BARNABAS ROBOTICS

<BUILD. LEARN. TEACH. INSPIRE>

DIVING INTO HARDWARE

The Young Inventor Series

OPEN-SOURCE CURRICULUM

Revision 1.02

BARNABAS ROBOTICS

<BUILD. LEARN. TEACH. INSPIRE>

Date: 11/13/2017
Author: Eric Lin
Contributors: Edward Li

Revision History:

- 1.00: First Revision
- 1.01: Added DC Motor
- 1.02: Added CCD and Nightlight synthesis project

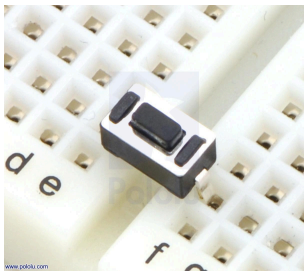
Materials

Item	Qty	Notes	How To Purchase
400 tie breadboard	1		
Male to Male Wires	15		
*Common Cathode Display	1		
470 Ohm Resistor	2		
4.7K Ohm Resistor	2		
10K Resistor	2		
Potentiometer (100K)	1		
LED	2		
4-Pin Button	2		
Arduino Uno	1		
USB Cable (A-B)	1		
2N7000 MOSFET	2	N-Channel MOSFET. TO-92 Package	
9V Battery	1		
9V Battery Holder	1		
*SSFGA6115	2	P-Channel MOSFET. TO-220 Package	
10 mm RGB LED (Common Cathode)	1		
*DIP 14 - NOR	1		

Forward

At this point your students are familiar with the level one Barnabas Bot and it's functionality. This curriculum is meant to expand on that kit by giving several loosely related electrical/programming projects which offer insight into previously unexplored concepts. This curriculum was made with the nominal age of 12 years old in mind and is intended to take place over eight 1:30 hour sessions. Please note that this curriculum can be taught as a stand alone class, however some assumptions have been made about prior experience with electronics and programming. One last thing I want to mention is that I will be showing both an Ardublock method and an Arduino method for all of the projects included in this curriculum. As always, please contact us at Barnabas Robotics with any questions or comments about this curriculum. We consider all of our material to be a work in progress, tweaking things as we receive feedback from people such as yourself.

UNIT 1: The Button



Up to this point the control that your students have had over their robots has left something to be desired. I am specifically referring to the lack of a start or stop button (short of connecting and disconnecting power). This means that as soon as code is uploaded to your robot it begins acting. If only there was some way to communicate with your robot that you want it to start at a specific time.

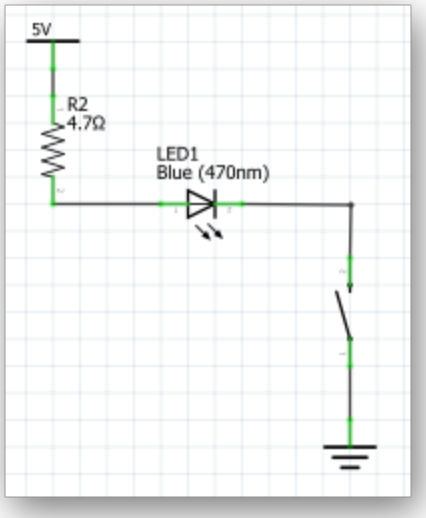
Luckily there is a way to do just that. Using a button we can tell our robot to start only after the button is pressed. Before going into detail about how to accomplish this I first want to explain how the button differs from all the other components of the robot they have used so far.

The LED, buzzer, and servo motors are what are known as outputs of the robot. The robot tells those pieces what to do. The button is fundamentally different as the robot can't send a signal to the button and make it press itself. The robot instead waits for the button to tell it that it has been pressed. The button, and components like it are known as inputs. I like to think that inputs tell the robot something while outputs listen to the robot and act accordingly.

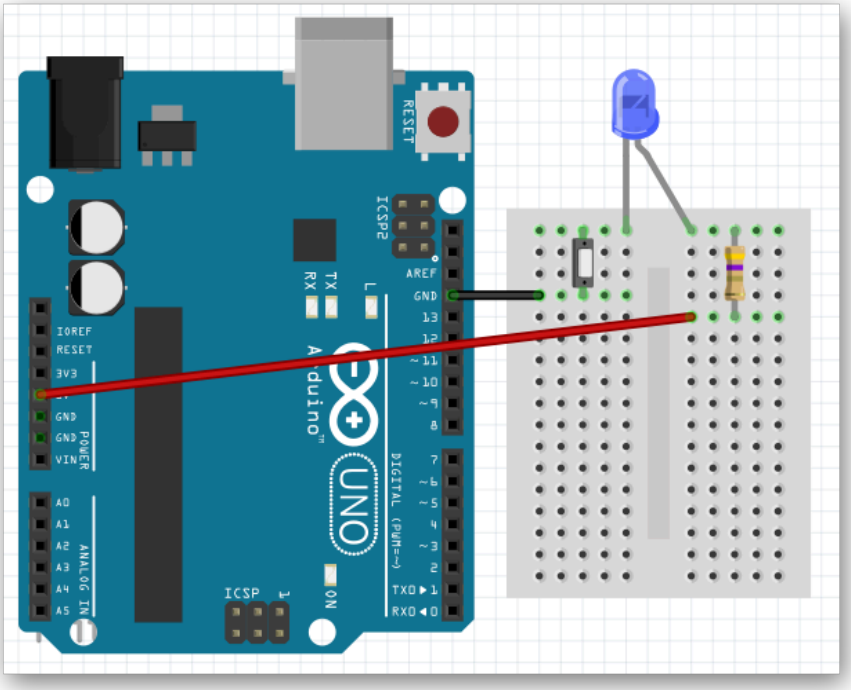
Week 1

Now that we understand the difference between outputs and inputs we are ready to begin wiring our button.

I want to start with a simple circuit to demonstrate how the button works. The button acts as an open in our circuit, no current can flow through it. However when the button is pressed metal to metal contact is made which closes the circuit and allows current to flow through the button.The analogy I like to use is a drawbridge. Traffic can only flow through a drawbridge when it is lowered, much in the same way that current can only flow through the button if it is pressed. The following schematic demonstrates this very well;



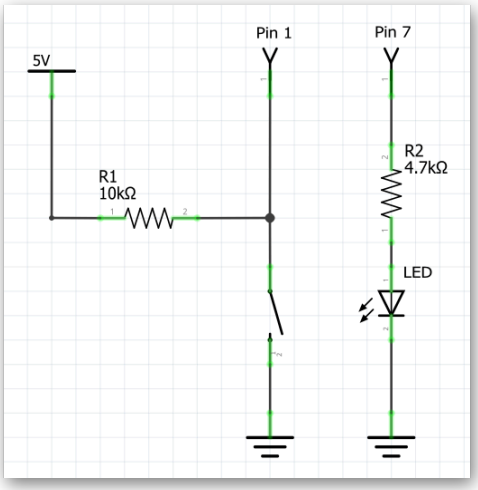
Have your students attempt to wire this schematic without the use of the circuit diagram below, only referencing it if they need help(If you are continuing with the fully built and wired Barnabas Bot don't be concerned that those components are not visible on this schematic);



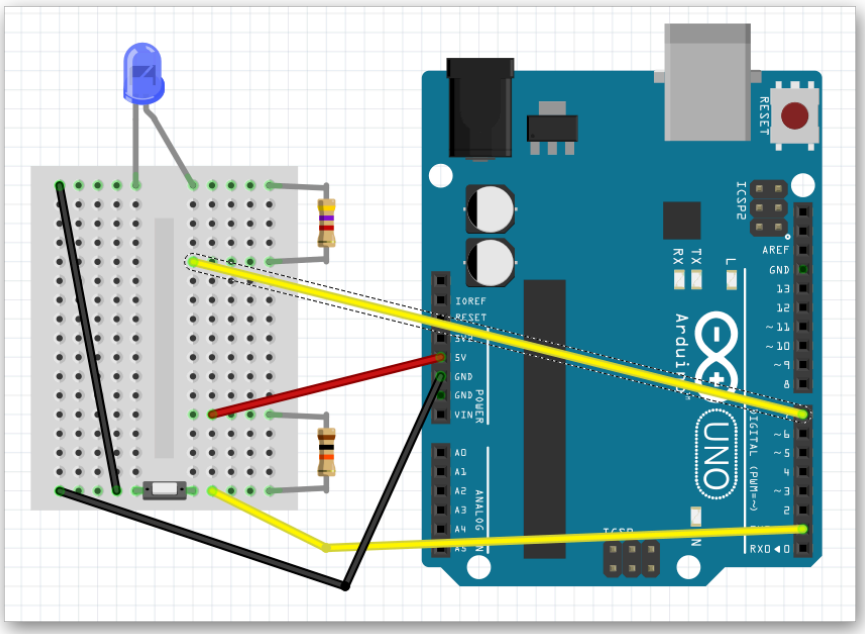
Have your students create this circuit and notice that the LED only turns on when the button is pushed. This is consistent with my explanation earlier that current only flows through

the button when it has been pressed. Unfortunately this circuit has nothing else to teach us so let's move on to the next circuit.
Note: Have the students build this circuit without an explanation of how the button works. Then ask them to explain the behavior of the circuit and see if they can guess how the button functions.

Below is a schematic of the button that instead has the LED wired to a programmable pin (likely the way it was wired before). Have your students attempt to wire this circuit;



If your students are having a hard time wiring the schematic above here is a finished circuit diagram;



In this circuit we have pin 1 connected to 5V through what's known as a pull-up resistor, a resistor that simply leads to 5V. Pin 1 is also connected to ground(GND) through the button, meaning that it is only connected to ground when the button is pressed. Notice that when the button is pressed there is a resistor between 5V and GND preventing us from shorting the circuit. Also, that resistor in effect adds 'distance' between 5V and pin 1. This means that when the button is pressed pin 1 is now 'closer' to GND and now will read GND rather than the 5V it reads otherwise.

Week 2

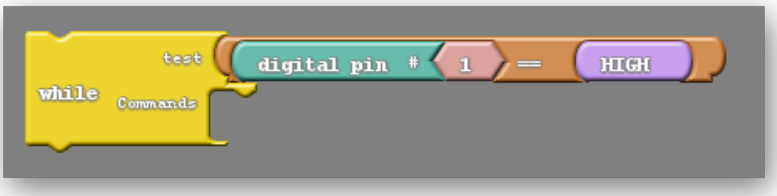
With our circuit completed last week we can move on to programming.

Ardublock:

I would like to reiterate the assumption of prior experience before delving into the coding specifics. If your students are not familiar with Ardublock consider introducing the basics and have them complete a simple task, like lighting an LED, before attempting to code the button.

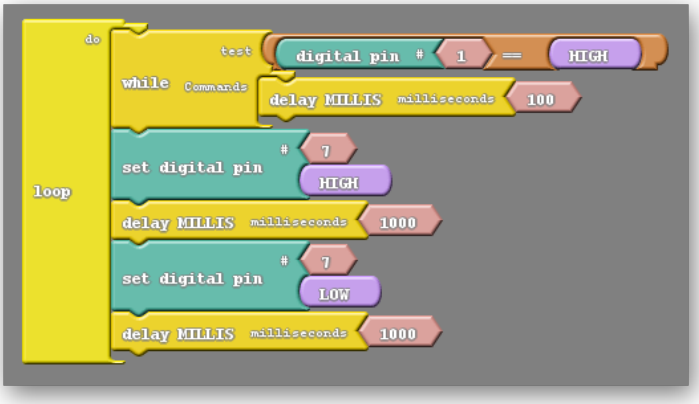
To complete the task of giving the robot a start button we will need to learn how to use the while loop:

The while loop is found in the controls tab of Ardublock. The while loop will run a list of commands over and over as long as a condition is met. In our circuit, because pin 1 is at 5V until the button is pressed we would like our tested condition to look as follows;



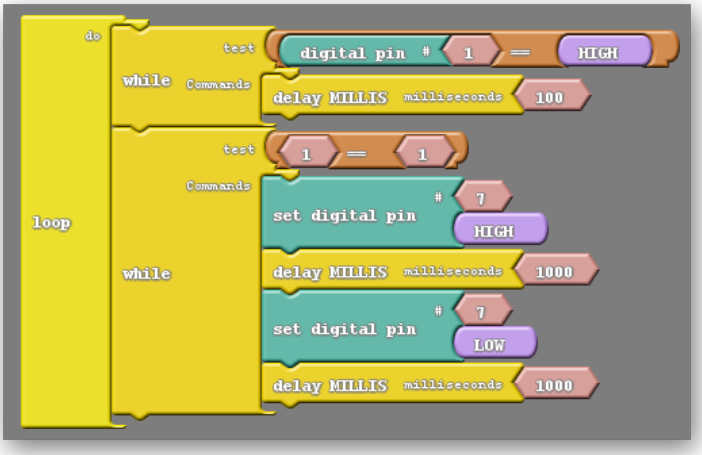
The digital pin, test, and HIGH blocks are found in the pins, tests, and variables/constants tabs respectively. The digital pin block will check the value of the pin specified, for us this is pin 1. Remember that because the button is an input we want to listen to what that pin is telling us. The strange '==' test block is simply asking, 'are these two things equal?'. An important note is that this is the test block that accepts rounded input blocks rather than the sharp edged or puzzle piece test blocks. Lastly, the HIGH block simply refers to 5V. So, what this test ultimately is asking is 'is pin 1 high(5V)?'. As long as the answer is yes the robot will continue to execute the commands inside of the while loop. When the answer becomes no the robot moves on to subsequent lines of code.

So let's construct the full code. At the moment let's just attempt to make our robot wait until the button is pushed to blink the LED.



The above code does just what we were asking for. Notice that the command inside the while loop is simply a delay block. Something must be input into the commands section of the while loop for Ardublock to allow your code to be uploaded and the delay block is a good candidate as it allows us to keep other pins free. It also means that the robot will only check the button every 10th of a second. There is however a small issue with our code. After the button is pressed the LED blinks only once before returning to the beginning of the loop do. This means that we have to press the button once for every time we want to see the LED blink. That seems extremely inefficient. instead I want the robot to blink indefinitely after the button is pressed.

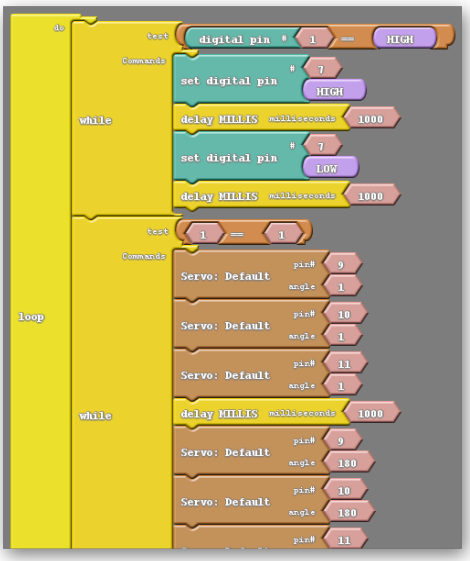
I'm going to accomplish this by making use of the while loop again. Remember that the while loop will continue to run as long as the condition is met. So, if we choose a condition that is always true the while loop will run forever, like so;

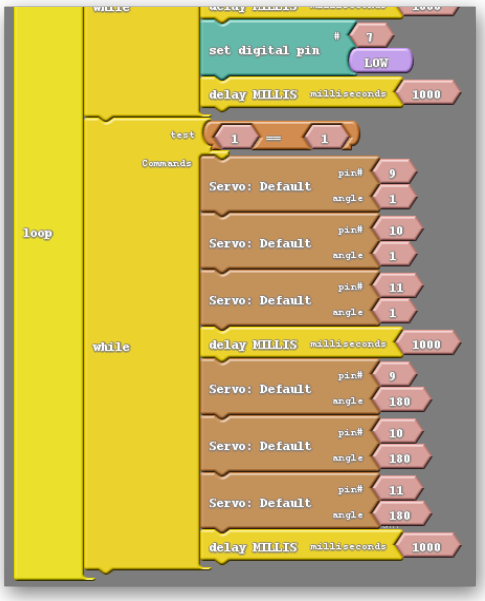


Now after the button is pressed the robot begins to blink the LED, just like before. The difference is now it will continue to do so as long as 1=1, which I assume will always be true.

You can reset the robot by pressing the reset button on the Arduino. This will revert the code back to the beginning of the loop do.

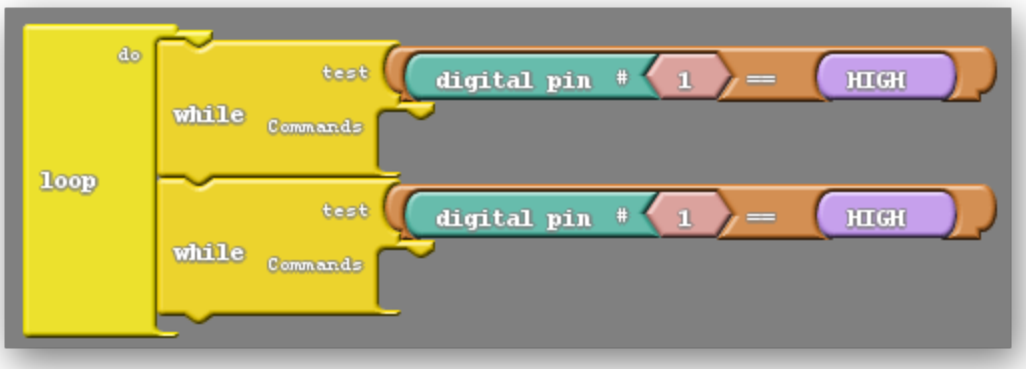
If your students have their Barnabas Bots they can do more complex things as shown below;





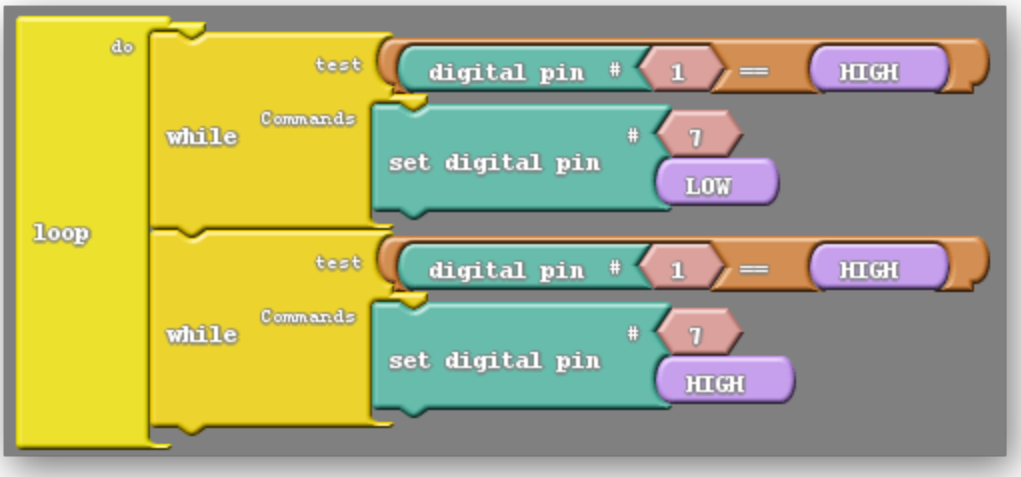
This code will blink before the button is pressed as a kind of standby, then begin dancing after the button is pressed.

The last method of programming the button that I will show you is perhaps the most useful. I can program the button to toggle between two or more states. For example, if I press the button once the LED will turn on, and if I press the button again the LED will turn off. The way I plan to do this is by using multiple while loops with the same condition;



As you can see these while loops are both using the condition of the button being pressed or not. If I press the button once the first while loop will be exited and the second will begin to run. If the button is pressed again the second while loop is exited, the loop repeats, and the first while loop begins to run again.

Now I have to choose the commands for each while loop. For this example I will keep things simple. I merely want to be able to toggle an LED between on and off. Furthermore I want the LED to begin in the off state. I meet those conditions by adding to the code in the way shown below;



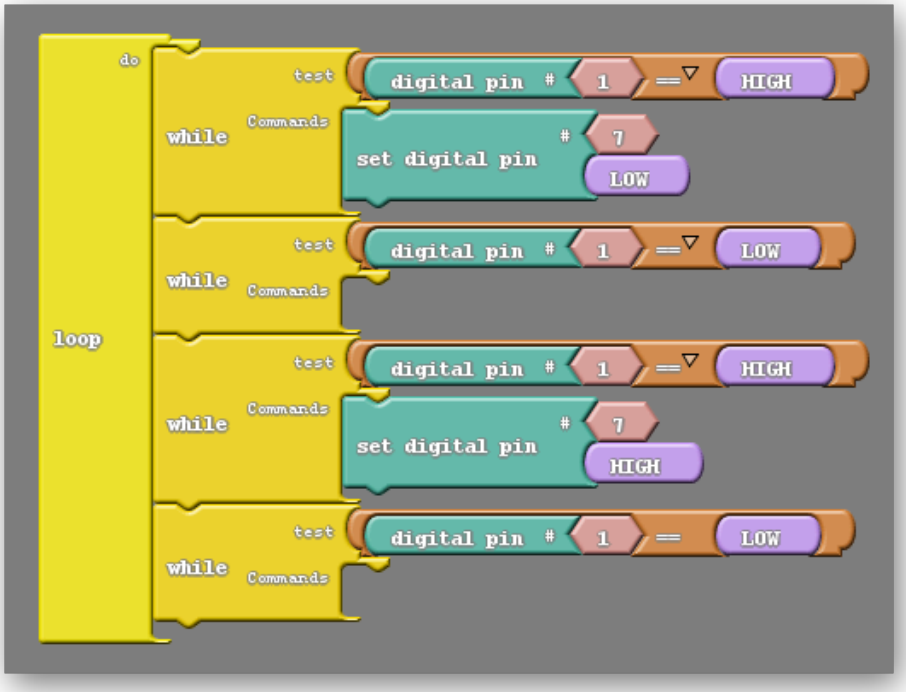
Notice that in the first while loop the pin is set low, meaning the the LED will begin in the off state. There is a small problem with the above code however, have your students upload this code and see if they can spot the unwanted behavior. Bonus points if they guess why it's happening.

So what is happening and why? You may notice as you and your students press the button in an attempt to change the state of the LED that sometimes the button seems unresponsive. Sometimes it works and sometimes it doesn't, giving the impression that the button is of low quality. As it turns out this behavior has nothing to do with the quality of the button. It is a direct result of an oversight in our code. If you think about a human being pressing the button, it doesn't happen instantaneously, does it? Their finger is on the button for some amount of time. During that time the program is constantly switching between the two while loops, as the condition for neither of them is true. When their finger releases the button, who knows which while loop will be running.

The solution for this is fairly simple, but allows for some amount of tinkering. I simply put delay blocks after each while loop as a buffer between the while loops;

The delay time needs to be a reasonable amount of time for a person to press down and release the button. If the time is too short, we'll have the same problem as before. If the time is too long, there will be some latency time between pressing the button and having the state of the LED actually change. Both of these problems, while not a matter of life and death to us, are important to people making products. The people who make your lamps, microwaves, phones, etc. care very much about these problems because they affect how many of your product is sold. These people put hours into figuring out what delay time is appropriate. We won't spend hours on it but I do encourage you to challenge the class to find what they feel is the best delay time.

As it turns out, there is an even better method of addressing this issue. What we can do instead is place additional while loops in place of the delays. Ones with the opposite condition as the ones currently in the code. That way pressing the button will exit the while loop containing the current state we are in, but the new while loop will hold the code in a standby state until the button is let go;



The new while loops have the button set equal to LOW as their condition. Meaning they are exited for the exact opposite reason that the HIGH while loops are exited. This ensures that the state of the LED only changes after a ‘full’ button press (both pressing and letting go), making the previous problem of skipping a state impossible. What’s peculiar about the new while loops is that they don’t contain any commands. This is fine as we are only using these loops as a standby while someone’s finger is still on the button.

Arduino:

As promised, I will provide an alternative method for people interested using the base Arduino IDE, however I will still assume a base level of knowledge when explaining the code. That being said, the level of prior experience I expect is not very high and I will be giving detailed explanations throughout the curriculum.

First of all, when using Arduino I use variables more liberally than in Ardublock. I find that assigning all my pins named variables in Ardublock becomes bulky and clumsy, while assigning variables in Arduino, while typing intensive, helps remind me of the robot’s physical specifications, how the robot is pinned, etc. If you or some of your students choose not to use variables they will need to keep track of the correct pin numbers throughout their code.

Because I will be using variables I will start like so;

```
int LED = 7;
int BUTTON = 1;
```

Simply giving the pins which have the LED and button attached names that reference what they do. It is much easier for me to remember to turn the LED on than to turn pin 7 on, even if it takes longer to type. Also notice the int written before both. Int is just there to tell Arduino what kind of number BUTTON and LED are. They are both integers. This is actually pretty important as many of the functions in Arduino only accept integers, particularly functions that relate to pins.

After this we begin writing inside the void setup;

```
void setup() {
        pinMode(BUTTON,INPUT);
        pinMode(LED,OUTPUT);
```

You can already see the use of naming the pins. Rather than remembering the number of the pins, I just need to remember that the button is an input and the LED is an output. Now there is one more thing I would like to place in the void setup;

```
void setup() {
        pinMode(BUTTON,INPUT);
        pinMode(LED,OUTPUT);
        while (digitalRead(BUTTON)==1){
                //Null Statement
        }
}
```

Wait! There's a while loop inside the void setup! YOU CAN'T DO THAT!!

Yes, yes I can. There are no laws of the universe being broken I promise. As a matter of fact I could place the pinMode() functions inside of the void loop if I felt like it. The only real difference is that the void setup only runs once. It is for this reason that I place the while loop inside of the void setup. The while loop is holding the code at that point, not allowing following lines to be run, until the button is pressed. If this while loop was placed inside the void loop, the robot would stop after the void loop was finished and wait for the button to be pressed again. I want to only have to press the button once to start my robot, and the reset button once to stop the robot.

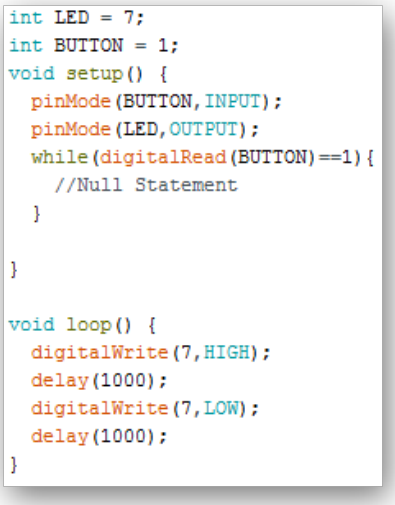
Question is, what is that while loop actually saying? The digitalRead(BUTTON) statement gives the boolean value of the button pin. Either high(1) or low(0). The ==1 asks if that value is 1 and, if it is, run the commands in the while loop. You'll notice that I didn't actually

put any commands in the while loop. I instead made the comment //Null Statement, to remind myself and to let anyone else that views the code that the while loop is empty on purpose.

The void loop is much simpler. I'm just blinking an LED;

```
void loop() {
digitalWrite(LED,HIGH);
delay(1000);
digitalWrite(LED,LOW);
delay(1000);
}
```

Here is the code written in the Arduino IDE;



```
int LED = 7;
int BUTTON = 1;
void setup() {
  pinMode(BUTTON,INPUT);
  pinMode(LED,OUTPUT);
  while(digitalRead(BUTTON)==1){
    //Null Statement
  }

}

void loop() {
  digitalWrite(7,HIGH);
  delay(1000);
  digitalWrite(7,LOW);
  delay(1000);
}
```

If your students are using their fully built and wired Barnabas bot they can consider working movement into this code using the robot's servo motors. Doing this in Arduino is fairly complicated though. This is because unlike Ardublock, Arduino does not tell our robot how to use it's servo motors. This is a task that we must do ourselves.

We must start by importing a functions library into Arduino;

```
#include <Servo.h>
```

Doing this allows us to appropriately call some of the objects we are trying to control servo motors, and Arduino now knows what we mean if we say that something is a Servo motor. Just to give you an idea of what is included in the Servo.h library I have included the source code. This should give you an idea of exactly why I want to include Servo.h. It lets me avoid having to

recreate that source code whenever I want to control a servo motor. Instead I am essentially telling Arduino; “hey, here is your homework. I need you to understand how to operate a servo motor.” This way I can just say servo whenever I want to treat an object as a servo.

Additionally I want to include my variable names for the LED and button like before;

```
int BUTTON = 1;
int LED = 7;
```

Also I want to create names for my motors;

```
 Servo leftArm;
Servo rightArm;
Servo head;
```

At this point Arduino knows nothing about these servo motors other than that they exist. At this point the motors are purely virtual. We will give the specifics of the motors in the void setup;

```
void setup() {
	pinMode(BUTTON,INPUT);
	pinMode(LED,OUTPUT);
	leftArm.attach(9);
	rightArm.attach(10);
	head.attach(11);
	while(digitalRead(BUTTON)==1){
		digitalWrite(LED,HIGH);
		delay(1000);
		digitalWrite(LED,LOW);
		delay(1000);
	}
}
```

Some things remain the same in the void setup. We are still assigning pin modes to the LED and button, and the while loop is still inside of the void setup. This time however I am including the LED blink inside the while loop. So now our robot blinks before the button is pressed to let us know it’s ready. In addition we have included some attach statements referencing our recently named virtual servo motors. Let’s take a closer look at one of those statements;

```
leftArm.attach(9);
```

First we can see that whatever this line of code is doing, it is referencing the leftArm Servo motor. Immediately after that is .attach. This is a good opportunity to talk more about coding conventions at large. Whenever you see thing.otherthing in code, the otherthing is performing some sort of action on the first thing. You can think of some real life analogies to this to help the students understand. For example my morning may look something like this if it took place in Arduino;

```
eric.wakeUp;
eric.getDressed;
eric.eatBreakfast;
eric.goToWork;
eric.contemplateExistance
eric.fendOffExistentialDread
eric.weep
```

You'll also notice that the (9) at the end of the .attach statement. That gives Arduino the information that the leftArm servo motor is attached to pin 9. Keep in mind that this is just how my robot is wired, if your left arm is attached to a different pin make sure that is reflected in your code.

The void loop is fairly simple as we just need to move some motors;

```
void loop() {
	leftArm.writeMicroseconds(1000);
	rightArm.writeMicroseconds(1000);
	head.writeMicroseconds(1000);
	delay(1000);
	leftArm.writeMicroseconds(2000);
	rightArm.writeMicroseconds(2000);
	head.writeMicroseconds(2000);
	delay(1000);
}
```

Notice that we are performing another action on each of our motors .writeMicroseconds. That action sends a signal of a certain frequency to that particular motor and the motor perceives that signal as a command to move to a certain position. 1000 and 2000 are at the extremes of the servo motors' range of motion with 1500 lying in the middle.

Once everything is put together the end result should look like the following;

```
#include <Servo.h>
int BUTTON = 1;
int LED = 7;
Servo leftArm;
Servo rightArm;
Servo head;
void setup() {
  pinMode(BUTTON,INPUT);
  pinMode(LED,OUTPUT);
  leftArm.attach(9);
  rightArm.attach(10);
  head.attach(11);
  while(digitalRead(BUTTON)==1){
    digitalWrite(LED,HIGH);
    delay(1000);
    digitalWrite(LED,LOW);
    delay(1000);
  }
```

```
}

void loop() {
  leftArm.writeMicroseconds(1000);
  rightArm.writeMicroseconds(1000);
  head.writeMicroseconds(1000);
  delay(1000)
  leftArm.writeMicroseconds(2000);
  rightArm.writeMicroseconds(2000);
  head.writeMicroseconds(2000);
  delay(1000);
}
```

THe final method of programming the button, the toggling method, is fairly straightforward. I can use this method to toggle between two or more states. For example, if I press the button once the LED will turn on, and if I press the button again the LED will turn off. I will start in the same place I always do, by creating the needed variables and setting up the pins;

```
LED=7;
BUTTON=1;

void setup(){
pinMode(LED,OUTPUT);
pinMode(BUTTON,INPUT);
}
```

Next I want to populate the loop do with two while loops, each with the same condition;

```
void loop(){
while(digitalRead(BUTTON)==HIGH){
  }
while(digitalRead(BUTTON)==HIGH){
  }
}
```

As you can see these while loops are both using the condition of the button being pressed or not. If I press the button once the first while loop will be exited and the second will begin to run. If the button is pressed again the second while loop is exited, the loop repeats, and the first while loop begins to run again.

Now I have to choose the commands for each while loop. For this example I will keep things simple. I merely want to be able to toggle an LED between on and off. Furthermore I want the LED to begin in the off state. I meet those conditions by adding to the code in the way shown below;

```
void loop(){
while(digitalRead(BUTTON)==HIGH){
  digitalWrite(LED,LOW);
  }
while(digitalRead(BUTTON)==HIGH){
  digitalWrite(LED,HIGH);
  }
}
```

Notice that in the first while loop the pin is set low, meaning the the LED will begin in the off state. There is a small problem with the above code however, have your students upload this code and see if they can spot the unwanted behavior. Bonus points if they guess why it's happening.

So what is happening and why? You may notice as you and your students press the button in an attempt to change the state of the LED that sometimes the button seems unresponsive. Sometimes it works and sometimes it doesn't, giving the impression that the button is of low quality. As it turns out this behavior has nothing to do with the quality of the button. It is a direct result of an oversight in our code. If you think about a human being pressing the button, it doesn't happen instantaneously, does it? Their finger is on the button for some amount of time. During that time the program is constantly switching between the two while loops, as the condition for neither of them is true. When their finger releases the button, who knows which while loop will be running.

The solution for this is fairly simple, but allows for some amount of tinkering. I simply put delay blocks after each while loop as a buffer between the while loops;

```
void loop(){
while(digitalRead(BUTTON)==HIGH){
  digitalWrite(LED,LOW);
  }
delay(500);
while(digitalRead(BUTTON)==HIGH){
  digitalWrite(LED,HIGH);
  }
delay(500);
}
```

The delay time needs to be a reasonable amount of time for a person to press down and release the button. If the time is too short, we'll have the same problem as before. If the time is too long, there will be some latency time between pressing the button and having the state of the LED actually change. Both of these problems, while not a matter of life and death to us, are important to people making products. The people who make your lamps, microwaves, phones, etc. care very much about these problems because they affect how many of your product is sold. These people put hours into figuring out what delay time is appropriate. We won't spend hours on it but I do encourage you to challenge the class to find what they feel is the best delay time.

As it turns out, there is an even better method of addressing this issue. What we can do instead is place additional while loops in place of the delays. Ones with the opposite condition as the ones currently in the code. That way pressing the button will exit the while loop containing the current state we are in, but the new while loop will hold the code in a standby state until the button is let go;

```
void loop(){
while(digitalRead(BUTTON)==HIGH){
  digitalWrite(LED,LOW);
  }
while(digitalRead(BUTTON)==LOW){
// Null Statement
  }
while(digitalRead(BUTTON)==HIGH){
  digitalWrite(LED,HIGH);
  }
```

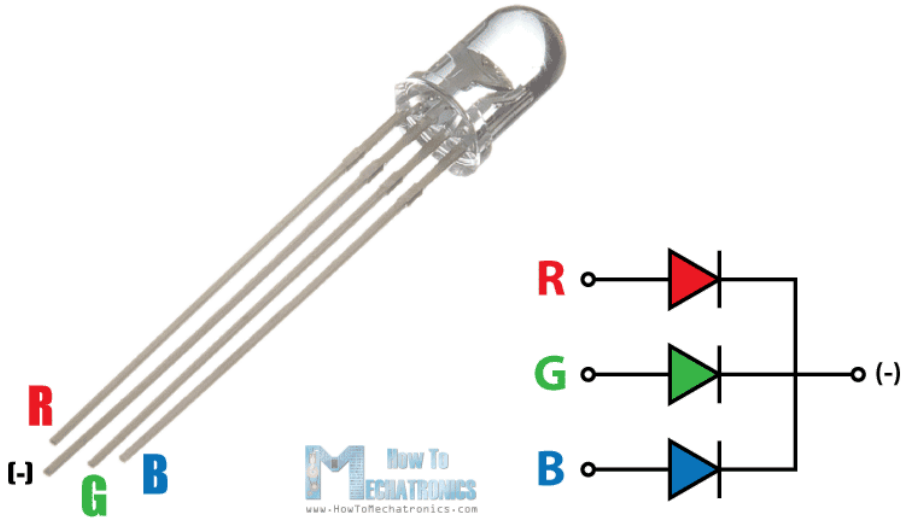
```
while(digitalRead(BUTTON)==LOW){
// Null Statement
  }
}
```

The new while loops have the button set equal to LOW as their condition. Meaning they are exited for the exact opposite reason that the HIGH while loops are exited. This ensures that the state of the LED only changes after a ‘full’ button press (both pressing and letting go), making the previous problem of skipping a state impossible. What’s peculiar about the new while loops is that they don’t contain any commands. This is fine as we are only using these loops as a standby while someone’s finger is still on the button.

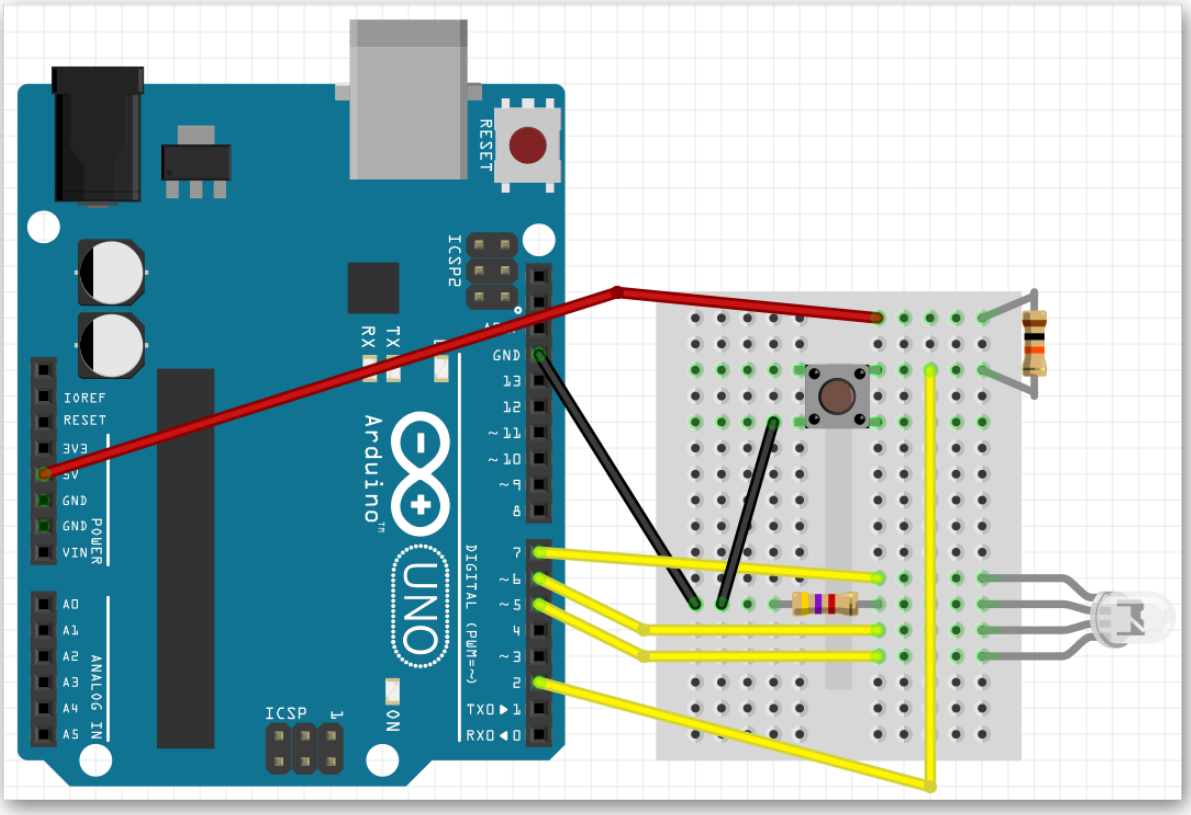
This ends the button lesson.

The RGB LED

Another tool to be used in tandem with the button, and just in general, is the RGB (red-green-blue) LED. This LED acts like three separate LEDs but with a common cathode. In other words, three LEDs that share their short (-) leg.



THe RGB LED behaves just like three individual LEDs, with only a slight change in how it is wired;



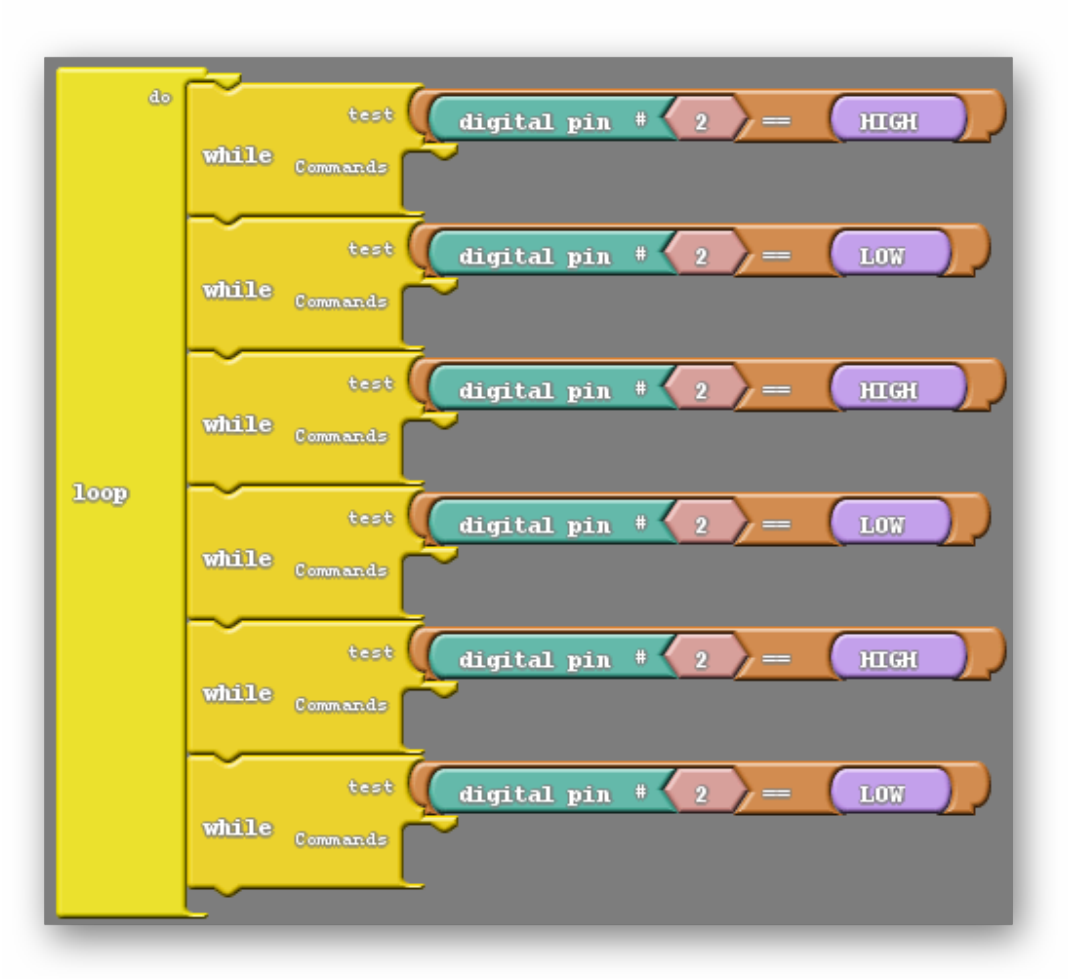
I now have access to all three colors on the LED through the pins 5, 6, and 7.

The RGB LED provided in your diving into hardware kit is capable of turning on all three colors at once by setting pins 5, 6, and 7 HIGH, but can turn on the colors in any combination. This piece of hardware can provide some flare to the button curriculum, allowing you to toggle color patterns rather than just toggling the single colored LED on and off.

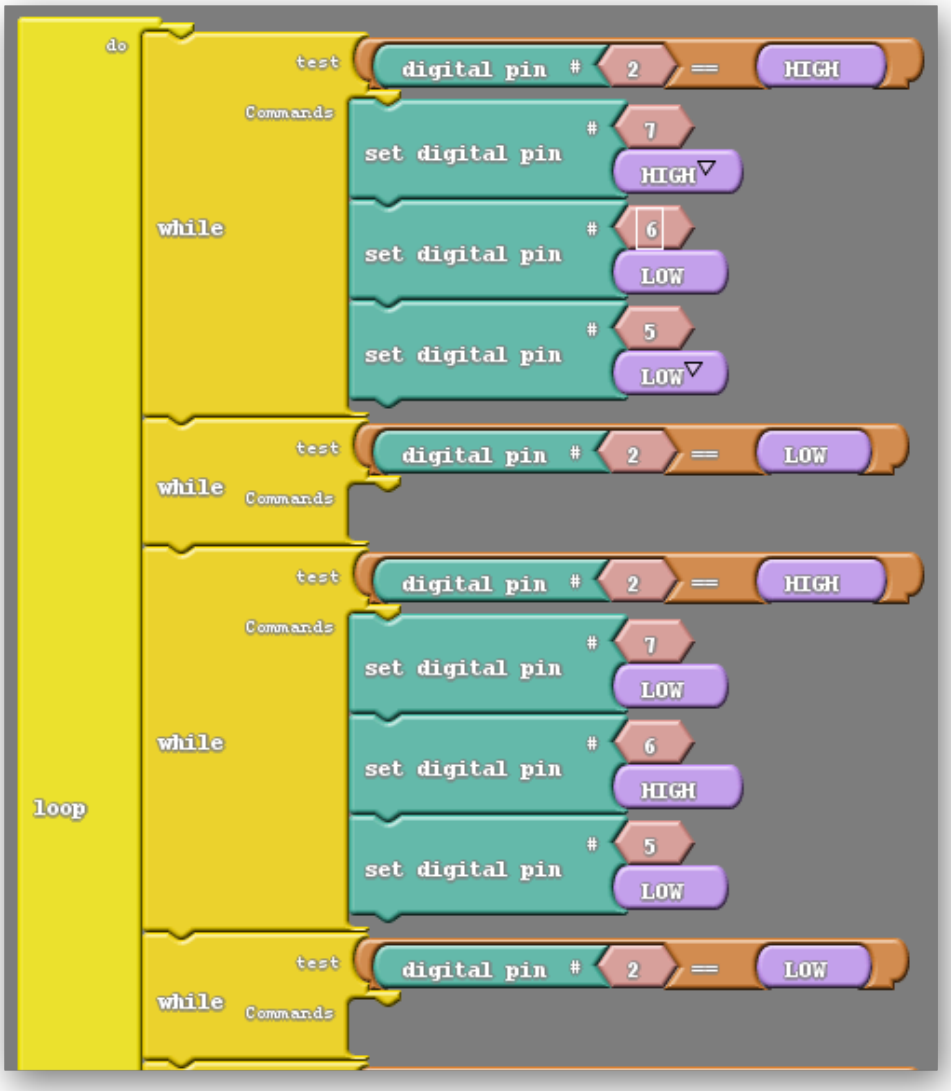
Let's try and utilize this new LED with in the last week 2 exercise.

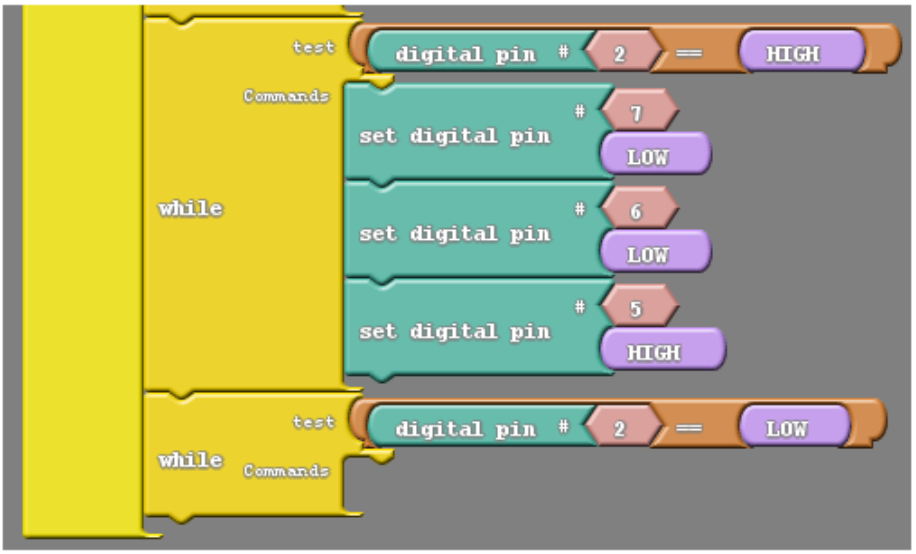
Ardublock:

Let's begin at this point;



Remember that by organizing our code into successive while loops with alternating conditions, like we have here, we get rid of the latency problem completely. Pressing the button will always have exactly the effect that we want as soon as it is pressed. Also remember that each of the while loops with LOW conditions will remain empty, they are only there to stop the while loop after from being skipped. Next is to just set the pins HIGH and LOW at the right time;


do
loop
while
test
digital pin # 2 == HIGH
Commands
set digital pin # 7 HIGH
set digital pin # 6 LOW
set digital pin # 5 LOW
while
test
digital pin # 2 == LOW
while
Commands
while
test
digital pin # 2 == HIGH
Commands
set digital pin # 7 LOW
set digital pin # 6 HIGH
set digital pin # 5 LOW
while
test
digital pin # 2 == LOW
while
Commands



The code above alternates between the first, second, and third LED being on. So this code will alternate between the blue, green, and red LED being on.

Arduino:

Here's where we will begin;

```
RED=7
GREEN=6
BLUE=5
BUTTON=2

void setup(){
  pinMode(RED,OUTPUT);
  pinMode(GREEN,OUTPUT);
  pinMode(BLUE,OUTPUT);
  pinMode(BUTTON,INPUT);
}
```

I have started by labeling each of the pins used with variables to help me keep track of things. Afterwards I assigned the pinmode of each of those pins. Each of the LED pins need a signal sent to them to function, so they have been labeled as OUTPUT. The button, for reasons that have been described before, has been labeled as an INPUT.

Next we create the structure of successive while loops that will let us toggle between states with our button;

```
void loop(){
  while (digitalRead(BUTTON)==HIGH){
  }
  while (digitalRead(BUTTON)==LOW){
  }
  while (digitalRead(BUTTON)==HIGH){
  }
  while (digitalRead(BUTTON)==LOW){
  }
  while (digitalRead(BUTTON)==HIGH){
  }
  while (digitalRead(BUTTON)==LOW){
  }
```

Notice that the condition of each while loop is the opposite of the last. Remember that by doing this we eliminate latency in our button presses as well as stop our code from inadvertently jumping ahead.

Finally we just need to populate the while loops with the correct digitalwrite statements, remembering that any while loop with a LOW condition will be left empty;

```
void loop(){
  while (digitalRead(BUTTON)==HIGH){
    digitalWrite(RED,HIGH);
    digitalWrite(GREEN,LOW);
    digitalWrite(BLUE,LOW);
  }
  while (digitalRead(BUTTON)==LOW){
    //Null Statement
  }
  while (digitalRead(BUTTON)==HIGH){
    digitalWrite(RED,LOW);
    digitalWrite(GREEN,HIGH);
    digitalWrite(BLUE,LOW);
  }
  while (digitalRead(BUTTON)==LOW){
    //Null Statement
  }
```

```
while (digitalRead(BUTTON)==HIGH){
  digitalWrite(RED,LOW);
  digitalWrite(GREEN,LOW);
  digitalWrite(BLUE,HIGH);
}
while (digitalRead(BUTTON)==LOW){
  //Null Statement
}
```

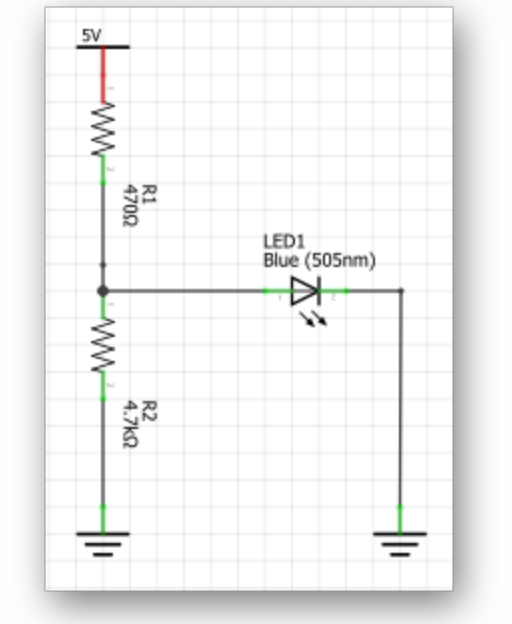
Looking through the code, you can clearly see that it toggles between RED, GREEN, and BLUE being HIGH. you should be able to verify for yourself that this is in fact true.

UNIT 2: Voltage Dividers & Their Applications

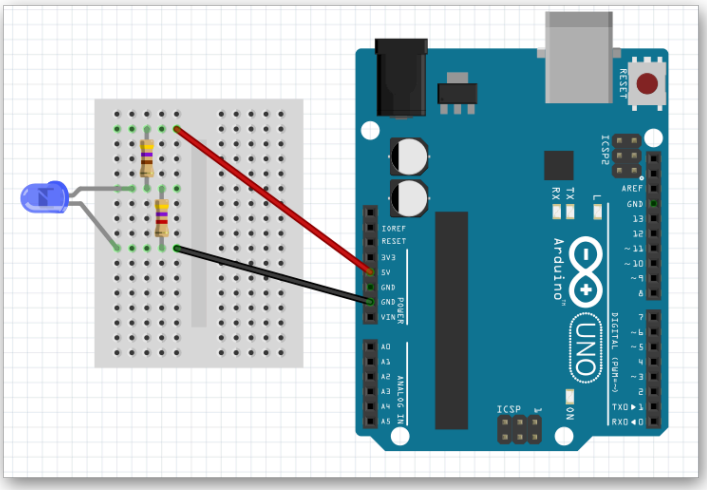
Week 3

The Voltage Divider

To begin this unit I want to introduce an incredibly common circuit called a voltage divider;



Start by drawing this diagram and having the students attempt to breadboard it themselves. The finished circuit is shown below for clarity;



The voltage divider does exactly what it sounds like, it divides the voltage in a circuit. By making the current travel across two resistors in series a certain amount of the total voltage is dropped across each resistor, so that after travelling through the first resistor the voltage in the circuit is no longer 5V. What is the voltage at that point in the circuit then? That depends on the ratio of the values of the two resistors. For example, if the first resistor in the circuit is ⅓ of the total sum of the resistances in the circuit then only ⅓ of the voltage will be dropped across that resistor and we would have ⅔ of the original 5V remaining. For the mathematically minded;

$$R_1/(R_1+R_2)x5V=\text{voltage through LED}$$

Because we have a 470 Ohm and 4.7k Ohm resistor we are only losing 1/11 of the voltage before reaching the LED, leaving us with about 4.5V.

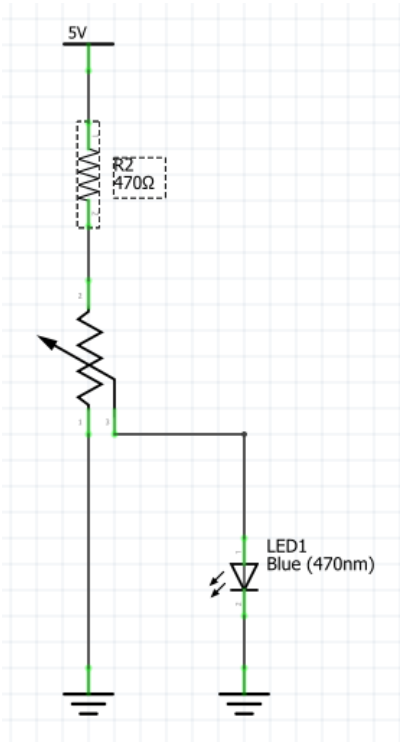
Now switch the 470 Ohm resistor with another 4.7k Ohm resistor and note the change in the LED's intensity. It may not even turn on! By giving both sides of the circuit equal amounts of resistance we have split the voltage in half, allowing only 2.5V to go through the LED.

This shouldn't come as a surprise to many of the students. After all, we did a similar thing in the level one curriculum to show that resistance affects the intensity of the LED. The interesting thing about the voltage divider is that we can change the resistor in the bottom half of the circuit to affect light intensity as well. Try replacing the grounded resistor with a 100K Ohm resistor and notice the difference in light intensity. By changing the resistance in the alternate path of the circuit we have somehow forced more current through the LED.

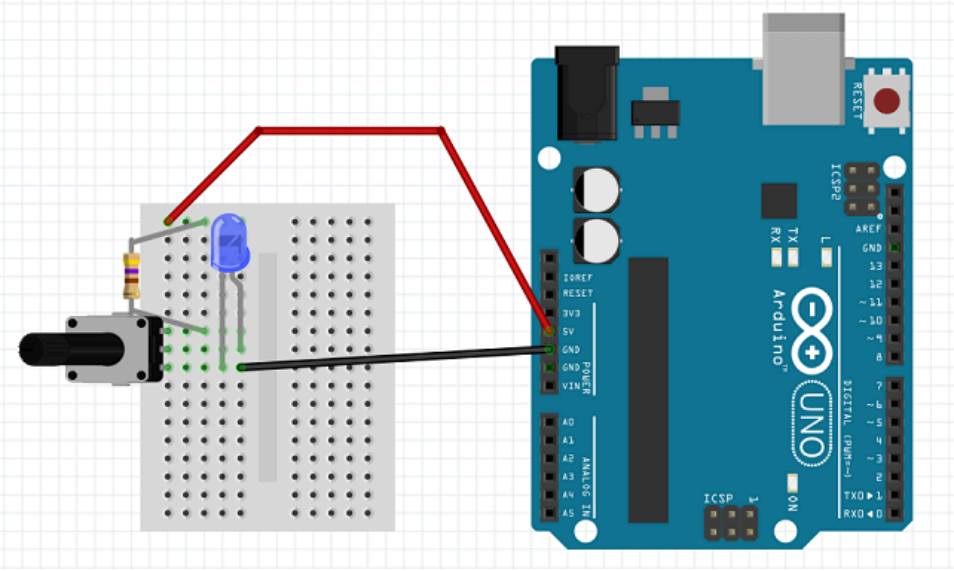
The change in intensity is easy to explain using the math of the voltage divider but I would like to approach this phenomenon more conceptually. Consider your circuit to be a river, our particular river splits into two branches. On one of these paths we have created a very large dam, so large that it increases the flow of water (electricity) through the other branch.

The Potentiometer

Now that we have explored the voltage divider we can tackle the potentiometer. Have your students attempt to create the following circuit;



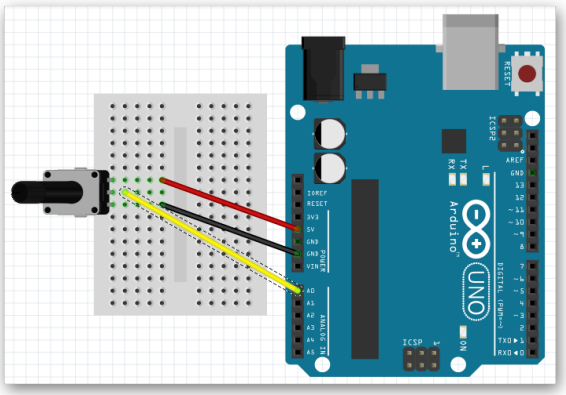
As always here is a completed circuit diagram for clarity;



So what exactly does the potentiometer do? I mentioned earlier that it was a king of variable resistor. That simply means that the value of its resistance can change. The potentiometer is a single resistor that is split somewhere into two resistors. As a matter of fact, you get to choose where the resistor is split by twisting the dial on the potentiometer. So we can split the circuit into a top and bottom just like we did with the voltage divider with one resistor for each half. On top of that, this voltage divider is much easier to manipulate if we want to see different behavior in our circuit.

Have your students play with the dial of the potentiometer and note the change in light intensity from the LED. Then ask them where they think the resistor is split for each kind of behavior (ex. When the light is bright the resistor is split so that almost all of it is at the bottom of the circuit). I have been using the words top and bottom to describe parts of our circuits for a little while now. All I mean by top is nearer to 5V and all I mean by bottom is nearer to GND.

Let's move on to a circuit that allows us to incorporate programming. This circuit is nearly identical to the one we just made;



The only difference is that we are diverting the second branch of the circuit into a programmable pin rather than through an LED. Using this circuit we can have our robot tell us exactly what the voltage is in the branch of the circuit that previously contained the LED. Let's get to programming!

Ardublock:

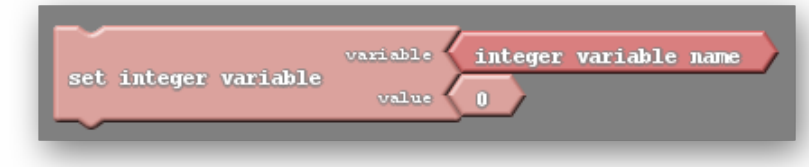
I want to start by discussing the goal of the code. Ardublock has a feature called the serial monitor which you can print lines of text to. I want the information printed on the serial monitor to be the voltage after going through the top of the circuit, through whatever amount of the potentiometer that corresponds to.

First I want to print **something** that corresponds to that voltage to the serial monitor, just as a proof of concept. It will help us to look back at the circuit that we just wired together. Notice that the programmable pin we are using is labelled A0. This is one of the analog input pins on the Arduino. The analog pins are capable of reading voltage values with a high degree of accuracy, giving a result between 0 and 1023, which correspond to GND and 5V, and any number in between corresponds to a voltage between 0V and 5V.

Ardublock:

I remember telling you that I generally don't like using variables when I use Ardublock. This is one of the exceptions. Without variables this section becomes much harder to understand. So in order to program this section I need to introduce you to some new block.

First I will introduce the block that lets us create and name variables in Ardublock;



This is the set integer variable block, which is found in the Variables/Constants tab. This block will create some variable, which you can name, and gives it some value. In the above case I did not bother to change the default name of the variable so my variable is called 'integer variable name', an absolute classic. I'm also giving it the value of zero. This means that if I were to put the variable 'integer variable name' anywhere after this block in my code the computer, and our

robot, will recognize it as zero. Not a particularly useful variable, but we'll need to know this for later, I promise.

You're probably wondering what integer variable means. It simply means that the variable can only be a certain kind of number, an integer. An integer is a number that can be expressed without decimals or fractions, like 1, 4, -10, 1000. Be careful with extremely large numbers though, those are considered long form numbers and the computer and robot treat them differently.

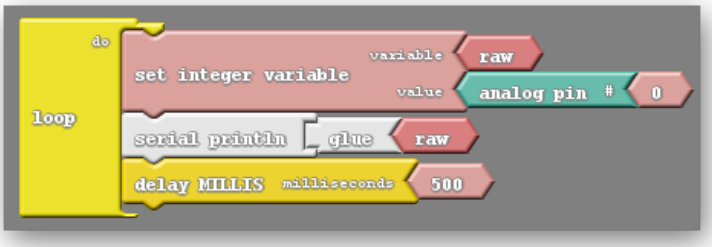
There is another block that we need to learn about before we get underway. It is the block that will allow us to print information onto the serial monitor. This block is the serial println block found in the communications tab;

What this block does is print a message to the serial monitor like I mentioned earlier. In this case, because I did not change the message from its default it will simply print the word 'message'.

Lastly, there is an additional block from the communications tab that we need. It is the glue block;

Note that this is the glue block with pointed corners rather than the rounded glue block. The glue block lets us glue a value to our message, such as a variable, and print it along with our message. I'm sure that you see where this is going.

Below is the full code that we will be using;



Notice that I have my variable's value set to analog pin 0, the pin we wired to our potentiometer. Now rather than having a variable that is one single value, we have a variable with a changing value. It changes as the voltage to pin A0 changes, which happens as we turn the dial on our potentiometer.
I have changed the name of my variable to 'raw' as the date we are going to receive is completely raw, we have not doctored or analyzed it at all.
In the serial println block I've just thrown the message away and simply glued the 'raw' variable value to that block, so the only thing being printed is the value of raw.
Lastly I put a half second delay at the end of the code to avoid printing raw values too rapidly.

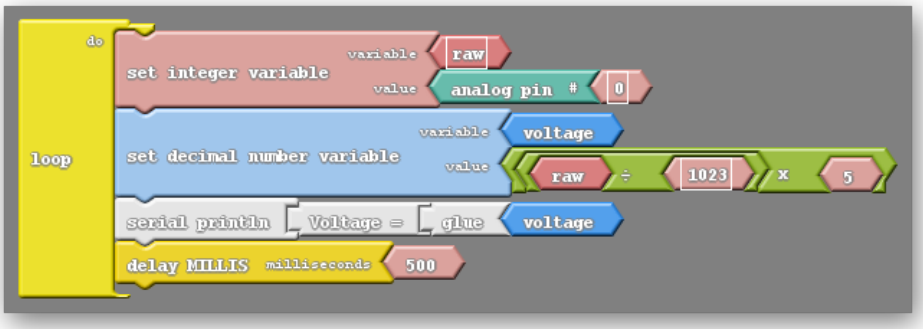
Upload this code to your robot and open the serial monitor, what do you see? As you turn the dial on the potentiometer what do you see? You should see a number between 0 and 1023 that fluctuates as the dial is turned.

That number, as I explained before, represents some voltage between 0V (GND) and 5V. I want to be able to read the exact voltage being supplied from the serial monitor directly, not have to do math to figure out what each number I see means. Luckily your computer and robot are incredibly good at math, well sort of. They can only do math that you teach them, but they can do it incredibly fast. So we need to understand the math first so we can teach our robot.

We know that the voltage we are getting is being split into 1023 equal parts, meaning that if the serial monitor reads 1 we are getting 1/1023 of the 5V. If it reads 1023 we are getting all of the 5V. So the voltage for any number the serial monitor can give us is;

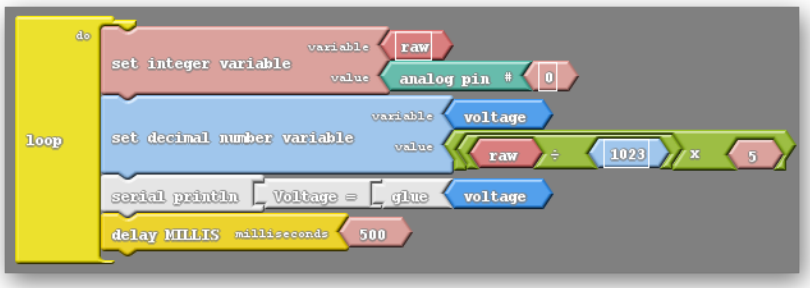
$$(raw)/1023 \times 5V = voltage$$

We just need to teach our robot this formula. To do this I need to teach you more of the blocks in Ardublock. This time we will delve into the math operators tab to find the mathematical operations I am sure you are familiar with: multiplication, division, addition, subtraction and some you are not so familiar with. You can see that each operation has room for two values to fit inside. Furthermore you can fit another operation inside of a first. Because we need to do division then multiplication we will use this little trick. We are also going to write another variable called voltage, which will be the actual 0-5 voltage at that point in the circuit;



I've only added a few things to our previous program. First I have created the voltage variable, which you'll notice is a different type of variable. It is a decimal variable. I did this because I want to be able to read values like 4.5V or 2.52V, not just 0, 1, 2, 3, 4, 5. The set decimal number variable block can be found inside the variables/constants tab. The value of the voltage variable looks a little wonky. I started with a multiplication operator and put a division operator inside that. Inside the division operator I have the variable raw divided by 1023 and that result is being multiplied by 5, just like in the formula. I am using a message this time just to make what's printed clear, and I've glued the voltage variable to that message.

So upload this, turn on the serial monitor and see what happens as you move turn the dial. ……. So, how'd that go? I'm guessing not too well. You likely only saw the values of 0.00V and 5.00V, what happened? I'll tell you what happened, we were not careful enough with our number types. We correctly called voltage a decimal number, but look at what we defined voltage to be. It's an integer divided by an integer multiplied by another integer. So the outcome is, you guessed it, an integer. There is one small easy change needed to make this work;



Can you see what I have changed? I changed the number type of 1023 to a decimal number (also found in the variables/constants tab). Beforehand, no matter what the value of raw was the outcome of raw/1023 was always either 1 or 0 because it was not allowed to be anything other than an integer. Now that 1023 is considered a decimal number (even though it doesn't have any decimals in it) the result of raw/1023 is allowed to be a decimal number. This is a good example of why keeping track of data types is so important. I generally write everything in terms

of decimal numbers to keep me out of trouble, although that is not always reasonable or even possible.

Arduino:

Getting our robot to print to the serial monitor in the C based Arduino in actually not very difficult. For starter there are no variables to define previous to the void setup, so let's jump right into the void setup;

```
void setup() {
	pinMode(A0,INPUT);
	Serial.begin(9600);
}
```

It's a pretty short and sweet void setup. The pin we have set as an input, A0, is a special pin on the Arduino. It is capable of reading voltage very accurately, which is why we are using it in our current circuit. We want this pin set as an input so that we can read the voltage at the midpoint of the circuit as we manipulate the potentiometer. The next line, Serial.begin(9600);, simply starts the serial monitor so that we have something to print our data to, and the 9600 is just referring to the rate of data capture (the maximum speed that the robot can print to the serial monitor).

Let's jump to the void loop now;

```
void loop() {
```

We need to create a variable that holds the value of pin A0, so that we can print that value to the serial monitor.

```
	int raw = analogRead(A0);
```

notice that I specified that the variable, which I named raw, as an integer.
I then set that variable to the value being read from pin A0 at that point in time.

I then want to give the actual order to print that value to the serial monitor.

```
	Serial.println(raw);
```

there are two print orders: .print and .println. .println will end the line so that anything printed after that will be printed on a new line.

```
        delay(500);
}
```

The delay block is just there to slow the rate of numbers being printed

Here is how the final code should look;

```
void setup() {
  pinMode(A0,INPUT);
  Serial.begin(9600);

}

void loop() {
 int raw = analogRead(A0);
  Serial.println(raw);
  delay(500);

}
```

Now upload this code to your robot and turn on the serial monitor, which is located in the tools tab in the Arduino window. What do you see? How about when you turn the dial on the potentiometer? You should see a number between 0 and 1023 that changes as the dial is turned.

That's great, but I know that number corresponds to a voltage, and I would really like my robot to just tell me what that voltage is. In order to do that we're going to have to teach our robots some math. You can probably guess that the number 0 refers to GND, or 0V, and 1023 refers to 5V. So what does the number 1 refer to? It refers to a voltage that is 1/1023 of the maximum 5V. So if I want to know the voltage that any raw number refers to I would use the following equation;

$$(raw)/1023 \times 5V = Voltage$$

So our code will look very similar to our previous code up to a point;

```
void setup() {
        pinMode(A0,INPUT);
        Serial.begin(9600);
}
```

```
void loop() {
        int raw = analogRead(A0);
```

Up to this point our code is identical to what we have already done but that's about to change. I'm going to introduce a new variable and a new data type in the next line;

```
        float voltage = (raw/1023)*5;
```

I first want you to notice that this new variable 'voltage' makes use of the math we figured out earlier to define its value. second , I put the word float in front of voltage to tell the computer and robot what kind of number voltage can be. A floating point number can have decimal places, and since I would like to read exact voltage values like 2.52V or 4.50V I have chosen to make voltage that kind of number. Now I am going to add a couple print statements;

```
        Serial.print("Voltage = ");
        Serial.println(voltage);
        delay(500);
}
```

The first print statement is just so that I can label my data. Notice that it is only a .print not a .println so everything will be printed on one line before moving on to the next line, like so; Voltage = some number (whatever the voltage happens to be at that time). Here is an example of the final code;

```
void setup() {
  pinMode(A0,INPUT);
  Serial.begin(9600);

}

void loop() {
  int raw = analogRead(A0);
  float voltage (raw/1023)*5;
  Serial.print("Voltage = ");
  Serial.println(voltage);
  delay(500);

}
```

Now upload this into your robot, turn on the serial monitor and turn the dial on the potentiometer. What happens? Well unfortunately you are probably only seeing Voltage = 0.00 or Voltage = 5.00. So something went wrong.

The problem was in this line;

Float voltage = (raw/1023)*5;

I correctly defined voltage as a float, and I know that is working correctly because the numbers in the serial monitor are displayed with decimals. The problem is when I divide raw by 1023. Remember that the computer and robot view both of those numbers as integers, and although we know that you can divide two integers and get a decimal number the computer sees integer/integer and says ...well that's obviously an integer. So the computer insists that raw/1023 is either 1 or 0, and since that number is being multiplied by 5 the only outcomes we can get are 5.00 and 0.00. So we need to change either raw or 1023 to a float. I chose to change 1023 to a float like so; raw/(float)1023. Just by putting the word float next to 1023 the computer will now recognize 1023 as a float. If you choose instead to change raw you must make sure to define it as a float to begin with rather than an integer;

float raw = analogRead(A0);

Here is my version of the finished code;

```
void setup() {
  pinMode(A0,INPUT);
  Serial.begin(9600);

}

void loop() {
 int raw = analogRead(A0);
 float voltage = (raw/(float)1023)*5;
 Serial.print("voltage = ");
  Serial.println(voltage);
  delay(500);

}
```

Now uploading this code and manipulating the potentiometer dial should provide the results expected.

Further Discussion:

1) What are some applications there might be for a potentiometer? One fairly common application is in light dimmers. As we saw early in the lesson the potentiometer can be used to dim or brighten a light conveniently, and you can imagine if your house has a dimmer that there is likely some similarities between its circuit and what we have done here today.
2) We ran into some trouble today. Our robot was not giving us the information we wanted it to give us. We had to troubleshoot the problem, as we often will throughout this course. We first looked at the code, the software, when attempting to fix the issue. This is often what engineers will do when they run into

a problem. The software problems are generally easier to fix as they don't involve rebuilding or rewiring anything so if the code should always be explored in more detail first when there is an issue.

3) The strange number 1023 came up many times in this lesson. Why is that? Remember me telling you that the pin A0 can read things very accurately. There is actually a name for the amount of accuracy it has, called 10 bit precision. The non analog pins on the Arduino have only 8 bits of precision. So what do bits of precision mean? We can start answering that question by asking another, what is a bit? A bit is the smallest unit of information that a computer or microcontroller can store. A bit is a location that holds a number, and that number can only be 1 or 0. So a pin capable of only one bit of precision is only capable of seeing two modes, such as high and low which we are familiar with (all of the Arduino pins are capable of at least 8 bit precision, even if we don't use all of that precision). An 8 bit number has eight locations instead of 1. Hopefully this puts a good picture in your head;

_ _ _ _ _ _ _ _

Each of those spaces can be either 1 or 0. So all together how many numbers can be expressed by an eight digit binary number? Well each digit is capable of only 2 different numbers itself so;

$$2x2x2x2x2x2x2x2=256$$

So instead of high and low I can cut 0V and 5V into 255 evenly spaced segments and each number can be related back to an appropriate voltage. I say 255 because zero is included in the 256 numbers that I can have, so 0-255 is 256 numbers. Just out of curiosity, how accurate is 8 bits of precision? Let's see;

$$(1/255) \times 5V = .02V$$

So every time you jump a number you jump .02 volts up or down. That is pretty accurate, but I imagine 10 bit precision will be far more accurate;

$$(1/1023) \times 5V = .005V$$

10 bit precision is four times as accurate! This makes alot of sense when we look at the numbers associated with 8 and 10 bit: 256 and 1024(remember 0-1023 is 1024 numbers). One is 4 times bigger than the other, allowing it to cut 0V to 5V into four times as many segments, quadrupling the accuracy.

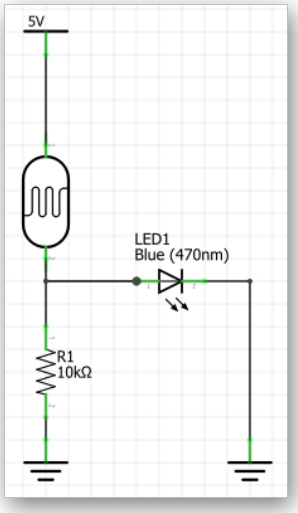
Week 4

The Photoresistor

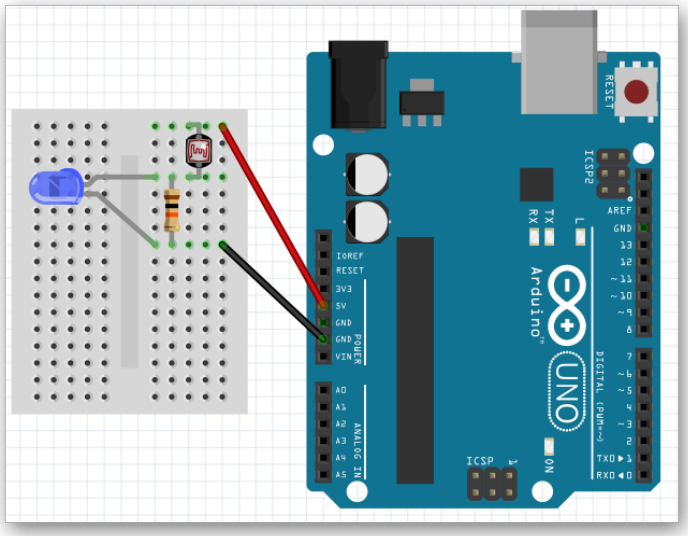
In this section we will be focusing on a particular component known as a photoresistor or light dependent resistor (LDR). This resistor is another variable resistor, and it varies depending on, you guessed it, how much light it receives. The photoresistor has a relatively small resistance when it is bright and and a relatively large resistance when it is dark.

Because of how the photoresistor functions we can utilize it in a voltage divider to create a signal that varies dependent on some outside stimulus, much like the potentiometer. However, unlike the potentiometer, the photoresistor does not need us to be the stimulus. The environment around your robot will be the stimulus.

Let's begin by asking the students to create the following circuit;

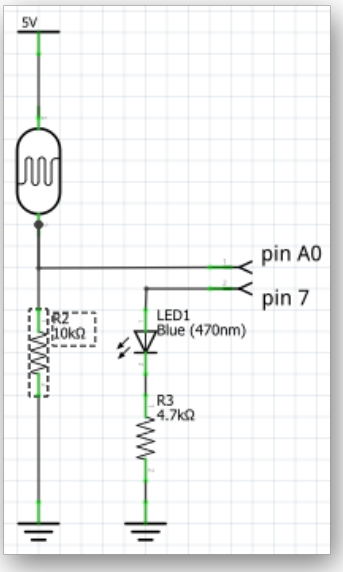


As always, allow the students to attempt building the schematic on their own before giving them access to the finished wiring diagram provided below;

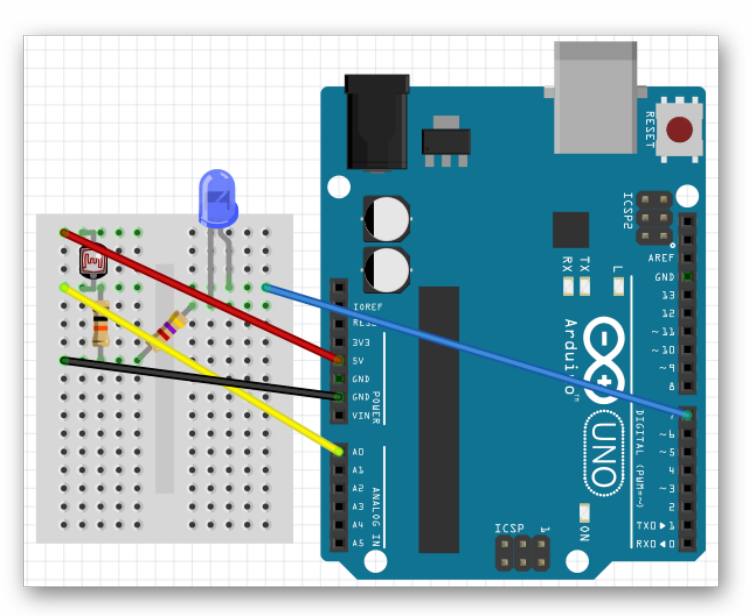


Test this circuit out! Cover the photoresistor with your hand and see if the behavior of the LED changes. Perhaps going into a room with no windows and turning the lights on and off may be more impactful, or conversely find an area of bright light. What you should see is that in dark areas the LED turns off and in bright areas the LED turns on. So we can clearly see the photoresistor at work, but it doesn't seem very useful, at least not in the way we're using it. It would be nice if the LED turned on in the dark when we need it, and off in bright light when we don't need it, rather than the other way around.

We're going to need to alter our circuit to accomplish this;



And the corresponding circuit diagram;



Before we get into the programming section of this lesson let's try to think critically about the behavior of this circuit. Remember that the stimulus we want to detect is darkness. When it is dark, the photoresistor has a very high resistance. Because the photoresistor is located at the top of the circuit, which acts like a voltage divider, it decides how much voltage is allowed through the top of the circuit is decided by the resistance of the photoresistor. So when it is dark and the resistance of the photoresistor is high, most of the voltage is dropped across the photoresistor. That means that when we try to read the voltage with the A0 pin we should only

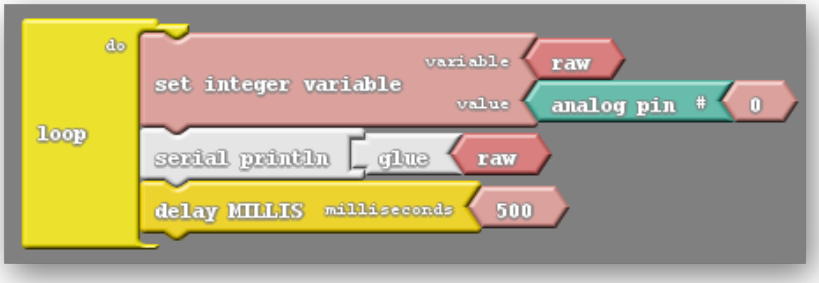
see a small amount of voltage. That will be our cue to turn on the LED, if the voltage measurement is small enough.

However, this begs the question, what is ‘small enough’? Doesn’t everyone have their own preferences about how dark is too dark? I want each student to make the decision for themselves how dark the area has to be before the LED lights up. So our task is no two fold:

1. Find that magic number that gives us the desired darkness before turning on the LED.
2. Code our robot to do just that.

Ardublock:

Ok, Let’s start with number one on our to do list. What we need to do is create a program that utilizes the serial monitor to tell us what number is associated with the appropriate level of darkness. We will end up making an identical code to part of last week’s lesson;

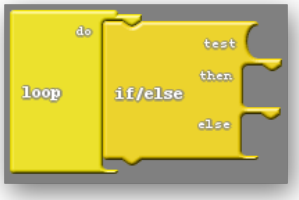


A good question to pose to the class would be ‘why is it appropriate to use this piece of code that we used last week even though the circuits are different?’ We didn’t have a photoresistor last week. We weren’t concerned about light levels. So why is our code the same?

The key to answering that question is to note that while the circuits are different, they have one very important similarity. They are both voltage dividers. Both last week’s circuit and this week’s circuit divide voltage by some form of varying resistor and attempt to read the remaining voltage.

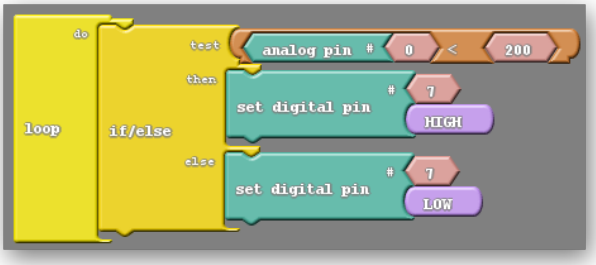
So upload this code into your robot, turn on the serial monitor, and read the values being shown. Note that they change if you cover the photoresistor with your hand. Have your students take their robots, while still attached to the computer via usb to an area they deem appropriately dark. Keep in mind that the computer screen should not be pointed towards the robot and in particular the photoresistor. Each student should now have a value on the serial monitor that relates back to the level of darkness needed to turn the LED on. Step 1 complete.

Now we need to use the information we just gained to accomplish the task we set out to do. I want to remind you of the first week's task. Remember that in week one we used a button to tell our robot what to do. Our current task is similar to that one in that we need to use a condition in our code to determine when our robot acts. That means making use of the **if/else** block in the controls tab again and one of the **test** blocks as well. Let's get started;



As always I first need to include a loop do or my code will not run or even upload. Next I placed a if/else block inside of my loop do because I need the robot to do two specific things dependent on a particular circumstance: turn the light off if it is bright, turn the light on if it is dark. So which test block will we use? I intend to use the less than test block because I want to turn on the LED for any level of darkness OR darker. Remember that the darker the area the lower the lower the reading from pin A0 will be. Inside that test block I want to include an analog pin block which must be less than the number we found earlier, like so;

I chose the number 200 as my 'preferred darkness'. You may have something different, and that's fine. Next we need to fill the then/else portions of the block. I want the LED on **if** the value read by the analog pin is low enough, **else** I want the LED off;



Note that I have completely done away with the variable assignment that I had in the previous code. That is just personal preference. It just feels clumsy in the current situation.

Arduino:

We will start by getting our serial monitor to display a value that corresponds to the resistance of the photoresistor. To do this we will actually use code identical to code used in an exercise done last week;

```
void setup() {
  pinMode(A0,INPUT);
  Serial.begin(9600);

}

void loop() {
 int raw = analogRead(A0);
  Serial.println(raw);
  delay(500);

}
```

Why would we be using code identical to what we did last week?

The answer is that each circuit, this week's and last week's, are both voltage dividers. Not only that, but they are both voltage dividers that use a variable resistor, although the type of variable resistor differs. Finally, in both cases we are attempting to read the voltage at the same point in the circuit.

Now upload this code into your robot and turn on the serial monitor. Now have the students take their robots to a place they think is sufficiently dark to need a light with the robot still attached to the computer. Have them read the number on the serial monitor with the computer screen pointed away from the robot. That number will be important in the upcoming code.

Now that we have a number associated with a particular level of darkness we can use that knowledge to turn on our robot's LED at the appropriate time. Let's get started;

I'll begin with a slightly different void setup that we had previously, removing the Serial.begin statement and adding a pinmode for the LED;

```
int LED = 7;
void setup() {
        pinMode(A0,INPUT);
```

```
        pinMode(LED,OUTPUT);
}
```

The void loop is going to have some new functions that we haven't used before. We need to use what is called 'conditional logic'. We want the LED to turn on only if it is dark enough, so we need our code to essentially say;

If it's dark->turn on LED
If it's bright->turn off LED

With this in mind let's get started;

```
void loop() {
        int raw = analogRead(A0);
        If(raw<200) {
                digitalWrite(LED,HIGH);
        }
        else{
                digitalWrite(LED,LOW);
        }
}
```

Here is a finished version of the code;

```
int LED = 7;
void setup() {
  pinMode(A0,INPUT);
  pinMode(LED,OUTPUT);

}

void loop() {
  int raw = analogRead(A0);
  if (raw<200) {
    digitalWrite(LED,HIGH);
  }
  else {
    digitalWrite(LED,LOW);
  }

}
```

The **if** and **else** statements are the conditional logic we talked about earlier. Notice the condition inside the if statement raw<200. Remember that as the area gets darker, the number 'raw' gets smaller. So the if statement is saying turn on the LED if it is this dark or darker, the else statement says if it's not at least that dark the LED will stay off.

I also want to emphasize the use of the bracket symbols {}, we're using many of them here. These symbols group actions together and let the computer and robot understand which commands go with which conditions. If we look carefully our void loop has brackets within bracket to help group the light on/off commands with the if/else logic rather than the void loop at large.

So try uploading this code and taking the robot into a dark area to see if the code is working. If it is, congratulations! You are done.

Troubleshooting:

The most common problem to have is the LED not lighting up in dark areas after implementing the final code. Remember that the number you placed in the **if** statement represents the threshold darkness needed to turn the LED on. If the LED is not responding to what you feel is the appropriate level of darkness consider making that number 10-50 larger. So, in my example rather than keeping the number 200 I could change it to about 250 and be much more likely to see my code work as intended.

What have we accomplished:

We have, in the past, described a robot as having human parts like: a body, a brain, a heart, a soul. I want to describe robots differently for just a moment. I want to describe a robot as having sensors and actuators. These are just fancy words for inputs (sensors) and outputs (actuators). A fully autonomous robot will take information from the world around it using its sensors, process what it sees/hears/feels with its brain, then uses it's actuators to react to what it has sensed. This week is the first time we have done this. We are well on our way to becoming real robot scientists and it is hard to understate this milestone to your students.

Wasn't there an easier way?

Yes, as a matter of fact there was an easier way to accomplish the task that I just praised you for. If you look back to the first circuit we made this week, you'll remember that it had the behavior opposite what we wanted. We wanted the LED to turn on in the dark and off in the light. Now if we were to switch the places of the resistors, putting the standard resistor at the top of the voltage divider and the photoresistor at the bottom of the circuit, we would have inverted the behavior of the circuit.

Why didn't I just have you perform that task instead? It would have been far easier. There are two main reasons I didn't approach this week's lesson in that manner:

1. Look at all the things we learned this week. We learned some new methods of coding that will be useful to us in the future as well as obtain a level of automation with the robots that I described earlier as being an incredibly important milestone.
2. By supplementing our electronics knowledge with our coding knowledge we were able to exercise a greater amount of control over what our robot was doing. We were able to tell it exactly how dark it needed to be in order for the light to turn on. In general this is true, that adding a software engineering aspect to our circuits offers a larger amount of control and therefore a larger amount of convenience.

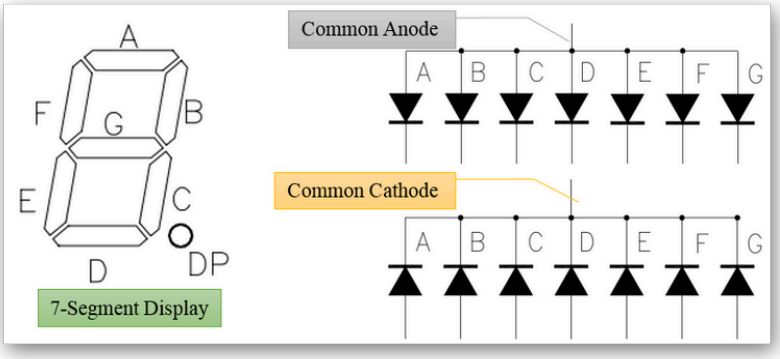
UNIT 3: The Common Cathode Display (CCD)

Week 5

The common cathode display (CCD) is a component that you are probably familiar with. At some point in your life you have seen a digital display of numbers or letters using the object in the picture above. This week you're going to learn how these things work.

I first want to explain what the name common cathode display means. A cathode is a type of electrode, which is just a conductor that current flows out of or into. For example the pins on the brain of our robot are electrodes. A cathode is just an electrode that current specifically flows to. There is another kind of electrode called an anode, which current flows from, and there is another type of digital display called a common anode display (CAD) which makes use of it.

Either of these displays, either the CCD or CAD, is simply several LEDs grouped together by one of these electrodes. As a matter of fact I have a picture that illustrates the difference between the two;



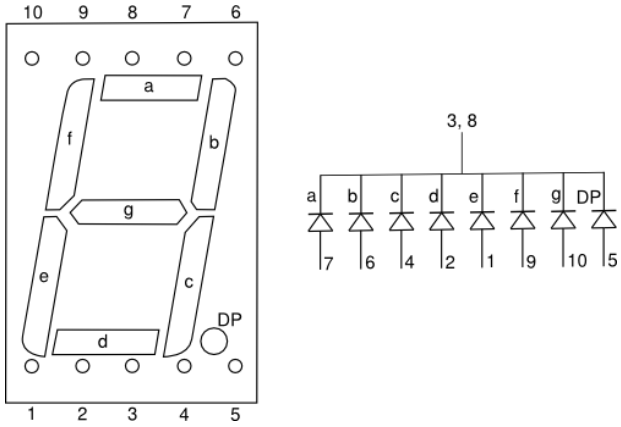
Notice that the only thing that is different between the two of them is the direction the LEDs face in relation to the electrode. In the common anode display the LEDs are all facing away from the anode while in the common cathode display all the LEDs are facing toward the cathode.

To better understand why we are using a CCD rather than a CAD we're going to need to familiarize ourselves with the CCD. So let's just keep the CAD in the back of our minds as we go through this lesson.

Understanding Datasheets:

With so many complex electronic components out there it's impossible to know how every one of them works. Luckily, any company that makes electronic components will offer what are known as data sheets online that you can open and download for free. A datasheet will tell you the function of each pin on a component and often tell you the peak operating conditions of the component. The CCD we are using is the LDS-C514RI Made by Lumex. You can squint and see that listed on the side of the part. Just by googling that you can find the datasheet for this part as a PDF, and I encourage you to challenge your students to practice their google fu, it may sound silly but it is an important skill. If they still have trouble here is the Datasheet PDF.

At the bottom of the datasheet we are given a system of numbers and letters that are associated with each other. Even better there is an illustration of the CCD that shows us which part of the display is associated with each letter. Unfortunately there is no illustration that shows which pin on the CCD is associated with each number. Luckily, CCD's are very common and I was able to find an illustration that does just that;

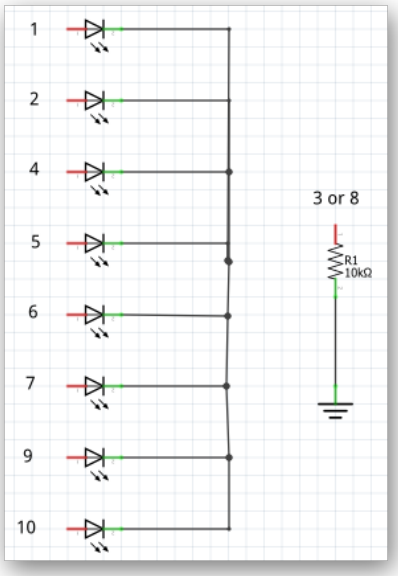


The above illustration will greatly assist in understanding how to write legible information to the CCD. For example, if I wanted to make the number 7 appear on the CCD I would need the a, b, and c segments to light up. Those segments are associated with the 7, 6, and 4 pin **of the CCD**, not of the Arduino. So I would need to write high signals to those pins of the CCD.

We need to wire our CCD together. If you're using a fully wired robot already you are probably wondering how we wire all of these to the current Arduino board as things are already a bit crowded. There are two answers to this question;

1) Use a completely new Arduino and breadboard for this activity.
2) Take all of the wires and components out of your current breadboard and Arduino/Noggin so you have the space needed.

Consider your options, and when you and your students are prepared, attempt to make the schematic below;



The above schematic is a little difficult to understand so I will try to explain. I want to match the pin number on the Arduino to the pin on the CCD with the same number. This means, for example, that I should wire pin 1 in the Arduino to the bottom left pin in the CCD, as it is also labelled pin 1.

As always I will include a finished circuit diagram, however because of how 'busy' this circuit is I will include a picture of the circuit rather than the usual circuit diagram I have been including.

(Picture of CCD wired to Arduino here)

Before moving on to next week's lesson I propose you attempt the following activity with your class;

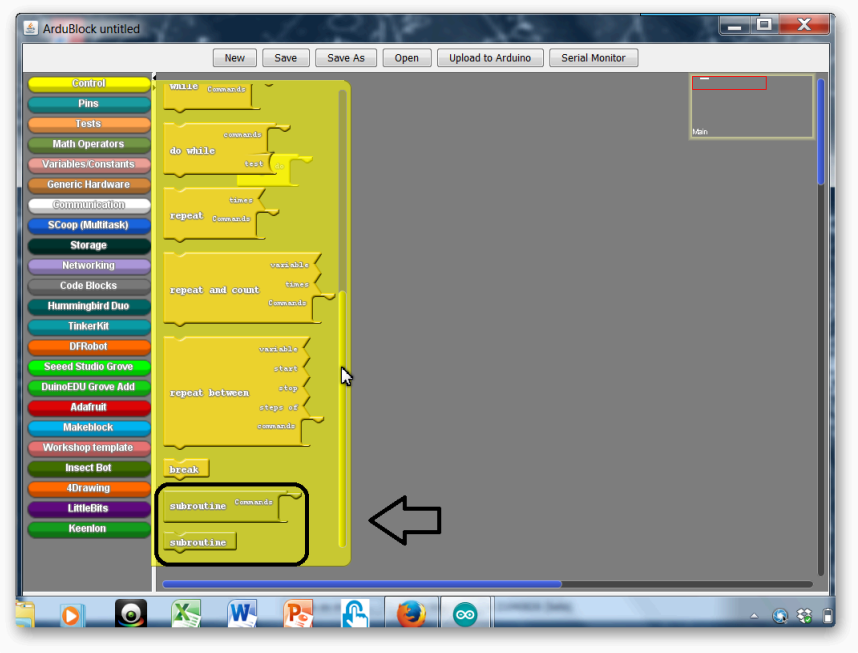
Using either programming or simply rewiring the circuit as needed, have your class collectively try and figure out which segment of the display is related to which pin of the CCD.

Week 6

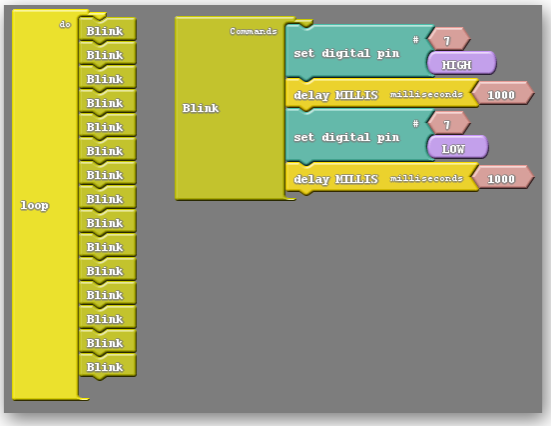
With the CCD wired we can move on to programming.

Ardublock:

Before we start programming the CCD I want to introduce the subroutine blocks;



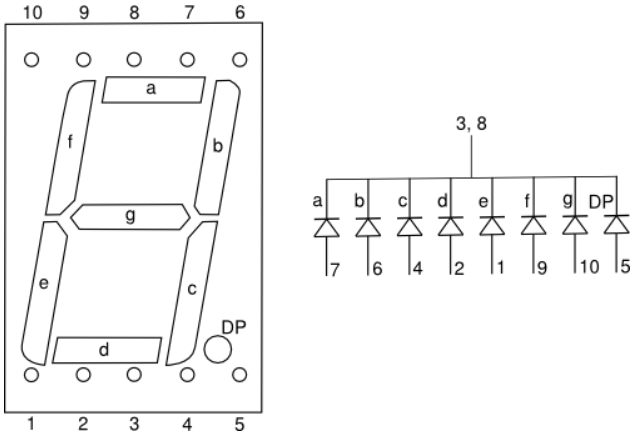
The subroutine blocks, as you can see, are found in the controls tab in Ardublock. They consist of the commands block and what I will call the run block. The subroutines blocks allow us to create longer segments of code that we can call using only one block. I'll give you an example;



In this program I have created a subroutine called Blink, which you probably recognize as the code needed to blink the LED. Then, inside the loop do I placed several run blink blocks, each one of which will do all of the things inside of the subroutine command block called Blink. So without doing much work I have made my robot blink 15 times.

You may already be thinking about how this can be useful for the CCD. Displaying a single number on the display is itself a process. To display each number multiple LEDs inside the CCD must be turned on. Furthermore, the ones that must be turned on are different for each number we want to display. In addition, we need to turn off the LEDs in between displaying numbers. Having access to subroutines would be very nice for all these things.

See if your students can use the illustration below to create 10 subroutines, one for each number;



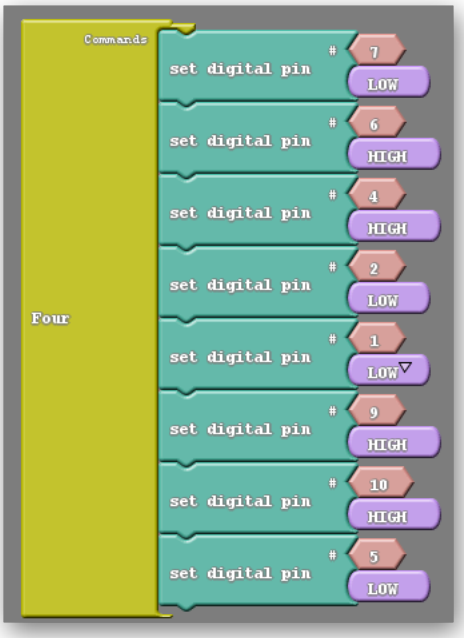
I'll list each of the subroutines here:

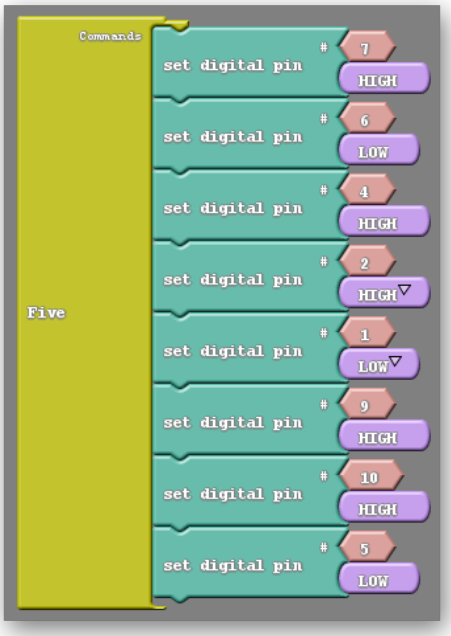

Commands
Zero
set digital pin # 7 HIGH
set digital pin # 6 HIGH
set digital pin # 4 HIGH
set digital pin # 2 HIGH
set digital pin # 1 HIGH
set digital pin # 9 HIGH
set digital pin # 10 LOW
set digital pin # 5 LOW

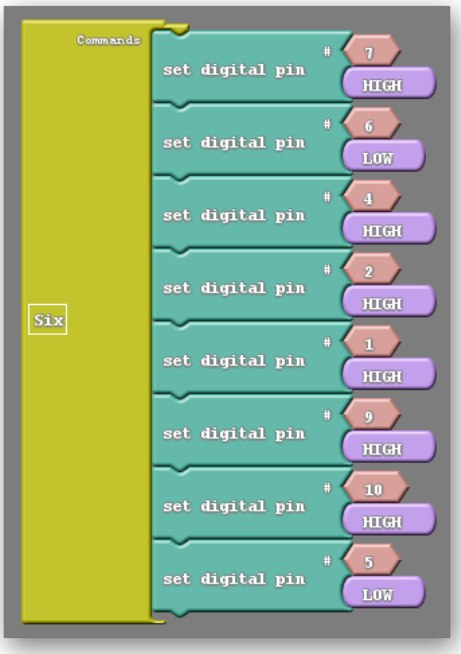

Commands
One
set digital pin # 7 LOW
set digital pin # 6 HIGH
set digital pin # 4 HIGH
set digital pin # 2 LOW
set digital pin # 1 LOW
set digital pin # 9 LOW
set digital pin # 10 LOW
set digital pin # 5 LOW

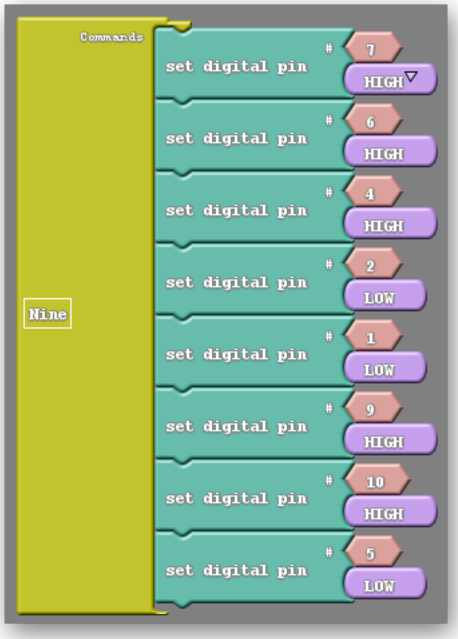

Commands
Two
set digital pin # 7 HIGH
set digital pin # 6 HIGH
set digital pin # 4 LOW
set digital pin # 2 HIGH
set digital pin # 1 HIGH
set digital pin # 9 LOW
set digital pin # 10 HIGH
set digital pin # 5 LOW


Commands
Three
set digital pin # 7 HIGH
set digital pin # 6 HIGH
set digital pin # 4 HIGH
set digital pin # 2 HIGH
set digital pin # 1 LOW
set digital pin # 9 LOW
set digital pin # 10 HIGH
set digital pin # 5 LOW


Commands
Four
set digital pin # 7 LOW
set digital pin # 6 HIGH
set digital pin # 4 HIGH
set digital pin # 2 LOW
set digital pin # 1 LOW
set digital pin # 9 HIGH
set digital pin # 10 HIGH
set digital pin # 5 LOW


Commands
Five
set digital pin # 7 HIGH
set digital pin # 6 LOW
set digital pin # 4 HIGH
set digital pin # 2 HIGH
set digital pin # 1 LOW
set digital pin # 9 HIGH
set digital pin # 10 HIGH
set digital pin # 5 LOW





One piece of advice I can give is to begin with each pin low and change the pins to high as needed for that given number. Keep in mind that you need all of these subroutines to exist in the same program at the same time, even though I showed only one at a time.

With these subroutines we can now easily get the CCD to display whatever numbers I want, in whatever order I want, at whatever speed I want. For example the following code will countdown from nine in one second intervals;

An additional challenge may be to integrate lessons from former weeks into this lesson. This could be done by having the CCD count how many times the button has been pressed, or give a number for relative brightness read from the photoresistor.

Arduino:

I want to start by introducing what are called functions. A function will allow us to act out many commands by only typing one line of code. The caveat is that we must define the function before using it, doing some work up front before being able to reap the rewards. For example below is code that will make the robot blink, only this time using a function to do so;

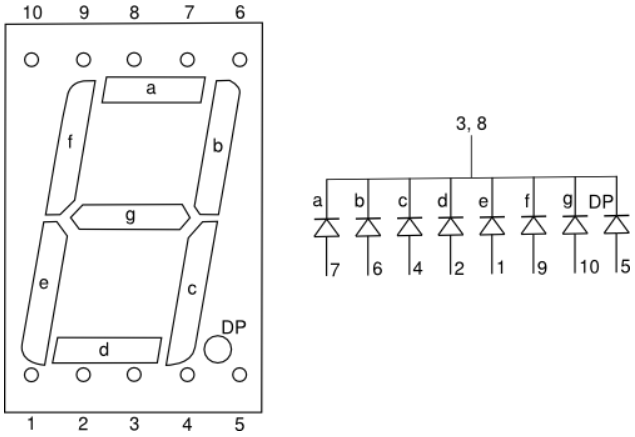
```
int LED = 7;
void setup() {
  pinMode(LED,OUTPUT);

}
void Blink() {
  digitalWrite(LED,HIGH);
  delay(1000);
  digitalWrite(LED,LOW);
  delay(1000);
}
void loop() {
  Blink();

}
```

You'll notice the usual things up through the void setup like naming pins and assigning pin modes. However in between the void setup and void loop I have inserted something we are unfamiliar with. I have called this thing void Blink(), and this is the function that I am defining. Just like LED needs int in front of it to identify what kind of number it is, the function Blink() needs void in front of it to identify what kind of function it is. Inside of the void blink you may recognize the lines of code that cause our robot to blink. These lines of code in this program do not make our robot blink but instead tell the robot that whenever the function Blink() is called to do those lines of code. So essentially when I tell the robot Blink() it will run those four lines of code which we know will make the robot blink.

Now I could have named that function something else, like void smile or void fly or something like that. The name just references those lines of code inside it and will not break if the name doesn't have anything to do with the code. The reason I gave that function the name Blink() is because it will be easier to remember what it does than if I name it something else.

I would like to make a set of functions for our CCD. If I don't do so we will have much more work to do and our code will be much more disorganized, making it difficult to find errors. What makes programming with the CCD so complicated? To answer that question you must remember that the CCD is basically 8 LEDs. This means that you must designate 8 pins to be high or low to display a given number, and which pins are high and which are low will affect what is ultimately displayed.

Let's take the number zero for starters. Looking at the CCD diagram again;



I can see that to display the number zero a, b, c, d, e and f must be turned on, while g and DP must be off. The lettered segments that must be on correspond to pins 7, 6, 4, 2, 1 and 9 while the segments that must be off correspond to pins 10 and 5. With that in mind our function for zero would appear like so;

```
void zero(){
  digitalWrite(7,HIGH);
  digitalWrite(6,HIGH);
  digitalWrite(4,HIGH);
  digitalWrite(2,HIGH);
  digitalWrite(1,HIGH);
  digitalWrite(9,HIGH);
  digitalWrite(10,LOW);
  digitalWrite(5,LOW);
}
```

Imagine if I wanted to display zero multiple times. Instead of writing out these eight lines of code every time, we can just say zero() and the robot will understand. Have your students attempt to create functions for the other numbers as well. I will list them below for additional guidance if needed;

```
void one(){
  digitalWrite(7,LOW);
  digitalWrite(6,HIGH);
  digitalWrite(4,HIGH);
  digitalWrite(2,LOW);
  digitalWrite(1,LOW);
  digitalWrite(9,LOW);
  digitalWrite(10,LOW);
  digitalWrite(5,LOW);
}
```

```
void two(){
  digitalWrite(7,HIGH);
  digitalWrite(6,HIGH);
  digitalWrite(4,LOW);
  digitalWrite(2,HIGH);
  digitalWrite(1,HIGH);
  digitalWrite(9,LOW);
  digitalWrite(10,HIGH);
  digitalWrite(5,LOW);
}
```

```
void three(){
  digitalWrite(7,HIGH);
  digitalWrite(6,HIGH);
  digitalWrite(4,HIGH);
  digitalWrite(2,HIGH);
  digitalWrite(1,LOW);
  digitalWrite(9,LOW);
  digitalWrite(10,HIGH);
  digitalWrite(5,LOW);
}
```

```
void four(){
  digitalWrite(7,LOW);
  digitalWrite(6,HIGH);
  digitalWrite(4,HIGH);
  digitalWrite(2,LOW);
  digitalWrite(1,LOW);
  digitalWrite(9,HIGH);
  digitalWrite(10,HIGH);
  digitalWrite(5,LOW);
}
```

```
void five(){
  digitalWrite(7,HIGH);
  digitalWrite(6,LOW);
  digitalWrite(4,HIGH);
  digitalWrite(2,HIGH);
  digitalWrite(1,LOW);
  digitalWrite(9,HIGH);
  digitalWrite(10,HIGH);
  digitalWrite(5,LOW);
}
```

```
void six(){
  digitalWrite(7,HIGH);
  digitalWrite(6,LOW);
  digitalWrite(4,HIGH);
  digitalWrite(2,HIGH);
  digitalWrite(1,HIGH);
  digitalWrite(9,HIGH);
  digitalWrite(10,HIGH);
  digitalWrite(5,LOW);
}
```

```
void seven(){
  digitalWrite(7,HIGH);
  digitalWrite(6,HIGH);
  digitalWrite(4,HIGH);
  digitalWrite(2,LOW);
  digitalWrite(1,LOW);
  digitalWrite(9,LOW);
  digitalWrite(10,LOW);
  digitalWrite(5,LOW);
}
```

```
void eight(){
  digitalWrite(7,HIGH);
  digitalWrite(6,HIGH);
  digitalWrite(4,HIGH);
  digitalWrite(2,HIGH);
  digitalWrite(1,HIGH);
  digitalWrite(9,HIGH);
  digitalWrite(10,HIGH);
  digitalWrite(5,LOW);
}
```

```
void nine(){
  digitalWrite(7,HIGH);
  digitalWrite(6,HIGH);
  digitalWrite(4,HIGH);
  digitalWrite(2,LOW);
  digitalWrite(1,LOW);
  digitalWrite(9,HIGH);
  digitalWrite(10,HIGH);
  digitalWrite(5,LOW);
}
```

In our excitement let's not forget to designate our pin modes in the void setup;

```
void setup(){
  pinMode(7,OUTPUT);
  pinMode(6,OUTPUT);
  pinMode(4,OUTPUT);
  pinMode(2,OUTPUT);
  pinMode(1,OUTPUT);
  pinMode(9,OUTPUT);
  pinMode(10,OUTPUT);
  pinMode(5,OUTPUT);
}
```

Keep in mind that you can define all of your functions before or after the void setup but they CANNOT BE DEFINED INSIDE THE VOID SETUP! They also can be defined after the void loop but BUT NOT INSIDE IT!

Now that all of our pins are setup and we have defined all of our functions we can finally do something with our CCD. I am going to attempt to count down from 9 in one second intervals;

```
void loop() {
  nine();
  delay(1000);
  eight();
  delay(1000);
  seven();
  delay(1000);
  six();
  delay(1000);
  five();
  delay(1000);
  four();
  delay(1000);
  three();
  delay(1000);
  two();
  delay(1000);
  one();
  delay(1000);
  zero();
  delay(1000);
}
```

Which brings us to the end of the lesson.

An additional challenge may be to integrate lessons from former weeks into this lesson. This could be done by having the CCD count how many times the button has been pressed, or give a number for relative brightness read from the photoresistor.

Synthesis Project - Night Light with CCD

The following is an optional synthesis project where you combine the the night light with the CCD.

Sample Code:

- Ardublock Code Template for students
- Ardublock Code
- C Code

Video Demo: https://youtu.be/2oYp1DwjNE4

UNIT 4: Logic Gates

Week 7

Now that we have some understanding of mosfets e can move on to larger logic structures. These logic gates that we will explore are constructed out of mosfets themselves. And while we'll leave how to actually construct logic gates for later we'll learn how each of them works in this lesson.

Before we go any further I want to make it clear that this week's lesson is far more pen and paper, more cognitive, than the last several weeks have been.

So what are logic gates? Aside from some mosfets put together. A logic gate is an electrical component that takes two distinctly different electrical signals and, based on those, output it's own high or low signal depending on the logic of that particular gate. For example an AND logic gate wants both incoming electrical signals to be on. One AND the other, hence its name. So if both incoming signals are high, then the outgoing signal is high. However the AND gate will not create a outgoing high signal for any other combination of incoming signals. Not if one is high but not the other, and not if neither is high. We can actually make a table for the logic of the AND gate;

Input A	Input B	Output Q
low	low	low
low	high	low
high	low	low
high	high	high

The letters A,B and Q are just convention. You'll get used to seeing them. I also want to familiarize you with the binary version of the above table, which appears thusly;

A	B	Q

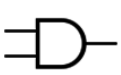
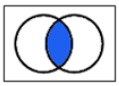
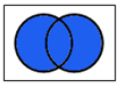
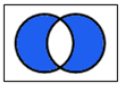
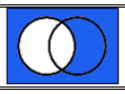
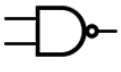
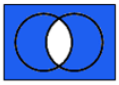
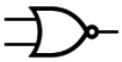
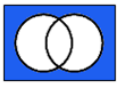
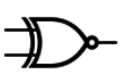
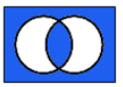
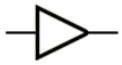
0	0	0
1	0	0
0	1	0
1	1	1

Strangely, there is a lot to absorb with this minimalistic table. First is the explanation of the 1s and 0s in place of the highs and lows. 0 just replaces low and 1 replaces high, but there's plenty more to talk about.

The 1s and 0s are the only characters in binary, which is in a way the most pure/basic language of computers. Those 1s and 0s are only representing high and low signals, that's it. So think about how many 1s and 0s your computer is juggling around in its head. Each and every single of of those numbers represents a very real electrical signal that exists in your computer. Every time your computer needs to store new information it does so by changing signals in circuits that contain logic gates. The very reason that your computer can do logical things is because the pieces it is made out of can do logical things!

That explanation is somewhat long winded and unnecessary for this week's lesson, but I think it is important to demystify an appliance that we use so regularly. I find that with added understanding, no matter how miniscule or trivial the information, so of the fear surrounding the object is dissipated. My hope is that with some added insight into how computers tick students will be more comfortable jumping into tech related subjects with both feet.

Let's get back on track. The table I presented last has a name. It is known as a **truth table**, and the logic that governs it is known as **truth table logic**. Now I have shown you the truth table for just one type of logic gate thus far but there are more! Take a look;

Expression	Symbol	Venn diagram	Boolean algebra	Values
AND			$A \cdot B$	A B Output 0 0 0 0 1 0 1 0 0 1 1 1
OR			$A + B$	A B Output 0 0 0 0 1 1 1 0 1 1 1 1
XOR			$A \oplus B$	A B Output 0 0 0 0 1 1 1 0 1 1 1 0
NOT			$\overline{A}$	A Output 0 1 1 0
NAND			$\overline{A \cdot B}$	A B Output 0 0 1 0 1 1 1 0 1 1 1 0
NOR			$\overline{A + B}$	A B Output 0 0 1 0 1 0 1 0 0 1 1 0
XNOR			$\overline{A \oplus B}$	A B Output 0 0 1 0 1 0 1 0 0 1 1 1
BUF			A	IN Output 0 0 1 1

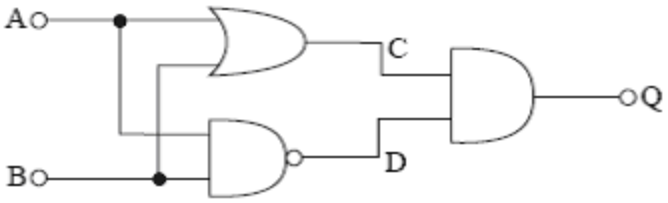
Venn Diagram for logic gates is a schematic representation of A and B overlapping each other inside a rectangle area, the diagram shows the relation of the boolean operators.

So understanding all of the information in the above diagram is unnecessary (looking at the boolean algebra column), however I particularly like the venn diagram visualization that this diagram chooses to use. I want you to focus on being able to relate the visual representation of each gate with the TTL or truth table logic associated with it, ultimately that will determine the behavior of a circuit using logic gates.

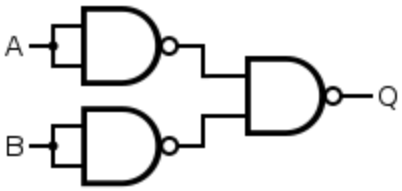
Exercises:

Below are a few circuits that use logic gates. The goal of these exercises is for the students to create a truth table for each circuit. Video solutions will be provided.

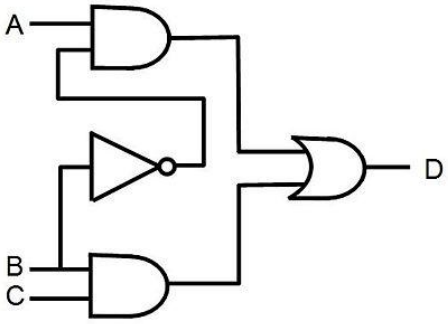
1)



2)

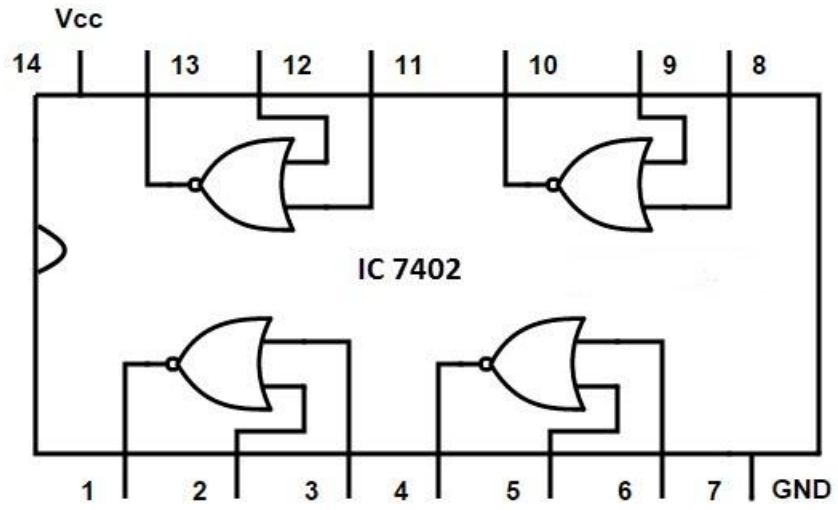


3)



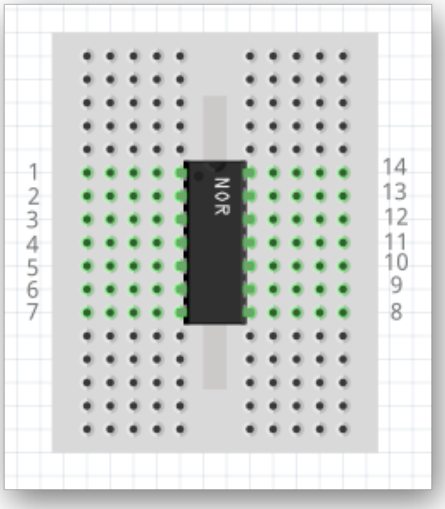
Within our hardware kits you should see a small chip. This chip contains within it several NOR gates. The first thing you'll notice when looking at the parts is that the NOR chip doesn't look anything like the NOR gates looked in the circuit diagram. That's not particularly surprising, after all a real resistor isn't just some squiggly lines, is it? The NOR chip is a bit more complex though, as there are 14 different pins on it. How are we going to be able to tell which pins are the inputs and which is the output? And why does the IC have 14 pins?

Well the NOR chip actually contains within it several individual NOR gates. The one we are using contains four as shown below;



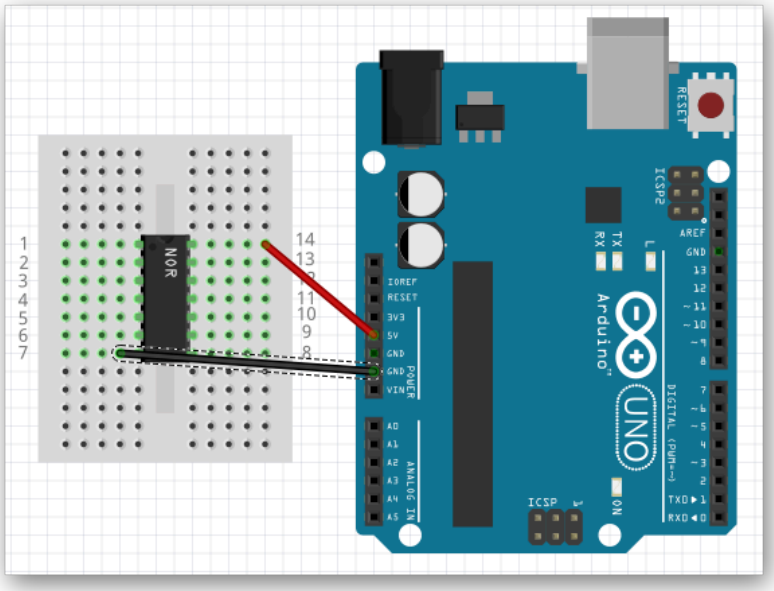
As you can see, this single chip contains four NOR gates, or at least that is what is suggested by the illustration. However, rather than just take my word for it, I would like the students to verify that this is fact the case by confirming the TTL for a NOR gate.

To do this I would instead recommend walking them through the wiring process, beginning with understanding the placement of the chip on the breadboard;

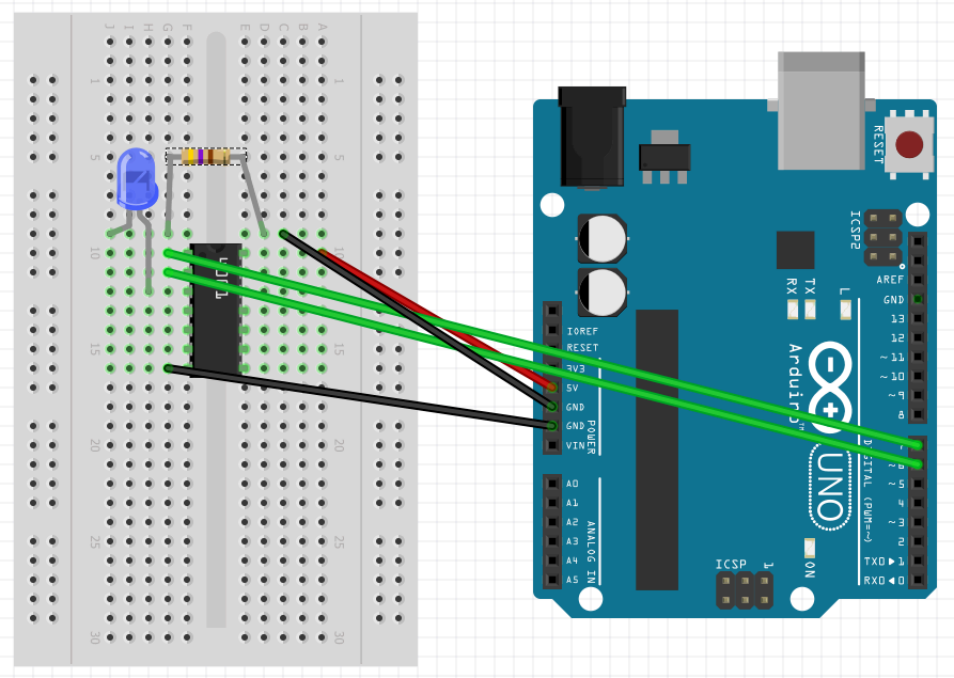


You can see that I've labelled the rows of the breadboard where the chip is sitting 1-14. Notice that the small notch of the chip is at the top in this picture. This is important as it is telling me that those top rows are the 1 and 8 pins on the chip. Also looking at the drawn illustration of the chip above We can pinpoint where each individual NOR gate is located. So 1, 2, and 3 are one NOR gate. 4, 5, and 6 are another. 8, 9, and 10 are yet another. 11, 12, and 13 are the last. The

obvious oddballs are pins 7 and 14, as they don't belong to any of the NOR gates in the chip. These pins are needed to power the chip, with pin 7 being attached to ground and pin 14 being attached to 5V. This is similar to the CCD. If you'll remember the CCD needed to be connected to power and ground in addition to having the other pins attached to programmable pins. The same is true for this chip we are using now, so as far as wiring this circuit together that is a good place to start;



We only need to verify one of the gates is NOR, from there I think it is safe to assume that the others are identical. So we must wire the two inputs of one gate to two programmable pins on the Arduino, as well s put an LED on the output of that gate so we can see whether it's on or off;



Now we just need to assign pins 6 and 7 to be HIGH and LOW to verify the TTL of the NOR gate from a couple of pages ago;

Pin 6	Pin 7	LED
LOW	LOW	?
HIGH	LOW	?
LOW	HIGH	?
HIGH	HIGH	?

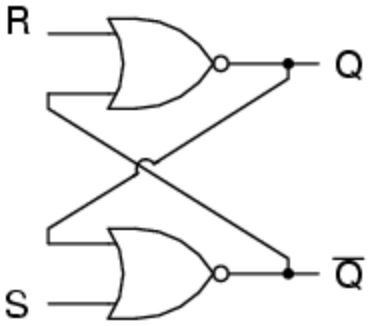
What behavior do you see? Is it what you expect?

Week 8

The SR Latch

For our last lesson I wanted to take what we've learned in the last few weeks of this class and use that knowledge to help us understand what I think is a really cool and important circuit. This circuit is the SR latch.

This circuit is awesome for reasons I'm going to keep to myself for now, so you're just going to have to trust me. Before we get to that I should probably introduce the actual circuit, so here it is;



S	R	Q	$\overline{Q}$
0	0	latch	latch
0	1	0	1
1	0	1	0
1	1	0	0

We have not only the circuit, which does look a bit weird, but the truth table as well, which also looks a little weird.

Why don't we start with the circuit. You can see the circuit is made of two NOR gates that are tied together in a peculiar way. By that I mean that what comes out of one is fed back into the other. This leads to this very weird chicken or the egg scenario where the output of one affects the output of the other, which affects the output of the first, which then affects the output of the other and oh no I've gone cross eyed.

The second thing to notice about the circuit is that it has two outputs. Normally we only consider one of the outputs but there are circuits that are built around this one that make use of both halves of the circuit. We're not going to use both halves at the same time but do notice that the circuit is symmetrical, so whatever stuff we're doing can be replicated on the mirrored half of the circuit.

So with all that said about the circuit, let's talk about the truth table, which should help us understand how the circuit actually functions. The last three rows look pedestrian enough. 1s and 0s, we understand what those do, what they mean. It's the first row that gives us trouble. Instead of just telling us what we get when both S and R are low, it just says latch | latch. That's not very helpful

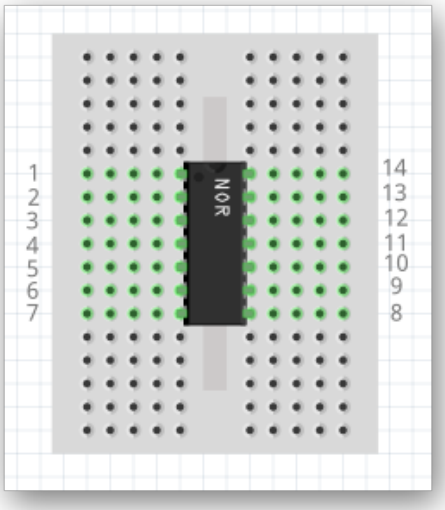
So I think it's finally time to reveal the big secret of the SR latch. The SR latch is a circuit that can remember. If you set S to 1 and R to 0 then set S to 0, the circuit remembers that S was at some point on and will keep Q high and Q bar (as I will now call it) low. Latch | latch simply means keep the last thing Q and Q bar were doing before S and R were both 0.

It goes a bit deeper than this. You're probably wondering at this point why the inputs are called S and R. Well S is often referred to as Set, and R is usually referred to as Reset. If you have turned on S at any time in the past Q will remember and stay on. S sets the memory. R will wipe Q, give it the value of 0 again. R resets the memory. Don't forget what I said earlier though, this circuit is symmetric. So S not only sets Q but resets Q bar, and R not only resets Q but sets Q bar.

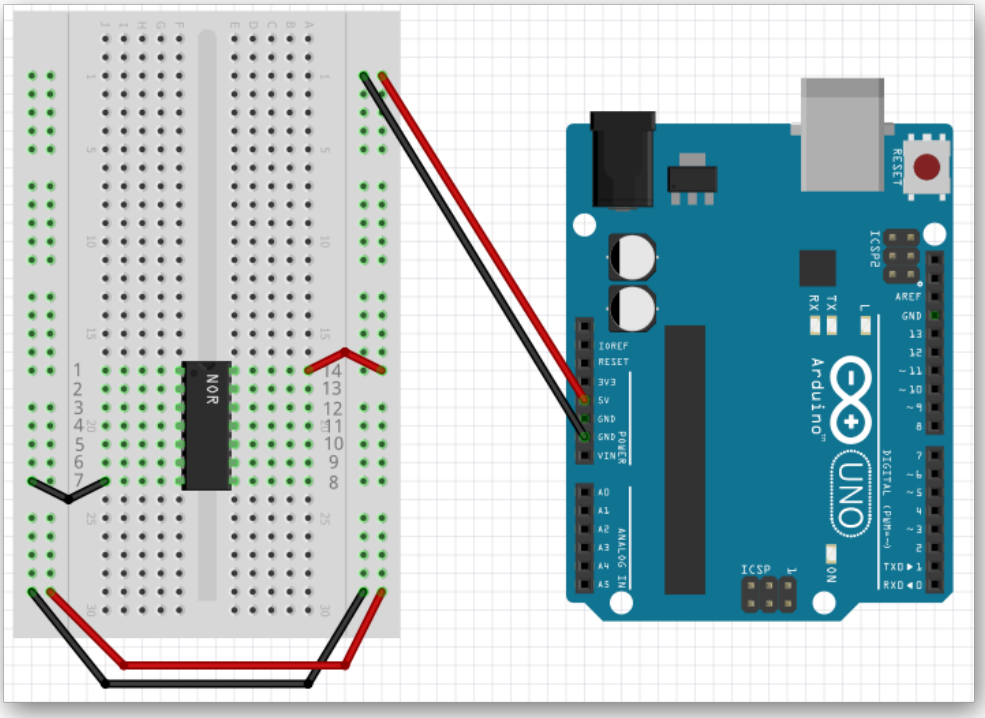
I don't want us to get lost in the technicalities of this circuit. Ultimately what it does is both very easy to understand and very powerful. It remembers. This circuit is the most basic memory circuit. Without it we would not have computers and many of our other everyday appliances. This circuit can store either a 1 or a 0, a single bit of memory, the smallest amount possible.

How many of these do you think your computer needs to remember all of the things it does? Well let's use my computer as an example. My laptop has 8 gigabytes of RAM. A gigabyte is one billion bytes, so I have 8 billion bytes of memory available for use. But how big is a byte? A byte is made of 8 bits, 8 SR latches. So my computer needs to contain 64 billion SR latches.

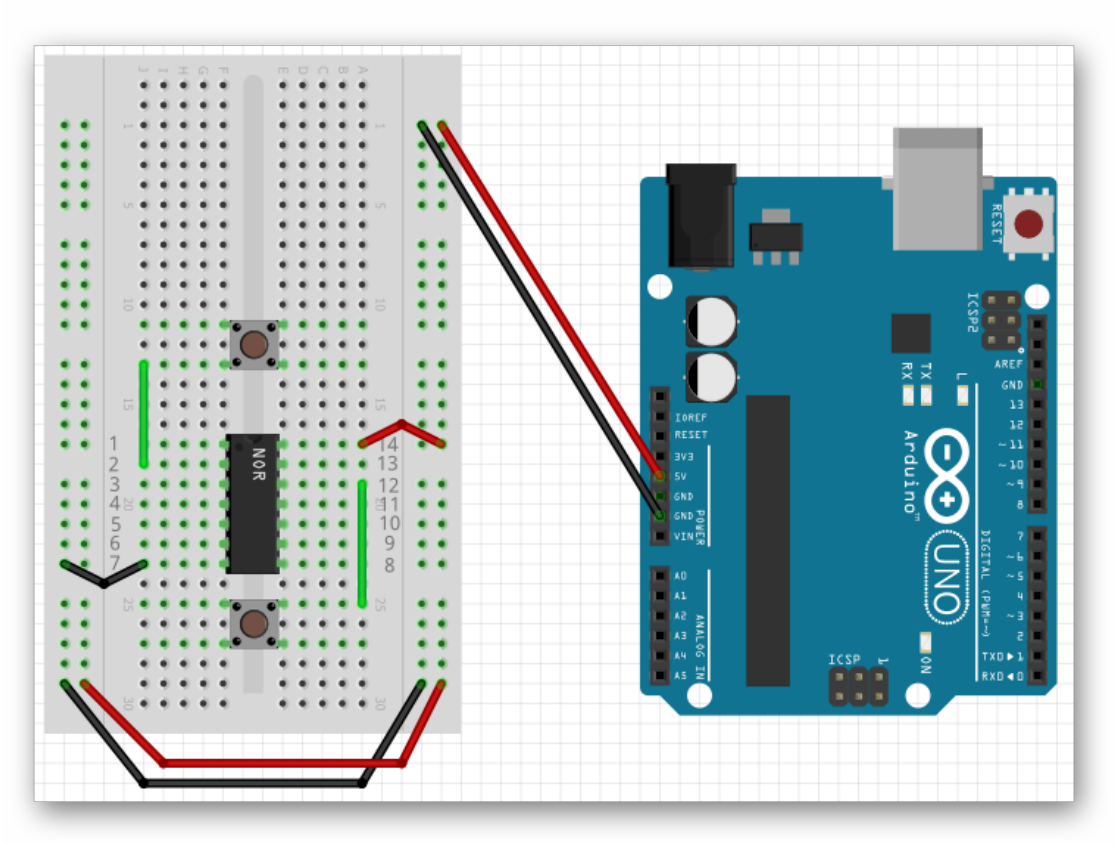
So let's attempt to create one of these circuits ourselves. Once again, let's begin with the placement of the chip on the breadboard;



And we will always need to connect both power and ground.

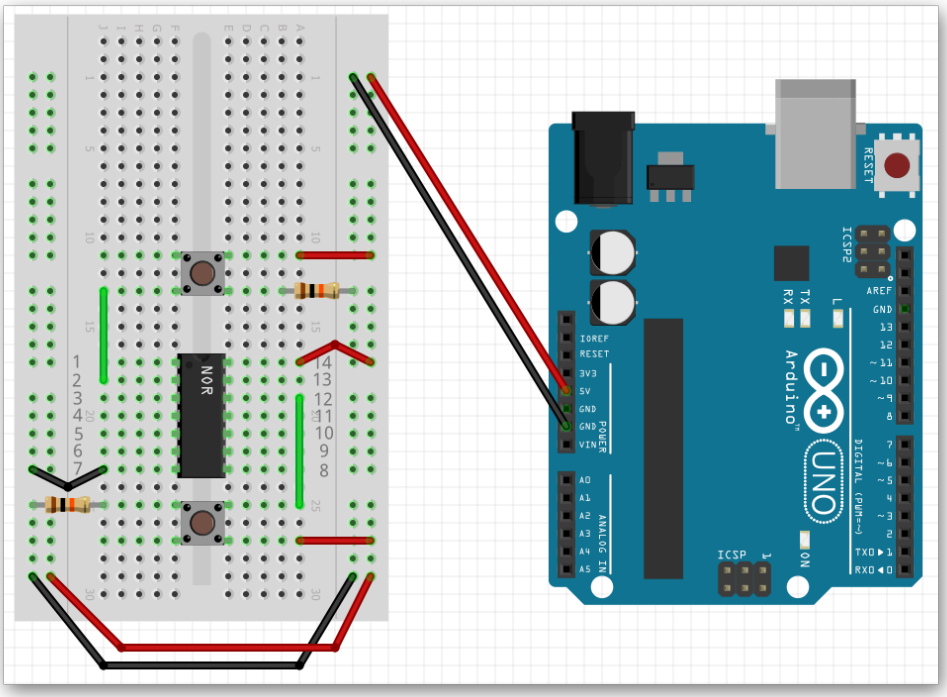


Next we need to decide which two NOR gates we are going to use. I'm going to use the 1,2,3 and 11,12,13 gates in the following illustrations. I'm going to start by attaching two inputs to buttons, one from each gate;

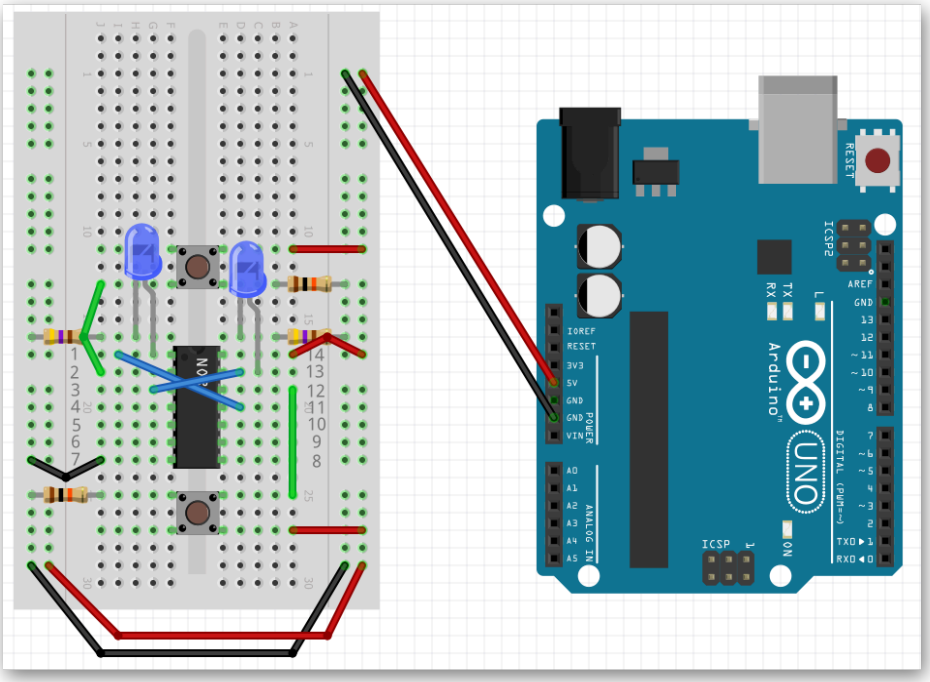


Notice that I've tried to keep things simple by connecting pin 2 and 12 on the arduino to the rows labelled 2 and 12.

Next we should wire up the buttons so they are functioning. Unlike in the past, when the default state of the button was HIGH, we need to wire our buttons this time so that the default state is LOW. This means we will wire a pull down resistor rather than a pull up resistor;

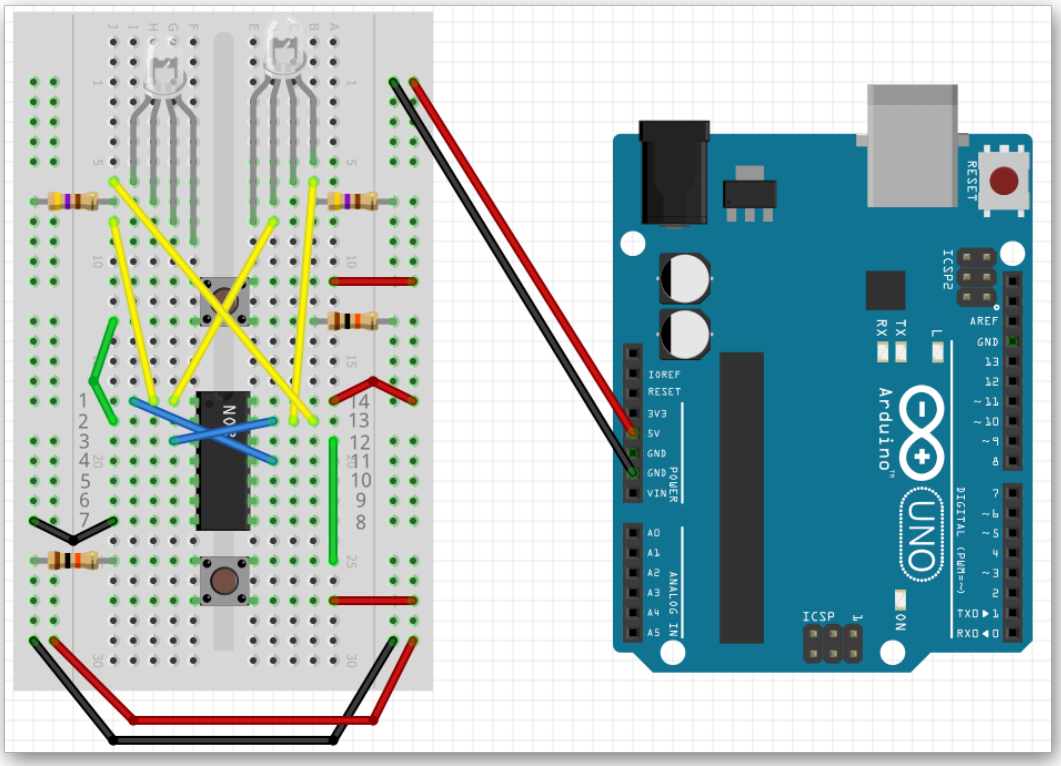


Lastly we have to tie the outputs and inputs of the two gates together and add the LEDs to the circuit;



In this configuration one button is acting as the set pin for one of the LEDs, but the reset pin for the other. So pressing one button will turn one LED on and the other off. However releasing the button will not cause the LEDs to revert back to their previous state, as the circuit is doing its job of remembering that it was on in the past. If you want to turn that LED off you will need to press the 'reset' button to get the desired effect. Remember that doing so will turn the other LED back on. The two LEDs are coupled at this point, and one cannot be effected without affecting the other.

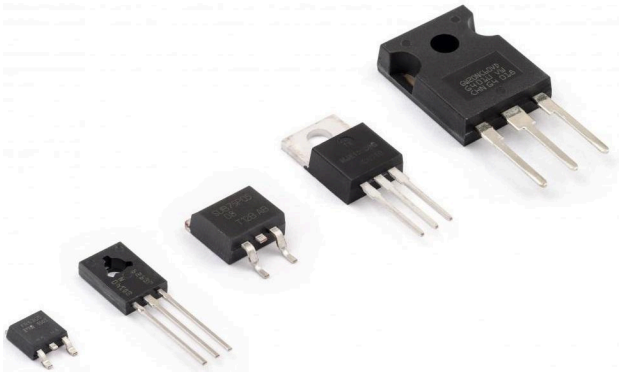
So we have built an SR latch, what can we do with it? Aside from being the basic building block of your computer, the logic inherent in the circuitry allows us to create some interesting things with this circuit, if we are clever enough. I'm going to utilize the RGB LED that we learned about much earlier in the class to create a simple stop light system for a four way intersection. All I'm going to do is replace the LEDs currently in the circuit and replace them with RGBs. I am going to connect both the green and red pins of the LEDs to the circuit, essentially putting four LEDs in the circuit. I'm also going to wire them in a way that ensures they always have the opposite behavior;



I want to point out that each NOR gate output is wired to different colors on the two LEDs. So pin one of the IC is wired to the green pin on one LED and the red pin on the other. Pin 13 of the IC is wired to the red and green pins left. Don't worry if it's not immediately obvious which pins of the LEDs need to be wired to the mentioned IC pins. This is a good opportunity to

experiment, as long as it is well understood which pin is GND the correct colors can be found through trial and error.

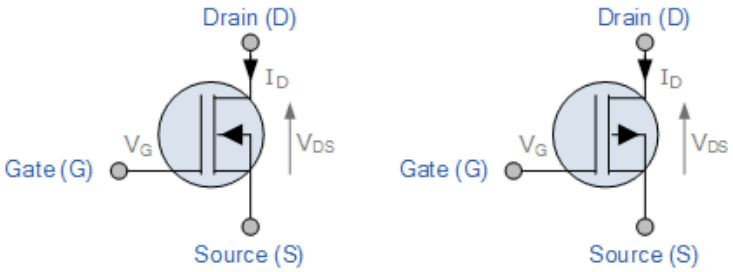
Mosfets



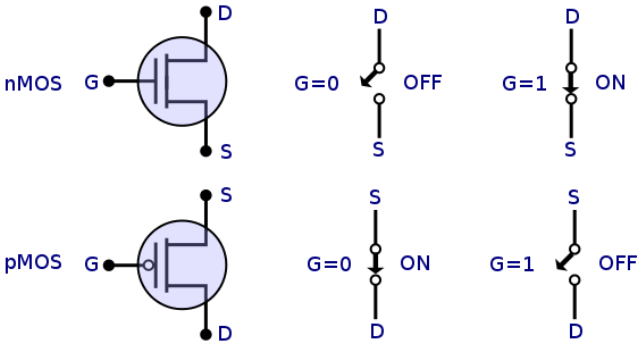
Mosfets are a type of electrical component called transistors. In all honesty, the solid state physics of how these things are able to do the things they do are a little beyond the scope of this class. We will however, be able to understand how they function in circuits by the end of this lesson. This opens the door to so many things in electrical engineering because mosfets allow circuits to perform basic logic. It is because of these components that our computers are able to “think”.

I want you to think back to our lesson about the button. The button had two leads, and current would flow through one to the other. It would only do so however, if the button was pressed. Imagine that instead of using our finger to allow current to pass we gave the button a third lead, and let an electrical signal ‘press’ a similar ‘button’ that lies inside the component. If you did did that, you would have a **mosfet**.

A mosfet will let current pass through the outer two leads only if the inner lead receives the correct signal, either high or low depending on the type of mosfet. Mosfets are split into P-channel and N-channel mosfets. P- channel mosfets will allow current to flow while the middle lead (known as the gate) is at low voltage, while N-channel mosfets will only allow current to flow if the gate is at high voltage. It turns out the middle lead is not the only lead with a name. In fact all of the leads of mosfets have names. Here is a schematic illustration to show you;

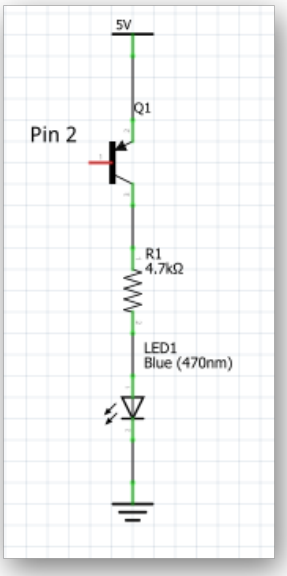


As you can see both the P-channel and N-channel mosfets have the same naming conventions, so you won't have to memorize different names for each. However the direction current flows is not the same between the two. Hopefully the illustration below can help us understand more thoroughly;

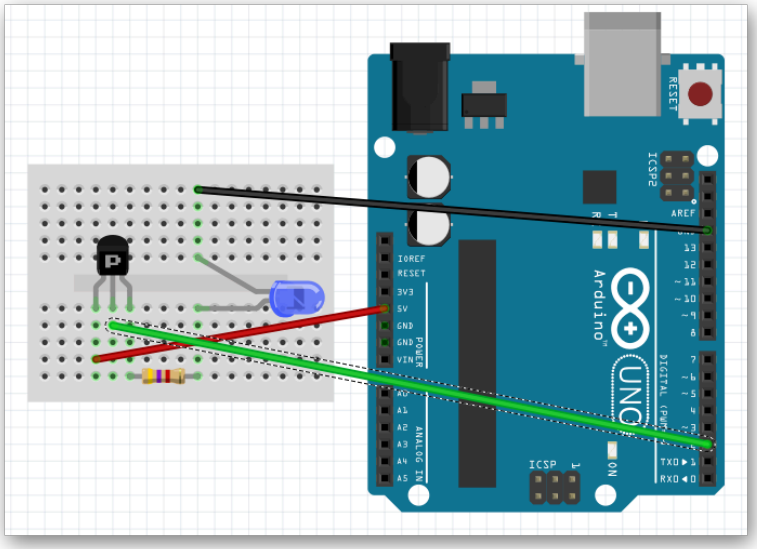


The first thing I want to address about the above illustration is the G=1 and G=0 segments. Those are just referring to the gate having a high voltage (G=1) or the gate having a low voltage (G=0). This picture kindly shows us when each type of mosfet is on and off with respect to the gate voltage. You'll also notice that current is not flowing in the same direction for both types of mosfet. The N-channel mosfet has current flowing from the drain to the source while the P-channel mosfet has current flowing from the source to the drain.

Now comes the challenge. Each of your students have been given four mosfets. Their assignment is to figure out which ar N-channel and which are P-channel, as well as which leg is the drain and which is the source (it is already assumed the middle leg is the gate). Along with those instructions you should give them this testing circuit schematic;



As always, here is a complementary illustration of the built circuit;

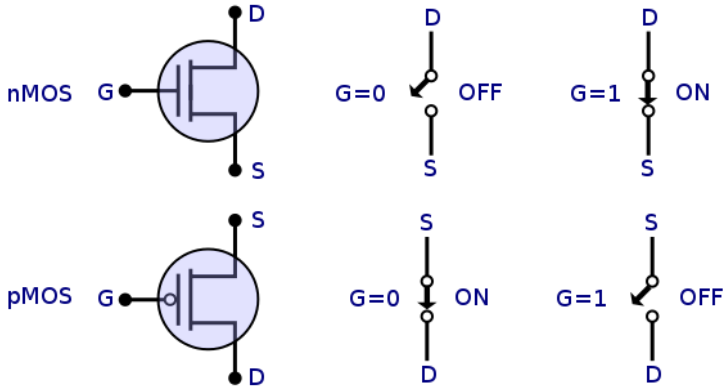


You'll notice that this circuit contains a P-channel mosfet, but you will not necessarily know which kind of mosfet you have in the circuit at first.

You can also see that the gate of the mosfet is being controlled by pin 2 of the Arduino. This is just for convenience so that I can change the the gate from high to low without changing the circuit. At this point I assume that setting a single pin to high or low in Arduino or Ardublock is a trivial exercise so I will not go into detail on how to do so here.

Some tips for identifying the mosfets:

- Use this picture;



- If the LED is not lighting up consider switching the orientation of the mosfet on the breadboard so that the gate lies in the same place but the outer two legs change places.
- If the LED is still not lighting up change the state of pin 2, either from low to high, or from high to low. Just the opposite of what is was originally.
- If it's still not working switch the orientation of the mosfet again, back to its original position.

The most important thing is to note the state of pin 2 when the LED finally turns on. If it is LOW, we know that it is a P-channel mosfet. If it is high, we know that it is a N-channel mosfet.

After that we need to decide which leg is the source and which is the drain. Once again I must refer to the above illustration to figure this out. If I have already found that it is a N-channel, then I know that the leg attached to 5V is the drain, do to how current is flowing in the illustration (look at the arrow). In the P-channel mosfet, it's the opposite. The 5V leg is the source in that case.

This brings us to the end of this week's lesson.

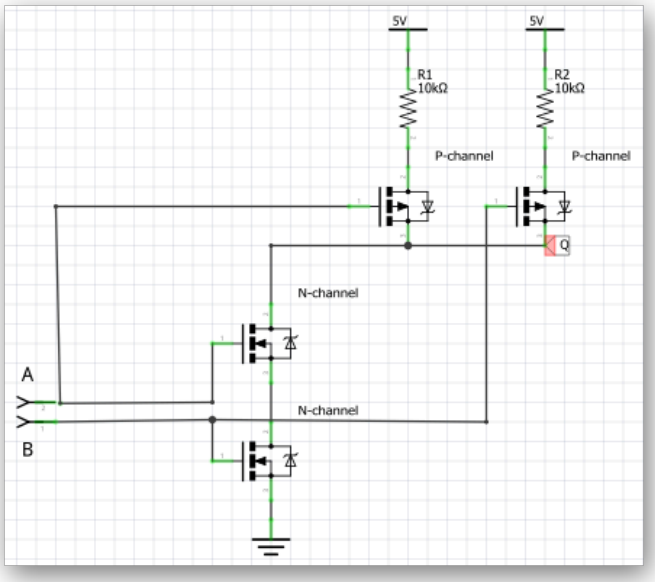
Continued Mosfets and Logic Gates

We now have some working knowledge of mosfets as well as logic gates, but you'll notice that we have yet to touch a logic gate. Today we are going to try to remedy this by creating our own logic gate.

The reason for having the last two lessons back to back is that logic gates and mosfets are closely related to one another.Mosfets are in fact the building blocks that make up logic gates. Several mosfets are put into a circuit together in some way to mimic the truth table logic of a particular logic gate.

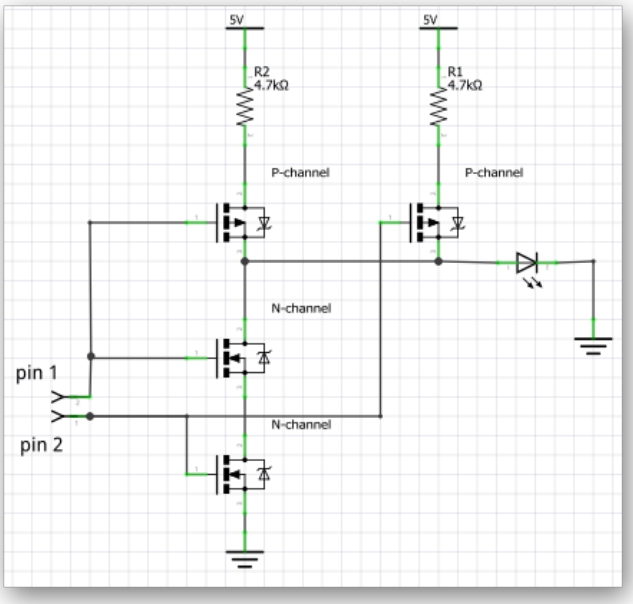
I want to create a NAND gate as today's lesson. As we learned during last weeks exercises NAND gates can make up any other logic gate if arrayed correctly. So, by learning how to make a NAND gate from mosfets we can then construct any logic gate we want from mosfet building blocks.

The general schematic for a NAND gate looks like this;

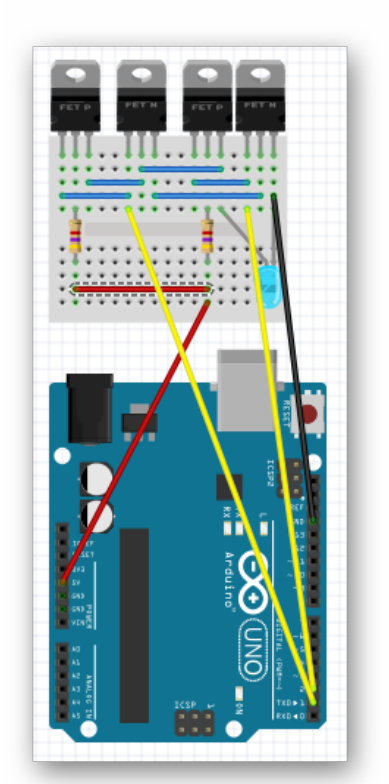


You can see where our two inputs A and B are as well as the output Q. Don't worry too much about the schematic representation of the mosfets, just remember that one pin acts like a switch, turning the mosfet on or off, and the two others are connected when the channel is on, allowing current to flow through the mosfet.

We are going to add an LED to this circuit as well as connect it to an Arduino so that we have control over the two inputs as well as verify that the logic of whatever we have made here does in fact match a NAND gate. We can do that by altering the schematic thusly;



As always I recommend letting your students attempt to construct this circuit straight from this schematic before showing the breadboard illustration. That being said constructing this circuit from the schematic is no easy task, although not impossible either. The key would be remembering which pins on the mosfet are the gate, drain, and source respectively. Of course the breadboard view is provided below for clarity;



Using this circuit I want the students to verify that this is in fact a NAND gate. How are we going to do that? Let's think for a minute. What do we know about NAND gates? Well the only thing we really know about NAND gates is their truth table logic;

A	B	Q
0	0	1
0	1	1
1	0	1
1	1	0

The only way we can compare our circuit to a standard NAND gate is to test it and see if it's truth table is the same as a NAND gate's truth table. You will assign pins 1 and 2 the values of high and low (which relate to the values of 1 and 0 in the table above) in a few different sequences and see if each sequence turns the LED on or off (which also relate to the values of 1 and 0 in the table above).

Start by giving the students the table below;

Pin 1	Pin 2	LED
LOW	LOW	?
HIGH	LOW	?
LOW	HIGH	?
HIGH	HIGH	?

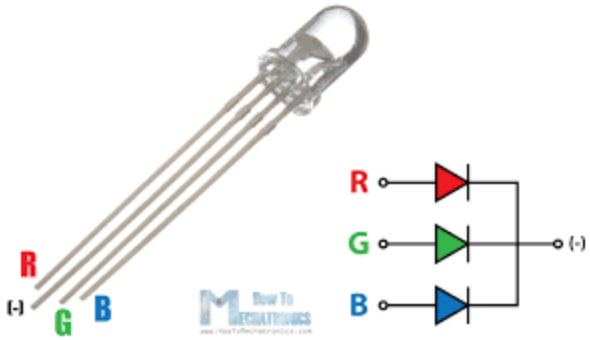
All that the students need to do is check if the LED is on or off in each case. Afterwards they will refer back to the previous table and compare the Q column to the LED column. The ons should match up with the 1s and the offs should match up with the 0s.

To accomplish this the students need to set both pin 1 and pin 2 to their correct states (either HIGH or LOW) by using two set digital pin blocks or set digital pin commands. That is all that is needed. So the students will set both pins LOW, check the state of the LED, then set one HIGH and the other LOW, check the state of the LED, and so on through each row of the table.

When the students have confirmed that this is in fact a NAND gate tell them to switch the places of the N-channel and P-channel mosfets in their circuits. Then have them attempt to make a truth table for this new circuit and challenge them to figure out what kind of logic gate this circuit is. (It should make an or gate)

Appendix

RGB LED Resources



Color Chart	R	G	B	Color Name
	0	0	0	Black
	255	255	255	White
	224	224	224	Light Gray
	128	128	128	Gray
	64	64	64	Dark Gray
	255	0	0	Red
	255	96	208	Pink
	160	32	255	Purple
	80	208	255	Light Blue
	0	32	255	Blue
	96	255	128	Yellow-Green
	0	192	0	Green
	255	224	32	Yellow
	255	160	16	Orange
	160	128	96	Brown
	255	208	160	Pale Pink

DC Motor Activity

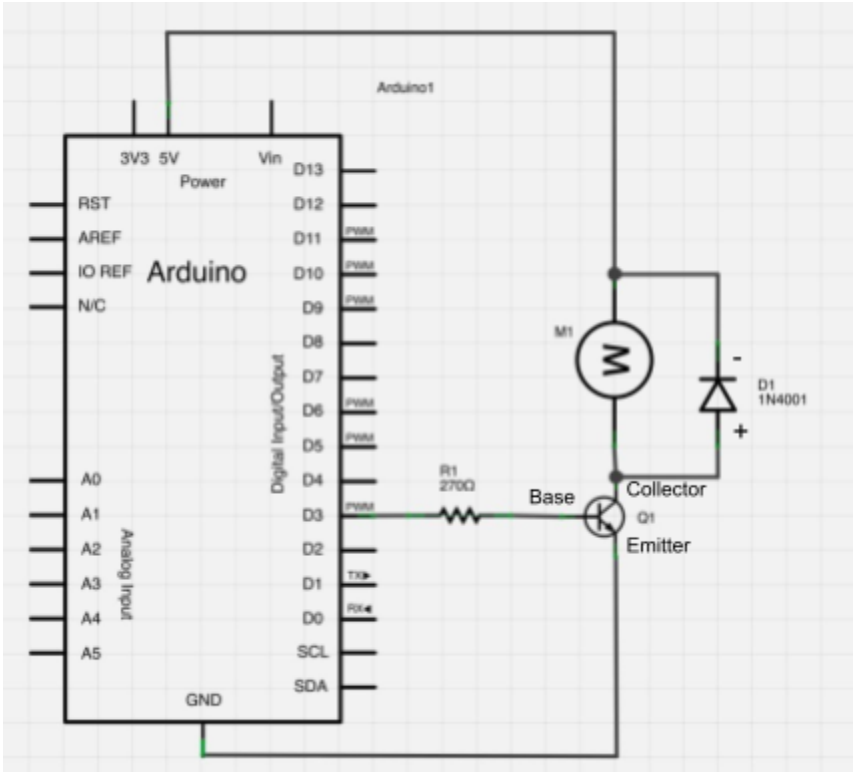
Materials needed

1. 130 Sized DC Motor with flying leads. Make sure that they are able to be plugged into a breadboard
2. PN2222 Transistor

3. 1N4001 diode
4. 270 Ω Resistor (red, purple, brown stripes)

NPN Transistor Pin out

Schematic View



Breadboard View (from Adafruit)