SeaGlide Arduino Kit:

An Introduction to Circuits, Code, and Physical Computing

Attribution – Thank you to Arduino and SparkFun for open source access to reference materials.

Contents

Parts Reference

Jumper Wires

Figure 1: Various sizes and colors of jumper wires are used to make solderless electrical connections.

Light Emitting Diodes (LEDs)

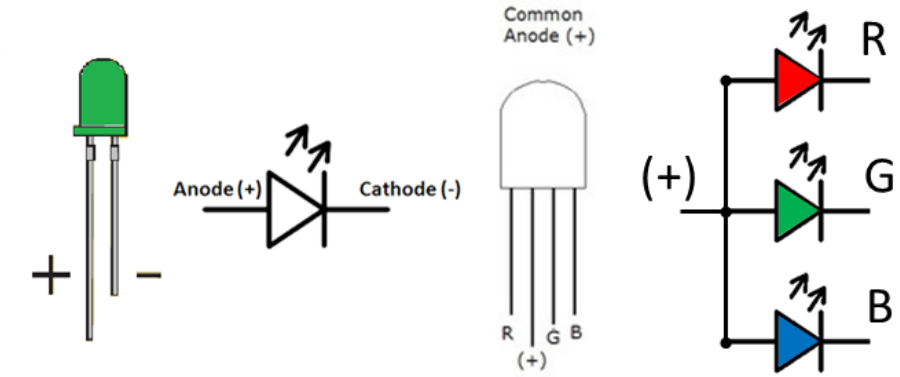


Figure 2: Green LED (left) and RGB LED (right).

470Ω and 10KΩ Resistors

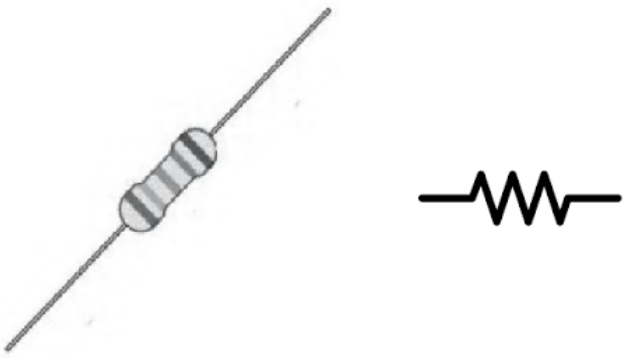


Figure 3: Resistor pictorial (left) and circuit diagram symbol (right).

Potentiometer

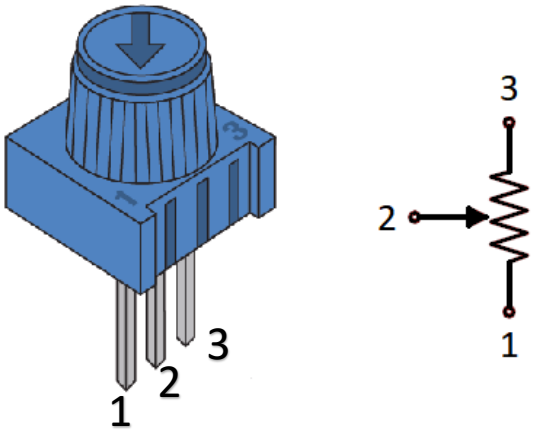


Figure 4: Potentiometer pictorial (left) and circuit diagram symbols (right).

Photoresistor / Photocell

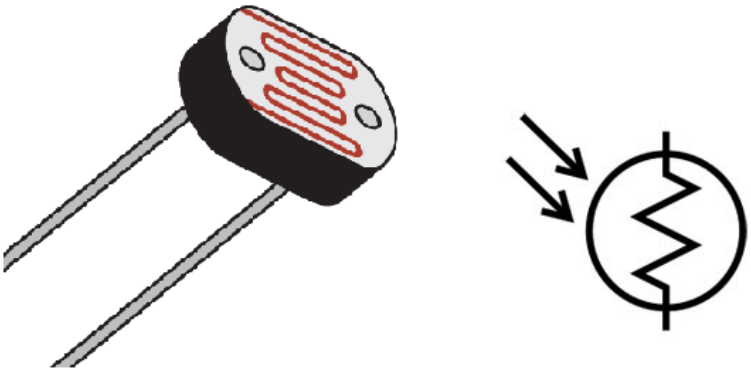


Figure 5: Photoresistor pictorial (left) and circuit diagram symbol (right).

Normally Open (NO) Push Button

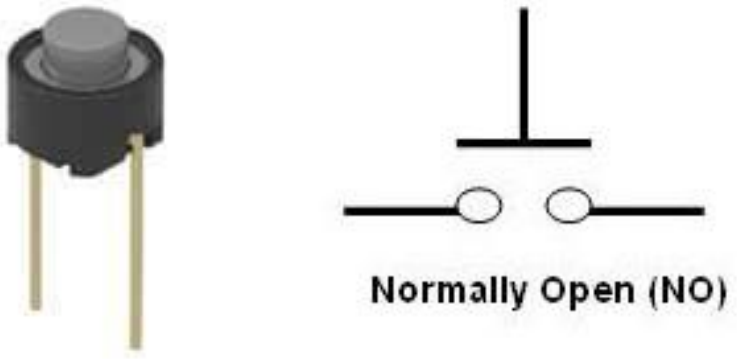


Figure 6: Push button pictorial (left) and circuit diagram symbol (right).

Normally Open (NO) Reed Switch

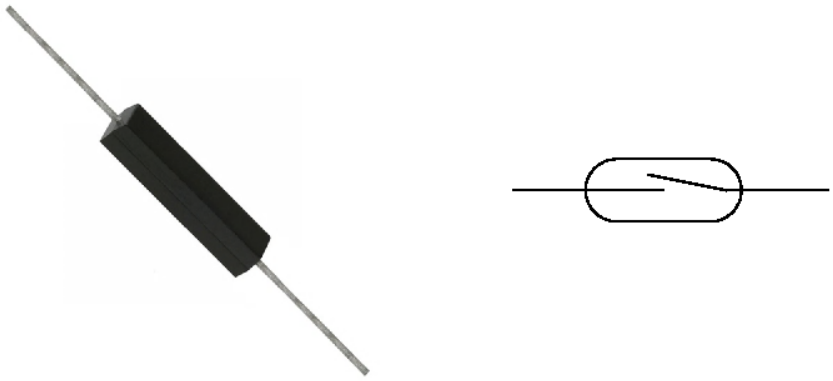


Figure 7: Reed switch pictorial (left) and circuit diagram symbol (right).

Servo Motor

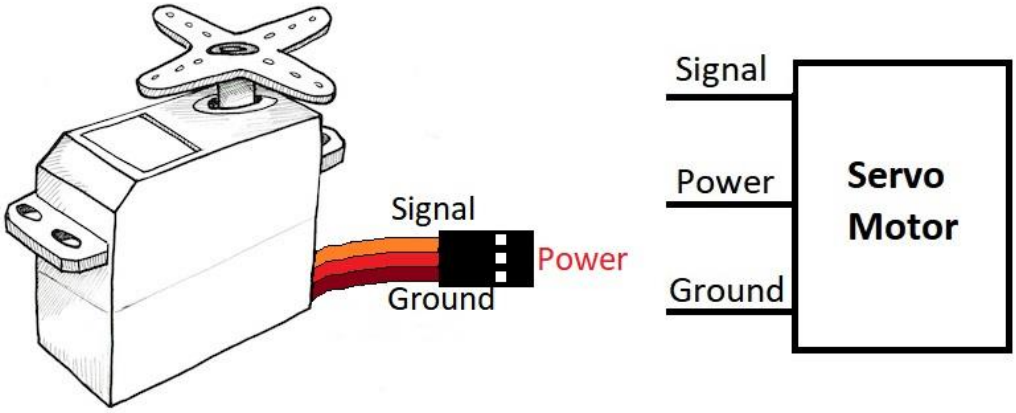


Figure 8: Servo motor pictorial (left) and circuit diagram symbol (right).

Mini USB Cable

Figure 9: Mini USB cable for programming and powering microcontroller.

Arduino Nano Microcontroller

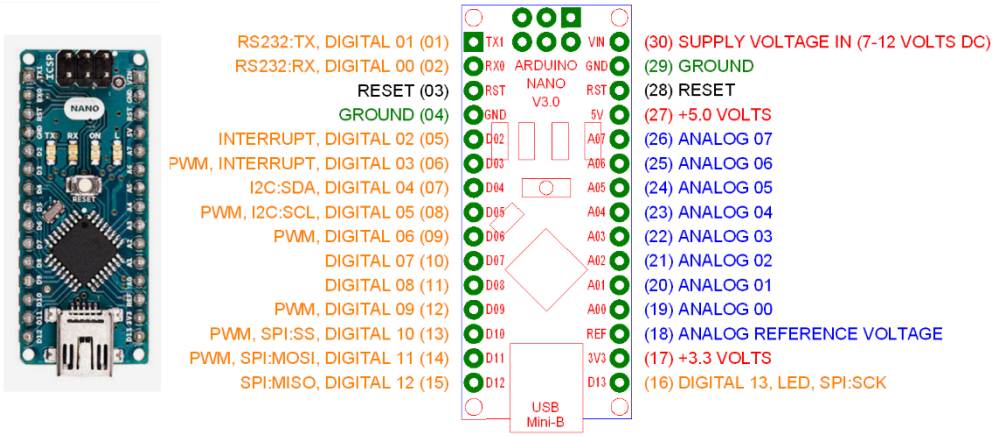


Figure 10: Arduino Nano (left) and pinout (right).

Solderless Breadboard

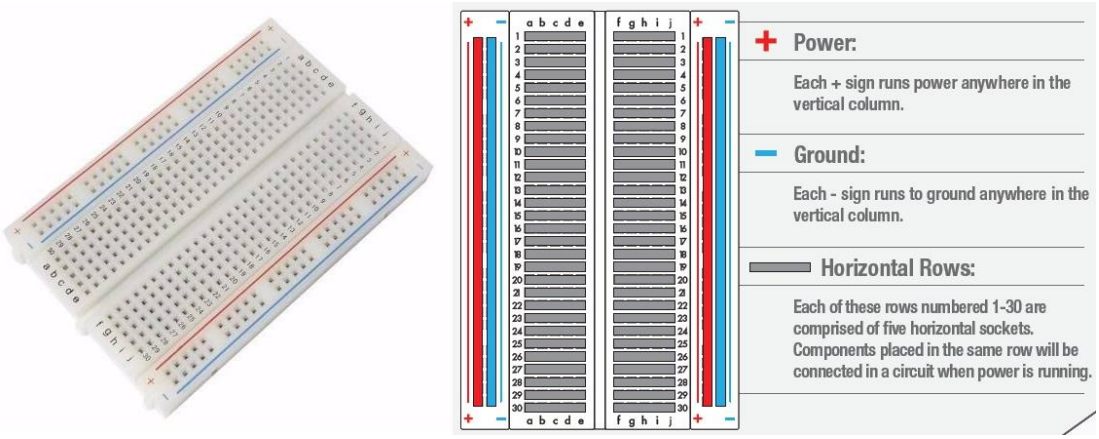
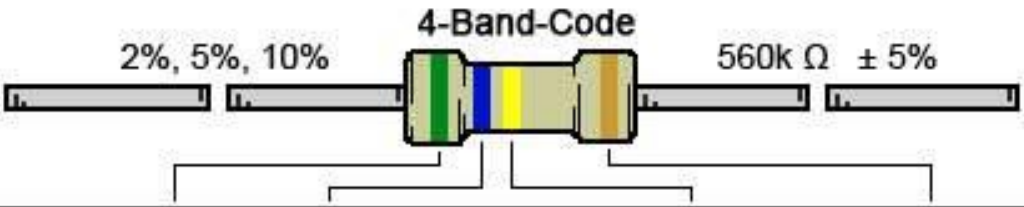


Figure 11: Solderless breadboard (left) and connection layout (right).

Multimeter

Figure 12: Multimeters can measure voltage, resistance, continuity, capacitance, and amperage.



COLOR	1ST BAND	2ND BAND	3RD BAND	MULTIPLIER	TOLERANCE
Black	0	0	0	1Ω	
Brown	1	1	1	10Ω	± 1% (F)
Red	2	2	2	100Ω	± 2% (G)
Orange	3	3	3	1KΩ	
Yellow	4	4	4	10KΩ	
Green	5	5	5	100KΩ	± 0.5% (D)
Blue	6	6	6	1MΩ	± 0.25% (C)
Violet	7	7	7	10MΩ	± 0.10% (B)
Grey	8	8	8		± 0.05%
White	9	9	9		
Gold				0.1Ω	± 5% (J)
Silver				0.01Ω	± 10% (K)

Figure 13 Resistor color band calculator

Arduino Programming Cheat Sheet

Primary source: Arduino Language Reference
http://arduino.cc/en/Reference/

Structure & Flow

Basic Program Structure

```
void setup() {
  // Runs once when sketch starts
}
void loop() {
  // Runs repeatedly
}
```

Control Structures

```
if (x < 5) { ... } else { ... }
while (x < 5) { ... }
for (int i = 0; i < 10; i++) { ... }
break;     // Exit a loop immediately
continue;  // Go to next iteration
switch (var) {
  case 1:
    ...
    break;
  case 2:
    ...
    break;
  default:
    ...
}
return x;  // x must match return type
return;    // For void return type
```

Function Definitions

```
<ret. type> <name>(<params>) { ... }
e.g. int double(int x) {return x*2;}
```

Operators

General Operators

=	assignment		
+	add	-	subtract
*	multiply	/	divide
%	modulo		
==	equal to	!=	not equal to
<	less than	>	greater than
<=	less than or equal to		
>=	greater than or equal to		
&&	and	||	or
!	not		

Compound Operators

++	increment
--	decrement
+=	compound addition
-=	compound subtraction
*=	compound multiplication
/=	compound division
&=	compound bitwise and
|=	compound bitwise or

Bitwise Operators

&	bitwise and	|	bitwise or
^	bitwise xor	~	bitwise not
<<	shift left	>>	shift right

Pointer Access

&	reference: get a pointer
*	dereference: follow a pointer

Variables, Arrays, and Data

Data Types

boolean	true | false
char	-128 - 127, 'a' '$' etc.
unsigned char	0 - 255
byte	0 - 255
int	-32768 - 32767
unsigned int	0 - 65535
word	0 - 65535
long	-2147483648 - 2147483647
unsigned long	0 - 4294967295
float	-3.4028e+38 - 3.4028e+38
double	currently same as float
void	i.e., no return value

Strings

```
char str1[8] =
  {'A','r','d','u','i','n','o','\0'};
  // Includes \0 null termination
char str2[8] =
  {'A','r','d','u','i','n','o'};
  // Compiler adds null termination
char str3[] = "Arduino";
char str4[8] = "Arduino";
```

Numeric Constants

123	decimal
0b01111011	binary
0173	octal - base 8
0x7B	hexadecimal - base 16
123U	force unsigned
123L	force long
123UL	force unsigned long
123.0	force floating point
1.23e6	1.23*10^6 = 1230000

Qualifiers

static	persists between calls
volatile	in RAM (nice for ISR)
const	read-only
PROGMEM	in flash

Arrays

```
int myPins[] = {2, 4, 8, 3, 6};
int myInts[6];   // Array of 6 ints
myInts[0] = 42;  // Assigning first
                 // index of myInts
myInts[6] = 12;  // ERROR! Indexes
                 // are 0 though 5
```

Built-in Functions

Pin Input/Output

```
Digital I/O - pins 0-13 A0-A5
  pinMode(pin,
    [INPUT, OUTPUT, INPUT_PULLUP])
  int digitalread(pin)
  digitalWrite(pin, [HIGH, LOW])

Analog In - pins A0-A5
  int analogRead(pin)
  analogReference(
    [DEFAULT, INTERNAL, EXTERNAL])

PWM Out - pins 3 5 6 9 10 11
  analogWrite(pin, value)
```

Advanced I/O

```
tone(pin, freq_Hz)
tone(pin, freq_Hz, duration_ms)
noTone(pin)
shiftOut(dataPin, clockPin,
  [MSBFIRST, LSBFIRST], value)
unsigned long pulseIn(pin,
  [HIGH, LOW])
```

Time

```
unsigned long millis()
  // Overflows at 50 days
unsigned long micros()
  // Overflows at 70 minutes
delay(msec)
delayMicroseconds(usec)
```

Math

```
min(x, y)   max(x, y)   abs(x)
sin(rad)    cos(rad)    tan(rad)
sqrt(x)     pow(base, exponent)
constrain(x, minval, maxval)
map(val, fromL, fromH, toL, toH)
```

Random Numbers

```
randomSeed(seed) // long or int
long random(max) // 0 to max-1
long random(min, max)
```

Bits and Bytes

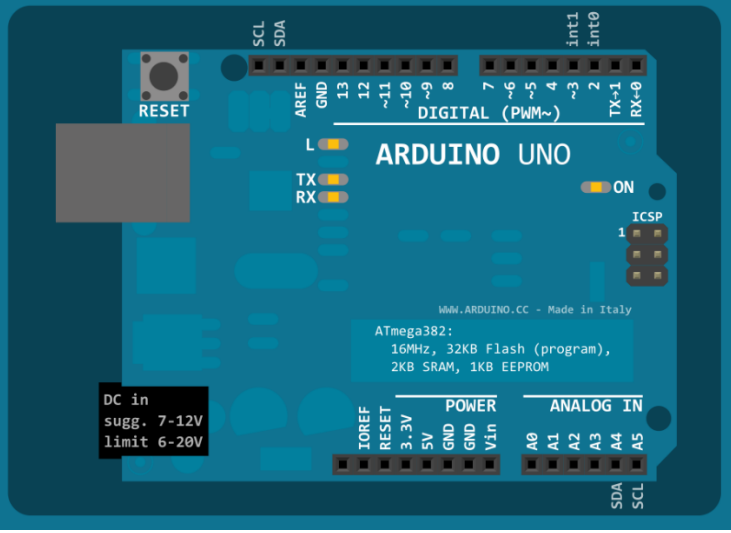
```
lowByte(x)    highByte(x)
bitRead(x, bitn)
bitWrite(x, bitn, bit)
bitSet(x, bitn)
bitClear(x, bitn)
bit(bitn)  // bitn: 0=LSB 7=MSB
```

Type Conversions

```
char(val)     byte(val)
int(val)      word(val)
long(val)     float(val)
```

External Interrupts

```
attachInterrupt(interrupt, func,
 [LOW, CHANGE, RISING, FALLING])
detachInterrupt(interrupt)
interrupts()
noInterrupts()
```



Libraries

Serial - comm. with PC or via RX/TX

```
begin(long speed)  // Up to 115200
end()
int available()  // #bytes available
int read()   // -1 if none available
int peek()   // Read w/o removing
flush()
print(data)    println(data)
write(byte)    write(char * string)
write(byte * data, size)
SerialEvent()  // Called if data rdy
```

SoftwareSerial.h - comm. on any pin

```
SoftwareSerial(rxPin, txPin)
begin(long speed)  // Up to 115200
listen()       // Only 1 can listen
isListening()  // at a time.
read, peek, print, println, write
  // Equivalent to Serial library
```

EEPROM.h - access non-volatile memory

```
byte read(addr)
write(addr, byte)
EEPROM[index]  // Access as array
```

Servo.h - control servo motors

```
attach(pin, [min_uS, max_uS])
write(angle)  // 0 to 180
writeMicroseconds(uS)
   // 1000-2000; 1500 is midpoint
int read()    // 0 to 180
bool attached()
detach()
```

Wire.h - I²C communication

```
begin()     // Join a master
begin(addr) // Join a slave @ addr
requestFrom(address, count)
beginTransmission(addr) // Step 1
send(byte)              // Step 2
send(char * string)
send(byte * data, size)
endTransmission()       // Step 3
int available() // #bytes available
byte receive()  // Get next byte
onReceive(handler)
onRequest(handler)
```

Installing Arduino

Instructions for installing and configuring the software you'll need to load programs onto your Arduino can be found at www.seaglide.net/firmware. The first step is a link to download and install the Arduino Integrated Development Environment (IDE) found in the *software* page of www.arduino.cc.

The SparkFun Inventor's Kit Manual also has instructions for how to download the Arduino IDE as well as tutorials similar to what will be presented in this document. The manual can be found at https://cdn.sparkfun.com/datasheets/Kits/SIK/V33/SIK%203.3%20Manual.pdf.

Unit 1: *LEDs, Resistors, & Buttons*

1.1 Blink *(Hello World)*

Key Concepts

- Ohm's Law
- Polarity
- Digital I/O

Parts

- LED
- 470Ω & 10KΩ resistors
- Jumper Wires
- Breadboard
- Arduino Nano
- Mini USB cable

Electricity is the movement of electrons. The number of electrons determines charge. The three "building blocks" of electricity are voltage (V), current (I), and resistance (R). Voltage quantifies the difference in charge between two points. Current is the rate at which charge flows. Resistance quantifies opposition to charge flow. An analogy to water is often used to explain electricity in more familiar terms, where: charge is volume, voltage is pressure, current is flow rate, and resistance is a flow restrictor. Ohm's Law governs the relationship between voltage, current, and resistance as:

$$Voltage = Current * Resistance$$

To start, try making the simple electrical circuit show in Figure 14. This circuit delivers power to an LED with just a battery and a 470Ω resistor. Replace the 470Ω resistor in your circuit with a 10kΩ resistor. The LED dims because the larger resistor is limiting current to the LED. Without a resistor, the LED could draw too much current and damage itself.

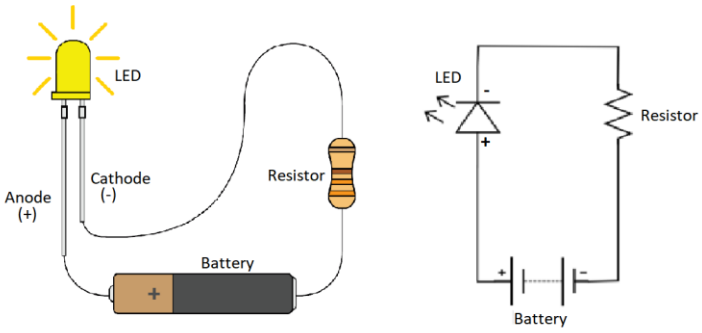
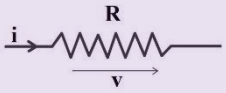


Figure 14: Light up an LED using a battery and a resistor pictorial (left) and circuit diagram (right).

❖ Exercise – Use Ohm's Law to calculate the difference in current delivered by the two resistors.



Circuit components that have positive and negative labels, like the battery and LED, have polarity; the orientation of these polarized components in a circuit is very important. Reversing the direction of a battery's terminals can ruin sensitive electrical components in a circuit. An LED is a light emitting diode, so like other diodes, it only allows current to flow in one direction. Diodes are useful in protecting other polarized parts that could be damaged by reverse voltage. Components that do not have polarity, like the resistor, are not required to be in a specific orientation. Now, replace the battery in the above circuit with +5V and GND (ground) from the microcontroller, as shown in Figure 16. The LED should be constantly on and powered by the microcontroller which is regulating the voltage from your computer's USB port. Look back at the breadboard connection layout in the Parts Reference section and check connectivity with a multimeter if needed.

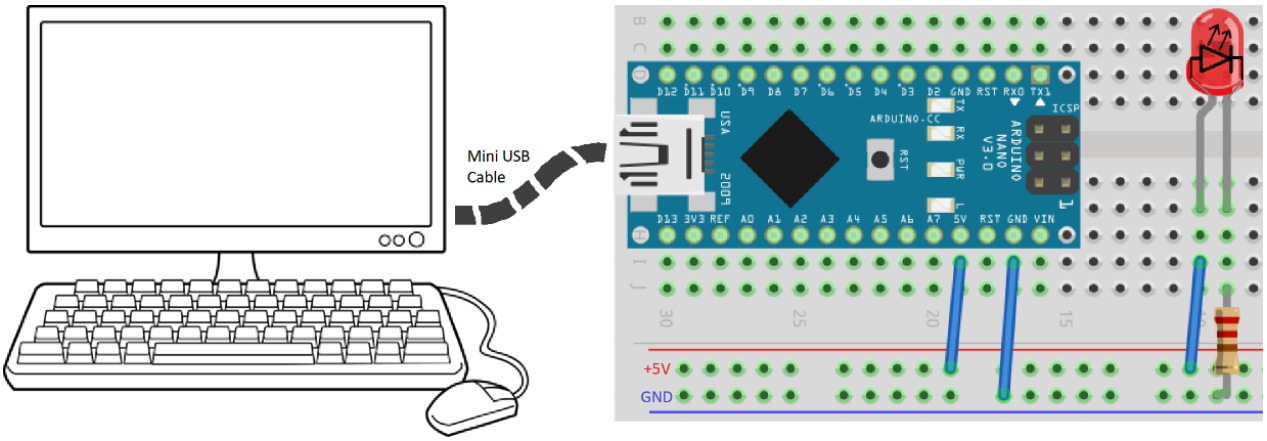


Figure 15: Use a microcontroller as the power supply to light up an LED.

Next, use a microcontroller to make the LED blink. Digital input/output (I/O) pins can "read" or "write" High or Low logic values which mean +5V or 0V from ground (GND) for the Arduino Nano. Figure 16 contains the layout diagram (made using the open source program Fritzing) of how to connect the electrical circuit as well as the circuit diagram to power an LED with the microcontroller.

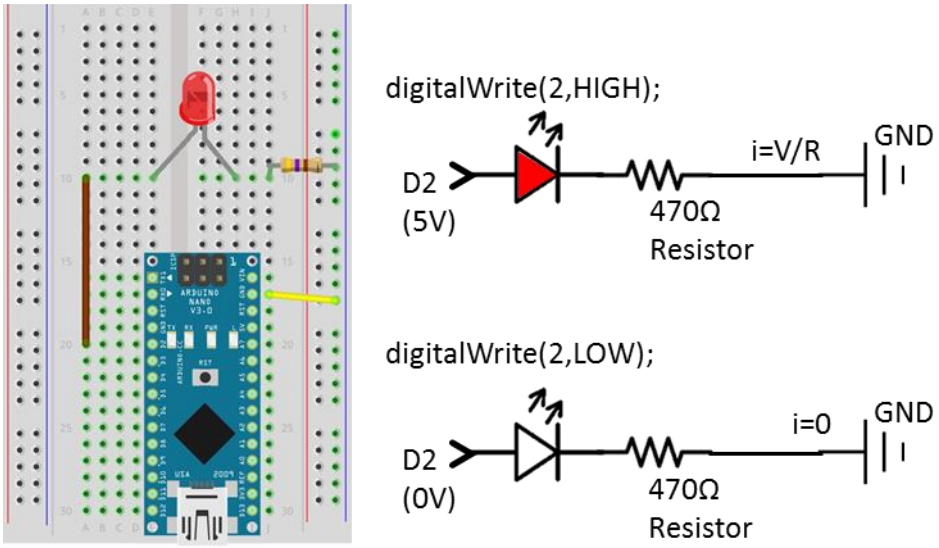


Figure 16: Blink layout diagram (left), blink circuit diagram (right).

Figure 17 contains code to program the Arduino Nano. Connect the Nano to a computer using a mini USB cable. Make sure to select the proper board and port in the "Tools" drop-down menu.

```
// This is a one-line comment

// Declare constants and variables for your program here
const int redLED = 2;           // create a new constant of type "integer" called redLED and assign it a pin number
const int nanoLED = 13;         // digital pin 13 is internally connected to an LED on the Arduino

void setup() {                  // The code inside this setup method will run once when the arduino starts up
  pinMode(redLED, OUTPUT);      // initialize digital pin "redLED" as an output so you can turn the LED on and off
  pinMode(nanoLED, OUTPUT);
}                               // the } marks the end of the setup loop closing the { which marks the beginning

void loop() {                   // This is the main method, the code inside here { will run repeatedly until the ardu
  digitalWrite(redLED, HIGH);   // turn the LED on (HIGH is the voltage level)  <----
  digitalWrite(nanoLED, LOW);   // Blink the Arduino LED opposite the external LED   |
  delay(1000);                  // wait for 1 second (1000 milliseconds)             |
  digitalWrite(redLED, LOW);    // turn the LED off by making the voltage LOW        |
  digitalWrite(nanoLED, HIGH);  //                                                   |
  delay(1000);                  // wait for 1 second                                 |
}                               // end main method, repeat to top --------------------
```

Figure 17: Blink Arduino Code.

Use device manager to check port if multiple options are available. Verify that the code compiles by clicking on the check mark button in the Arduino IDE's top left corner. Upload the code to your Arduino by clicking the arrow button to the right of the check mark ().
Are the LED on the breadboard and the LED on the Arduino blinking opposite one another every second? If not, it is time to start troubleshooting! Check electrical connections on the breadboard – use a multimeter if necessary. The "Help" menu in Arduino has an offline code reference with explanations and examples for all of Arduino's built-in functions, including: setup and loop; HIGH, LOW, and OUTPUT; and pinMode and digitalWrite.

❖ Exercise – Modify the code so that one (or both) of the LEDs blink S.O.S.

1.2 Button

Key Concepts

- Floating Voltages
- Pull-up/ pull-down resistors
- If…Else Conditional Statement
- Frequency

Parts

- NO Push Button
- LED
- 470Ω & 10kΩ resistors
- Jumper Wires
- Breadboard

This lesson will teach you how to control power to an LED with a button, using the Arduino Nano. A simple input for a microcontroller is a button. Normally open (NO) push buttons (also called momentary switches) electrically disconnect a circuit unless pressed. This type of switch is like a keyboard key – it only conducts electricity while pressed. You can check continuity with a multimeter; when the button is pressed the multimeter should beep.
In lesson 1.1, the Arduino digital I/O pins were used as outputs that delivered +5V or 0V. These voltages represent the microcontroller's logic levels. When these digital logic pins are used as inputs, they can only read HIGH or LOW. But what happens if the circuit has 4V or 2.3V at the pin? Since there is no in-between state for digital logic pins, the Arduino has to pick high or low by comparing the pin voltage against an imperfect threshold, between 1.0V and 2.3V. If an input pin is left disconnected this it is subject to an unpredictable floating voltage. A floating voltage is often an unintentional electrical potential that comes from previous circuit activity or environmental static electricity, which can trigger false inputs.

Using a pull-up or pull-down resistor can eliminate floating voltages and make your circuits more reliable. A pull-up resistor is a highly restrictive path to +5V that pulls the input to HIGH when the lower-resistance path to ground (through the button) is not connected. Similarly, a pull-down resistor is a highly restrictive path to ground (0V). It might be easier to understand this concept by thinking about the resistors as spring-loaded doors; to open the door you must overcome spring tension, but then the door self-closes when you let it go instead of flopping around partially open. Figure 18 shows a 10k external pull-up resistor which pulls the voltage on Digital input 12 to HIGH unless the button is pressed. The Arduino has internal pull-up resistors that activate when enabled in code.

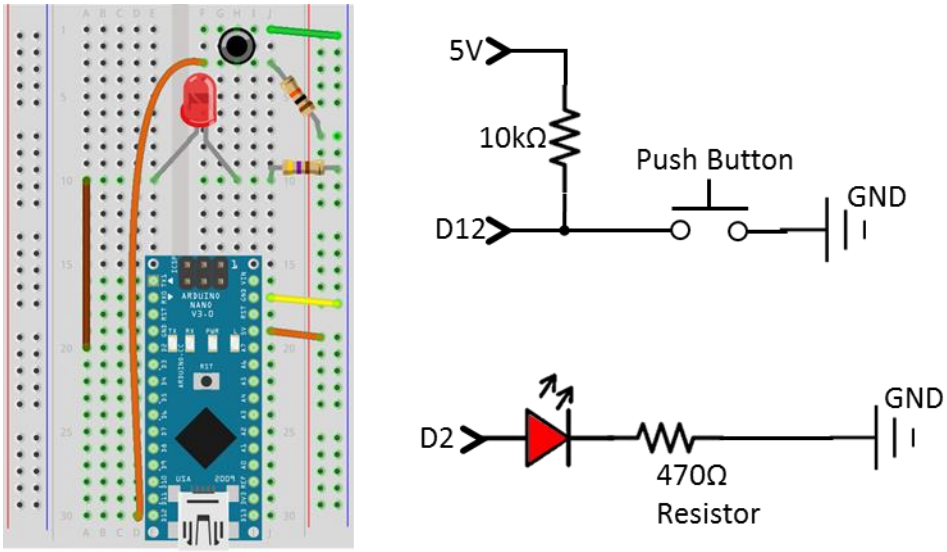


Figure 18: Button breadboard layout (left) and schematic (right).

Figure 19 shows the code associated with this lesson; it contains "if…else" conditional statements that control how the button input is used. In this lesson, when the button is not being pressed the input is read as HIGH (due to the pull-up resistor) and no power is given to the LED. The microcontroller registers a button press as a LOW logic level and powers on the LED.

- Exercise – Change the pull-up resistor to a pull-down resistor. With the existing code, explain why the LED stays on when the button is not pressed and turns off when it is pressed.

```
// Constants
const int buttonPin = 12;              // Button connected to pin 12
const int redLED = 2;                  // Red LED on pin 2

void setup() {                         // Start setup, this loop runs once at startup
  pinMode(redLED, OUTPUT);             // Initilize pin for output control
  pinMode(buttonPin, INPUT);           // Initialize pin as an input
}                                      // End setup

void loop() {                          // Start main loop <-------------------------------------
  if (digitalRead(buttonPin) == HIGH)  // Conditional statement: if (this) is true              |
  {                                    // then {code} will execute. if not it will be skipped   |
    digitalWrite(redLED, LOW);         // Turn the LED off by making the voltage LOW            |
  }                                    // End of the 'if' conditional statement                 |
  else                                 // The code in {} after 'else' will only run when        |
  {                                    // the previous 'if' statement is not true               |
    digitalWrite(redLED, HIGH);        // Turn the LED on (HIGH is the voltage level)           |
  }                                    // End 'else'                                            |
  delay(50);                           // Make the program run at a frequency of 20Hz           |
}                                      // End main method, repeat to top  ----------------------
```

Figure 19: Code for button lesson.

A delay statement at the end of the void loop can be used to run the code at a specific frequency, defining how many times the void loop cycles per second (units of Hertz, Hz). This can be helpful for data collection and post-processing.

> ❖ Exercise – Use the Arduino's internal pull-up resistor on pin 6 instead of the external resistor.
> In the Arduino IDE go to: *Help > Reference* then click *pinMode()*

1.3 Traffic Light

Key Concepts

- Global vs. Local Variables
- Boolean Data Type
- While loops

Parts

- Red, yellow, & green LEDs
- Button
- 470Ω & 10kΩ resistors
- Jumper Wires
- Breadboard
- Arduino Nano
- Mini USB cable

Some traffic lights use a combination of timing and sensors that can detect the presence of waiting traffic to regulate traffic flow. This lesson builds on previous key concepts to simulate this kind of traffic light. Wire up the circuit shown in Figure 20. Look over the code in Figure 21.

As in previous lessons, global constants are defined at the beginning so that they can be accessed in any part of the program.

Within the void loop function a local variable, bState, is created. Local variables stay within the function that they are created – it is possible to have local variables with the same name in different functions. The decision to make a variable or constant global versus local can be based on several factors, including memory management and unique name control.

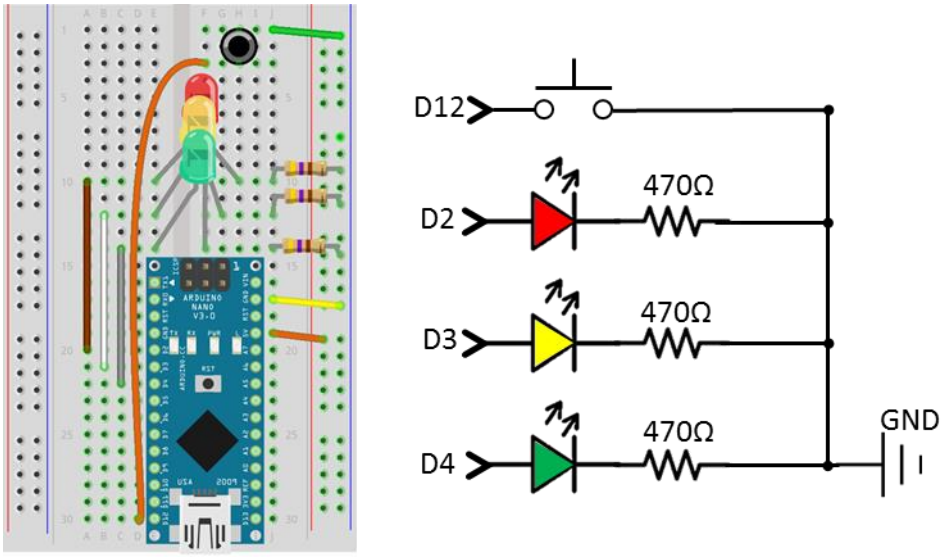


Figure 20: Traffic light layout diagram (right) and schematic (left).

The variable bState is defined to be a Boolean data type, meaning that it can either be "HIGH" (equivalent to "1" or "True") or "LOW" (equivalent to "0" or "False"). Assigning variables and constants the appropriate data type is important for memory management and precision.

```
// Declare constants and variables for your program here
const int buttonPin = 12;           // Button connected to pin 12
const int redLED = 2;              // red LED on pin 2
const int yellowLED = 3;           // yellow LED on pin 3
const int greenLED = 4;            // green LED on pin 4

void setup() {                     // the code inside this setup method will run once
  pinMode(redLED, OUTPUT);         // set redLED as an output
  pinMode(greenLED, OUTPUT);       // set greenLED as an output
  pinMode(yellowLED, OUTPUT);      // set yellowLED as an output
  pinMode(buttonPin,INPUT_PULLUP); // set buttonPin as input, activate internal pull-up resistor
}                                  // end setup

void loop() {                      // this is the main method, the code inside here repeats
  digitalWrite(greenLED, HIGH);    // turn on the green LED <--------------------------------
  delay(3000);                     // wait 3 seconds                                        |
  digitalWrite(greenLED, LOW);     // turn off the green LED                                |
  digitalWrite(yellowLED, HIGH);   // turn on the yellow LED                                |
  delay(1000);                     // wait for 1 second                                     |
  digitalWrite(yellowLED, LOW);    // turn off the yellow light                             |
  digitalWrite(redLED, HIGH);      // turn on the red LED                                   |
  delay(3000);                     // wait 1 green cycle for opposing traffic               |
  bool bState = HIGH;              // declare local boolean variable for button state       |
  while(bState){                       // while button is not-pressed(HIGH)--------)        |
    bState = digitalRead(buttonPin);   // read button state to local variable      )        |
  }                                    // repeat-----------------------------------)        |
  delay(1000);                     // wait for opposing traffic to stop                     |
  digitalWrite(redLED, LOW);       // turn off the red LED                                  |
}                                  // end main method, repeat to top ------------------------
```

Figure 21: Traffic Light Arduino Code.

The traffic light code contains a while loop. While loops run repeatedly until a certain logical condition is met – in this case the loop runs until bState == LOW. While loops are very advantageous for certain things; however, they can be tricky. If a while loop's break condition is never met, the code will be stuck in an infinite loop and is only recoverable by power cycling.

❖ Exercise – Replace the button with a reed switch and activate it with a magnet.

Unit 2: *Sensors, Serial, & Methods*

2.1 Sensors

Key Concepts

- Analog Inputs & ADC
- Digital Information Size
- Voltage Dividers
- Pulse Width Modulation (PWM)

Parts

- Red, yellow, & green LEDs
- Potentiometer
- Photoresistor
- 470Ω & 10kΩ resistors
- Jumper Wires
- Breadboard
- Arduino Nano

Digital pins are great for discrete inputs and outputs, but what if you want to measure a range of values? Most Arduino's are ready to handle that challenge they can measure a voltage level between 0V & 5V as an analog input. Since microcontrollers are digital (ones and zeros) processors, analog (continuous/smooth) values have to be quantized into integer values using the analog to digital converter (ADC) built into the processor. The Nano has a 10-bit ADC which means that it can read a continuous voltage between 0-5V as 1024 (2^{10}) discrete levels resulting in integer values from 0-1023. Bits are a unit of digital information size. Eight bits make up one byte. In this lesson, we'll use a potentiometer to vary the input voltage to pin A0 from 0 to 5V.

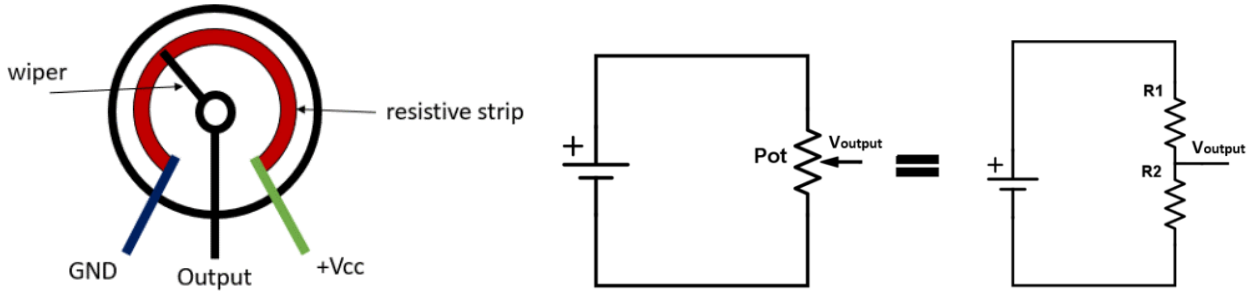


Figure 22: How a potentiometer works. A potentiometer, like other resistors, does not have polarity so the GND and +V_{cc} legs can be switched; however, the output leg cannot be switched.*

Potentiometers vary in resistance when turned and are commonly used in circuits as an input for tuning, for example a volume knob on a stereo. A volume knob may function on a logrythmic scale, but the potentiometer in the kit is a linear one, which means it will change evenly across its range. The potentiometer has 3 pins that can be set up as a voltage divider to vary the output from 0-5V. A voltage divider is just a set of resistors that split a voltage based on differences in resistance. This lesson uses the potentiometer's variable voltage as an input to control the brightness of an LED using the Arduino's built-in functions, *map* and *analogWrite*. *Map* relates the input, which varies from 0-1023, to the output range, which is 0-255. The 0-255

range exists because the “analog” output is simulated using pulse width modulation (PWM). PWM is a technique for getting analog results with digital means; digital control is used to create a square wave (see Figure 23), a signal switched between on and off with variable times (duty cycle). This happens so fast that it creates the effect of having a continuous average voltage.

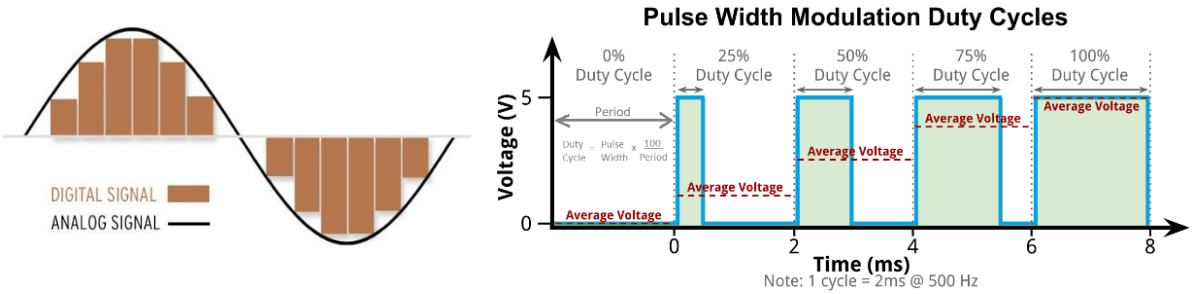


*Figure 23: Digital vs. analog signals** (left) and PWM (right).*

Wire up the circuit shown in Figure 24 and upload the code in Figure 25 to the Arduino Nano.

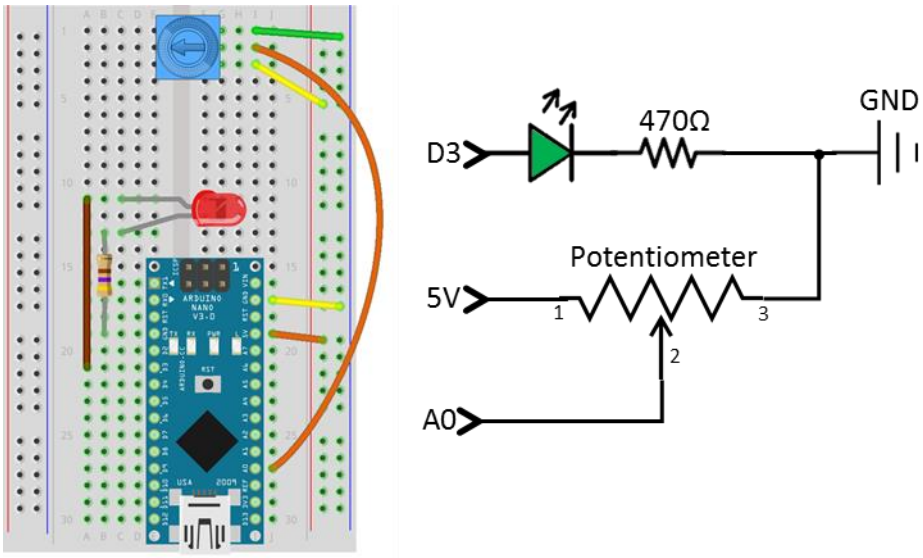


Figure 24: Sensors layout diagram (left) and schematic (right) with potentiometer.

Photo credits – *https://randomnerdtutorials.com/electronics-basics-how-a-potentiometer-works/ and **http://www.klipsch.com/blog/digital-vs-analog-audio.

```
// Constants
const int LED = 3;                    // LED on pin 3
const int sensorPin = A0;             // sensor attached to pin A0 (analog input #0)

// Variables
int sensorValue = 0;                  // variable used to store the value coming from the sensor
int outputValue = 0;                  // variable to store the brightness value for the LED

void setup() {                        // start setup, this code runs once at startup
  pinMode(LED, OUTPUT);               // set greenLED as an output
}                                     // end setup

void loop() {                           // start main loop
  sensorValue = analogRead(sensorPin);    // read value sensor (10 bit range(0-1023)) and store
  // map the sensorValue of 0-1023 to outputValue in a 0-255 range
  outputValue = map(sensorValue, 0, 1023, 0, 255);
  analogWrite(LED, outputValue);          // vary PWM duty cycle to LED
}
```

Figure 25: Sensors code.

Does the LED dim and brighten when you turn the potentiometer's knob? Now, replace the potentiometer in the circuit with a photoresistor and a 10kΩ resistor in series as shown in Figure 26. The photoresistor's resistance value changes in response to varying light exposure. Have you seen a smart phone screen brighten when it is placed in the sun? This is because screen backlights need to be brighter in brighter environments to remain visible, so most phone screens rely on light sensors to help them adjust. The LED in this lesson simulates the behavior of such a screen. Does the LED change brightness well? The next lesson discusses how you can get quantitative feedback from the sensor in order to regulate brightness.

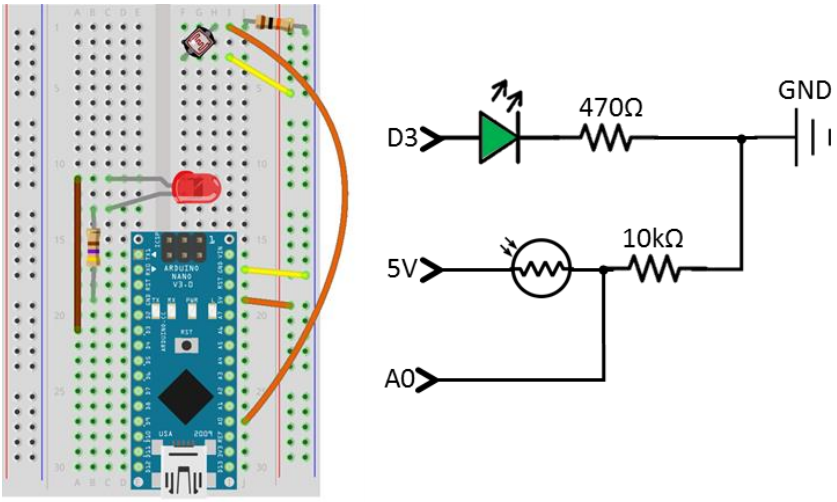


Figure 26: Sensors layout diagram (left) and schematic (right) with photoresistor.

❖ Exercise – Replace light sensor with a resistive flex sensor.

2.2 Serial

Key Concepts

- Sensor Calibration
- Data Sheets
- Number Bases (Radix)

Parts

- LED
- Photoresistor
- 470Ω & 10kΩ resistors
- Jumper Wires
- Breadboard
- Arduino Nano
- Mini USB cable

Serial communication is a key component of physical computing, it provides a digital means of understanding what the processor is thinking. In this lesson, we will use the serial output to help us perform sensor calibration for the photocell. First, perform a manual sensor calibration by reading the sensor's useful range in the environment, and then mapping that range to the output so the LED has a more noticeable response to the varying light in the room. Adding a

serial print line (Serial.println), not a serial print (Serial.print), statement into the sensors code (green Box) will display sensorValue and a carriage return to the serial monitor (Figure 27).

```
// Constants
const int LED = 3;                  // green LED on pin 3
const int sensorPin = A0;           // sensor attached to pin A0 (analog input #0)
const int sensorMin = 0;            // minimum value measured by sensor Default = 0
const int sensorMax = 1023;         // maximum value measured by sensor Default = 1023

// Variables
int sensorValue = 0;                // variable used to store the value coming from the sensor
int outputValue = 0;                // variable to store the brightness value for the LED

void setup() {                      // start setup, this code runs once at startup
  pinMode(LED, OUTPUT);             // set LED as an output
  Serial.begin(9600);
}                                   // end setup

void loop() {                                       // start main loop
  sensorValue = analogRead(sensorPin);              // read the value of the potentiometer and
  Serial.println(sensorValue);                      // print out the sensor value, then go to t
//  Serial.println(sensorValue, BIN);                  // print out the sensor value in binary
  outputValue = map(sensorValue, sensorMin, sensorMax, 0, 255);  // map the sensor value from m
  outputValue = constrain(outputValue,10,255);      // constrain output to values we like
  analogWrite(LED, outputValue);                // vary PWM duty cycle to LED
  delay(50);
}
```

Figure 27: Manual sensor calibration can be done with the help of Arduino's serial monitor.

If nothing is displaying on the serial monitor or if the serial monitor shows random characters, check that its baud rate (drop-down menu in the bottom right corner) matches the baud rate set in the code – in this case 9600 bits/second. Completely cover the photocell and then shine a bright light on it. Estimate the minimum and maximum values using the serial monitor output and then test these values into the code.

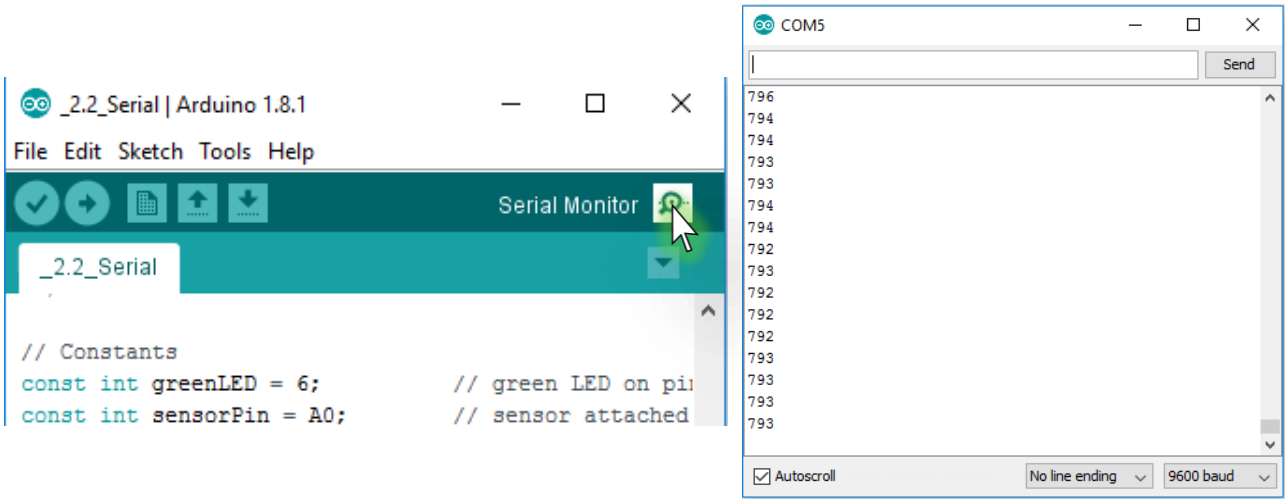


Figure 28: How to navigate to the Arduino serial monitor.

Do you feel confident in your estimate? Click on the "Tools" menu and then select "Serial Plotter" to open Arduino's serial plotter. It displays the serial output graphically. Additionally, it is possible to get a good idea of a sensor's range without testing by checking its data sheet (Figure 29). Typically, it is good practice to check a part's data sheet before buying or using it, but calibration is still necessary to get the most accurate results.

Ambient light like…	Ambient light (lux)	Photocell resistance (Ω)	LDR + R (Ω)	Current thru LDR +R	Voltage across R
Dim hallway	**0.1 lux**	600KΩ	610 KΩ	0.008 mA	0.1 V
Moonlit night	**1 lux**	70 KΩ	80 KΩ	0.07 mA	0.6 V
Dark room	**10 lux**	10 KΩ	20 KΩ	0.25 mA	2.5 V
Dark overcast day / Bright room	**100 lux**	1.5 KΩ	11.5 KΩ	0.43 mA	4.3 V
Overcast day	**1000 lux**	300 Ω	10.03 KΩ	0.5 mA	5V

This table indicates the approximate analog voltage based on the sensor light/resistance w/a 5V supply and 10KΩ pulldown resistor.

Figure 29: User-friendly datasheet example from http://learn.adafruit.com/photocells.

Humans generally use the base 10 number system, or decimal. However, machines use binary, or base 1. There are other number base (radix) systems as well. Learn more about number bases at http://www.purplemath.com/modules/numbbase.htm. Understanding number bases and how to convert between different bases is a very good to know.

❖ Exercise – Change the code so sensorValue prints in binary instead of decimal.

2.3 Methods

Key Concepts

- Modularity
- Hard vs. Soft Coding
- Libraries

Parts

- LED
- Photoresistor
- 470Ω & 10kΩ resistors
- Jumper Wires
- Breadboard
- Arduino Nano
- Mini USB cable

Using methods, or functions, is a good way to break up a daunting coding task. This lesson focuses on how to take the code from lesson 2.2 and break it up into segments that are handled by methods. One of the segments will even do the calibration process all on its own, but the main script will not get any longer! Using the circuit from 2.2, upload the code in Figure 30. Methods are beneficial for performing repetitive tasks with less code; instead of having multiple blocks of the same code in the main script, the code block is written once as a function outside of the main script and then only called within the main script. Methods can also be easily re-used from one project to another, because it is essentially a small pre-packaged set of code. This is called modularity. Modularity is generally a good coding practice as is soft coding – which avoids coding values and functions directly into source code unlike hard coding. Both modularity and soft coding allow code to be applicable in more than one situation without intensive changes. In a lot of cases with Arduino, developers and fellow open-source coders will write solution blocks and share them on the internet or embed them in the default Arduino software IDE. For example, “pinMode” is a method that has many smaller commands running behind the scenes. Larger collections of these packages build as libraries and classes.

```
// Constants
const int LED = 3;           // green LED on pin 3
const int sensorPin = A0;    // sensor attached to pin A0 (analog input #0)
int sensorMin = 1023;        // minimum sensor read, initialize as opposite extreme
int sensorMax = 0;           // maximum sensor read, initialize as opposite extreme

// Variables
int sensorValue = 0;         // variable used to store the value coming from the sensor
int outputValue = 0;         // variable to store the brightness value for the LED

void setup() {               // start setup, this code runs once at startup
  pinMode(LED, OUTPUT);      // set LED as an output
  Serial.begin(9600);        // initiate Serial connection
}                            // end setup

void loop() {                              // start main loop <-----------------------------------------
  sensorValue = analogRead(sensorPin);     // read the value of the potentiometer into "sensorValue"  |
  outputValue = convertInput(sensorValue); // pass input value to method returning desired output     |
  analogWrite(LED, outputValue);           // vary the output to the LED                              |
  printValues();  // use this method to display labled values in the serial monitor or serial plotter |
  delay(50);                               // short delay to give the serial port a break.            |
}                                          // End main method, repeat to top --------------------------

// this is a method, it is a block of code that you can call by typing "printValues();" -------------
void printValues() {                       // will run once when called & return no value (void)      |
  Serial.print("Sensor Value: ");          // print "Sensor Value: " to the Serial console            |
  Serial.print(sensorValue);               // print the sensorValue (0-1023)                          |
  //  Serial.print(" Output Value: ");     // print " Output Value: "                                 |
  //  Serial.print(outputValue);           // print the LED Value (0-255)                             |
  Serial.print(" Min: ");                  //                                                         |
  Serial.print(sensorMin);                 //                                                         |
  Serial.print(" Max: ");                  //                                                         |
  Serial.print(sensorMax);                 //                                                         |
  Serial.println();                        // carriage return                                         |
}                                          //---------------------------------------------------------

int convertInput(int _input) {         // converts input to desired output and returns an integer ---
  if (_input > sensorMax) {            // this updates the max value if the current is greater       |
    sensorMax = _input;   }            //  be careful global variables can be modified by methods    |
  if (_input < sensorMin) {            //                                                            |
    sensorMin = _input;   }            // be careful global variables can be modified by methods     |
                                       // map the input value from 0-1023 range to 0-255 range       |
  int out = map(_input, sensorMin, sensorMax, 10, 255);
  out = constrain(out, 10, 255);       // limit the output to values we like                         |
  return out;                          // this method returns an int to the function call            |
}                                      //-----------------------------------------------------------
```

Figure 30: Methods code.

Unit 3: *Servos, IR, & RGB*

3.1 Servo Motors

Key Concepts

- Feedback
- For loops
- Objects

Parts

- 2 x Buttons
- Potentiometer
- Servo
- Continuous Servo
- Jumper Wires
- Breadboard
- Arduino Nano
- Mini USB cable

This lesson introduces physical outputs using servos. A servo is a small electromechanical device that uses motors and sensors to produce a controlled motion. These motions are typically a specific commanded angle or speed. They achieve reproducible results by having internal feedback, (often an internal potentiometer) which tracks shaft position for the servo drivers. Servos also use a controller to actuate and hold at the commanded position or speed. Position controlled servos are commonly used in throttles or control surfaces on small vehicles. Continuous rotation servos can be used as easily controllable gear-motors. Wire up the circuit in Figure 31. The Arduino IDE comes with numerous example sketches in the "File" tab under "Examples." Open the one named "Sweep" in the "Servo" option (Figure 32).

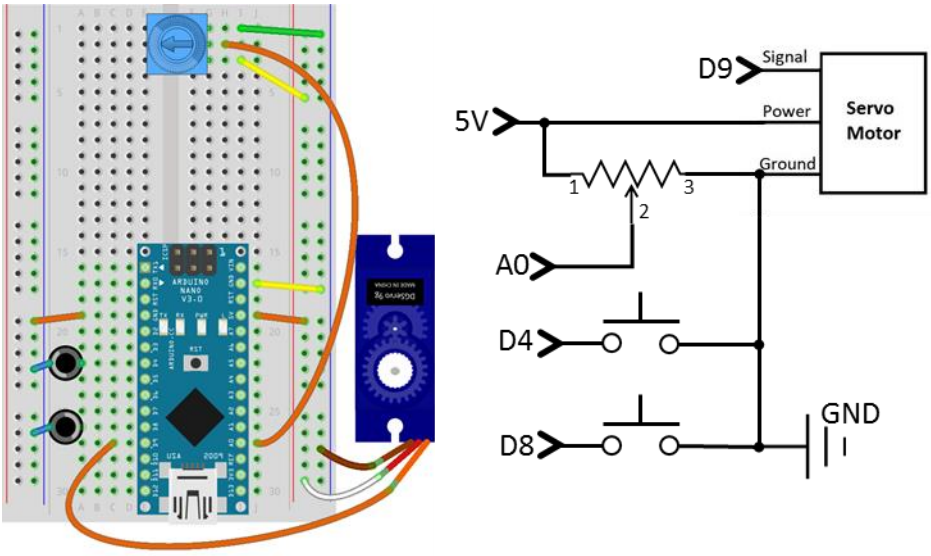


Figure 31: ServoMotors layout diagram (left) and circuit diagram (right).

The “Sweep” sketch increments a servo’s command angle back and forth until it is unplugged. The example uses a for loop to increment the servo angle. This control structure is a looping container that uses an initial condition, end condition, and increment to produce a fixed number of cycles.

```
for (pos = 0; pos <= 180; pos += 1)
```

means that the code in {} after for() will run when the variable ‘pos’ equals 0, 1, and so on 181 times. After the condition is met, the rest of the main loop is completed. This should make the servo sweep back and forth.

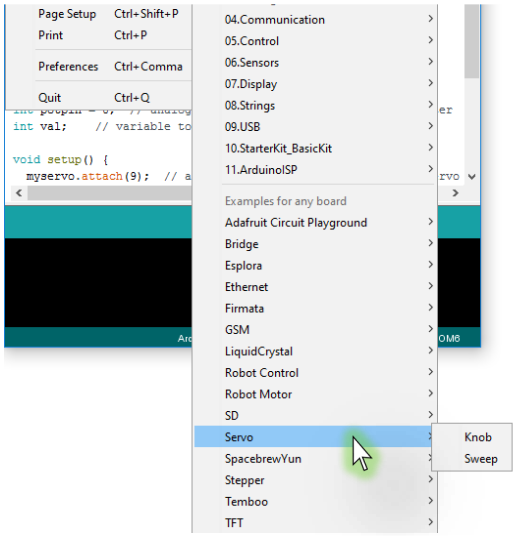


Figure 32: Arduino has helpful example sketches under the "File" tab.

> ❖ Exercise – try changing the step size in the for loop and see how the behavior of the servo changes.

Next, look over and upload the code in Figure 33.

This ServoMotors code uses two buttons to make a servo swing back and forth – like a rudder control. Notice the content of the second line: ‘Servo myServo.’ This line is the creation of a servo object named myServo, which is an arbitrarily chosen name. Creating objects in code is a helpful way to group sets of variables and functions together. The setup loop’s last line ‘myServo.attach(servoPin)’, similar to the general methods from earlier, is a myServo specific method concerning the Servo class that now connects all parts of that object with the pin. ‘Attach’ activates the servo signal on that pin and correlates that pin with any ‘myServo.write’ functions. This is especially convenient when there are more than one of an object, because it keeps the instances organized.

```
#include <Servo.h>              // import the servo library, gives you access to servo code
Servo myServo;                  // create "Servo" object called "myServo" to control a servo

// Constants
const int buttonPin = 8;         // button connected to pin 8
const int buttonPin2 = 4;        // button connected to pin 4
const int servoPin = 9;          // servo attached to pin 9

void setup() {                           // start setup, this code runs once at startup
  pinMode(buttonPin, INPUT_PULLUP);    // initialize buttonPin as an input and turn
                           // on the arduino's internal pull-up resistor on buttonPin
  pinMode(buttonPin2, INPUT_PULLUP);  // initialize buttonPin as an input and turn
                           // on the arduino's internal pull-up resistor on buttonPin
  myServo.attach(servoPin);            // attaches the servo on pin "servoPin" to the
                          // servo object. servo must be attached to send it a command.
// When a servo is attached, it will drive to the command position
}                                        // end setup

void loop() {                            // start main loop
  // this is called a conditional statement.
  // if the true/false condition inside the () is true the code in the {} will execute,
  // if not it will be skipped
  if (digitalRead(buttonPin) == false){
    myServo.attach(servoPin);          // Tells servo object the servo pin.
    myServo.write(175);                // drive the servo to 175 deg
    delay(50);                         // a .05 seccond delay, reduces jitter
  }                                    // end if
  else if (digitalRead(buttonPin2) == false){
    myServo.attach(servoPin);
    myServo.write(5);                  // drive the servo to 5 deg
    delay(50);
  }
  else{                                // if the first two cases are false, execute
  myServo.attach(servoPin);
  myServo.write(90);                   // drive the servo to 90 deg (center)
  }
}                                      // end main method, repeat to top
```

Figure 33: ServoMotors code.

Now, upload the other servo example sketch, 'knob.' This is a good illustration of higher resolution actuator control. It would be difficult to steer a boat using only 2 buttons, so the

knob code implements a potentiometer to act more like a steering wheel. The servo signal looks like a PWM signal; it communicates the desired angle with a varying pulse length. Finally, switch out the servo with the continuous (360^{o}) servo – the type of servo in a SeaGlide buoyancy engine – and try the three different code examples to see how they differ. If the continuous servo does not stop moving at 90^{o} output, adjust the bias-control-potentiometer on the back until it does. Remember, detaching the servo will reliably stop a continuous rotation servo.

❖ Exercise – what is an oscilloscope?

3.2 IR and RGB

Key Concepts

- Color Mixing
- Case Statements
- Structures
- Pointers

Parts

- 3 x LEDs (red, green, blue)
- IR Remote & Receiver
- 3 x 470Ω Resistors
- Jumper Wires
- Breadboard
- Arduino Nano
- Mini USB cable

Infrared (IR) is just outside of the visible light spectrum, so it cannot be detected by an unaided human eye. Remote controls for most TVs and cable boxes use IR to communicate. SeaGlide also utilizes an IR remote for control. The circuit in Figure 34 will respond to IR commands from a remote, decode the remote's button ID, and then print it to the serial monitor along with values for the three LEDs.

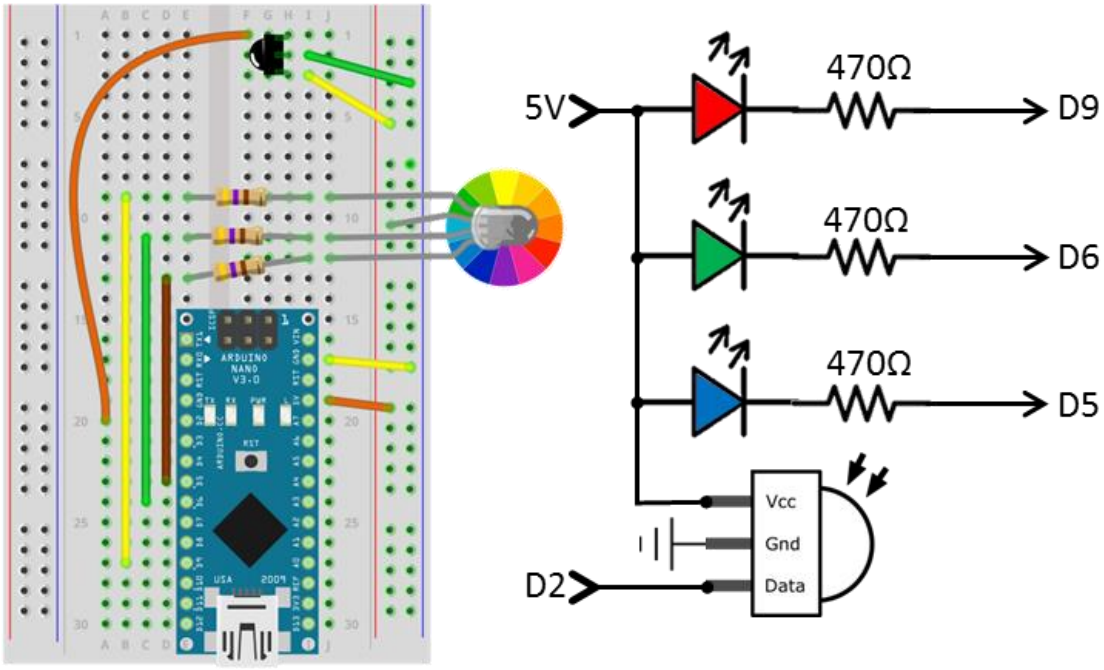


Figure 34: IR-RGB layout diagram (left) and circuit diagram (right).

First, install the external Infrared (IR) Remote library as per the instructions on seaglide.net/firmware. After looking over and uploading the code in Figure 35, look at the serial stream to get desired IR ID. Modify case statement to have different buttons and then increment different LEDs.

If the LEDs were close enough, they would act like an RGB LED (like the one in SeaGlide) and their colors would blend according to the diagram in Figure 36. Color Mixing is how Liquid Crystal Display (LCD) screens can display almost any color – each pixel has subpixels for red, green, and blue that vary in intensity to produce any color.

```
// Pins
static byte POT_PIN = A3;            // Potentiometer signal pin
static byte RECV_PIN = 2;            // IR reciever signal pin

static byte RED_LED = 9;             // Red LED cathode
static byte GREEN_LED = 6;           // Green LED cathode
static byte BLUE_LED = 5;            // Blue LED cathode

#include <IRremote.h>                // include IRremote library installed in libraries folder

IRrecv irrecv(RECV_PIN);             // create an object, IRrecv, named irrecv

decode_results results;              // create a custom structure named results
                                     // Structures are like a folder of variables
void setup()
{
  Serial.begin(9600);                // start serial communication
  irrecv.enableIRIn();               // Start the IR receiver
  Serial.println("Waiting for IR command");
}

byte i = 0;                          // create a counter variable
byte r = 7;                          // byte variables have a max range of 0-254
byte g = 7;                          // perfect for outputting to the analog write hardware
byte b = 7;                          // if the number you put in a byte is large than 254 then i

void loop() {
  if (irrecv.decode(&results)) {     // if decode returns true, a signal was recived and the res
    Serial.println(results.value);   // print IR code, irrecv.decode modified results.value beca
    irrecv.resume();                 // setup watch to receive the next value
    //    switch (results.value){
    switch (i) {                     // begin switch case; i is the variable being matched. chan
      case 0:                        // run this segment when i == 0
        r += 8;                      // r = r+8; this increments the analog value we will write
        break;                       // designates the end of the case segment, breaks out of th
      case 1:
        b += 8;
        break;
      case 2:
        g += 8;
        break;
    }
    i++;                             // i=i+1, increment the counter value so each button press
    if (i == 3) {                    // if i is too high reset it to zero
      i = 0;
    }
    Serial.print(r);                 // print the value for r to the serial monitor
    Serial.print(", ");              // print a comma and space for value seperations
    Serial.print(g);                 // print the value for g
    Serial.print(", ");
    Serial.println(b);               // print the value for b
  }
  analogWrite(RED_LED, r);           // output pmw(0-255) to RED_LED pin
  analogWrite(GREEN_LED, g);         // output will seem reversed, the led will get brighter as
  analogWrite(BLUE_LED, b);
  delay(100);
}
```

Figure 36: Color mixing diagram.

Initially, any code received will cause the Nano to cycle through decreasing the brightness of each LED in a cycle. By copying button codes from the serial monitor, you can make different buttons respond to different segments of the code through the case statement and control the LEDs independently. A switch case statement is basically a streamlined group of 'if…else' statements. Instead of typing out an 'if…else' statement for each option, you can use a switch case statement to compare a single variable with a long list of values. In this circumstance, it would be useful to compare the decoded IR code switch (results.value) with the selected fixed codes that we selected to control our lights (Case 039487493789283:).

Interestingly, the IR receive code creates an object irrecv, then a structure of type decode_results named results, which gives results a value it passes in a pointer to results(&results) into a method of the irrecv object. This process is like telling the method where to deliver the information instead of duplicating the information into and out of the function. It is a useful way to handle more variable outputs of functions and it is very memory efficient, but can be very confusing. Using a pointer to refer to a variable is like referring to a page number instead of reproducing all the information on that page every time you want to reference it.

3.3 BuoyancyEngine

Key Concepts

- Debugging & Troubleshooting
- Component & Subsystem Testing

Parts

- 3 x LEDs (red, green, blue)
- Button
- Potentiometer
- IR Remote & Receiver
- 3 x 470Ω Resistors
- Jumper Wires
- Breadboard
- Arduino Nano
- Mini USB cable

The circuit layout in Figure 37 should allow the code in SeaGlideV1_0.ino to be benchtop tested. Having all the parts working in a flexible environment is useful for debugging electronics including those in SeaGlide. Debugging is the process of trying to first *isolate* a problem in order to then correct it. Having a benchtop interface that is easy to modify is very helpful when troubleshooting; not needing to solder or de-solder to add individual components to a system for testing saves time and parts. Another good way to test is to take parts that you know work and replace some of your suspicious parts to see what is not working. This process is closer to component and subsystem testing. Components from all three lessons are in the SeaGlide buoyancy engine. Component and subsystem testing should be done throughout the entire SeaGlide build. Buoyancy engines are a critical subsystem that should be benchtop tested before going in a glider.

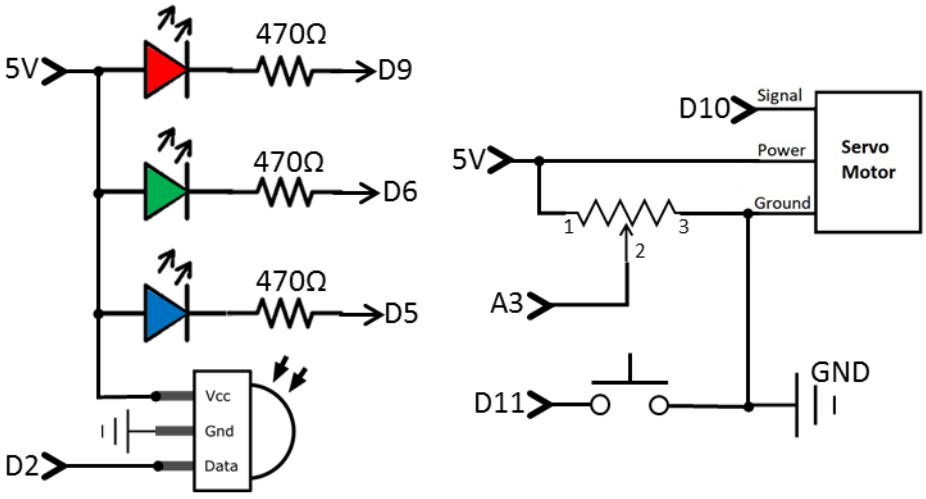


Figure 37: Buoyancy engine layout diagram (left) and circuit diagram (right).

The basic SeaGlide buoyancy engine does not have position feedback; the plunger extends and retracts based on a combination of timing and limit switch triggers. In this example, the screw and plunger are not connected to the continuous servo. So, when the servo turns, pretend the plunger is moving and hit the limit switch. Run the continuous servo one way until button press, stop (glide), and then run it the other way, wait (glide), then re-start from the beginning.