EE 1301: Introduction to Computing Systems

IoT Laboratory #2

Getting Started with Sensors and Actuators
Background
As discussed in IoT Lab 1, a smart device interacts with the world with 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'll familiarize yourself with the inputs and outputs available on your Photon. Using these inputs/outputs we'll 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 (Phototransistor)
- Device Description - Temperature Sensor (TMP36)
- Device Description - Individually Addressable LEDs (8mm 2811 RGB LED)
- Device Description - Simple Speaker (Piezo speaker)

Pre-Lab Requirements
Before coming to lab there is a fair amount of reading material you should review. They 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. Then complete the Moodle quiz for lab 2 online.

Pre-Lab Checklist
- 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
- Read the second half of the SparkFun Tutorial - “How to read a schematic” https://learn.sparkfun.com/tutorials/how-to-read-a-schematic#name-designators-and-values
- Complete the Moodle quiz for “IoT - Lab 2”

Required Components:
- Phototransistor (Light Sensor)
- 200 Ohm Resistor
- 4.7k Ohm Resistor
- 100k Ohm Resistor
- 3 Individually Addressable LEDs (WS2811s)
Push button switch
Potentiometer
Decoupling Caps

Lab Procedure

First Sensor - Light Sensor
A light sensor can be extremely useful for future 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 phototransistor. A phototransistor conducts an amount of 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 we can read. The schematic in Figure 1, shows the sensing circuit in the upper right corner. As light hits the photo-transistor (Q1) it is converted into current. The current flows through the resistor dropping the voltage on the sense pin A1. Therefore, more light means a lower measured voltage.

Note: Infrared detecting LEDs (black) are originally designed to detect signals from IR remote controls; however they make for great sunlight detectors in projects (sunlight has a lot of infrared compared to light from fluorescent bulbs.)

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 sensors response and calibrate our final
code (see “Quick Lesson - Power Supplies, Analog, and Digital Pins” if the term “serial port” seems unfamiliar).

Procedure

1.) Wire up your Particle and the phototransistor as shown below (Note: Your phototransistor should be clear (all wavelengths) if you’re working in the computer lab. (You may use the tinted black phototransistor (infrared only) if you’re working in a space with sunlight.)

![Breadboard Layout of the test circuit for a phototransistor](image)

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

```cpp
int data0;
```

5.) **In the setup()** function we first need to setup our serial port

```cpp
// 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 the serial port is always available and the library is always pre-loaded by the compiler. As such we don’t 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 the Analog pin into the variable `data0`.

```cpp
// 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

```cpp
// 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 easily implement a heartbeat LED.
   a.) **In setup()**

```cpp
// Setup D7 pin to output a heartbeat
```

b.) **At the beginning of loop()**

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

c.) **At the end of loop()**

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

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

9.) **Save, Verify, and Flash your Code to your Particle.**
10.) When your Particle has reset and is “breathing cyan”, open PuTTY and connect to the appropriate serial port (i.e., COM3)
Take some data with the sensor to prove it is working (for example, position the sensor so it can see light, cover and uncover the sensor with a very small top hat.)

<table>
<thead>
<tr>
<th>Condition</th>
<th>DAC reading</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure: One possible version of AnalogRead2Serial.ino
Exercise 1 - Temperature Sensor

This time, on your own, build a serial evaluation setup for your TMP36 temperature sensor using the “Device Descriptions / Temperature Sensor” document (should be available on Moodle).

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

<table>
<thead>
<tr>
<th>Condition</th>
<th>DAC reading</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

NOTE: If you are only doing the sensors for this week, look at the guidelines at the end of this document on how to write the lab report.

Actuators - Display Elements

In this section we'll work on getting some information out of the microcontroller. Today 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

These LEDs 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 rest of the data on, etc.

If you want to see what regular LEDs are capable of check out the news report and YouTube of CSE’s Winter Light Show this year. Imagine what you could do with an RGB LED!

https://www.youtube.com/watch?v=1zJrMUFZtwI
Pin Description
Note: These devices need **+5V power supplies**.

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

For now wire up your LEDs as follows:

**Figure 1: Example Circuit Wiring Diagram**

**Libraries**
Libraries are collections of code that is commonly utilized. They are usually maintained by companies or interested individuals. They make complicated tasks easier. In the spark development environment they are handled in a special way.
1.) Go back to the build.particle.io development environment. Start by creating a fresh App
(click the Code button, click the **CREATE NEW APP** button.)

2.) To view a list of supported libraries, click on the **Libraries** button.

3.) In this example we’ll be using a library called the NEOPIXEL library. Click on this library.
Your current program window will be temporarily hidden and the library code will be
brought up for you to review. Note that there are tabs at the top of the window, a library
often consists of several files, including example code for you to review. Take a moment
to look around the files.

4.) When you are finished reviewing the code, click the **INCLUDE IN APP** button, select your
newly created app and click **ADD TO THIS APP**.

5.) You’ll notice a couple things:
   a.) First the following lines were added to your main file

```cpp
#include "neopixel/neopixel.h"
```
   b.) Second, on the left pane you’ll notice a section labeled, “Included libraries”, that
includes “NEOPIXEL”. This is important as the library contains a number of files
that are required for your program to function.

6.) Your app should now look something like this:

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. An **object** is an
instance of a class that contains unique embedded data and functions. Each of these
embedded functions is called a **method**.

We will need some basic information in order to set up the NeoPixel object:

- The PWM capable **digital pin number** (D4) to which the string of pixels is attached
- The **number of pixels** (3) in the string
- The **type of controller chip** (WS2811, https://www.adafruit.com/products/1734)
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. Like so:

```c
// IMPORTANT: Set pixel COUNT, PIN and TYPE
#define PIXEL_PIN D4
#define PIXEL_COUNT 3
#define PIXEL_TYPE WS2811

Adafruit_NeoPixel strip = Adafruit_NeoPixel(PIXEL_COUNT, PIXEL_PIN, PIXEL_TYPE);
```

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

In C (and many other programming languages), we access the methods of an object with the syntax “object.method()”. Think of these methods simply as functions. We then call the method “strip.begin()” to initialize the strip. We only need to do this once, so the best place for it is in the setup() function of our Photon code. As so:

```c
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.

<table>
<thead>
<tr>
<th>Method (Function) Definition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void = strip.setPixelColor(&lt;uint16&gt;,&lt;uint32&gt;)</td>
<td>Store the PackedColor (a 32-bit ‘packed’ RGB integer) for the n-th PixelID (zero indexed integer) in internal memory</td>
</tr>
<tr>
<td>Example: strip.setPixelColor(PixelID, PackedColor)</td>
<td></td>
</tr>
<tr>
<td>&lt;uint32_t&gt; = strip.Color(&lt;uint8&gt;,&lt;uint8&gt;,&lt;uint8&gt;)</td>
<td>Returns a 32-bit ‘packed’ integer representing the RGB values (ie., Red<em>256^2 + Blue</em>256 + Green)</td>
</tr>
<tr>
<td>Example: PackedColor = strip.Color(Red,Blue,Green)</td>
<td></td>
</tr>
<tr>
<td>void = strip.show()</td>
<td>Send stored configuration to LED strip in single burst transmission</td>
</tr>
<tr>
<td>Example: strip.show()</td>
<td></td>
</tr>
</tbody>
</table>

We will define a couple 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.

---

1 You can actually see the definition of this method in the file “neopixel.c” on line 699. This is a useful place to look if you need to figure out how an undocumented library functions.
void loop() {
    //Setup some colors
    int PixelColorBlue = strip.Color(0, 0, 128);
    int PixelColorRed = strip.Color(80, 0, 4);
    int PixelColorGold = strip.Color(60, 50, 5);
    //Set first pixel to blue
    strip.setPixelColor(0, PixelColorBlue);
    //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
    //set second and third pixel to Gopher Gold! and Red
    strip.setPixelColor(2, PixelColorRed);
    strip.setPixelColor(1, PixelColorGold);
    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 continually update them.

**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, there are several pitfalls. We’re going to start by looking into 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 following circuit. 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 very 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 supplies (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.

Now wire up a push-button and a potentiometer as described above and shown in the following circuit:

So, we now have two inputs. We want to grab the state of those inputs and output them to the serial port for debugging purposes. Rapidly throwing together some code, it might look like this:

```cpp
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 (PuTTY, CoolTerm, etc.). Press the button a couple times, turn the Pot. setting and press it again. You should notice a couple things immediately.

Events vs. State
You may have noticed above that pressing the button once, results in a "lot" of button counts. In fact, the number of button counts is actually dependant on the speed of your microcontroller and the complexity of the code being run (Not a good thing!)

In our case, we are really interested in the event “Button is pushed” not really 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:

```cpp
ButtonNow = digitalRead(ButtonPIN);

if(ButtonNow == HIGH && ButtonLast == LOW) {
    //Do our work here;
    ButtonLast = HIGH;
} else if (ButtonNow == LOW) {
    ButtonLast = LOW;
}
```

Modify the previous example to only dump information to the serial port once per button press.

Debouncing
Often times Human-Interface is made more complex because the physical buttons used actually “bounce” several times before settling on outputting a value (see oscilloscope output on the right.) 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 last 10’s or even 100’s of milliseconds.)

The Photon has an internal loop max execution rate of 1ms. This can mask bouncing effects very nicely. Unfortunately, this can sometimes results in false sense of security. So, if your App:

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

you will need handle debouncing in your project.
otherwise you can probably ignore it for now.

The PCB mount momentary switches included in your kit,
paired with the processing time of the loop() function are
fairly robust, but it’s still possible to run into problems.

Exercise 2 - Micro Project

Read through the device descriptions on 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
responds (speaker, servo, led, etc.). Use any of the actuators in this lab or devices you have
figured out on your own. A couple examples of potential micro-projects are:

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

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, give it a browse and see if it is
relevant to you.

Lab Report

Put together a very brief report on the work in this lab. It should include the following:

● A paragraph on your experience with each of the devices (sensors, actuators) that you
  connected.
  ○ Did it work the first time you connected it and programmed it? What mistakes did
    you make? How did you resolve the issue. 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
    to method of implementation you used.
  ○ For the sensors, include the table that shows DAC values under different
    conditions.
● A description of your micro project.
● The Lab TA must see a demo of your circuits. Make sure s/he checks off your work
  before you leave the lab (or, setup a time outside the lab to demo if you don’t have time
  to finish).