EE 1301: Introduction to Computing Systems

IoT Laboratory #2

Getting Started with Sensors and Actuators

Created by: David Orser, Kia Bazargan, and John Sartori

Many thanks to the students, teaching assistants, and faculty that work to continually improve this document. Together we make better labs!

Please send comments and suggestions to orser@umn.edu
Copyright 2018

Background

A smart device interacts with the world using Sensors and Actuators. A **sensor** is a component that detects changes in the environment. A thermometer, light detector, and soil humidity detector are all examples of sensors. An **actuator** is a device that can manipulate the world around your smart device. A valve, light, motor, or display are all actuators.

Purpose

In this lab, you will familiarize yourself with the inputs and outputs available on your Photon. Using these inputs/outputs we will explore how your Photon can interact with the world around it. The idea is to give you an overview of what is possible, and in turn, to stimulate ideas on what your project may contain.

Supplemental Resources

Device Description - Light Sensor (NOTE: Not yet updated for new LDR.)
    https://docs.google.com/document/d/1CZS5gcCSr3VjYKRbaNx_3vLojI4gn_NwTIVp8csDyis/export?format=pdf
Device Description - Temperature Sensor (TMP36)
    https://docs.google.com/document/d/1aU-czttYtcsZGxNUK-tyo3LB-An_cs-UyPaWxzwSQwc/export?format=pdf
Device Description - Individually Addressable LEDs (8mm WS2811 RGB LED)
    https://docs.google.com/document/d/15UqLJ_mDqQ16eRNtT0lnVlmQ6QVm4BX43DxwAUHWk1A/export?format=pdf
Device Description - Simple Speaker (Piezo speaker)
    https://docs.google.com/document/d/1jHnLRklvXFc-_g4nmLzca9y7KzvwRADfOfDVTBDKwI4/export?format=pdf
Device Description - Servo Motor

Pre-Lab Requirements

Before coming to the lab, there is a fair amount of reading material you should review. Reading materials are provided in a "Quick Lesson" format -- stand-alone documents that cover a single topic. Please read through all the materials on the pre-lab checklist below.

Pre-Lab Checklist
- ❏ Complete Homework problem 3B
- ❏ Read the Quick Lesson - Electrical Circuits
- ❏ Read the Quick Lesson - Getting to Know Your Pins: Power Supplies, Analog, and Digital Pins
- ❏ Read the SparkFun Breadboard Tutorial - https://learn.sparkfun.com/tutorials/how-to-use-a-breadboard$(ADC - 620)$flat
- ❏ Read the second half of the SparkFun Tutorial - "How to read a schematic" http://learn.sparkfun.com/tutorials/how-to-read-a-schematic#name-designators-and-values

Reminder: If you use your space efficiently, you should never need to take apart your breadboard. You may have a temperature sensor attached to the A0 pin and the ILED's on D4 at the same time and the different code you flash will determine what is run. There will not be a need to unplug devices and rewire it for each new lab.

Required Components:

LDR (Light Sensor) 200 Ohm Resistor 4.7k Ohm Resistor 100k Ohm Resistor	3 Individually Addressable LEDs (WS2811s) Push button switch Potentiometer

Lab Procedure

Exercise 1: First Sensor - Light Sensor

A light sensor can be very useful for projects. For example, a light sensor can tell us when someone enters a lab (turns on the light), when the sun shines into an office, or when a cupboard or locker is opened.

The light sensor utilized in this lab is called a light-dependent resistor or LDR. An LDR conducts current depending on the amount of light that hits it. Since our Particle devices only measure voltage, we use a resistor to convert the current into a voltage that we can read (V=I*R). The schematic in Figure 1 shows the sensing circuit. As light hits the photo-transistor (LDR), it is converted into current. The current flows through resistor R1, dropping the voltage on the sense pin A1. Therefore, more light means a **lower** measured voltage.

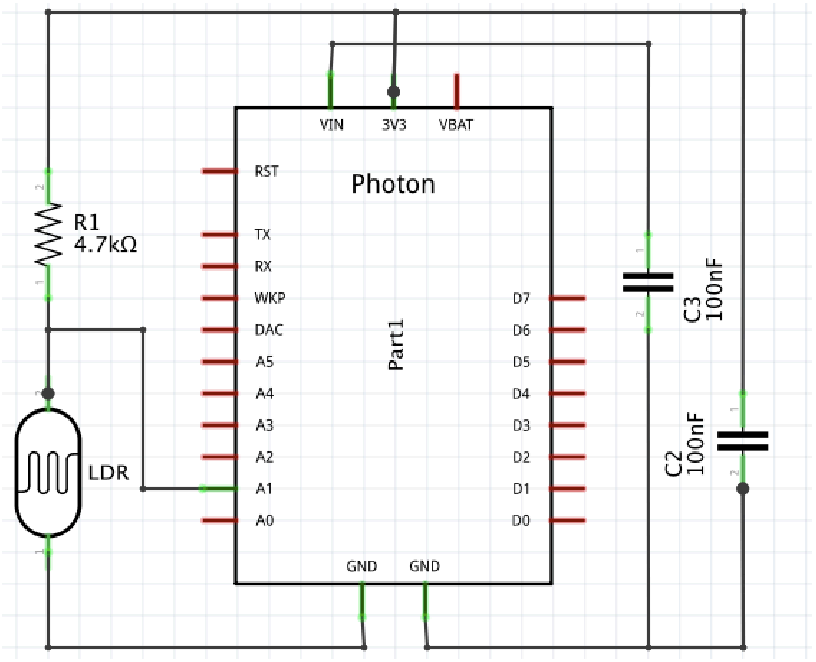


Figure 1: Schematic diagram of the test circuit for an LDR

Testing and retrieving data from a sensor is the first task in evaluating a sensor. In the example below, we will be using the serial port to evaluate our sensor's response and calibrate our final code (see "Quick Lesson - Power Supplies, Analog, and Digital Pins" if the term "serial port" seems unfamiliar).

Note:
Many things can affect sensor readings: temperature, battery voltage, variations between different manufactured parts, and noise. These can all affect the values read by your Particle. If you have issues later in the lab, it is a good idea to come back to this step and verify that the sensor is reporting the values you expect.

Procedure

1) Wire up your Particle and the LDR as shown below.

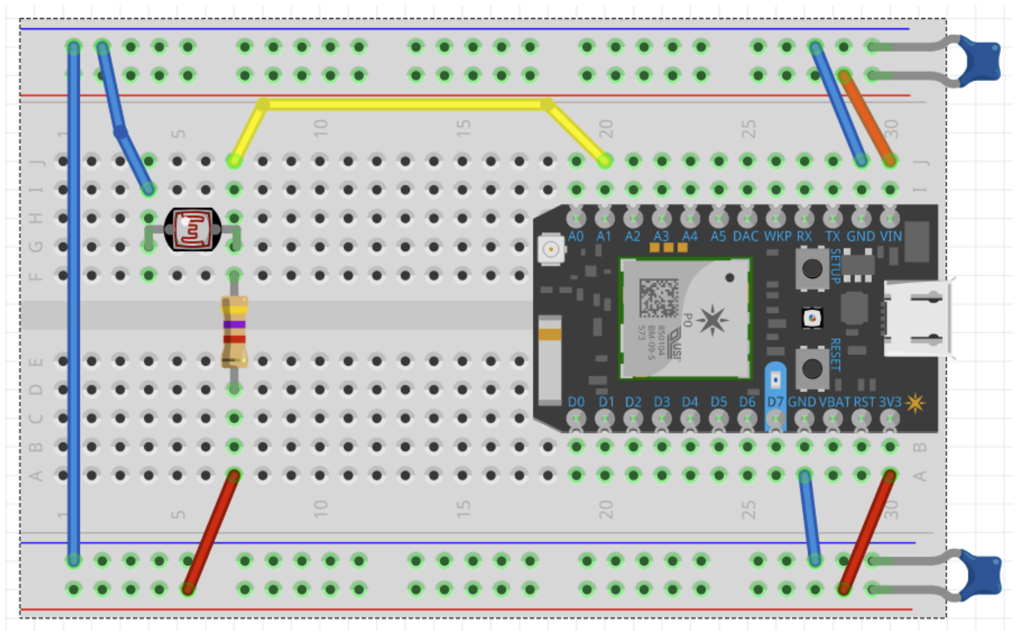

A0 A1 A2 A3 A4 A5 DAC WKP RX TX GND VIN
D0 D1 D2 D3 D4 D5 D6 D7 GND VBAT RST 3V3
SETUP
RESET

Figure 2: Breadboard Layout of the test circuit for an LDR

2) Connect a USB cable from your computer to your Particle device.
3) Open the Particle Workbench IDE (VS Code) and create a new project.
4) Before we declare any functions, we should declare a variable (type: int) to hold the results of our measurement.

```
int data0;
```

5) **<u>In the setup()</u>** function, we first need to set up the serial port.

```
// Open the serial port for communication with the computer
Serial.begin(9600);
```

Note: The line above may look a little cryptic. (Where did the function Serial.begin() come from?) The Particle IDE is designed to be a *very* user-friendly programming environment and is only designed to be used with the Particle line of devices; as such, it makes many assumptions. Two of these assumptions are that the serial port is always available, and the library is always pre-loaded by the compiler. As such, we do not need to declare “Serial” or specify its type, just initialize it.

6) The loop() function will contain the working payload of our program. First, we read from the Analog pin into the variable `data0`.

```
// Read data from analog pins (returns a number from 0 to 4095)
data0 = analogRead(A1);
```

7) Next, we print this data in a readable format to the serial port.

```
// Print the data to the serial port
Serial.print("My Data is: ");
Serial.print(data0);
Serial.println(";");
```

8) Add a heartbeat LED

A heartbeat LED is a useful construct to verify that our program has successfully loaded (or reloaded after a change). Adding three pieces of code to our App will allow us to implement a heartbeat LED easily.

a.) In setup()

```
// Setup D7 pin to output a heartbeat
pinMode(D7, OUTPUT);
```

b.) At the beginning of loop()

```
// Heartbeat, show we're alive
digitalWrite(D7, HIGH);
delay(250);
```

c.) At the end of loop()

```
// Heartbeat, show we're alive
digitalWrite(D7, LOW);
delay(250);
```

Note: When your Particle Photon is behaving oddly, the heartbeat LED can be very useful. Try changing the heartbeat rate of the LED significantly and re-flashing your Photon. You will immediately know if the code has been updated and is running.

9) Save, Verify, and Flash your Code to your Particle Photon.

10) When your Photon has reset and is “breathing cyan”, check the serial monitor by doing the following:
    a.) Open a Particle CLI Terminal using CTRL + SHIFT + P "Particle: Launch CLI"
    b.) Type in the terminal:

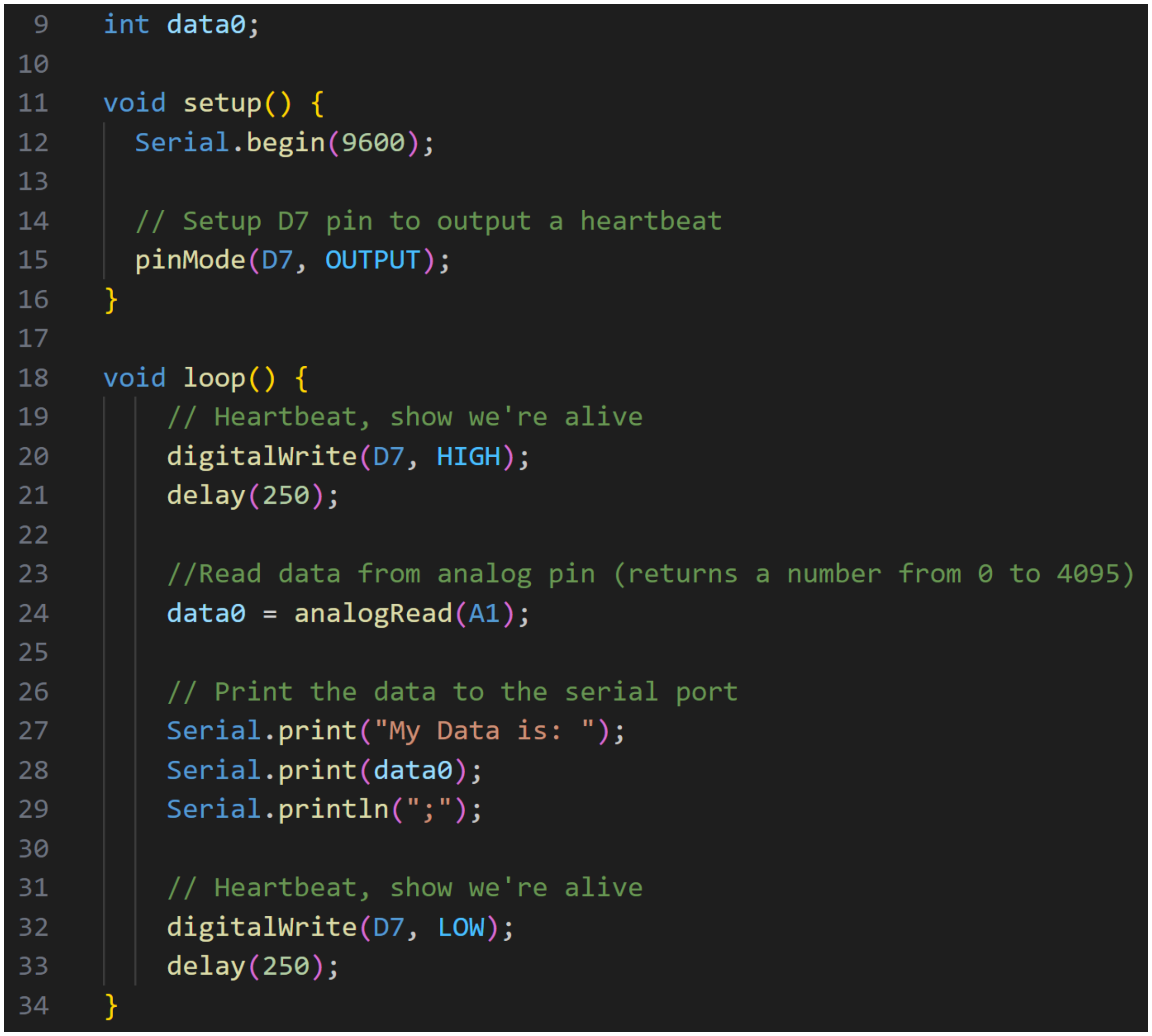
```
particle serial monitor
```

NOTE: You should be in the Particle CLI rather than PowerShell, cmd, or bash.
NOTE: You can also use the “Launch CLI” button in the top right or via Particle Button.

11) Take data with the sensor to prove that it is working.

For example: position the sensor so it can see the room lights then cover and uncover the sensor with your hand. Try using your phone flashlight to see even higher light levels.

Condition	ADC reading
Covered by your hand	
Room with bright fluorescent lights	
…	
Strong Phone Light	



```
int data0;

void setup() {
  Serial.begin(9600);

  // Setup D7 pin to output a heartbeat
  pinMode(D7, OUTPUT);
}

void loop() {
    // Heartbeat, show we're alive
    digitalWrite(D7, HIGH);
    delay(250);

    //Read data from analog pin (returns a number from 0 to 4095)
    data0 = analogRead(A1);

    // Print the data to the serial port
    Serial.print("My Data is: ");
    Serial.print(data0);
    Serial.println(";");

    // Heartbeat, show we're alive
    digitalWrite(D7, LOW);
    delay(250);
}
```

Figure: One possible version of AnalogRead2Serial.ino

Exercise 2: Temperature Sensor

This time, on your own, build a serial evaluation setup for your TMP36 temperature sensor using the schematic and example code offered in Device Description - Temperature Sensor

Take some data from the sensor to prove that it is working (for example, air temp and temp after being pinched for 20 seconds). Note: Be careful not to touch the leads on the bottom of the package as your body impedance might alter your measurements.

Condition	ADC reading
Room Temperature	
Finger Temp	
…<other>...	

Actuators - Display Elements

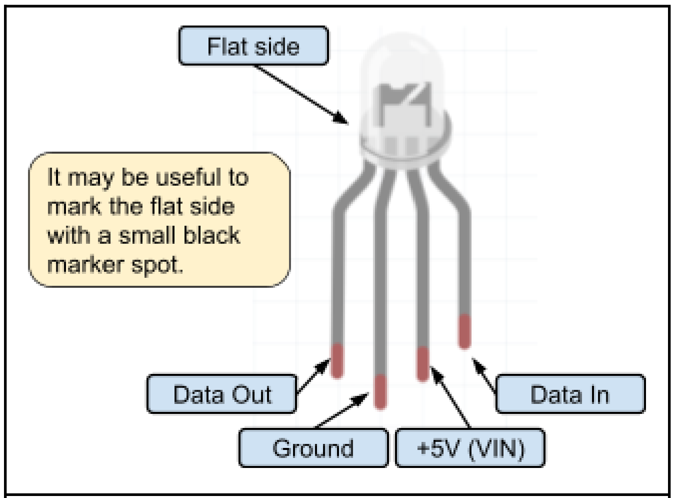
In this section, we'll display some information on external display elements connected to the microcontroller. The advent of very cheap integrated circuits and new packaging technologies has allowed the embedding of encoders and decoders directly into display elements. No longer do you need 8+ wires to communicate in a very basic way with a display. It is common today to use a variation of the serial port (4 wires) utilized above to transmit configuration information (often dozens of variables) to external devices.

Individually Addressable LEDs

We will now look at our first complex output device (sometimes called an actuator). iLEDs, RGB LEDs, or NeoPixels are part of a class of devices that are called individually addressable. This means that when wired up in a chain, every LED can be individually set to a different color. Each LED requires a shared power supply and ground pin. Additionally, each LED has a Data_In and Data_Out pin. When connected in series, the first LED in the string grabs the first 24-bits (color setting) and then passes the rest of the data to the next LED down the chain, which grabs the next 24-bits (second color setting), passing the remaining data, etc. In this context, 24-bits means 24 ones and/or zeros. You do not need to worry about the details of how 24 bits are represented, as there are predefined library functions that handle those details. But why do we have 24 bits to represent the color? Well, each LED internally has three small LEDs: Red, Green, and Blue. Each of these mini-LEDs can show its color with intensity from 0 (off) to 255 (the brightest)[1]. So, if I set the Red value to 128 and the Green and Blue values to zero, I get a halfway bright red color. If I set Red=255, Green=0, and Blue=255, I get the brightest purple.

[1] Remember that 8 binary digits (also called bits) are needed to represent numbers between 0 and 255.

If you want to see what regular LEDs are capable of, check out the YouTube of CSE's Winter Light Show. Imagine what you could do with an RGB LED!
https://www.youtube.com/watch?v=HkN5-Qfiy7s
https://www.youtube.com/watch?v=1zJrMUFZtwI


Flat side
It may be useful to mark the flat side with a small black marker spot.
Data Out
Ground
+5V (VIN)
Data In

iLED Pin Description

NOTE: These devices need a +5V power supply (on the Photon this pin is labeled "VIN"). For more information, please read Quick Lesson - Power Supplies, Analog, and Digital Pins

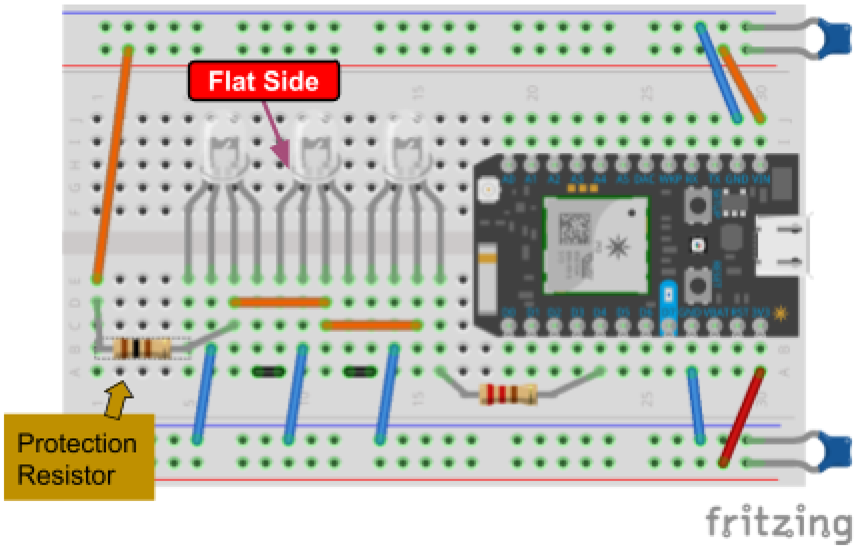
WARNING: Plugging these LEDs in backward (even for a second) will destroy them!!!

WS2811 vs WS2812 NOTE: This year, the Depot ordered two different types of neopixels, and unfortunately, they have slightly different "signaling standards." The lab document was written assuming a WS2811 neopixel, and notes have been added below describing what to change if you have WS2812 neopixels in your kit. There is currently no way to determine if your NeoPixel is a WS2811 or WS2812. If you see strange behavior, please try changing the signaling standard. If that doesn't work, consult your TA.

Exercise 3---Step1: Build the iLED Circuit

For now, wire up your LEDs as follows. **Do not connect the circuit to power until you have finished reading this paragraph in its entirety!** After you are done wiring it up, double check to see that all iLEDs have their flat sides to the left. Connecting the circuit to a power source (USB port of your computer, or a USB adaptor plugged into the power outlet) would result in all three iLEDs turning on with a default blue color. So, if you connect the power and one or more of them do not light up, it means you have not wired up the circuit correctly and you might burn the iLEDs if you keep the circuit connected to the power. Disconnect power immediately and have someone (or yourself) check your wiring. If all three light up blue, then your circuit is probably OK. Disconnect the power and move on to the next step.
WS2812 NOTE: The WS2811 neopixels may not turn on to a default blue color and may only power up once you upload code to your board to enable them.

Figure: iLED Wiring Diagram



Note: The resistor at the bottom-left corner of the circuit above is only used for protecting your iLEDs, in case you connect them in reverse. The resistor causes the iLEDs to flicker. **Once you write the program to light up the iLEDs with different colors (next section) and verify that your circuit generates the expected colors, you can replace the "protection resistor" with a wire and fix the flickering issue.** (If your lab kit doesn't have a 100 Ohm resistor use a 220 Ohm resistor.)

Note: The blue ceramic capacitors on the top and bottom power rails are there to keep the voltage supplies stable under varying demand for current, similar to how water towers keep the water pressure constant under varying demand for water.

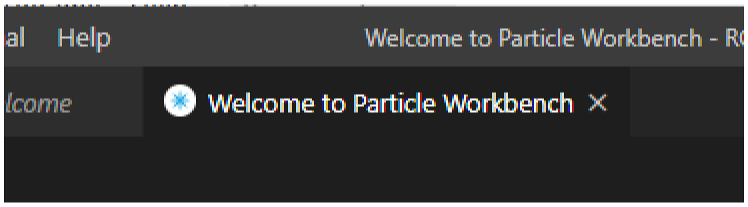
Libraries

Libraries are collections of predefined variables and functions that are commonly utilized. They are usually maintained by companies or interested individuals. They make complicated tasks easier. In the Particle development environment, they are handled in a special way.

Exercise 3---Step2: Writing Code for the iLED

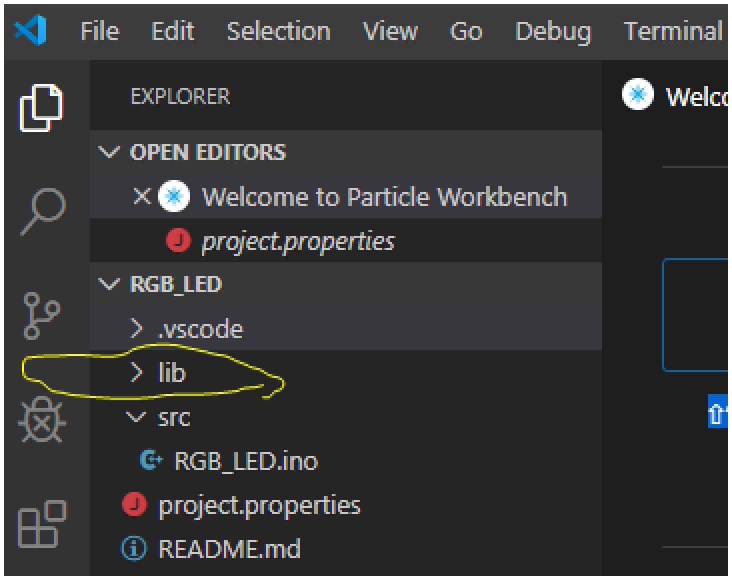
1) Go to VS Code and create a new project

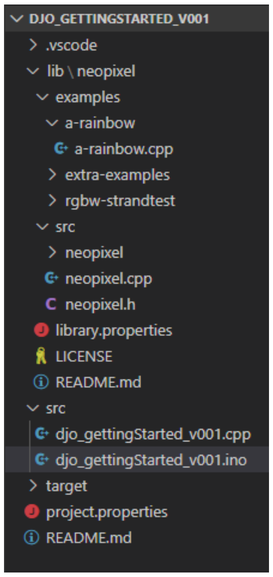
a.) Click on the “Welcome to the Particle Workbench” tab


al Help
Welcome to Particle Workbench - R
lcome
Welcome to Particle Workbench ×

b.) Click on “Create a Project” under the “Getting Started” section.
c.) Choose the parent folder and the name of the project.

2) Bring up the command pallet (⇧⌃P (Windows, Linux) | ⇧⌘P (macOS))
3) Type “libraries” and choose “Particle: Find Libraries”
4) When asked what the library name is, type “neopixel”. When we bought the iLED, its manual said we could use it with the neopixel library. That’s how we knew what library to search for.
5) You can see that in the list of found libraries, the first item is called “neopixel”, and the description shows that it is **[verified]** and compatible with Photon.
6) Bring up the command pallet again, and this time use “Particle: Install Library”.
7) Enter “neopixel” without the quotation marks as the library name.
8) Notice that a new item called “lib” was added to your project:

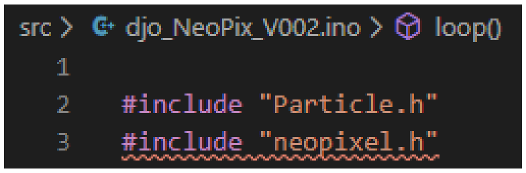




9) If you click on the arrow next to lib, you will notice subdirectories for “examples” and “src”. The src directory contains the *source code* for controlling the “neopixel” color LED.
10) Look through the examples directories and look at a couple of the *.cpp files. You may see the following lines. These lines are necessary to integrate the neopixel library with your code.

```
#include "Particle.h"
#include "neopixel.h"
```

11) Add these lines to the top of **your** source file (the one that was originally created when you created your new project, something like: “djo_TestNeoPixel_v001.ino”).



```
#include "Particle.h"
#include "neopixel.h"
```

12) If you see a red underline beneath “neopixel.h” close and re-open VS Code.

Object Models

Going into detail on object-oriented programming is beyond the scope of this document. It is sufficient to understand that a class is another data type in C++ programming (just like an int, a bool, etc., but a bit more complex, in the sense that it holds both values AND actions to be taken on those values). The internal actions/functions of an object are sometimes called methods. An object is an *instance* of a class.[2]

We will need some basic information in order to create and set up our NeoPixel object:

- The pin number (D4) to which the string of pixels is attached
- The number of pixels (3) in the chain
- The type of controller chip (WS2811, https://www.adafruit.com/products/1734)
  - **WS2812 NOTE:** If you have WS2812 neopixels in your kit, you will use WS2812 as the type of controller chip.

Exercise 3---Step3: Creating a NeoPixel Object and Using it

You can then create a new NeoPixel object similar to how we declare a new integer, except we use a function to fill the object with its initialization data. The int variables defined below are used for better readability when we **instantiate** the new NeoPixel object below. Note that in the next few sections, we are giving you pieces of the code. You have to put together these pieces for the code to work.

```
Adafruit_NeoPixel strip = Adafruit_NeoPixel(PIXEL_COUNT, PIXEL_PIN, PIXEL_TYPE);//
These lines of code should appear AFTER the #include statements, and before
// the setup() function.
// IMPORTANT: Set pixel COUNT, PIN and TYPE
int PIXEL_PIN = D4;
int PIXEL_COUNT = 3;
int PIXEL_TYPE = WS2811;
// int PIXEL_TYPE = WS2812;
// WS2812 NOTE: use WS2812 if you have them
```

This creates “strip”, an object, based on “Adafruit_NeoPixel”, a class, which is defined in the “NeoPixel” library (confused yet?, hold onto your shorts...)

Think of methods simply as functions.

[2] Generally speaking you can have many objects of one class in a program. For example multiple iLED strips you needed to control, each with a different pin, number of LEDs, col

As you saw with Serial, in C++ (and many other programming languages), we access the methods of an object with the syntax “object.method()”. To initialize the strip, we call the method “**`strip.begin()`**”. We only need to do this once, so the best place for it is in the setup() function of our Photon code. As so:

```
void setup() {
    strip.begin();
    ...
}
```

Finally, we have declared and initialized our object; we’re ready to use it to do something useful!

We will be using three methods from the Adafruit_NeoPixel class of objects. In the table below are definitions of the methods we are going to be using.

Method (Function) Definition	Description

`void = strip.setPixelColor(<uint16>,<uint32>)`[3]
`Example:`
`strip.setPixelColor(PixelID, myColor)`
	Store the Color for the n-th PixelID (zero-indexed) in memory. PixelID refers to the n-th physical LED in the chain of serially connected LEDs. The LEDs do not light up yet (see the last row).

`<uint32_t> = strip.Color(<uint8>,<uint8>,<uint8>)`
`Examples:`
`myColor2 = strip.Color(255,0,0)`
`myColor = strip.Color(Red,Green,Blue) // WS2811`
`/* NOTE: WS2812 neopixels set color values as G,R,B instead of R,G,B */`
`myColor = strip.Color(Green,Red,Blue) // WS2812`
	Returns an encoded color representing the RGB values that neoPixel can use. In the second example, Red, Green, and Blue are variables specifying an intensity between 0 and 255.
	NOTE: For WS2812 neopixels specify the color components in the G,R,B order instead of R,G,B order

`void = strip.show()`
`Example:`
`strip.show()`
	Sends the colors stored in memory to the physical LED strip in single burst transmission. This statement causes the lights to change color!

We will define a couple of colors, then store the desired data into the strip object. Once everything is set, we’ll call the show() method to dump the data over the digital data link to the string of pixels. If you forget to use the show() method, the iLEDs will NOT change colors.

Loop() Example Code

```
void loop() {
    /* NOTE: Two versions of the color code are specified below for WS2811 and
```

[3] You can actually see the definition of this method in the file “neopixel.c” on line 699. Opening the library definitions is a useful place to look if you need to figure out how an undocumented library function works.

```
                 WS2812 neopixels. Use the version according to the type of neopixels in
                 your kit and delete or comment the other version. */
    //Setup some colors, WS2811 version
    int PixelColorCyan = strip.Color(   0 , 255, 255);
    int PixelColorRed  = strip.Color(  80,   0,   0);
    int PixelColorGold = strip.Color(  60,  50,   5);
    //Setup some colors, WS2812 version
    /*
    int PixelColorCyan = strip.Color(   255 , 0, 255);
    int PixelColorRed  = strip.Color(  0,   80,   0);
    int PixelColorGold = strip.Color(  50,  60,   5);
    */

    //Set first pixel to cyan
    strip.setPixelColor(0, PixelColorCyan);
    //set second pixel to red
    strip.setPixelColor(1, PixelColorRed);
    //set third pixel to Gopher Gold!
    strip.setPixelColor(2, PixelColorGold);
    strip.show();
    delay(1000);  //wait 1sec

    //flip the red and gold
    strip.setPixelColor(0, PixelColorCyan);
    strip.setPixelColor(1, PixelColorGold);
    strip.setPixelColor(2, PixelColorRed);
    strip.show();
    delay(1000);  //wait 1sec
}
```

Verify and flash your code to your Photon, check that the LEDs function as intended.

It is important to note that NeoPixels hold their color settings until the next show() method call or 5V power goes away. So, there is no need to continuously update them.

Finally, when your circuit generates the expected colors, you can **replace the "protection resistor" with a wire** to fix any flickering issue.

Exercise 4

Based on the circuit you built-in Exercise 3, finish one of the following two exercises.

Exercise 4A: Changing Brightness in Steps of 50

(Create a new project: so you have a separate copy of your code.)

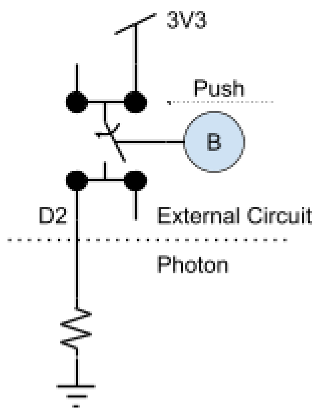
Now change the program so that the first LED shows red, the second one green, and the third one blue. We want all three to have a light intensity of 0, then after a second, all three change intensity to 50, then 100, 150, 200, 250, and back to 0. (VERY STRONG HINT: Do not use the pixel.setBrightness() function. It is not intended for this purpose and very likely will break and/or confuse you later.)

Exercise 4B: Flickering Candle Exercise

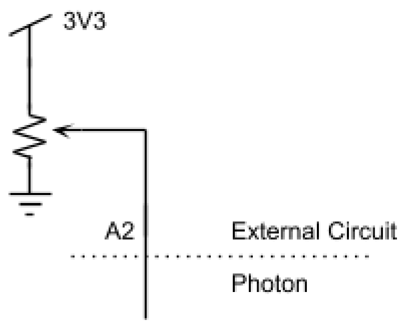
Copy your HW3 function "**`int randWalk(int oldValue, int updateSize);`**" to VS Code. Use it to set the brightness of your LED(s). Try making all three flicker with the same brightness. Play with the updateSize to create different effects. Try making all three flicker independently (they will each have a separate "oldValue" and call randWalk() separately.)

Exercise 5: Sensors - Human Input Devices

Accepting input from a human being is a useful feature for microcontrollers. While this can appear to be a simple task, several pitfalls exist. We will start by setting up a single push button and potentiometer (knob) as input devices.

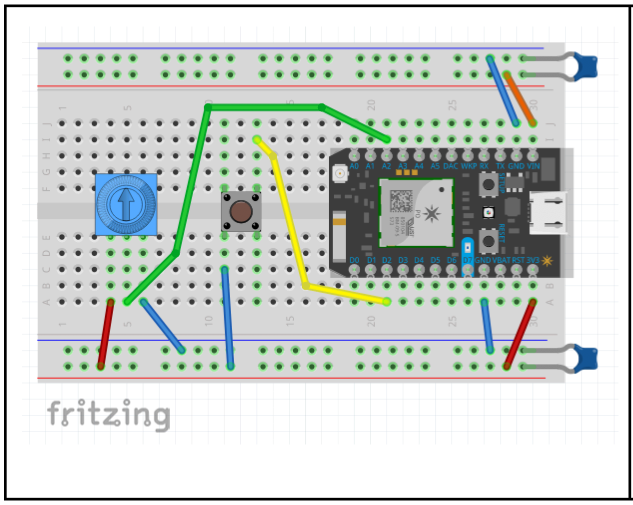


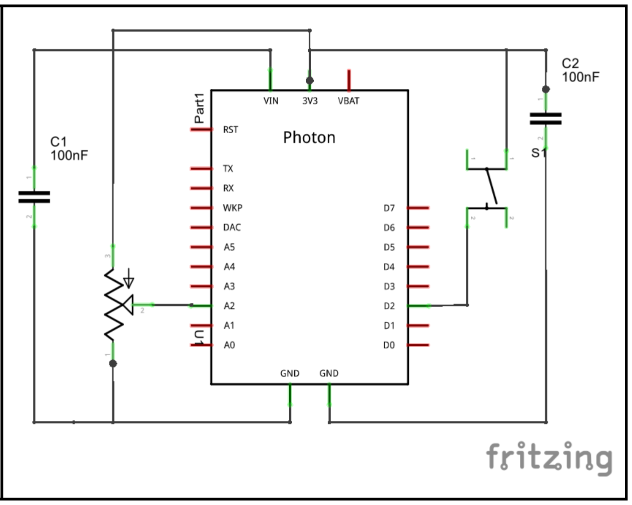
The **push button** is a normally open momentary switch; this means that if we tie one terminal of the push button to 3V3 and the other terminal to an input of the Photon set to "INPUT_PULLDOWN" it results in the circuit on the right. When the button is "not pushed", it results in the Photon's internal "pulldown" resistor pulling the input to ground, 0.0V, or a "low" state. Think of this as a default state. When the button is pushed, the input gets connected to 3V3 with a low resistance. The resistors have a tug of war, and the lower resistance path wins, pulling the input to 3.3V or a "high" state.



The **potentiometer** is a three-terminal device. It is implemented physically as a long resistor with a contact on each end (terminals 1 and 3) and a "swiper" that can contact anywhere in between (terminal 2). When the two fixed terminals are connected to the power rails (GND and 3V3), the "swiper" terminal will output a voltage between 0.0V and 3.3V, depending on the position of the swiper. This can easily be wired into an analog input terminal that samples the voltage value.

Wire up a push button and a potentiometer as described above and shown in the following circuit:





We now have two inputs. Note that the pushbutton provides a digital input (D2) and the potentiometer provides an analog input (A2). We want to sample the states of those inputs and then output them to the serial port for debugging purposes. The following code shows one way to do this.

```
int ButtonPIN = D2;
int PotPIN = A2;

int PotOut = 0;
bool ButtonOut = FALSE;
int ButtonCount = 0;

void setup() {
    pinMode(ButtonPIN, INPUT_PULLDOWN);
    pinMode(PotPIN, INPUT);
    Serial.begin(9600);
}

void loop() {
    ButtonOut = digitalRead(ButtonPIN);
    PotOut = analogRead(PotPIN);

    if(ButtonOut == HIGH) {
        ButtonCount = ButtonCount + 1;

        Serial.print("Button Count = ");
        Serial.print(ButtonCount);
        Serial.print(" , Level = ");
        Serial.println(PotOut);
    }
}
```

Go ahead and run this code on your Photon, then connect to the Photon with your terminal program (VS Code, PuTTY, CoolTerm, etc.). Press the button a couple of times, turn the potentiometer, and press the button again. You should notice a couple of things immediately.

Events vs. State

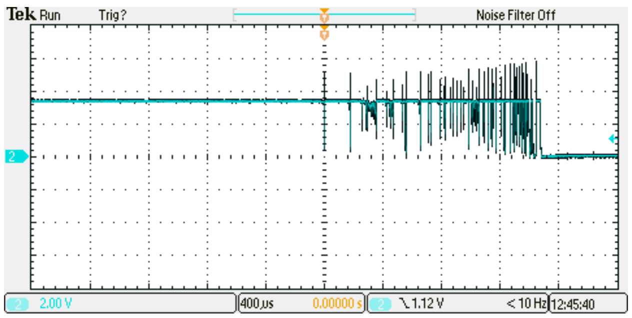
You may have noticed that pressing the button once may result in reporting more than one button count. In fact, the number of button counts depends on the speed of your microcontroller and the complexity of the code being run (not a good thing!).

In our case, we are interested in the event “Button is pushed,” not the current state of the button “Down.” Inherently the event “Button is pushed” requires knowledge of two pieces of information -- the previous state of the button “Up” and the current state of the button “Down.” We can build code to capture these pieces of information and build on them. For example:

```
ButtonNow = digitalRead(ButtonPIN);

if(ButtonNow == HIGH && ButtonLast == LOW) {

    //Do our work here;

    ButtonLast = HIGH;
} else if (ButtonNow == LOW) {
    ButtonLast = LOW;
}
```

Finally, to complete Exercise 5, modify the previous example to only send information to the serial port once per button press.

Debouncing



Often, the human interface is made more complex because the physical buttons used actually “bounce” several times before settling on a steady output value (see oscilloscope output on the right, which shows the voltage sampled from a button when it is pressed.) This can result in detecting multiple button presses for a single press or an artificial button press when the button is released. Usually, this bouncing lasts less than 2ms (but can sometimes last 10s of milliseconds).

The maximum execution rate of the loop function for a Photon device is 1 ms. This can mask bouncing effects very nicely. Unfortunately, this can sometimes result in a false sense of security. You can ignore debouncing as long as your App doesn’t:

- Use a switch other than the PCB mount momentary switch supplied by the ECE Depot
- Operate on the “release” of the PCB mount switch rather than the “press”
- Sample the same input multiple times in a single loop() function
- Operate your Photon in SYSTEM_MODE(MANUAL)

Exercise 6: Micro Project

Read through the device descriptions for the Speaker and the Servo Motor. Using what you've learned in this lab, write an app that senses something (a button, pot, temp, light, etc.) and somehow responds (speaker, servo, led, etc.). Use any of the actuators in this lab or devices you have figured out on your own. A couple of examples of potential micro-projects are:

- Automated Light: Light level rises → Turn off an LED lamp
- Automated Fan: Temperature rises → Turn on a fan
- Music Box: Press a button → play a song or multi-tone siren

While the choice of a specific micro project you do for this section of the lab is up to you, there are a couple of concepts that are useful for embedded systems design. Now is potentially a good time to read Quick Lesson - Programming Constructs. Take a look and see if it is relevant to you.

Lab Report

You are required to submit a written report on your Micro Project. This is only Exercise 6 and is the open-ended part of this lab. Your report should contain your well-commented code and a brief (no more than half a page) description of your project.

Be prepared to discuss the following:

- Your experience with each device (sensors, actuators) that you connected.
    - Did it work the first time you connected and programmed it? What mistakes did you make? How did you resolve any issues? If you didn't have any issues, just say so.
    - Include the relevant code you used to get it to work. When describing code, it is important to break the code into sections and describe how and why you chose the particular implementation you used.
    - For the sensors, include the table that shows the ADC (analog read) values under different conditions.

No submission is necessary for the other exercises, but the Lab TA must see a demo of your circuits. Make sure s/he checks off your work before you leave the lab (or, set up a time outside the lab to demo if you don't have time to finish).