Jan 2021

Embedded Systems & IOT: Photon Labs

James Peyton Jones. James.peyton-jones@villanova.edu
Professor, Electrical & Computer Engineering, Villanova, PA 19085.

Contents

Introduction

The ParticleIO Photon is a low-cost ($19), yet remarkably powerful embedded target which includes a 32-bit ARM-M3 120 MHz processor, with 128 KB RAM and 1 MB flash memory. It has on-board wifi, and USB communications as well as 18 General Purpose Input/Output (GPIO) pins and standard interface drivers including I2C, I2S, 2xSPI, UART, CAN, 8xADC, 2xDAC and 9xTimer/PWM. As such it is a cutting edge device for cyber-physical system designs linked to the Internet of Things (IOT). This series of lab exercises are designed to use the device as a platform to learn how to build embedded systems, and to learn how they work.

The labs are designed around a small breadboard shown in Fig. 1 below which includes:

- Two (active high) 0 – 3.3V **pushbutton** inputs.
- One (blue) 0 – 3.3V **potentiometer** input.
- One Open Collector (ie. active low) **Reflective Light Sensor** (RLS)
- Two (active high) **LED** outputs, one onboard (D7), one external.
- One **buzzer** output
- One external **transistor** drive (used for example to drive a small dc motor also included in the kit).

Note: The board uses the following wire color coding:

- Red (**Left side**) = **5**V; Red (**Right side**) = **3.3V** ← **CARE!** Not to confuse or connect these together!!!
- **Black** = 0V Ground reference.
- **White** = Inputs to the Photon
- **Yellow** = Outputs from the Photon

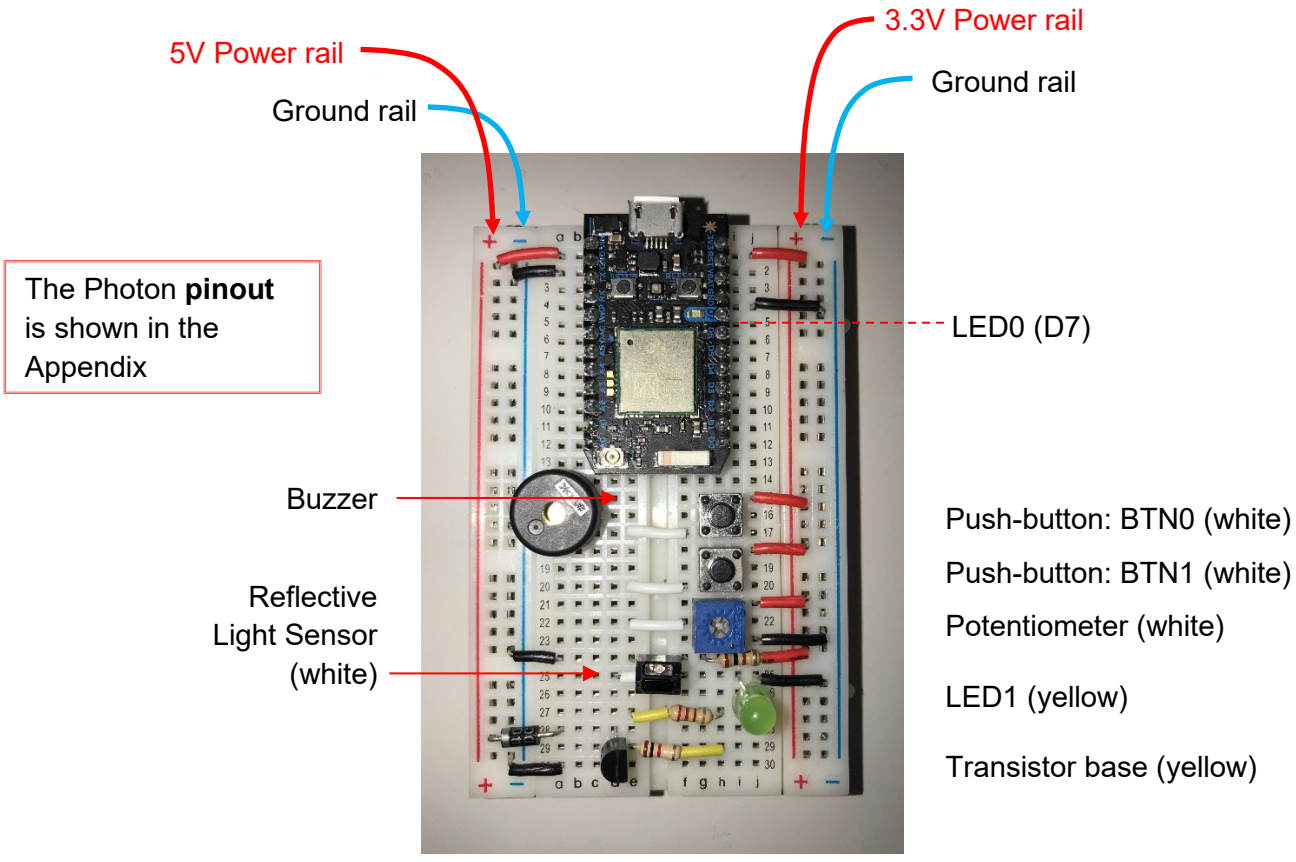


Fig. 1: Photon breadboard

1. Getting Started – Lab1a (onboard LED)

Many of the steps required to set up a Particle user account, and to configure the photon have already been completed for you. The aim of this section is to prove everything works by getting a simple program to flash an LED compiled and running on your device ☺!

1. Install **Particle Dev**, and Integrated Development Environment (IDE) for editing programs and flashing them to the photon via the cloud. You can download the IDE from here (ignore that it says it is discontinued): https://www.particle.io/products/development-tools/particle-desktop-ide
   - OR use the web-based IDE called **Particle Build** – see instructions in Appendix
2. From your start menu, start **Particle Dev** which brings up the Integrated Development Environment (IDE).
3. Under the rightmost menu-bar item, select **Particle**>**LoginToParticleCloud** and enter your account details:

   Email address: **<your VU email address>** … eg. username@villanova.edu
   Password: **photonXX** ….where **XX** denotes the number of your device, eg. **50**, 51, etc.

   Your email address should then appear in the bottom 'status' strip of your IDE window => logged in ok.
4. Under the rightmost menu-bar item, select **Particle**>**SelectDevice** which brings up a list of devices 'owned' by the account. In this case you own only one device, previously named as **photonXX**. Click on it. The device name should appear in the bottom 'status' strip with a colored dot which matches the device's LED.
5. Click on the **StartNewProject** icon  , then name your project **Lab1a** and specify an appropriate **parent directory** on your machine for the project files to be saved.
6. A new project directory called **Lab1a** should then appear underneath the parent directory. A **Lab1a.md** file describing all the files within the project may also open automatically, but you can **File**>**CloseTab**, or **Ctrl+W** to close it.
7. In the Project directory tree pane, click on the **/src** sub-directory and then click on the **Lab1a.ino** file in order to edit the Lab1a.ino code template for your program that has been created for you. Fill in your details in the header comment section at the top, and then **edit** the remainder as follows:

```
#define LED0 D7                 // now we can call pin D7 by the more meaningful name 'LED0'

void setup() {
  pinMode(LED0, OUTPUT);        // sets LED0 pin D7 mode to Output
}

void loop()  {
    digitalWrite(LED0, HIGH);   // turn LED0 on
    delay(1000);                // wait 1 second
    digitalWrite(LED0, LOW);    // turn LED0 off
    delay(1000);                // wait 1 second
}
```

8. Always **File>Save** or **Ctrl+s** to save your program, *then* click the **CompileInCloud** icon  to send your saved file to the cloud for compilation. You can watch progress in the bottom 'status' strip of your IDE window.
9. If successful a ***.bin** compiled executable should appear in your /src directory. If not successful, a list of **errors** will be shown. Try deliberately inserting an error (eg. missing semicolon, or misspelling), **save**, and recompile to see this…

10. Your program is now ready to be flashed to your photon. Check your photon is powered on and connected to the network (ie. '**breathing cyan**' on the central led, and in the bottom status strip of your IDE). Click the **Compile&UploadCode** icon to recompile and flash your code to your device. Your photon may 'blink magenta' while flashing is in progress, and once complete will immediately start executing your program ☺!
11. Modify your code to make the LED flash **more quickly** (eg. every 1/5 second), re-compile, and re-flash to see your modified code execute on the device. [Use this new flash-rate for subsequent exercises, eg. Lab1b].
12. **Optional** challenge: How could you modify your loop() code, to achieve the same result only with half the lines of code – ie. get it to flash with only two, rather than four, lines of code within the main loop ?
13. Check you program has a well-documented 'header' and nicely-aligned meaningful comments (see grading rubric in appendix). Demonstrate your program to your instructor (sign-off), **change** the file extension to *.**cpp** and submit it to Blackboard as a *.cpp file – (this allows Blackboard to display submitted files).

1.1 Simple Digital Output – Lab1b (external LED)

Embedded systems are fundamentally digital, so it is important to be able to input and output digital/logic signals. You are already doing this by writing to the onboard LED0 on pin D7. The aim of this lab is to extend this by writing to an external **LED1** on your board.

1. Read the firmware documentation for the following functions: **pinMode**(); **digitalWrite**()
   https://docs.particle.io/reference/device-os/firmware/photon/#input-output
   https://docs.particle.io/reference/device-os/firmware/photon/#digitalwrite-
2. Create a new project using **File>NewWindow**. Then click **StartNewProject** (), and name the new project **Lab1b**. In the Project directory tree pane, select the **/src/ Lab1b.ino** file for editing and copy/paste your solution from **Lab1a** as the start point for this exercise.
3. Modify the previous program so that it also flashes an external **LED1** connected to pin **D0** at a **200 ms** rate. Your external LED1 and the onboard LED0 should turn on and off together.
4. Use another jumper wire to **connect** pin D0 output to the LED1 input (**yellow** wire on the **left** side of the board). Then compile, flash and **test** your program.
5. Check you program has a well-documented 'header' and nicely-aligned meaningful comments. Demonstrate your program to your instructor (sign-off), and submit it to Blackboard as a ***.cpp** file.

1.2 Simple Digital Input – Lab1c (push-button input)

Embedded systems often use digital inputs from push-buttons to control the operation of the system (like the buttons on a digital watch for example). The aim of this lab is to read a pushbutton input and use it to control when flashing occurs.

1. Read the firmware documentation for the following functions: **pinMode**(); **digitalRead**()
   https://docs.particle.io/reference/device-os/firmware/photon/#digitalread-
2. Create a new project using **File>NewWindow**. Then click **StartNewProject** (), and name the new project **Lab1c**. In the Project directory tree pane, select the **/src/ Lab1c.ino** file for editing and copy/paste your solution from **Lab1b** as the start point for this exercise.

3. Assuming a push-button **BTN1** will be connected to pin **A1**, modify your code so that **LED0** and **LED1** will flash as before, but only if **BTN1** is pressed (and held pressed-down).
   - Read the push-button status into a variable called **flashEnabled** and only execute your LED flashing code **if** this flag is 'set' true, (ie. if its value is logic 1).
   - Note **BTN1** is **active-high** (ie. goes high when pressed). When not pressed the signal is not driven…
   - You therefore need to configure the **BTN1** input with an appropriate **input pull-up** or **pull-down** resistor .
   - You may be tempted to use a **while (**flashEnabled**)** loop. Why is this a bad idea? – see notes p. 25
4. Connect the white output from the 'lower' **push-button1** on your protoboard to pin **A1**, and ensure that output pin **D0** is still connected to the (yellow) input of **LED1**. Compile, flash, and **test** your program.
5. Check you program has a well-documented 'header' and nicely-aligned meaningful comments. Demonstrate your program to your instructor (sign-off), and submit it to Blackboard as a ***.cpp** file.

1.3 Simple Digital Input - Lab1d (reflective light sensor)

Pushbuttons are not the only digital inputs. Other sensors / alarms may also provide digital inputs to your system. The aim of this lab is to enable flashing when ***either*** BTN1 is pressed ***or*** when an (active low) Reflective Light Sensor (**RLS**) detects the presence of an object.

1. Study the RLS circuit diagram below and (optionally) read the RLS **datasheet** posted on Blackboard.
2. Create a new project using **File>NewWindow**. Then click **StartNewProject** (), and name the new project **Lab1d**. In the Project directory tree pane, select the **/src/ Lab1d.ino** file for editing and copy/paste your solution from **Lab1c** as the start point for this exercise.
3. Assuming the **RLS** will be connected to pin **A4**, modify your code so that **LED0** and **LED1** will flash either if BTN1 is pressed (as before), or if the sensor detects a nearby object.
   - Note: the **RLS** has an **Open-Collector** output which can only pull **low** when it detects reflected light. You therefore need to set the input **pull-up** or **pull-down** mode accordingly
   - Read the RLS sensor status into a variable called **RLSval** and modify your logic so that the LED flashing code only executes your LED flashing code if BTN1 is pressed or **RLSval** indicates an object is present.
4. Connect the white output from the Reflective Light Sensor to pin **A4**, and ensure that output pin **D0** is still connected to the (yellow) input of **LED1**. Compile, flash, and **test** your program. The LEDs should flash if you put your hand close to the sensor (within an inch or so) or if you press BTN1.
5. Check you program has a well-documented 'header' and nicely-aligned meaningful comments. Demonstrate your program to your instructor (sign-off), and submit it to Blackboard as a ***.cpp** file.

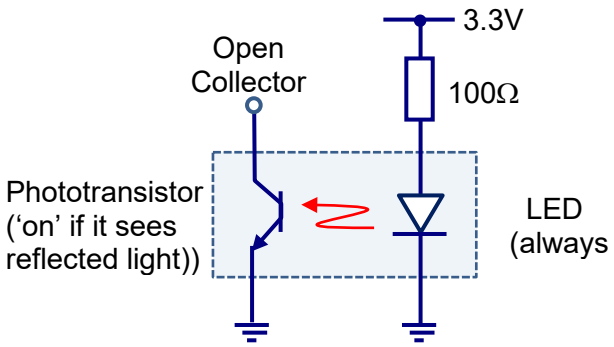


Fig. 2: Reflective Light Sensor (RLS) Circuit

2. 'Analog' Input / Output

Digital I/O is great for turning things on/off, or measuring if things are on/off, but many sensors output an analog signal. Also it is often useful to be able to turn things partially on, like a dimmed light or a motor that is not running at full speed. While a true analog output is possible using a DAC, a good approximation can be achieved using a Pulse-Width-Modulated (PWM) output signal.

2.1 ADC Input and PWM Output – Lab2a (pot-controlled LED & Motor)

The aim of this lab is to use an Analog-to-Digital Converter (ADC) to read the voltage from a potentiometer, and use it to control the brightness of an external LED, and the speed of a small dc motor, by regulating the duty cycle of a PWM output signal – see Fig. 3.

1. Read the firmware documentation for the following functions: **analogRead**(); **analogWrite**()
   https://docs.particle.io/reference/device-os/firmware/photon/#analogread-adc-
   https://docs.particle.io/reference/device-os/firmware/photon/#analogwrite-pwm-
2. Create a new project using **File>NewWindow**. Then click **StartNewProject** (), and name the new project **Lab2a**. In the Project directory tree pane, select the **/src/Lab2a.ino** file to edit your program.
3. Assuming pin **A5** is going to be used as an **ADC** and that **LED1** is connected to pin **D0**, write a program so that the brightness of **LED1** is controlled by the voltage measured by the **ADC**.
   - Note that you do **not** need to set the pinMode for analog inputs, but you **must** do so for PWM outputs.
   - Read the value from the ADC into a variable named **adcVal**, and then use this value to control the PWM duty cycle.
   - Note that the maximum output from analogRead is **4095**, but the maximum input to analogWrite is **255**, so you will need **scale** your **adcVal** before you output it to **LED1.**
4. Connect the white output signal from the square **blue potentiometer** on your protoboard to pin **A5** of the photon, and connect output pin **D0** to the (yellow) input to the **LED1**. Compile, flash, and **test** your program. Try turning the potentiometer with a screwdriver. Does the LED change brightness?
5. Make a tiny edit to your program in order to drive the on-board LED (pin **D7**), rather than pin D0. Re-compile, flash and **test**. Does it still work? Can you explain what is happening? Revert back to using **D0**, but add a **comment** to your program explaining why you use D0 rather than D7.
6. The PWM digital output can drive more than one small load, providing the total current load is less than $\pm$**25 mA**. However this is insufficient to drive something bigger, like a motor. To deal with this issue, connect your PWM output **D0** to both the LED (as before) *and* the (yellow) **base of the transistor** on your board. Then connect a **motor** between **+5V** power (on the left side of your board) and the **collector** of the transistor (ie. put the motor in parallel with the flyback diode on the left side of your board as show in Fig. 3).
7. Try turning the potentiometer with a screwdriver. Does the motor change speed?
8. Check you program has a well-documented 'header' and nicely-aligned meaningful comments. Demonstrate your program to your instructor (sign-off), and submit it to Blackboard as a ***.cpp** file.

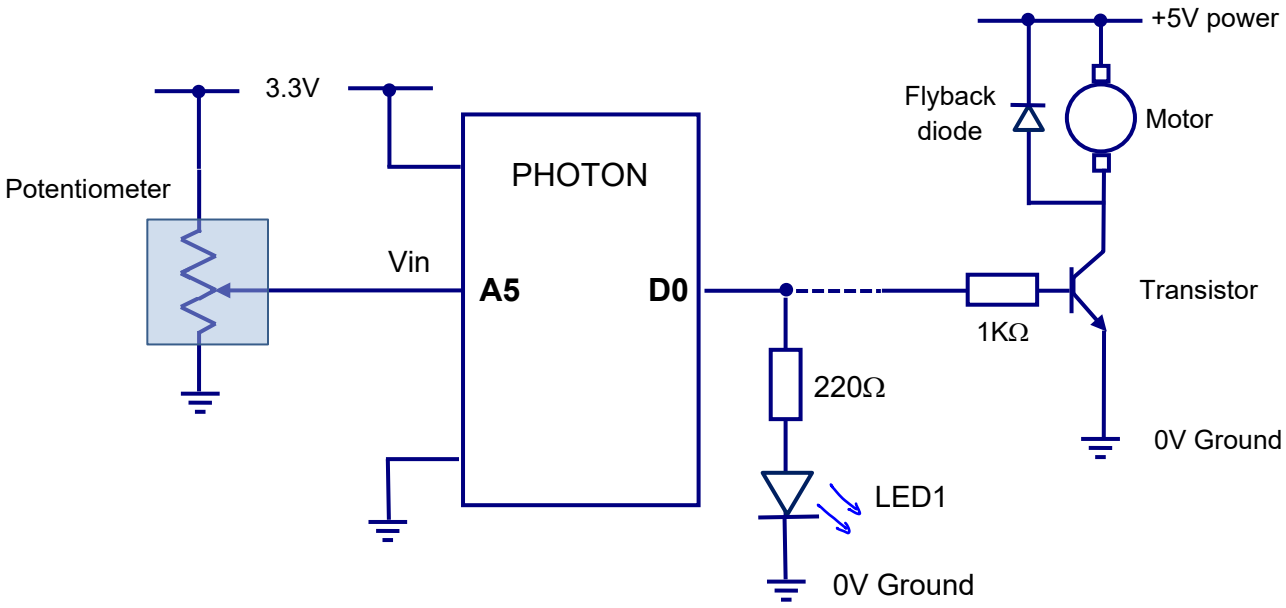


Fig.3: Circuit diagram for Lab2xx

2.2 PWM, Servo, and Tone Outputs – Lab2b (pot-controlled LED, Servo, Tone)

The aim of this lab is to use the potentiometer input to control not only the **LED** brightness, but also the angular position of a small **servo** motor, as well as the frequency or **tone** of a buzzer.

1. Read the firmware documentation for the **tone**() function, as well as the functions within the **Servo** class: https://docs.particle.io/reference/device-os/firmware/photon/#tone- https://docs.particle.io/reference/device-os/firmware/photon/#servo
2. Create a new project using **File>NewWindow**. Then click **StartNewProject** (), and name the new project **Lab2b**. In the Project directory tree pane, select the **/src/Lab2b.ino** file for editing and copy/paste your solution from **Lab2a** as the start point for this exercise.
3. First modify your program so that it controls the tone / frequency of the buzzer on your board (as well as the brightness of **LED1**), assuming the **BUZZER** is connected to pin **A7** (also labeled as WKP).
   - You should call the **tone** function in such a way that the tone sounds **continuously**.
   - Note that the output from analogRead is in the range **0** – **4095**, and the range of frequencies accepted by the tone() function is **20 Hz** – **20 kHz**, so you may need to scale/shift your **adcVal** before sending it to the **BUZZER.**
4. Use another jumper wire to **connect** the **A7** / **WKP** output pin to the Buzzer input (on the **left** side of the board). Also ensure the **potentiometer** is still connected to **A5**. Then compile, flash and **test** your program. Does the frequency and LED brightness change together as you turn the potentiometer?

5. Now further modify your program so that it controls the angular position of a servo motor (as well as the brightness of **LED1** and the tone of the **BUZZER**!), assuming the **SERVO** is connected to pin **D2**.
   - You will need to declare a **Servo** object called **myServo** and attach it to the **SERVO** pin.
   - Note that the output from analogRead is in the range **0** – **4095**, and the range of angles accepted by the myServo**.write()** function is **0** – **180** degrees, so you will need to scale your **adcVal** before sending it to the **SERVO.**
6. Borrow a servo motor from the instructor, and carefully connect it to your board:
   - First connect the **Brown** servo motor wire to the **Black/Ground** of your breadboard
   - Then connect the **Red** servo motor wire to the **5V** line on the **Left** side of your breadboard.
   - Finally connect the **Yellow** servo motor wire to the **SERVO** output pin **D2**
7. Compile, flash and **test** your program. Do the servo position, frequency and LED brightness all change together as you turn the potentiometer?
8. Check you program has a well-documented 'header' and nicely-aligned meaningful comments. Demonstrate your program to your instructor (sign-off), and submit it to Blackboard as a ***.cpp** file.

2.3 PWM, Servo, and Tone *Signals* – Lab2c (oscilloscope waveform analysis)

The aim of this lab is to use the **Oscilloscope** to compare and contrast the square wave signals that are generated by **PWM**, **Servo** and **Tone** functions in your **lab2b** solution. These signals are ultimately defined by their period, high-pulse-width, and duty cycle, but they differ according to which of these parameters act as variables, and which remain constant. You will **report** your results in a Word document. A brief review of oscilloscope controls can be found in the **Appendix**.

Please ensure this project is entirely your own work. If you need help, ask an instructor. Do not ask for, or give any help to a classmate.

It is not possible to run this lab as originally intended since it requires use of oscilloscopes etc. Instead, download the Lab2c Oscilloscope results from the Blackboard > Lab Assignments folder and continue as if you had made these measurements yourself. You will need to read the instructions so you understand how these results were obtained, and then use them to write and submit a report as detailed below.

1. Check that **potentiometer** is connected to input pin **A5**, and that output **D0** is connected to **LED1**. Then compile, and flash your **lab2b** solution and check that the LED1 brightness changes as you turn the potentiometer. [It is not necessary to have the servo and buzzer connected, providing the **BUZZER** tone signal is being outputted on pin **A7** / WKP, and that the **SERVO** signal is being outputted on pin **D2**],
2. Write a **section** of your report for each type of signal (**PWM**, **Servo**, **Tone**) as described below. In each case:
   - Observe the signal on the oscilloscope, and describe how it changes as you turn the potentiometer.
   - Clearly state what is the **primary** variable that changes as you turn the potentiometer
   - …and also state what other signal-defining parameters **Do Not** change (significantly) in this case.
   - Take at least **two** photo's / screen-capture snapshots to illustrate your description (paste in your report)
   - …and use oscilloscope **cursors** to take **quantitative** measurements of these waveform snapshots.

   More specific instructions are provided below…

3. **PWM**: Connect **Channel1** of the oscilloscope to the LED1 / **PWM** output on pin **D0**, [and connect the oscilloscope **ground** to the ground of your breadboard]. Hit **Autoscale** on the oscilloscope, and then fine-tune the display so that:
   - The x-axis time-scale is **1ms** / division.
   - The y-axis scale is **2V** / division
   - The rising edge of the waveform is nicely lined up with the display grid
   - The ground level of the waveform is nicely lined up with the display grid

   Now **describe / measure** your waveform as specified in Step #2 above. In this case, for example, your two snapshots might include a relatively low duty cycle, and a relatively high duty cycle. Your quantitative measurements should include all measurements necessary to compute the precise duty cycle in each case. Do not simply rely on the oscilloscope's ability to compute the duty cycle for you.
4. **SERVO**: Now also turn on **Channel2** of the oscilloscope, and hook it up to the **SERVO** output signal on pin **D2**. Use the **same** x- and y-axis scales as before, but this time set the oscilloscope to **Trigger** using **Channel2** as the Source. **Describe / measure** your waveform as specified in Step #2 above. Also **contrast** your results with those obtained for the PWM signal (still shown on Channel1).
5. **TONE**: Now move the **Channel2** probe to the **BUZZER** tone output on pin **A7** / WKP. Use the **same** x- and y-axis scales as before, but reset the **Trigger** so it uses **Channel1** as the Source. The waveform may appear to be a bit jumpy (because in this case the two counter-timers are running independently) so you may find it helpful to freeze / unfreeze the display by pressing the **Run / Stop** button on the oscilloscope.
   Describe / measure your waveform as specified in Step #2 above. Also **contrast** your results with those obtained for the PWM signal (still shown on Channel1).
6. Write brief **introduction** and **conclusion** sections of your report, outlining the purpose of this work, the main results obtained, and any other practical details or additional observations. **Submit** on Blackboard.

3. Robot Control Project

The aim of this project is to apply the techniques learned in Section 1, and Section 2, to control a small mobile robot. In particular you will interface the Photon to a **TB6612 H-bridge** driver to control the speed and direction of the Left and Right motors of the robot in order to steer it as desired. **Please ensure this project is entirely your own work. If you need help, ask an instructor. Do not ask for, or give any help to a classmate**.

3.1 Part1: A function to 'drive' the robot motors fwd/back at specified speed

1. Create a new project using **File>NewWindow**. Then click **StartNewProject** (), and name the new project **Robot1**. In the Project directory tree pane, select the **/src/Robot1.ino** file to edit your program.
2. Read the **TB6612 H-bridge** datasheet which is posted on Blackboard. You will have to extract the information you need (ignore the rest). Note **STBY** is permanently wired High.
3. Use the following pin definitions and connections for your program (below). Remember to initialize pins within **setup()** using pinMode() as necessary:
   - **D1**: LeftMotorPWM (**PWMA**)
   - **D2**: RightMotorPWM (**PWMB**)
   - **Ground**
   - **D3**, **D4**: LeftMotorControls (**In1A**, **In2A**)
   - **D5**, **D6**: RightMotorControls (**In1B**, **In2B**)
   - **5V** Power ← Care! This is the **Left** side power of your breadboard!

4. Now write a function **drive(int LeftSpd, int RightSpd)** where the two arguments, **leftSpeed** and **rightSpeed** should set the left and right motor speeds ***and*** their direction according to the specified speed arguments, ie. Positive values mean the corresponding motor should rotate forwards and negative values should mean it rotates backward. [Setting the motors to drive at identical speeds will make the robot go straight. Setting one faster than the other will cause it to turn…]
5. Check your function and setup code to your instructor, before proceeding to Part 2 where you will call your drive function to get the robot to do smart things! [No need to submit to Blackboard yet].

3.2 Part2: Calling your 'drive' function for Line-following robot control

The robot is equipped with a down-facing Reflective Light Sensor (RLS) like the one you have on your breadboard. The aim is now to use this RLS to design a robot that will follow a black line painted on a white surface, using the RLS to differentiate between the black and white surfaces.

1. Continue to develop your code from Part1. Use the same pin definitions and connections as before, but add the following:
   - **A4**: Reflective Light Sensor input (**White**)
   - **A5**: Potentiometer input
2. In the main loop of your program, write code for your robot controller as follows:
   - Use analogRead() to read the reflected light intensity into a variable called **RLSval**, and compare this to the **ADCval** read from the potentiometer input. If the RLSval is **less than** ADCval, then the RLS is considered to be seeing a white floor. Otherwise, it is considered to be seeing the black line. You can **adjust** the black / white decision threshold by adjusting the potentiometer ☺!.
   - For debugging purposes, **LED0** (D7) should be lit whenever the RLS sees the **black** line, and **LED1** (D0) should be lit whenever the RLS sees the **white** floor.
   - If the RLS sees a **white** floor, the robot should move forwards while turning **left** (to seek out the black/white edge of the line). If it sees a **black** floor, it should move forwards while turning **right** (again to find the black/white transition). In this way the robot will always follow black/white line edge.
   - To achieve these forward-moving turns, call your **drive()** function to set each motor to have a nominal **baseSpeed** of **25%** but add a **diffSpeed** of **5%** to one motor, and subtract it from the other in order to turn in the desired direction while continuing to move forwards.
3. Compile and flash your program to the Photon before connecting it to the robot.

   We will try to schedule times for you to test your robot in CEER-206, Tues 6:30-7:30 pm, or Thur 5:00-7:30pm, but if not you can show your code to your instructor who can verify if it would work or not. If you are not able to test your code in person we *may* be able to do so for you and send you a video of the results.
4. Disconnect power from your photon. Then connect your board to the robot as indicated in **part 1, step #3** and **part2 step #1**. Connect the **Ground first**, and the **5V power last**. The Photon should re-boot, connect, and then start executing your program as soon as you supply the 5V power. To **power-off** either pull out the 5V line, or double click the battery 'on/off' button. Please **do not** disconnect the robot USB lead.
5. Disconnect power from your photon. Then connect your board to the robot as indicated in **step #2**. Connect the **Ground first**, and the **5V power last**. The Photon should re-boot, connect, and then start executing your program as soon as you supply the 5V power.

6. Hold the robot with its wheels just **off** the ground, first with the sensor over white ground, and then over the black line. Use the potentiometer to **adjust** the RLS threshold so that the robot reliably detects the black / white surfaces, as indicated by your **LED's**. Then place the robot on the ground and watch it go ☺!
7. To **power-off** either pull out the 5V line, or double click the battery 'on/off' button. Please **do not** disconnect the robot USB lead.
8. Check you program has a **well-documented** 'header' and nicely-aligned meaningful **comments**. Demonstrate your program to your instructor (sign-off), and submit it to Blackboard as a ***.cpp** file.

4. Dealing with Events & Interrupts

4.1 A first attempt: The need for interrupts – Lab4a (toggling flashing on/off)

In Section 1.2 (Lab1c) you wrote a program which flashed both LED's when **BTN1** was pressed (and *held* on). Now we want to modify the program so that pressing (and *releasing*) **BTN1** toggles the flashing behavior of the LED's **on** or **off**. To do this you will need to detect the **0 → 1 event** that occurs as the button is pressed.

1. Create a new project using **File>NewWindow**. Then click **StartNewProject** (), and name the new project **Lab4a**. In the Project directory tree pane, select the **/src/Lab4a.ino** file for editing and copy/paste your solution from **Lab1c** as the start point for this exercise.
2. Modify your program to **toggle** the flashing behavior **on/off** when **BTN1** is pressed momentarily:
   - In order to catch when **BTN1** goes from **0** to **1** you will have to declare two new variables, **btn1Status** and **btn1PrevStatus**, initializing them appropriately.
   - Each time **loop()** is called you should use these variables to read / poll the **current** status of **BTN1**, and compare it with its **previous** status the last time around.
   - If this comparison shows that a **0 → 1 event** has occurred then toggle the value of **flashEnabled**. The flashing of your LED's should then respond as desired without any further code modification.
3. Make sure your external **LED1** is connected to **D0** with a jumper wire. Then compile, flash, and test your program. Does it behave as desired? Does it always capture the button-press if pressed only very **briefly**? Include your observations as a **comment** within the 'Description' section of your program header.
4. Check the rest of your program is **well-documented** with nicely-aligned meaningful **comments**. Demonstrate your program to your instructor (sign-off), and submit it to Blackboard as a ***.cpp** file.

4.2 Using a hardware interrupt – Lab4b (easier toggling of flashing on/off)

The aim of this program is to write a much more effective / elegant program to do the same thing as the previous program, only this time using a hardware interrupt and associated Interrupt Service Routine (ISR).

1. Read the firmware documentation for the following functions: **attachInterrupt**(); https://docs.particle.io/reference/device-os/firmware/photon/#attachinterrupt-
2. Create a new project using **File>NewWindow**. Then click **StartNewProject** (), and name the new project **Lab4b**. In the Project directory tree pane, select the **/src/Lab4b.ino** file for editing and copy/paste your solution from **Lab4a** as the start point for this exercise.

3. Modify your program as follows:
   - Write a very brief function **void toggleFlashEnabled()** to be used later as an interrupt handler when BTN1 is pressed. The function should simply toggle the value of **flashEnabled** when it is called.
   - Also add the keyword **volatile** to your declaration of **int flashEnabled** to warn the compiler that your toggleFlashEnabled interrupt handler could modify the contents of flashEnabled at any moment.
   - In the **setup**() function **attach** your **toggleFlashEnabled** function to a **Rising** edge interrupt on **BTN1**. Pressing the button will then automatically cause your interrupt handler to be called ☺!
   - You no longer need to poll the **btn1Status** in your main program loop. Both the code and variables used when trying to catch the **0 → 1 event** are no longer needed!
   - Your main program **loop()** function can now revert to something closer to the original program you developed in **Lab1c**.
4. Make sure your external **LED1** is connected to **D0** with a jumper wire. Then compile, flash, and test your program. Does it behave as desired? Does it always capture the button-press if pressed only very **briefly**? Include your observations as a **comment** within the 'Description' section of your program header.
5. Check the rest of your program is well-documented with nicely-aligned meaningful **comments**. Demonstrate your program to your instructor (sign-off), and submit it to Blackboard as a ***.cpp** file.

4.3 Using timer interrupts – Lab4c (dimmable flashing of LED1)

Ok, so now our main program **loop()** is busy flashing the LED, while **BTN1** can be used to turn flashing on / off. Suppose now we want to flash LED0, while *simultaneously* using the **potentiometer** to control the brightness of the external **LED1** to make it dimmable (as we did in Section 2.1, **Lab2a**).

The problem is the program cannot respond to the potentiometer during the **delay**() period which is needed for the LED timing. The aim of this program is to use the onboard timer-counters to generate **timing events** / interrupts to resolve this problem.

1. Read the firmware documentation for instantiating a **Timer** object by calling its **timer()** constructor to generate **timing events**, and to **callback** a user-specified interrupt service routine when these timing events are triggered. Also read about the **timer.start**() and **timer.stop**() functions: https://docs.particle.io/reference/device-os/firmware/photon/#software-timers
2. Create a new project using **File>NewWindow**. Then click **StartNewProject** (), and name the new project **Lab4c**. In the Project directory tree pane, select the **/src/Lab4c.ino** file for editing and copy/paste your solution from **Lab4a** as the start point for this exercise.
3. Modify your program as follows:
   - First, forget about using the potentiometer to control LED brightness (we will do that later). Instead modify Lab4a to flash **both** LED's using timer interrupts as outlined below…
   - Write a function called **void flashLEDs()** which **digitalRead**s the current LED values, and **digitalWrite**s the opposite values back to each LED if **flashEnabled** is set.
   - Declare and instantiate a **Timer** object, calling its **timer()** constructor to generate events every **200 ms**, and calling **flashLEDs()** as an interrupt handler whenever these events occur.
   - At the end of your **setup()** function, **timer.start()** the timer you have created.
   - Your main program **loop()** no longer needs to worry about flashing or delays and can simply be empty!

4. Compile, flash, and test your program. Do the LEDs still flash normally? You may find it possible to stop the flashing, but with both LEDs **frozen** in their '**on**' state, rather than being 'off'. Why is this? Modify your **flashLEDs()** function to correct for this problem, explaining in the **adjacent comments** why this is necessary. Compile, flash and re-test.
5. Optional: Modify your program so the two LEDs flash at different rates. Without interrupts, this is very hard(!), but with timer interrupts it should be fairly straightforward.
6. Now modify your program so that **LED1** no longer flashes, but instead its brightness can be controlled using the potentiometer input.
   - Modify your **flashLEDs()** interrupt handler so that it no longer flashes LED1.
   - insert code in your (currently empty) main program loop to **analogRead()** your potentiometer input into a variable called **adcVal** and then use it (appropriately scaled) to write a PWM signal to **LED1** to control its brightness.
7. Make sure your external **LED1** is connected to **D0** and that your **potentiometer** is connected to **A5** with a jumper wire. Then compile, flash, and test your program. Does it behave as desired?
8. Check you program has a **well-documented** 'header' and nicely-aligned meaningful **comments**. Demonstrate your program to your instructor (sign-off), and submit it to Blackboard as a ***.cpp** file.

5. Communicating with the cloud

Only use Chrome as your browser in this lab! Obscure problems can occur with other browsers.

5.1 Reading Photon variables from the cloud – Lab5a (uses http GET)

Your **Lab4c** program from Section 4.3 reads the potentiometer value and uses it to control the LED brightness. Suppose we want to be able to view the value read from the potentiometer remotely from the cloud.

1. Read the firmware documentation for exposing a variable on your photon to the cloud https://docs.particle.io/reference/device-os/firmware/photon/#particle-variable-
2. Create a new project using **File>NewWindow**. Then click **StartNewProject** (), and name the new project **Lab5a**. In the Project directory tree pane, select the **/src/Lab5a.ino** file for editing and copy/paste your solution from **Lab4c** as the start point for this exercise.
3. Modify your program as follows:
   - We assume you are using **analogRead()** to read the potentiometer voltage into a variable **adcVal**.
   - Within your **setup()** function, call **Particle.variable()** to register your **adcVal** variable in the cloud with an associated string key eg. **"adcValue"**.
4. Compile and flash your program. It should respond to **BTN1** and the potentiometer as before, but now the **adcVal** should be visible as **"adcValue"** from the web. To test this, first login to Particle **Console**: https://console.particle.io/
5. Within **Console**, double-click on your device: If your device is online, you should see your variable listed as "**adcValue**" in the right hand pane. Try turning the potentiometer with a screwdriver. There may be a little delay, but you should see the **adcValue** changing as well as the brightness of the LED. Note, you could view this from the far side of the world if you wanted to.
9. Check you program has a **well-documented** 'header' and nicely-aligned meaningful **comments**. Demonstrate your program to your instructor (sign-off), and submit it to Blackboard as a ***.cpp** file.

5.2 Calling Photon functions from the cloud – Lab5b (uses PUT)

Your **Lab4c** program from Section 4.3 switches flashing behavior on/off whenever **BTN1** is pressed. Suppose we want to do the same thing remotely, from the cloud.

1. Read the firmware documentation for calling a user function on your photon from the cloud https://docs.particle.io/reference/device-os/firmware/photon/#particle-function-
2. Create a new project using **File>NewWindow**. Then click **StartNewProject** (), and name the new project **Lab5b**. In the Project directory tree pane, select the **/src/Lab5b.ino** file for editing and copy/paste your solution from **Lab5a** as the start point for this exercise.
3. It would be nice just to expose your existing **toggleFlashEnabled** Interrupt handler function to the cloud (since this already flips the value of **flashEnabled**), but interrupt handlers cannot accept or return any parameters. Instead, modify your program as follows:
   - Write a function **cloudToggleFlash()** which does exactly the same thing as toggleFlashEnabled, but which accepts a **String cmd** argument and return an **int**, (in order to conform to the specification for cloud-callable functions). [The body of your function can ignore the cmd argument, but should **return(0)** to indicate successful execution].
   - Within your **setup()** function, call **Particle.function()** to register your **cloudToggleFlash** function in the cloud with an associated string key eg. **"toggleFlash"**.
4. Compile and flash your program. It should respond to **BTN1** and the potentiometer as before, but now flashing should be controllable from the web. To test this, first login to **Particle Console**: https://console.particle.io/
5. Within **Console**, double-click on your device: If your device is online, you should see your function "**toggleFlash**" listed on the right hand pane (if not, click the 'refresh' icon). Then try invoking the function by clicking on the function **call** button. There may be a little delay, but you should be able to turn flashing on/off. Note, you could do this from the far side of the world if you wanted to.
6. Now create a minimal **web page** as another example of how you can invoke your **toggleFlash** function from the cloud:
   - Copy the html code listed below into a text editor and **File>Save** it as **myPage.html**.
   - You will need to substitute your **DeviceID** number and **AccessToken** in the html code.
     - The **DeviceID** is listed in Particle Console>Devices.
     - The **AccessToken** can be found under Particle **Console>Web IDE** (lowest icon in left-hand menu strip), followed by **>Settings** (again lowest icon in left hand menu strip)
   - Here is the html code:

     <form
     action="https://api.particle.io/v1/devices/**<yourDeviceID>**/**toggleFlash**?access_token=**<yourAccess Token>**" method="POST">
     <input type="submit" value="Toggle LED Flashing">
     </form>
7. Open your myPage.html file with a web browser and click on the button in the top left of the web page. Again you should be able to turn flashing on/off.
8. Check you program has a **well-documented** 'header' and nicely-aligned meaningful **comments**. Demonstrate your program using both Console and Web page control to your instructor, (sign-off), but submit only your program to Blackboard as a ***.cpp** file.

5.3 Publishing Photon Events to the cloud – Lab5c (using publish)

Suppose (from somewhere in the cloud), we want to get some notification that the **BTN1** has been pressed without needing to repeatedly poll the photon. Let's get the Photon to publish an event to the cloud…

1. Read the firmware documentation for publishing events to the cloud https://docs.particle.io/reference/firmware/photon/#particle-publish-
2. Create a new project using **File>NewWindow**. Then click **StartNewProject** (), and name the new project **Lab5c**. In the Project directory tree pane, select the **/src/Lab5c.ino** file for editing and copy/paste your solution from **Lab5b** as the start point for this exercise.
3. Modify your program as follows:
   - Modify your **toggleFlashEnabled** function (which you previously attached as **BTN1**'s interrupt service routine), so that it also sets a global **volatile** variable called **publishFlag** to be **true**.
   - Modify your main program loop function so that, if **publishFlag** is **true**, it **publish**es an event called **"btnPressed"** and passes the additional information **"High"**.
   - Important: Within this if-statement, you should also reset the **publishFlag** to **false**, so you don't keep on publishing btnPressed events!
4. Compile and flash your program. It should behave exactly as before, only this time when **BTN1** is pressed, it should also publish an event to the cloud. To test this, first **login** to Particle **Console**: https://console.particle.io/
5. Within **Console**, double-click on your device: You should see details of your device, including an empty list of events. Press **BTN1**. You should see a new event generated within Particle Console.
6. Check you program has a **well-documented** 'header' and nicely-aligned meaningful **comments**.Demonstrate your program using Console to your instructor, (sign-off), and submit to Blackboard as a ***.cpp** file.

5.4 Going beyond Particle Cloud – Lab5d (using IFTTT)

How can we get our published event to do something more useful, eg. to send us an email notification, log data to a spreadsheet, or send a notification to our mobile phone? One possible answer is to setup **IFTTT** to listen for our published events and to perform actions when they occur. [This does not require any more programming of our photon].

1. **Sign In** (or **Register** for a free **IFTTT** Account): https://ifttt.com/
   You may be asked to select some 'favorite' services. Just follow through till the process is complete.
2. Under the **PersonalProfile** menu (at the top, next to 'Explore'), click on **Create**, and then click on the **If This** icon. This will lead you through a sequence:
   a. Choose service: Type **Particle** in the search field, and then click on the Particle Icon that appears. You may need to **authenticate** with Particle using your Villanova email and photonXX password.
   b. Choose a **Trigger**: Click on **New Event Published** to tell IFTTT to listen for our events
   c. Fill in the fields that describe our **btnPressed** event and the **High** data that comes with it, as well as the source (**photonXX)** of these events. Then **Create Trigger**!
   d. Now click on the **Then That** icon to define what **Action** we want to have happen

e. Define the **Action** you desire: eg. type **Email** in the search field, and then click on the Email icon. You may need to **authenticate** to provide access to your gmail account.

f. Fill in the fields that define your email message: You can use a mixture of normal text, and some 'Ingredients' that embed the data about your event. Then **Create Action**!

3. Under **My Applets** you should see your new applet listed. Click on it, and click on the **check now** button to see if it is live – [This will take a few minutes after your applet is first created].

4. Now press the **btn1** (once!) on your photon, and check you see the event appearing on **Particle Console**. Then login to your **gmail** account and check your mail! [It will take a few minutes to appear].

5. Demonstrate your program to your instructor, (sign-off), and submit a copy of the email you receive to Blackboard as a ***.cpp** file.

6. Serial Communications

Although communicating with the cloud opens new possibilities for IOT, most communications with other devices occur locally, so it is important to understand how these communications protocols work.

6.1 Using a UART– Lab6a (talking to a friend)

So far, we have used a USB cable as a convenient way to power the Photon, but we have not attempted to use it for communication. The aim of this lab is to establish serial communications between our Photon and the PC host using a USB-based virtual COM port. It is often useful to be able to send such messages when debugging a program.

1. Read the firmware documentation for serial communications: https://docs.particle.io/reference/device-os/firmware/photon/#serial

2. Create a new project using **File>NewWindow**. Then click **StartNewProject** (), and name the new project **Lab6a**. In the Project directory tree pane, select the **/src/Lab6a.ino** file to edit your program.

3. Write a program to echo any data / messages received on the USB Serial port back to whoever was sending this data.
   - You will need to initialize the USB virtual **Serial** port by calling **begin** within your **setup()** function.
   - Only try reading from the Serial port while it is **connected()** and there are bytes **available()** to read.
   - Store the data **read** from the **Serial** port in a *char* called **inByte**, then write it back to the **Serial** port.

4. Flash your program to the photon, and ensure it has rebooted and that your program is **running** and ready to receive data ***before*** you attempt to connect to it from your host PC.

5. You now have a program executing on the photon waiting to talk, but you need to have a program executing on the PC to provide the other end of this conversation. There are a number of public domain programs (eg. putty) that can talk to a Virtual COM port on a PC, but there is also a serial monitor utility within Particle Dev itself. Select the **Particle > SerialMonitor** menu item, or click on the **USB icon** within Particle Dev to open this utility. Then click **Connect** in the Serial Monitor pane that appears.

6. Check your program works by typing some message, eg. "**Hello!**" in the message entry strip at the bottom of the Serial Monitor pane. When you hit **return**, the message is sent. The photon should echo the same message back, and you should see it appear in the main area of the Serial Monitor pane.

We now want to use the true UART called **Serial1** which communicates through the **Rx** and **Tx** pins of the photon, rather than using a virtual USB serial port.

7. Extend your program so that as well as echoing the message back to the PC, it also **transmits** the same message out through the **Serial1** port. Likewise, if any message is **received** through the **Serial1** port, your program should send it to the USB-**Serial** port, so that you can see it on the Serial Monitor.
   - Within **setup()**, configure the **Serial1** port to be **9600** baud, **8**-bit, even parity, **1** stop bit.
   - Insert a **200** ms **delay** after transmitting the message to **Serial1** before you check to see if any response to your message has been received back from **Serial1**.
   - Insert a **10** ms **delay** at the end of your main **loop()** just to add a break between messages.
8. Flash your program, and **check** it still echoes messages back to the Serial Monitor. Nothing much will seem to have changed because nothing is connected to the Serial1 port.
9. Now use jumper wires to connect your board to a friend's board who is at the same stage in the lab: First connect the **ground**, then connect the **Tx** and **Rx** lines appropriately – think carefully what '**appropriately**' might mean!

   [If you cannot find a partner, you can test your program by using a jumper wire to pipe your **Tx** line directly **back into** your **Rx** input. The messages you receive will obviously be identical to the ones you sent (which is a bit boring), but you should be able to verify that your program works].
10. When looking at the results in the Serial Monitor, it would be nice to be able to distinguish **received** messages from those you **sent**. Modify your program, so that it pre-pends a '**>** ' before the start of any received message, so the results look something like this:

    Hello …this is the message we sent
    > Hi there! …this is the reply we received, as indicated by the initial '> '
11. Check you program has a well-documented 'header' and nicely-aligned meaningful comments. Demonstrate your program to your instructor (sign-off), and submit it to Blackboard as a ***.cpp** file.

6.2 UART *Signals* – Lab6b (oscilloscope waveform analysis)

The aim of this lab is to use the **Oscilloscope** to look at the waveform transmitted from the **Tx** pin, and to write a brief **report** / analysis of your results in a short Word document.

Please ensure this project is entirely your own work. If you need help, ask an instructor. Do not ask for, or give any help to a classmate.

1. Create a new project using **File>NewWindow**. Then click **StartNewProject** (), and name the new project **Lab6b.** In the Project directory tree pane, select the **/src/Lab6b.ino** file to edit your program.
2. Write a simple program that repeatedly sends the message "**VU**" to the **Serrial1** port, using **9600** baud, **8**-bit, even parity, **1** stop bit. Also insert a delay of **10** ms between each transmission. Flash your program to the photon.
3. ~~Connect **Channel1** of the oscilloscope to the **Tx** output pin, [and connect the oscilloscope **ground** to the ground of your breadboard]. Hit **Autoscale** on the oscilloscope, and then fine-tune the display so that:~~
   - ~~The x-axis time-scale is **300us** / division.~~
   - ~~The y-axis scale is **2V** / division~~
   - ~~The **falling** edge of the waveform is nicely lined up with the display grid and all the trace is visible~~
   - ~~The ground level of the waveform is nicely lined up with the display grid~~
4. ~~Use oscilloscope **cursors** to measure the **baud rate** of the transmitted signal and take a photograph or screen capture of your waveform (with cursors on).~~

   Download the Lab6b Oscilloscope results from the Blackboard > Lab Assignments folder and continue as if you had made these measurements yourself.
5. Paste your image into your report, and **annotate** the waveform showing where the start, stop, data and parity bits all appear. Read off the binary **data** that is being transmitted and explain how this relates to the characters "**VU**". Also confirm that any **parity** bits have the right value.
6. Now edit your program so that you repeatedly transmit a numeric value, **x**, where **int16_t x = 27099**; Flash your program and observe the result [see Lab6b Oscilloscope Results]. Why doesn't this work as intended?
7. We need to re-interpret the bit pattern of our **16-bit** integer as two **8-bit** characters which we can do as indicated on **p. 25** entitled "Type Reinterpretation" which is the last page of the first big packet of note.
   - Declare a **union** called **myData** with two overlapping fields, an **int16_t intVal**, and an array of two **char txBuff[2]**.
   - In your program, assign **x** (ie.the value **27099**) to myData's **intVal**, and then transmit it as **txBuff**.
8. Once you have this working, **repeat** steps #4 and #5, only explaining in this case how the results relate to the numeric value of x. Use the results to confirm whether the photon is a **big-endian** or **little-endian** machine.
9. Write brief **introduction** and **conclusion** sections of your report, outlining the purpose of this work, the main results obtained, and any other practical details or additional observations. **Submit** both your program from step #7 and your report on Blackboard.

7. Register Bit Manipulation

Many hardware devices, including the onboard timers, serial IO, and GPIO of the photon are configured using bits within a register to act as switches that control the device operation. Often the necessary low-level bit manipulation is hidden from the user by higher level functions like pinMode(), digitalWrite(), digitalRead(). However, like any good mechanic, it is important to be able to go under the hood…

7.1 Programming GPIO Registers Directly – Lab7 (Lab1c revisited)

The aim of this lab is to rewrite Lab1c (flashing LED's controlled by the push-button) without using higher level functions like pinMode(), digitalWrite(), digitalRead(). Instead, you will need to set/clear the appropriate bits in the appropriate GPIO registers in order to give the same overall functionality. To do this you will need to know the following pin mappings:

Photon Pin **D0** = GPIO Port**B** pin **#7**; Pin **D7** = GPIO Port**A** pin **#13**; Pin **A0** = GPIO Port**C** pin **#5**;

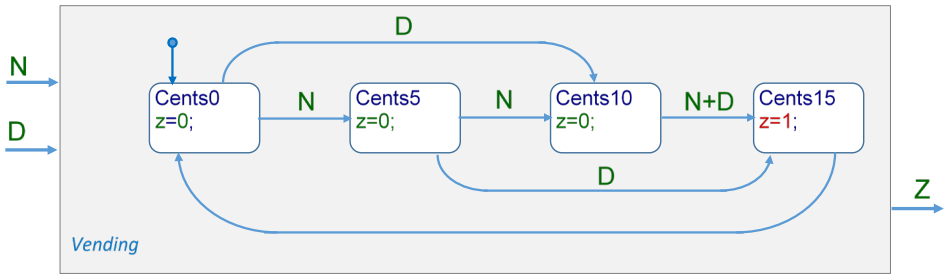
1. **Read** the lecture notes about "Registers & Bit flags" on p.5 of the Boolean Logic note packet.
2. Create a new project using **File>NewWindow**. Then click **StartNewProject** (), and name the new project **Lab7**. In the Project directory tree pane, select the **/src/ Lab7.ino** file for editing and copy/paste your solution from **Lab1c** as the start point for this exercise.
3. Compile and test your program to check it still works as before.
4. Now try to re-program pin **D0**, so that you no longer use pinMode() or digitalWrite() for this pin. Instead you will need to:
   - set/clear the appropriate bits in the **GPIOB** Mode Register **MODER**… ← within setup()
   - …and also set/clear the appropriate bits in the Output Type Register **OTYPER**. ← within setup()
   - set/clear the appropriate bit within the Output Data Register **ODR** ← within loop()
     (in order to turn the LED on/off as desired).
   - **Important**: In all cases, the other bits in these registers should be left **unchanged**.
5. Compile and test your program to check it still works as before.
6. Repeat steps #4, #5, only this time reprogramming pin **D7**.
7. Then do the same again for pin **A0**. In this case you will need to mask/extract the appropriate bit from the Input Data Register **IDR** in order to read the A0 logic input level.
8. Check you program has a well-documented 'header' and nicely-aligned meaningful comments. Demonstrate your program to your instructor (sign-off), and **submit** it to Blackboard as a ***.cpp** file.

8. Finite State Machines

Embedded systems need to be able to respond to *sequences* of events that drive the system into different behavior *states*. A finite state machine provides a useful framework for encoding such behavior. This lab is optional, but you might find that doing it will give you insight and help with your final Alarm system project.

8.1 A first FSM– Lab7a (Vending Machine)

The aim of this lab is to implement a FSM to emulate the behavior of a vending machine for dispensing 15c gum packets in response to users inserting nickels, or dimes into the machine as indicated in the diagram below:

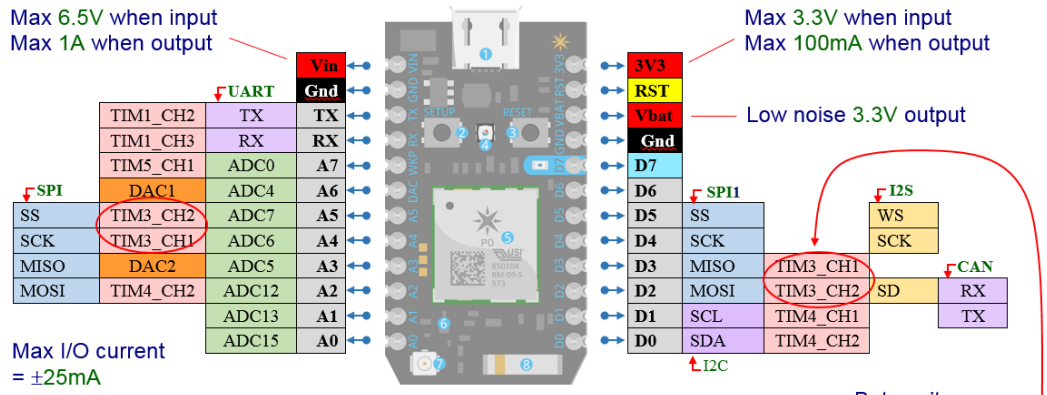


The rising edge of BTN0 will be used to simulate the insertion of a nickel (N), and the rising edge of BTN1 simulates the insertion of a dime (D). Activation of the LED0 is used to simulate actually dispensing the gum. The FSM should execute every 0.2 seconds.

1. Read the course notes for implementing the vending machine
2. Create a new project using **File>NewWindow**. Then click **StartNewProject** (), and name the new project **Lab7a.** In the Project directory tree pane, select the **/src/Lab7a.ino** file to edit your program.
3. Develop your program incrementally as suggested below:
   - Declare a timer object and use it to call a function setTickFlag() every 200 ms. You also need to write this function which should set a global variable tickFlag whenever it is called.
   - Within loop(), respond whenever the tickFlag is set by calling your vendingFsm() function. This function still needs to be written, but you can create an empty vendingFsm() function for now.
   - Set up your two buttons as inputs that generate interrupts, and write interrupt service routines to set flags N and D representing the occurrence of a Nickel or Dime insertion event.
   - Write the body of your vending Fsm() function, lighting LED1 whenever 'gum' is dispensed
4. Compile and test your program. You may find that the LED flashes so briefly it is almost invisible, so go back and modify your state machine design in order to keep the LED lit for 2 seconds while 'gum' isi being dispensed. Modify your vendingFsm() function accordingly, and re-compile and test.
5. The management like your work so much they want to you add an extra feature: Assuming the vending machine is loaded with 5 packs of gum, the machine should keep track of how many packs are remaining, and stop dispensing gum (lighting the LED) if there are none left.
6. Check you program has a well-documented 'header' and nicely-aligned meaningful comments. Demonstrate your program to your instructor (sign-off), and submit it to Blackboard as a ***.cpp** file.

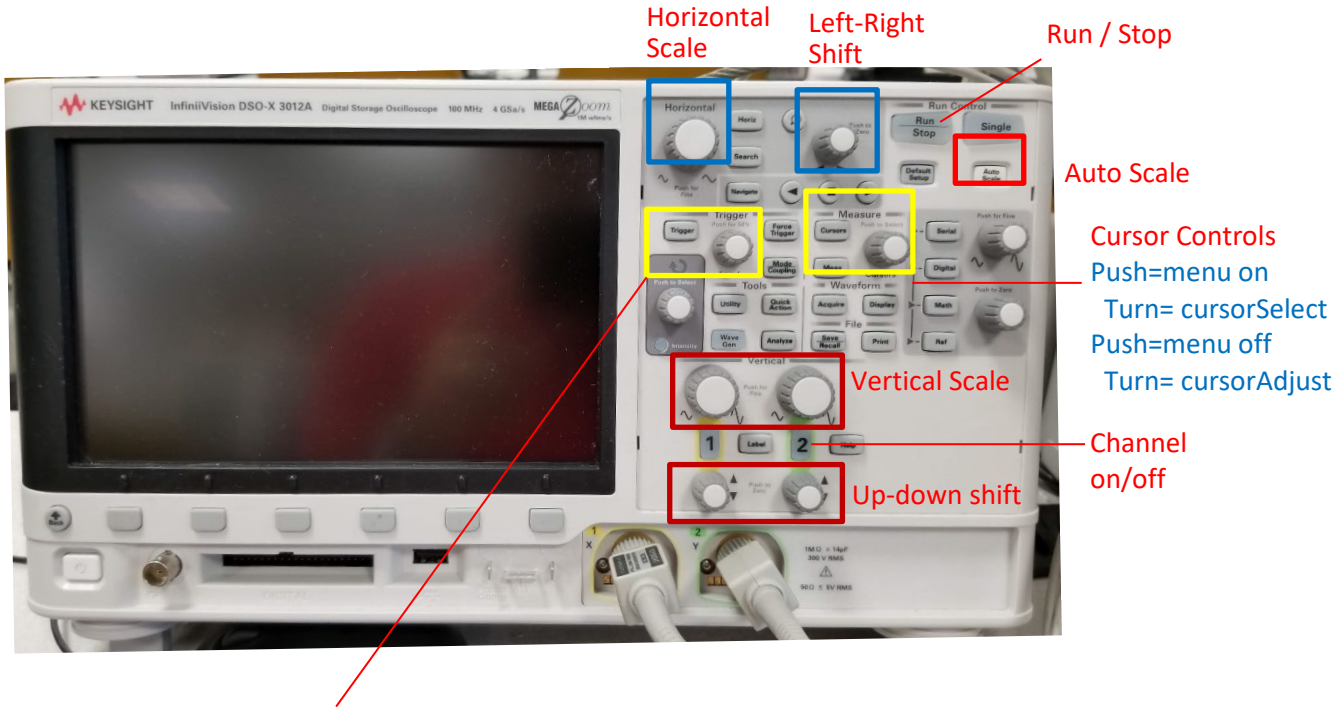
9. APPENDIX

9.1 Photon Pinout



- All GPIO pins can be used as simple Digital I/O + Interrupts
- Pins (except DAC1,2) are 5V tolerant as inputs, but output 3.3V
- Counter / Timer channels used for PWM, Tone, Servo Motor control etc.
- PWM on A5 / D2 and on A4 / D3 are not independent channels (same timer)

9.2 Oscilloscope overview



P

9.3 Particle Build (Web-based IDE)

1. Log in to Particle Build https://build.particle.io/build but **skip** any offer of setting up dual authentication:

   Email address: **<your VU email address>** … eg. username@villanova.edu
   Password: **photonXX** ….where **XX** denotes the number of your device, eg. **50**, 51, etc.

2. Enter **Lab1a** as the name for the Current App, and then in the right-hand editor pane enter your source code as shown below.

```
/* Project Lab1a
 * Description:  <what it does + why + important points>
 * Author: <your name>
 * Date:    <data written>
 */

 */#define LED0 D7                  // now we can call pin D7 by the more meaningful name 'LED0'

void setup() {
  pinMode(LED0, OUTPUT);           // sets LED0 pin D7 mode to Output
}

void loop()  {
    digitalWrite(LED0, HIGH);      // turn LED0 on
    delay(1000);                   // wait 1 second
    digitalWrite(LED0, LOW);       // turn LED0 off
    delay(1000);                   // wait 1 second
}
```

3. **Save** your code by clicking the 3rd from top 'Save' **folder icon** in the left-hand strip.

4. **Compile** your code by clicking the 2nd from top 'Verify' **check-mark icon** in the left-hand strip. The result will be shown in the bottom 'status' strip of your window. If there are **errors**, click on **Show Raw** and scroll down to the first reported error and fix it. Then recompile and Repeat.

5. Click the 3rd from bottom '**Devices**' compass icon in the left-hand strip which brings up a list of devices 'owned' by your account. In this case you own only one device, previously named as **photonXX**. If not already selected, Click on **star** next to its name to select this device as the one to be programmed. The device name should appear in the bottom 'status' strip with a colored dot which matches the device's LED

6. Your program is now ready to be flashed to your photon. Check your photon is powered on and connected to the network (ie. '**breathing cyan**' on the central led, and in the bottom status strip of your IDE). Click the **flash** icon ⚡ to recompile and flash your code to your device. Your photon may 'blink magenta' while flashing is in progress, and once complete will immediately start executing your program ☺!

7. Modify your code to make the LED flash **more quickly** (eg. every 1/5 second), re-compile, and re-flash to see your modified code execute on the device. [Use this new flash-rate for subsequent exercises, eg. Lab1b].

8. **Optional** challenge: How could you modify your loop() code, to achieve the same result only with half the lines of code – ie. get it to flash with only two, rather than four, lines of code within the main loop ?

9. Check you program has a well-documented '**header'** and nicely-aligned meaningful **comments** (see grading rubric in appendix). Demonstrate your program to your instructor (sign-off), then submit as detailed below:

10. Click the 4th from top 'Code' **<> icon**, and then next to '**Files'** click the '**Download sources**' icon. **Change** the file extension to ***.cpp** and submit it to Blackboard as a *.cpp file – (this allows Blackboard to display submitted files).