EE 1301: Introduction to Computing Systems

IoT Laboratory #4

Simple Multi-Tasking on the Photon
Background
The programs you have written so far for your Photon board have all been performing one task at a time. For example, a typical program might perform the following tasks in a loop: read an input value from a sensor, then perform some computations, then write output values to an actuator. This simple sequential flow is fine when all the tasks need to be performed at the same rate. However, to implement more complicated projects, you may need to perform multiple tasks at different rates, and it may not be acceptable to wait for one task to finish before starting the next task. For example, if you want your program to blink an LED once per second and also check if the user has pressed a button, you should not wait until the button has been pressed to proceed and blink the LED; otherwise the LED might not blink at a rate of once per second. Likewise, you should not wait until you blink the LED to check if the button has been pressed; otherwise the user might press the button during the 1 second delay while waiting to blink the LED, and the button press could be missed. In this lab, we will learn how to process multiple tasks at the same time when the tasks may occur asynchronously (i.e., at random times) or at different rates.

Purpose
In this lab, you will learn some basics about how the Photon’s firmware handles executing user code (in the setup() and loop() functions) and other functionalities such as wifi connectivity, serial communication to a terminal, and communication with the cloud. You will learn to use “state” variables to remember the state of the execution of your program. Finally, you will learn to use timers to trigger events periodically at regular intervals.

Pre-Lab Requirements
The quantity of new material for this lab is substantially less than for Labs 2 and 3. However, the application of the concepts is more complex. You should read this lab carefully and consider how you might use these concepts in your final project.

Pre-Lab Checklist
- Read the Particle documentation on the millis() function
- Read the Particle documentation on the use of INPUT, INPUT_PULLDOWN, and INPUT_PULLUP

Required Components:
- Push Button Switch
- Decoupling Capacitors

Background Processes in Photon
The processor on your Photon board comes with a pre-programmed “firmware”, which is the underlying software that takes care of serial communication through the USB port, wifi
connectivity, and cloud connectivity. As mentioned in previous labs, the firmware calls the `setup()` function once after it first connects to the wifi network. Then, the firmware keeps calling the `loop()` function repeatedly every 1ms. After each iteration of the `loop()` function, the firmware performs background tasks to maintain wifi connectivity and check with the cloud to see if there are any messages for the device (e.g., if a new user software is available to flash onto the device or if a cloud event has been published that the device subscribes to). If an iteration of the `loop()` function takes much longer than 1 ms, the device will not be able to perform these background tasks for a long time, and bad things can happen, like losing wifi or cloud connectivity. You will see this in the following experiment.

Wire up the following circuit and flash the code that follows onto your Photon. The code is supposed to sample the button and flash the on-board LED at a regular rate. 

**Note** that we have defined the button input not as a simple INPUT pin, but as an INPUT_PULLDOWN pin. Your TA will explain why you need a pull-down resistor (it’s important that you learn this!).

```cpp
// This program will toggle the on-board D7 LED on and off every second.
// By holding down the button, you temporarily stop the flash and show a constant blue
// light on the D7 LED. However, the way the code is written is not correct. It stalls
// the loop() function when the button is held down, and if the button is held down for
// too long, it causes a time-out.
// Try it: Keep pressing the button for about 20 seconds, and you will see that the
// board loses cloud connectivity and starts breathing green as opposed to breathing cyan.
```

### Pull-down Resistors

The keyword `INPUT_PULLDOWN` asks Photon to use its internal pull-down resistors. Using a pull-down resistor will weakly connect the pin to GND (0V). This will result in a value of LOW whenever the button is not pressed. Since the other side of the button is connected to 3.3V, the button will read HIGH when it is pressed.
// First, define the pins we are going to use.
int led = D7; // The D7 on-board LED
int button = D3; // The input button (we will set it up with the internal pull-down resistor)

// The setup function runs only once, when the device boots up or is reset
void setup() {
  // Setup the D7 pin as an output (on-board LED), and
  // D3 as a button input with pull-down
  pinMode(led, OUTPUT);
  pinMode(button, INPUT_PULLDOWN); // instead of INPUT, we used INPUT_PULLDOWN, which activates the internal pull-down resistor. When the button is not pressed, the button pin is connected to LOW (GND) through the pull-down resistor. When the button is pressed, the button pin is connected to HIGH (3.3V).
}

// The loop function is called repeatedly, as often as possible, after your device boots up. Firmware interleaves background tasks associated with WiFi + Cloud activity with the code in your loop function by performing the tasks after each loop function iteration.
// Note: Code that blocks (stalls) for too long (like more than 5 seconds), can make bad things happen (like losing cloud connectivity). As a result, the while loop below can cause trouble, since it can prevent the loop function from finishing for an arbitrarily long time, causing the device to lose its cloud connectivity.
//
// Note 2:
// The built-in delay function is safe to use, even for arbitrarily long delays.
// It is implemented (by Particle) in a way that it safely interleaves required background activity with loop function execution.
void loop() {
  while (digitalRead(button) == HIGH){ // As long as the button is pressed down, wait here and do nothing. This is called a blocking loop, and is not a good idea, in general.
    digitalWrite(led, LOW); // Turn OFF the LED
delay(1000); // Wait for 1000mS = 1 second
    digitalWrite(led, HIGH); // Turn ON the LED
delay(1000); // Wait for 1000mS = 1 second
  }

  // If you want to wait for the button to be released without causing the photon to lose cloud connectivity, replace the while loop above with the one below.
  //
  // while (digitalRead(button) == HIGH) // as long as the button is pressed
  //  // down, wait here and do nothing.
  //  Particle.process(); // allows the firmware to do some background tasks such, as maintaining wifi and cloud connectivity while the while loop is waiting for the button to be released
  //}

As you experiment with running the code on your Photon, notice the following behavior. If you don't press the button, the code will repeatedly toggle the LED -- on for one second and off for one second. However, if you press and hold the button for a few seconds, the blinking will stop and the blue LED will remain on.

Press and hold the button for a few seconds and then release it. You will see that the blinking stops as long as you hold down the button. Next, try holding the button down for more than 20
seconds. You will see that the board will stop breathing cyan (which indicates cloud connectivity) and will start breathing green (wifi connectivity only). The reason is that you have stalled the processor in the while loop (while(digitalRead(button) == HIGH){}), and it is unable to complete this iteration of the loop and check in with the cloud. (Remember that checking in with the cloud happens after an iteration of the loop function finishes.) The while loop that you used is called a **blocking loop**. In general, blocking loops are not a good idea, unless you are sure it is OK for your particular application to stall indefinitely.

Now, change the code based on the recommendation in the blue comments right below the `loop()` function, and then repeat the experiment. You will see that no matter how long you hold down the button, the Photon board never loses cloud connectivity (i.e., it continues breathing cyan). The reason is that the `Particle.process()` function lets the Photon’s firmware do the background processing, such as wifi token renewal and cloud message checking as the while loop is executing.

Although the code does what it is supposed to do (i.e., stop blinking as long as the user holds down the button), its structure is not suitable for all applications that need to multi-task. We will look at an example of such applications in the next section.

### Detecting Button Presses based on Previous Button State
(Recap from IoT Lab 2)

The previous example uses a while loop to detect when the user has pressed the button. As mentioned above, that might be acceptable in some applications that do not need to perform other actions, but if you want to be able to perform other tasks while waiting for the button to be pressed or released, then you should use an approach that avoids using while loops. We can do that by checking the status of the button using an if statement instead of a while loop.

If you remember, we did just that in IoT Lab 2: “Sensors - Human Input Devices”. You might recall that if we just checked the **value** of the button in a particular loop function iteration, we would get multiple button presses every time the button was pressed, since the loop function executes multiple times before the button is released. Our solution was to compare the value of the button across two successive loop iterations. We added a **state variable** to record the state of the button in the previous loop iteration, and compared the current and previous button values to check if the button changed from LOW to HIGH since the last loop iteration (just pressed between the two calls). This state variable needs to be initialized either as a **global variable** or a **static local variable** so that its value persists from one loop iteration to the next. We have chosen the global variable option in the following code because it is simpler to understand, but using static variables is the preferred method. More information on **static local variables** is available in the **Appendix**.
Below is an example of how we will be implementing the button check in the next version of the code, read through the code and make sure you understand the code -- especially the concept of using a state variable to record the previous state of the button. If you have any questions, stop here and discuss with a neighbor or the TA before proceeding.

```cpp
// initialization code -- prevButton is declared as a global variable
int prevButton = LOW;  // state variable to store if button was HIGH or LOW 
                        // last time we checked

// inside loop() code...
int curButton = digitalRead(button);
if(curButton == HIGH && prevButton == LOW){
  // a transition happened -- the user has just pressed the button
  // respond to the button press in this code block
}
prevButton = curButton;  // record the state of the button for the next round
```

Storing Blinking Rate in a State Variable and Using Button Press to Change Blinking Rate

Now, we will use the improved button press code to change the blinking rate of an LED. The blinking rate should cycle between two different rates (let’s call them rate 0 and rate 1), changing each time the button is pressed. When the LED is blinking at rate 0 and the button is pressed, the LED should start blinking at rate 1, and vice versa. To decide how to set the blinking rate when the button is pressed, you can use another state variable to keep track of the current blinking rate for the LED. Based on the value of the state variable, you will know how to set the new blinking rate when the button is pressed.

The program below will blink the on-board LED (D7). By pushing a button (D3), you can toggle the rate at which the LED blinks between 0.25 Hz (2 seconds on, 2 seconds off) and 0.5 Hz (1 second on, 1 second off). Take a look at the code below.

```cpp
// This program blinks the on board LED (D7) at different rates
// When the button is pressed, the rate of blinking changes.
// Use state variables to store the previous button value and the current blinking rate

// Define the pins we are going to use
int led = D7;  // The D7 on-board LED
int button = D3;  // This is the input button (use an internal pull-down resistor)
int LedFreqState = 0;  // set to 0 for 0.50 Hz, set to 1 for 0.25 Hz
int prevButton = LOW;  // state variable to store if button was HIGH or LOW last time

void setup() {
  pinMode(led, OUTPUT);
  pinMode(button, INPUT_PULLDOWN);  // INPUT mode with internal pull-down resistor
  Serial.begin(9600);  // Use Serial port for debugging
}
In the code above the new state variable that stores the blinking rate is, `LedFreqState`. When a button press is detected, this state variable is updated (`LedFreqState++`). This state variable is also checked to decide whether to delay for 1 or 2 seconds between turning the LED off and on (`if (LedFreqState == 0){...} else{...}`).

Flash the program above to your Photon and try it out! As you test the program, see if you can observe any abnormal behavior. First, hold down the button for awhile and you should observe that the blinking rate changes as expected. Now, try pressing and releasing the button quickly while the LED is off (easier to do this when the rate is 0.25Hz). You will see that the program does not detect your button press. This is because the Photon is busy executing one of the code blocks for the if/else statement that blinks the LED. Since the shorter code block takes at least 2 seconds and the longer code block takes at least 4 seconds, if you do not hold the button for several seconds, your button press can be missed.

The LED blinking is handled using blocking code, making it impossible to detect a button press when the LED is being toggled. As discussed before, blocking code is not a good solution when you want to perform multiple tasks asynchronously or at different rates (like checking for button presses and blinking the LED). We will address this deficiency in the code in the later section on “Using a Timer”, but first, add some functionality to the current code by adding an additional blinking rate, as described below.
Exercise 1 -- More LED Blinking Rates

Before continuing on, demonstrate for your TA that you can add a state to the program above. Change the LED blinking program such that you can cycle through three states: 1Hz, 0.5Hz, and 0.25Hz. As before, your program should change states when the button is pressed.

Using a Timer to know when to toggle the LED WITHOUT using the delay() Function

As noted before, the problem we are trying to fix in the code is that the program cannot detect button presses when the LED is being toggled. This is because the \texttt{delay()} function blocks the Photon from continuing to execute the \texttt{loop()} function until it finishes\(^1\). It is similar to a while loop that keeps iterating and doing nothing until the specified amount of delay has elapsed. Like before, to make this code non-blocking, we will use an if statement to check when to toggle the LED instead of using the blocking \texttt{delay()} function. The if statement will check whether the correct amount of time has elapsed or not. If it has, we will toggle the LED; if not, we will continue executing the rest of the code.

The programming construct we will use to accomplish this is called a \texttt{timer}. A timer is a special piece of hardware that continuously counts up from the instant your Photon is turned on. You can access the current value of the counter with the function \texttt{millis()}, which returns the elapsed time in milliseconds since your device was turned on. Here is a link to the Photon help page on \texttt{millis}:

\url{https://docs.particle.io/reference/firmware/photon/#millis-}

We will use a variable to determine when to toggle the LED. Each time the LED is toggled, we will calculate and store the next time to toggle by reading the current value of the timer and adding the amount of time until the next time the LED should toggle. Then, each time through the loop function, we can check if the current timer value is greater than our stored variable that tells us the next time to toggle.

As an example, the following program will blink the on-board LED (D7) at a rate of 0.5Hz. It does this by checking the current time and comparing it to the saved state variable, which stores the next time to toggle. If the current time is greater than the state variable, the LED is toggled and the next time to toggle is computed and saved in the state variable. The program below is non-blocking, since there is no way to block an iteration of the loop function from completing for a long time.

---

\(^1\) As we saw before, the \texttt{delay()} function \texttt{does} let background tasks in the firmware be executed, but not user code such as checking the state of the button and comparing it to the previous state of the button.
// This program will blink the on-board D7 LED at 0.5 Hz.
// It does not use the delay() function
// Note that there are two state variables -- one to store the next time to toggle
// the LED, and one to store the current LED state (ON/OFF)

int led = D7; // The D7 on-board LED

// Declaring the state variables as global variables.
// You could also declare state variables as local static variables inside the loop
// function (preferred).

bool LED_state = FALSE;

// This variable stores the next time to toggle the LED.
// It will be initialized to 1 second in the future in the first iteration of the loop function
unsigned long int timeToToggleLED = 0;

void setup() {
    pinMode(led, OUTPUT);
    digitalWrite(led, LOW);
}

void loop() {
    unsigned long int currentTime = millis(); // get the current time

    // Check if it is time to toggle the LED "Are we there yet? Are we there yet? ..."
    if(currentTime > timeToToggleLED){
        // Time to Toggle!
        LED_state = !LED_state;
        digitalWrite(led, LED_state);

        // Calculate and store the next time to toggle the LED
        timeToToggleLED += 1000;
    }

    // The rest of your Photon code goes here
}

The statement, timeToToggleLED += 1000 sets the next time to toggle the LED at 1000 ms (1 second) in the future. Flash the code above to your Photon and test it out.

**Note:** In these examples, an unsigned long int is used to store the amount of time in milliseconds since the device was reset. On the Photon, an unsigned long int has 32 bits. The unsigned keyword indicates that the variable only stores positive values and cannot store negative values. Thus, the maximum value that the 32-bit variable can hold is $2^{32}-1=4,294,967,295$. This means that the counter will rollover to 0 again after 4,294,967,295 milliseconds (approximately 49.7 days). You can safely ignore rollover for most of this course. However, if your project uses timers and is going to run for longer than 49 days at a time, please talk to the instructor or the TA about how to handle rollover.
Exercise -- Checking for Button Presses and Changing the Blinking Rate, both using Non-blocking Code

Now, take what you've learned in the examples above and create a non-blocking program that detects button presses and changes the blinking rate of the LED. Your program is required to meet the following criteria:

- **No blocking code**
- Have at least 3 different states, each with different blinking rates or patterns
- **No delay() function calls**

If you’d like, feel free to experiment with your RGB LEDs or add other sensors/actuators in this exercise.

**HINT** - Use of preprocessor directives can be very handy for enabling/disabling debug code, as demonstrated in the following example:

```cpp
#define __DEBUG

setup() {
}

loop() {
    //<blah>
    #ifdef __DEBUG
    Serial.println("Button press detected.");
    #endif
    //<blah>
}
```

All code within `#ifdef` and `#endif` lines is only compiled into your executable if `__DEBUG` is defined. To enable debug code, define `__DEBUG` using the statement `#define __DEBUG`. To disable the debug code, simply comment the line that says `#define __DEBUG`. Then, `__DEBUG` will not be defined, and the Serial print statement will not be executed.

**Lab Report**

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

- Describe some potential problems with using blocking code on your Photon device.
- Describe how state variables were used to create non-blocking code.
- 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).
Appendix I: Static Variables

As you have learned in the lectures, a local variable in a function is only visible until the end of the function’s code block, i.e., the variable’s scope is local to the function. On the other hand, a global variable can be seen by all functions, i.e., its scope is global, as the name suggests. Take a look at the following code example (it is written for a generic platform, and not necessarily for the Photon board).

```c
int gVar = 1;

// Alice wrote this function at Company X.
void f1()
{
    int lVar = 0;
    gVar++;
    lVar++;
    return;
}

// Bob wrote this function at Company X.
void f2()
{
    gVar=0;    // Let’s assume this is an error. Bob didn’t mean to reset gVar.
    return;
}

int main()
{
    gVar = 5;
    f1();    // 1st call
    f1();    // 2nd call
    f2();
    return 0;
}
```

When the program starts, the global variable gVar is initialized to 1, and then main() starts executing. The first line of main() changes the value of gVar to 5. From this point on, every function, including f1() and f2() will see the value of gVar as 5.

The first time main() calls f1(), the function f1() creates a local variable lVar and initializes it to zero. gVar is incremented to 6, and lVar is incremented to 1. Remember these values. The function f1() relinquishes control over the memory location containing lVar before returning to main (think of it as destroying lVar). The second time main() calls f1(), the function f1() first creates the local variable lVar again and initializes it with a value of 0. Note that the previous copy of lVar from the previous call was lost and this would be a new variable with a new initialization. However, gVar++ is going to increment the previous value of gVar, which results in gVar being assigned the value 7. The local variable lVar is incremented to 1 before being destroyed. Note that gVar kept its value from the last call of f1().
When f1() returns, main will see the value of gVar as 7. The next line in main() calls f2(). Let’s assume that Bob, who wrote f2(), was not careful and made an error by resetting gVar. Alice, who wrote f1() has no control over other people messing up the value of gVar, which could be critical to the proper operation of f1().

In summary, global variables have the advantage that they “remember” what happened to them in the previous calls to a function, but have the main disadvantage that they are not protected from access by other functions. Since coding is often a collaborative process, use of global variables can be problematic.

Static variables were introduced in the C language to address this issue. A static variable is a local variable to a function in terms of scope, which means that no other functions can access the variable. However, static variables “remember” their previous value from a previous call to the function, which is similar to the way global variables behave. The following example illustrates this point:

```c
void f1()
{
    static int sVar = 3;
    sVar++;
    return;
}

int main() {
    f1();     // 1st call
    f1();     // 2nd call
    f1();
    return 0;
}
```

When main() calls f1() for the first time, sVar is created as an int variable and initialized to 3. From this point on, whenever f1() is called in the future, the initialization of sVar will be skipped. In that first call, after initialization, sVar is incremented to 4 before returning to main(). Unlike a regular local variable, sVar is not destroyed and will retain its value.

The next line in main() calls f1() again. In the second call, sVar is not initialized to 3. Instead, it retains its value of 4. The next line of f1() increments sVar to 5.