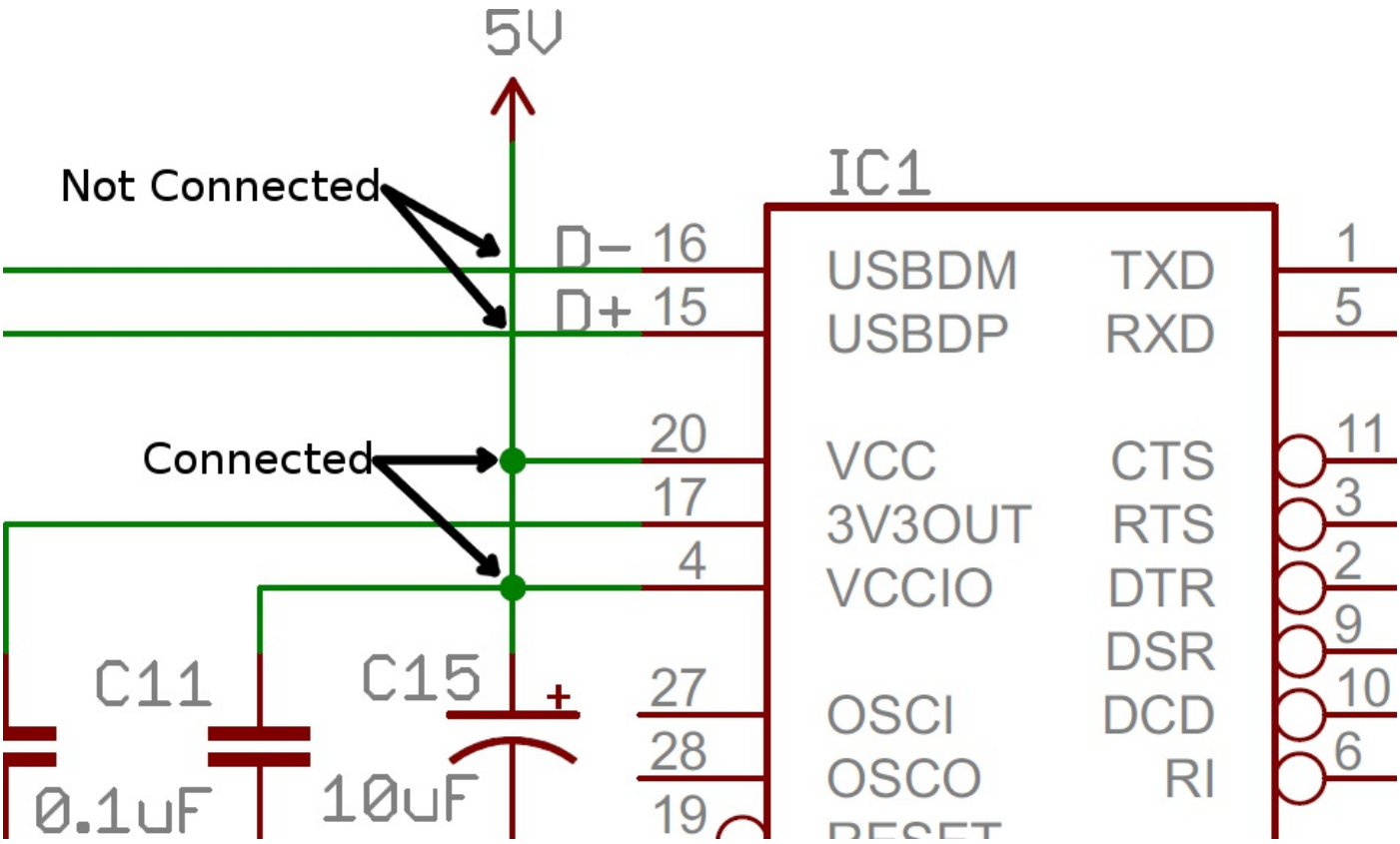


Junctions and Nodes

Wires can connect two terminals together, or they can connect dozens. When a wire splits into two directions, it creates a junction. We represent junctions on schematics with nodes, little dots placed at the intersection of the wires.

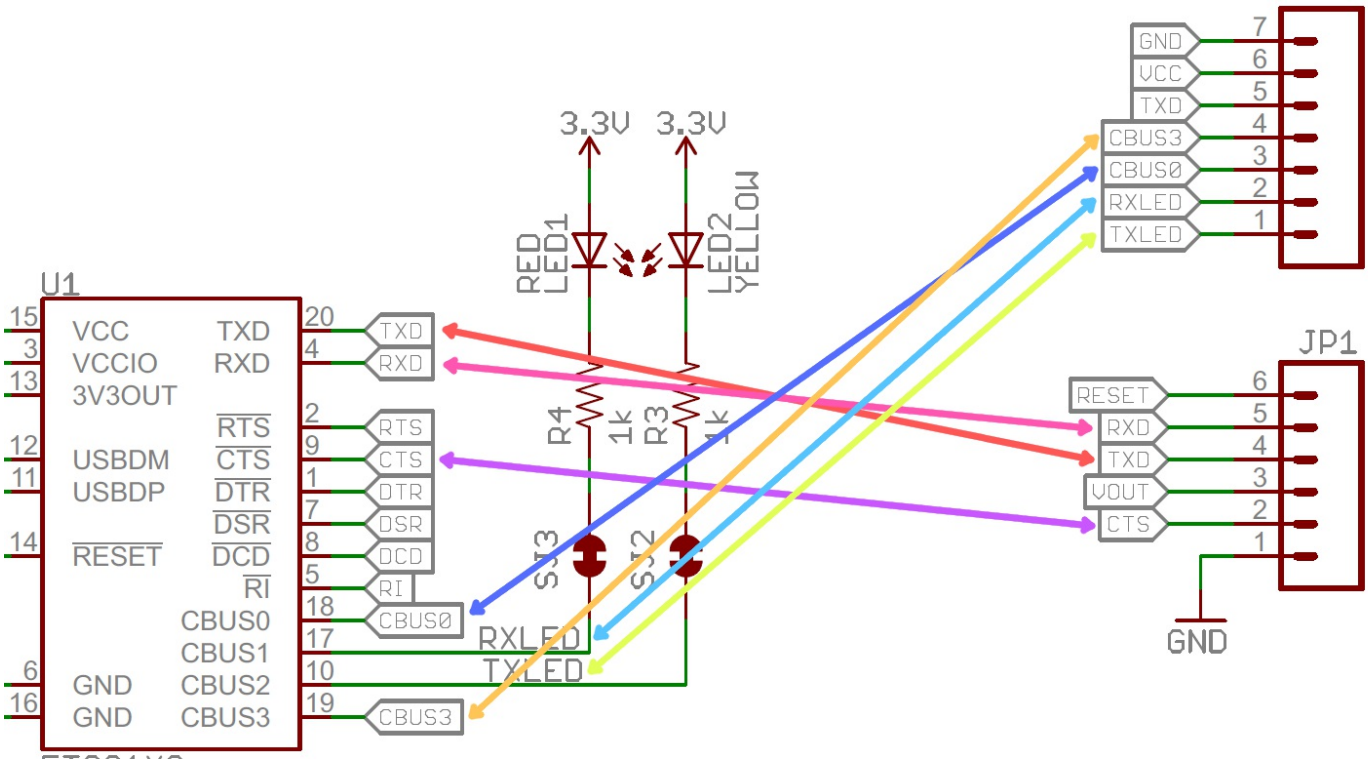


Nodes give us a way to say that “wires crossing this junction are connected”. The absences of a node at a junction means two separate wires are just passing by, not forming any sort of connection. (When designing schematics, it’s usually good practice to avoid these non-connected overlaps wherever possible, but sometimes it’s unavoidable).



Net Names

Sometimes, to make schematics more legible, we'll give a net a name and label it, rather than routing a wire all over the schematic. Nets with the same name are assumed to be connected, even though there isn't a visible wire connecting them. Names can either be written directly on top of the net, or they can be "tags", hanging off the wire.



Each net with the same name is connected, as in this schematic for an FT231X Breakout Board. Names and labels help keep schematics from getting too chaotic (imagine if all those nets were actually connected with wires).

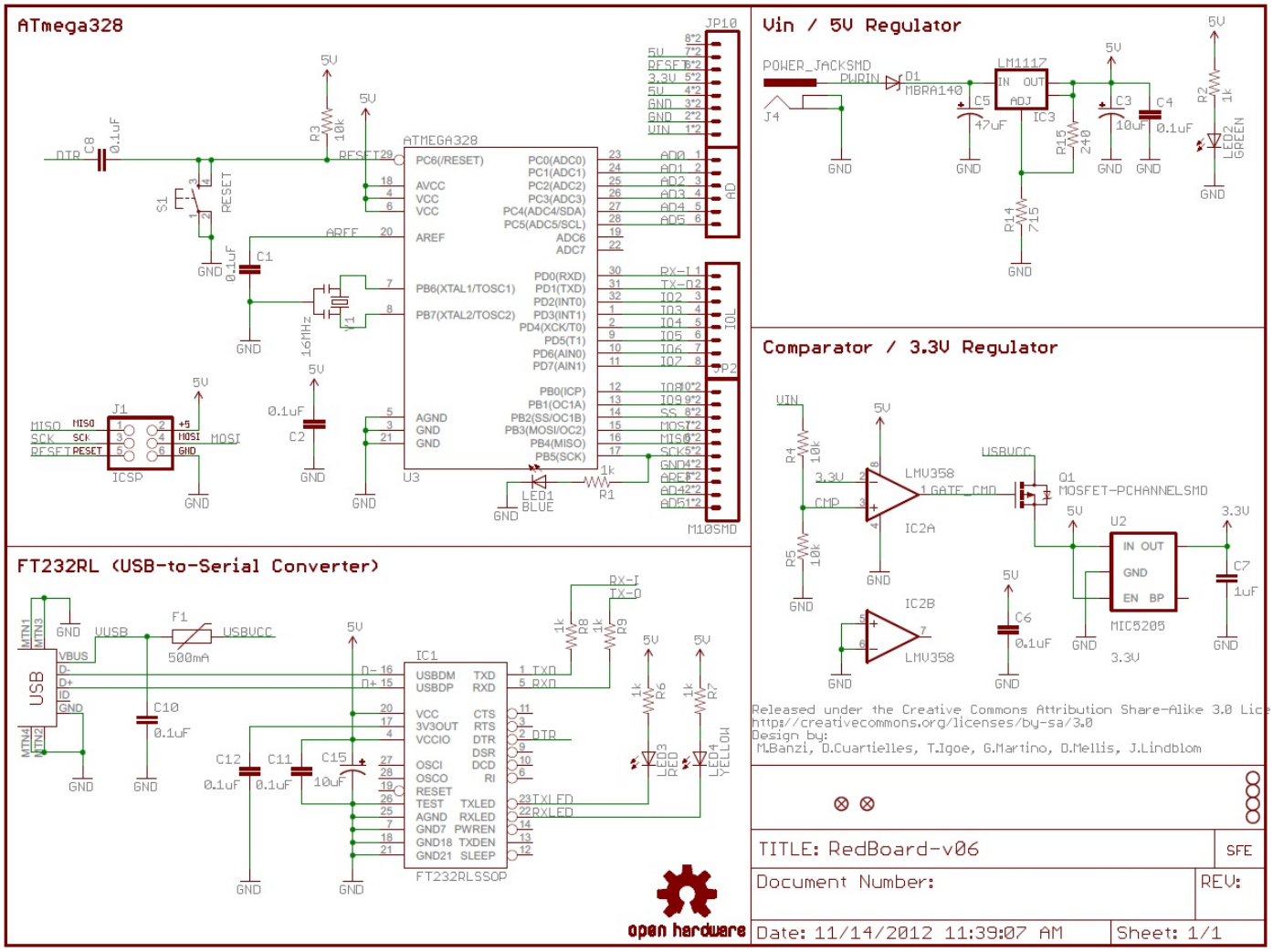
Nets are usually given a name that specifically states the purpose of signals on that wire. For example, power nets might be labeled “VCC” or “5V”, while serial communication nets might be labeled “RX” or “TX”.

Schematic Reading Tips

Identify Blocks

Truly expansive schematics should be split into functional blocks. There might be a section for power input and voltage regulation, or a microcontroller section, or a section devoted to connectors. Try recognizing which sections are which, and following the flow of circuit from input to output. Really good schematic designers might even lay the

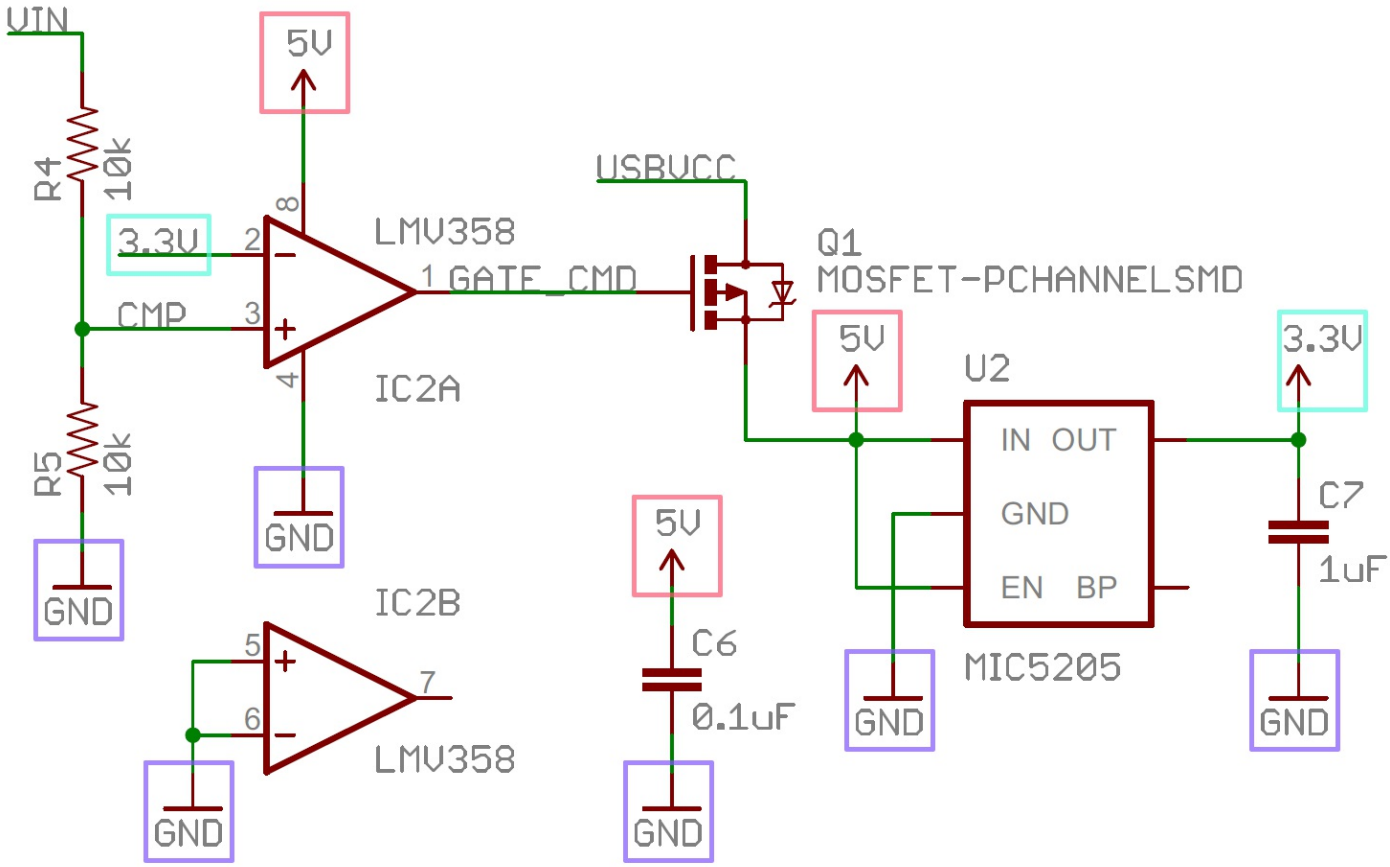
circuit out like a book, inputs on the left side, outputs on the right.



If the drawer of a schematic is really nice (like the engineer who designed this schematic for the RedBoard), they may separate sections of a schematic into logical, labeled blocks.

Recognize Voltage Nodes

Voltage nodes are single-terminal schematic components, which we can connect component terminals to in order to assign them to a specific voltage level. These are a special application of net names, meaning all terminals connected to a like-named voltage node are connected together.



Like-named voltage nodes – like GND, 5V, and 3.3V – are all connected to their counterparts, even if there aren't wires between them.

The ground voltage node is especially useful, because so many components need a connection to ground.

Reference Component Datasheets

If there's something on a schematic that just doesn't make sense, try finding a datasheet for the most important component. Usually the component doing the most work on a circuit is an integrated circuit, like a microcontroller or sensor. These are usually the largest component, oft-located at the center of the schematic.