# Lab 2: Flashing an LED/Driver Development

Due Date: September 20, 2024 @ 11:59 PM

Estimated Complexity: Medium

Estimated Time to Complete: 2 weeks

**START EARLY!**

(Hypothetical) Engineering Request: Provide Driver support for configuring peripherals and flashing LEDs. These drivers will make it easier for other embedded software engineers to use the same driver for different applications and projects.

Prerequisites:

- Completion and complete understanding of the Lab Lecture
- Knowledge of structs in C
- How to deal with pointers
  - How can we have a pointer act as a struct?
- How to create filesets
- How to use the documentation to find:
  - Memory/Register Maps
  - Bit fields and bit masks
- How to write C code
- How to write switch statements in C
- How to follow coding standards and guidelines

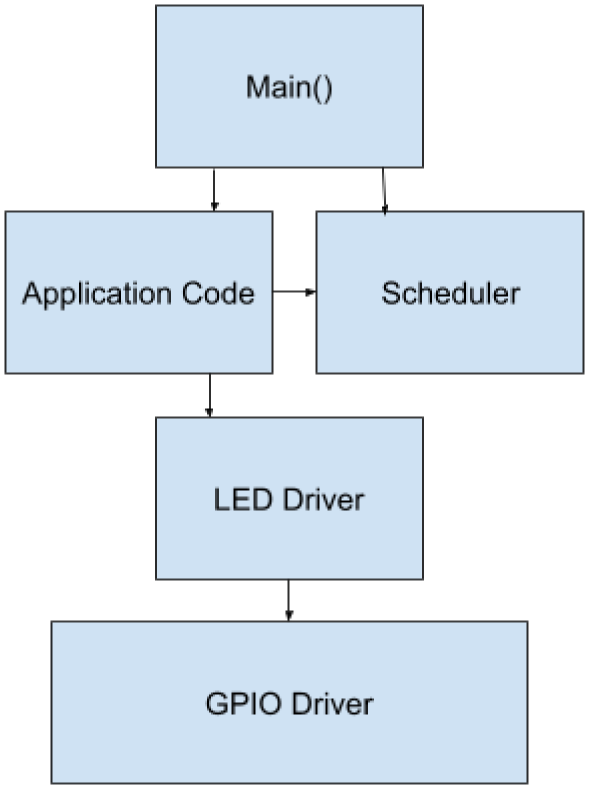
High-Level Overview:

This lab will configure the GPIO (General Purpose Input-Output) pins to toggle an LED after a fixed delay. The program should cause the LED to flash at a fixed rate. The rate at which the LED flashes will be determined based on the length of your name multiplied by some magnitude specified later in this document.

This will also be tied together using a scheduler. The purpose of the scheduler is to have the "super loop" execute functions in a more organized manner. We will have an application code that will add scheduler events and execute specific code based on these scheduled events. We will have two events for this lab: an event to "handle" the LED and an event to execute the delay. To summarize, after each iteration of the super loop in main.c, the scheduler will need to find the events scheduled and run specific functions accordingly. This concept will be explained in more depth later on.

Note: When instructed to add prototypes, assume no return type unless a return type is explicitly stated. If the data type is underlined, we expect it to be named that.

Coding hierarchy:



Remember, files can only directly include files they “point” to in the diagram.
For example, for the application code fileset (.c and .h file), in the ApplicationCode.h, there should be an include statement for “Led_Driver.h”.
The source files should only include their respective header file. So LED_Driver.c should only include LED_Driver.h while LED_Driver.h will include GPIO_Driver.h

Remember, if things are in *italics,* that is just information I want you to know. They are not action items to be completed.

Lab Instruction:

1. **Create a new project**
    a. In the IDE, right-click the space within “Project Explorer” and go to New > STM32 Project
        i. Or go to file>new>stm32 project
    b. Select the board we are using
    c. Have the exact new project details as the last lab besides the project name
        i. The project name should be similar to the last lab but just with the current lab number
        ii. Make sure the project template is selected to empty
2. Create the following filesets (source and header file). Make sure to put the header files in the ‘inc’ directory and the source in the ‘src’ directory. Remember, the files must be named this!
    a. LED_Driver.*
    b. GPIO_Driver.*

c. Scheduler.*
d. ApplicationCode.*
   Where * indicated h or c

3. Create the following file in the inc directory:
   a. STM32F429i.h
4. Create the following prototypes and macros as directed in the specified files:
   a. In “ApplicationCode.h”:
      i. Create two macros
         1. One that will be the length of your name
         2. Another one that will be defaulted to have a value of 250000
            a. *This will be used for magnify the delay that will be created later*
      ii. Create the following function **prototypes**. If an input or output argument is not specified, you can assume there is none for that specific argument. Unless explicitly stated, they do not have to be named the way I name them. This will be true for creating all function prototypes in this lab and later labs
      iii. applicationInit
         1. Brief: This will act as a function that is used to initialize the application code. This includes initializing hardware components as well
      iv. greenLEDInit
         1. Brief: Initializes the green LED
      v. redLEDInit
         1. Brief: Initializes the red LED
      vi. toggleGreenLED
         1. Brief: Toggles the Green LED
      vii. toggleRedLED
         1. Brief: Toggles the Red LED
      viii. activateGreenLED
         1. Brief: Turns on the Green LED
      ix. activateRedLED
         1. Brief: Turns on the Red LED
      x. deactivateGreenLED
         1. Brief: Turns off the Green LED
      xi. deactivateRedLED
         1. Brief: Turns off the Red LED
      xii. appDelay
         1. Brief:  This will act as a function to do a software delay
         2. Input Arguments: A uint32_t to determine the amount of tie to delay
   b. In “Scheduler.h”:
      i. Create the following prototypes **verbatim** (ie, name the prototypes how they are named below):

(void);

1. getScheduledEvents
    a. Brief: This will return the scheduled events
    b. Return Arguments:
        i. A uint32_t that will be the scheduled events
2. addSchedulerEvent -
    a. Brief:  Adds an event to be scheduled
    b. Input Arguments:
        i. A uint32_t value that will be the event to be scheduled.
3. removeSchedulerEvent -
    a. Brief: Removes event to be scheduled
    b. Input Arguments:
        i. A uint32_t value and will be an event to be removed

c. In “GPIO_Driver.h”:
    i. Create the following function prototypes. Note that some of these data types will not be created/defined yet, but we will define them later. *Note: Your code will not compile until we get those types created*
    ii. GPIO_Init
        1. Brief: Initializes the GPIO peripheral based on the configurations provided
        2. Input Arguments:
            a. A pointer of type GPIO_RegDef_t that will be the port to be configured
            b. A struct of type GPIO_PinConfig_t that will be the pin configuration
    iii. GPIO_ClockControl
        1. Brief: Enables or disables the appropriate clock given the GPIO port
        2. Input Arguments:
            a. A pointer of type GPIO_RegDef_t that will be the port to be configured
            b. A value of uint8_t that indicates if we are enabling or disabling
    iv. GPIO_ReadFromInputPin
        1. Brief: Reads the value from the specified input pin from the specified GPIO port
        2. Input Arguments:
            a. A pointer of type GPIO_RegDef_t that will be the port to be configured
            b. A value of uint8_t that indicates which pin to read
        3. Return Values:
            a. A uint8_t that will be the value of the pin
    v. GPIO_WriteToOutputPin

   1. Brief: Writes to the specified output pin of the provided GPIO port
   2. Input Arguments:
      a. A pointer of type GPIO_RegDef_t that will be the port to be configured
      b. A value of uint8_t that will indicate the pin number
      c. A value of uint9_t that will indicate the value to be written to the pin

vi. GPIO_ToggleOutputPin
   1. Brief: Toggles the specified pin in the specified GPIO Port
   2. Input Arguments:
      a. A pointer of type GPIO_RegDef_t that will be the port to be configured
      b. A value of uint8_t that will indicate the pin number to be toggled

d. In “LED_Driver.h”:
   i. Create two macros, one for the green LED and one for the red LED. Of course, these macros will need different values (preferably 1 and 0).
   ii. Create the following function prototypes. **For this lab, all prototypes in this driver will have the same input argument, that is, a uint8_t that will dictate which LED is being interacted with**
   iii. LED_Init
      1. Brief: Initlaizes the specified LED
      2. Input Argument:
         a. A uint8_t value that indicates which LED
   iv. ToggleLED
      1. Brief: Toggles the specified LED
      2. Input Arguments:
         a. Same as the function above
   v. TurnOffLED
      1. Brief: Turn off the specified LED\
      2. Input Arguments:
         a. Same as the function above
   vi. TurnOnLED
      1. Brief: Turn on the specified LED
      2. Input Arguments:
         a. Same as the function above

5. Create microcontroller-specific macros and register maps
   a. We will use the reference manual, specifically the memory map sections (Section 2.3).
   b. We care about two components for this lab: Resets, Clocks, and Control, and the other will be a GPIO peripheral.
      i. To find out which GPIO peripheral we want, we need to locate the electrical schematic for our microcontroller.

        1. Find out which GPIO port (and pin) the LED (Green and/or Red) is connected to. *Refer to the prelab if needed*

c. In STM32F429i.h,
    i. *We need to make base address macros. The naming convention should be the peripheral or bus it is on with '_BASE_ADDR' appended to it.*
        1. *For example, RNG_BASE_ADDR (from the prelab)*
    ii. Include stdint.h. *This will allow us to use uintXXX_t data type*
        1. *Where XXX is 8,16, or 32*
    iii. Create two macros,  each of which will evaluate to the **bus base address** of the two peripherals that we will be utilizing
        1. We will be using the RCC peripheral and GPIOG peripheral
    iv. Create two macros, each of which will be the base addresses of the RCC peripheral and the GPIOG peripheral
        1. Note the value the macro takes should not be hardcoded; it should use the bus peripheral address and add an offset.
            a. For example, (XXX_BASE_ADDR + 0x1800)
    v. Create the typedef struct 'GPIO_RegDef_t' that will represent the register map for GPIO
        1. The struct should have 9 members. The alternate function registers should be combined into a single array of size 2.
    vi. Create the typedef struct 'RCC_RegDef_t' that will represent the register map for RCC
        1. This will be a lot of typing and careful coordination. There will be numerous reserved sections, which you will also need to include
            a. Although they cannot be "accessed," they still have a role in the register memory map
        2. There should be 32 members of this struct, this is counting all the reserved registers that are in continuous memory being in a single array. If you have 30, are you sure you are copying the right data for OUR specific board?
            a. *Hint - Is our board the only board for which the reference manual is?*
    vii. Create a macro for GPIOG that will act as a pointer of type GPIO_RegDef_t to the GPIOG base address. It will look like:
        #define GPIOG ((GPIO_RegDef_t*) GPIOG_BASE_ADDR)
        1. *Feel free to review pointer casting in C if this is severely confusing you*
    viii. Create a macro for the RCC that will act as a pointer type of 'RCC_RegDef_t' to the RCC base address.
    ix. Create two macros, one to enable and one to disable, the clock for GPIOG
        1. This will need to access the RCC macro and set or clear a bit in the appropriate register responsible for enabling the bus which

GPIOG is on. For example, to enable the clock for GPIOB, it would look like:

#define GPIOG_CLK_ENABLE()     (RCC->AHB1ENR |= (1 << 1))

x. Due to us doing a lot of binary actions and setting or verifying things are set to 1 or 0, you should add the following macros:

```
#define ACTIVE          1
#define NON_ACTIVE      0
#define SET             1
#define RESET           0
#define ENABLE          SET
#define DISABLE         RESET
```

6. Create GPIO driver code
    a. In the header file (preferably above the function declarations), add the following:

```
typedef struct
{
    uint8_t PinNumber;       // Pin Number
    uint8_t PinMode;         // Pin Mode
    uint8_t OPType;          // Output Type
    uint8_t PinSpeed;        // Pin Speed
    uint8_t PinPuPdControl;  // Pin Push up/ Pull Down Control
    uint8_t PinAltFunMode;   // Alternate Function mode
}GPIO_PinConfig_t;
```

        i. *GPIO_PinConfig_t acts as the configuration struct that clients will use to configure a GPIO. When developing the LED code, we will go more in-depth on the purpose of these struct members*
        ii. Create a set of macros for each of the GPIO pin numbers
            1. There should be a total of 16 Macros in this set
            2. One macro will be:

```
#define GPIO_PIN_NUM_0  0
```

        iii. Create a set of macros for all GPIO pin mode configurations
            1. There should be 4 modes
        iv. Create a pair of macros for GPIO output type (OPType) configurations
        v. Create a set of macros for the GPIO pin speed configurations
        vi. Create a set of macros for the GPIO pin push-up/pull-down control configurations
        vii. Be sure to include ‘Stm32f4291.h”
    b. In the function responsible for clock control:
        i. You must check if you are enabling or disabling a clock
        ii. Then you must determine which GPIO clock you are operating on based on the input argument of type ‘GPIO_RegDef_t’

            1. *Hint: the GPIO_RegDef_t will be the same type of one of the Macros you created in "Stm32f429i.h" file*
      iii. Then do the appropriate operation (enabling or disabling the clock)
            1. Use the macros you created in Stm43f429 file
   c. In the function responsible for writing a value to a pin:
      i. You must first determine which value (1 or 0) is getting written to the pin
      ii. Then you must set or clear a bit in the Output Data Register of the provided port
            1. This means you will need to use the shift operator to set/clear the proper bit properly
            2. *Refer to the Reference Manual if needed*
   d. In the function responsible for toggling a pin:
      i. You will need to toggle the appropriate bit of the provided port by using the shift operator to toggle the correct bit
      ii. *Refer to the Reference Manual if needed*
   e. In the function responsible for initializing the GPIO:
      i. Create a local variable of type uint32_t that will be used as a 'temporary" (temp)  variable
      ii. Configure the port mode  (MODER) register with the appropriate mode from the pin configuration struct
            1. Assign the temp variable the pin mode shifted by the pin number times 2.

```
temp = (GPIO_PinConfig.PinMode << (2 * GPIO_PinConfig.PinNumber));
```

                  a. *Note -  GPIO_PinConfig_PinNumber is the name of the input argument. Yours will most likely have a different name*
                  b. *The multiplication by two is needed because each pin number has two configurable bits in this register. If we only shift by the pin number, we would access the incorrect bit field*
                  c. Note - the 2 in the above line of code counts as a magic number so make a proper macro for it
            2. Clear the bits in the port mode register that ONLY correspond to the appropriate pin number.
                  a. We will need to shift a bit mask that will clear the two bits at the correct location

```
pGPIOx->MODER &= ~(0x3 << (2 * GPIO_PinConfig.PinNumber));
```

            3. Set the mode in the port mode register
                  a. Set the appropriate bits in the port mode register using the temp value

```
pGPIOx->MODER |= temp;
```

      iii. Configure the output speed register

        1. Same process as configuring the port mode register, but instead using the pin speed configuration provided in the RegDef struct

iv. Configure the pull-up/pull-down register
    1. Same process as configuring the port mode register, but instead, using the push-pull control of the RegDef struct

v. Configure the output type register
    1. It's the same process as configuring the port mode register; however, use the output type member of the RegDef struct. AND there are only two possible configuration options, which means only one bit needs to be set rather than two.
        a. This means you will need to change your bitmask to reflect that when accessing the correct bit
        b. Also, multiplying the pin number by two is unnecessary since it is only one bit.

vi. Configure the alternate function register only if the pin mode is configured to use the alternate function
    1. Use an if statement to check this condition
        a. Check if the pin mode equals the alt function macro
    2. Create a variable of type uint32_t that will be used to select the proper register in the alternate function registers
        a. *It is noted that we will be treating two registers (AFR registers) as one big one that is broken into two halves or parts. We can do this because both registers are next to each other in memory and closely intertwined regarding functionality. AFR[0] will represent the alternate function low register (which will only have pins 0-7), and AFR[1] will represent the alternate function high register (which will have only pins 8-15).*
        b. *According to the documentation, there are 16 different alternate functions a given pin can have, meaning each pin will have 4 bits that need to be configured*
    3. Determine the proper register ( the lower or upper) and store that number in the variable above.
        a. This is simply the pin number divided by 8.
        b. *This is because 8 is the "cut-off" for whether we need to use the high or low registers. Remember how division in C works; if you have something like 7 / 8 (7 divided by 8), that will be computed as 0 (given that 7 and 8 are declared as integers). If you were to do 9/ 8 (9 divided by 8), you would get 1. This is also known as taking the floor of the number.*
        c. *So doing that division will either result in a 1 or 0, which then will dictate which register (high or low, AFR[1] or AFR[0]) we will access*

        4. Create a variable of type uint32_t that will be responsible for accessing the correct bit fields
            a. *Instead of shifting by the pin number, we will shift based on this value because we refer to the AFRs as one register with two parts. For example, say we want to change the alternate function for pin 9, we would use AFR[1], which is responsible for pins 8-15, but if we shift by 9 multiplied by 4 (4 being the number of bits for each pin's configuration), we will access the wrong location*
            b. *Rather, we need to know we are already at pin 8's bit field when we access AFR[1] and only shift 1 set of bits (4 bits) over*
        5. Determine the pin location in the targeted register (AFR[0] or AFR[1]) and assign it to the variable above
            a. This is simply pin number modulo 8
            b. *Recall what the % operator does in C. That is, it will calculate the remainder. For example, 9 % 8 (9 modulo 8) is 1. 8 % 9 (8 modulo 9) would return 8*
            c. *So if we had pin number 9, we would get a pin location of 1, which means that in the AFR register we access (AFR[1]) we will shift 1 pin location from the base.*
        6. Similar to configuring other registers in this function, we will pull the alternate function mode from the pin configuration struct, shift that value by the pin location value multiplied by 4 (the configuration bits for each pin), and store that in the temp variable.
        7. Clear the appropriate bit fields in the correct AFR register
            a. We do this by using the variable that selected which register we would use as the array accessor.
            b. We will need to shift a bit mask that will clear the 4 bits at the correct location
        8. Set the appropriate bit fields in the correct AFR register
            a. Instead of clearing, you will just set the correct AFR register with the value you stored in the temp variable
        9. *Make sure you understand this. The instructional staff can and will ask questions regarding these concepts in the interview grading session*

7. Create LED interface/driver code
    a. *Note the below instructions will refer to "appropriate LED" and have similar language, the logic should still be in your code to support the opposite LED then what is expected for this lab, but the code will not need to be populated.*
    b. In the LED Initialization function
        i. Create a switch statement that takes the input argument and determines which LED needs to be initialized
        ii. Initialize the appropriate LED

1. Populate the GPIO pin configuration struct elements with the appropriate configurations
    a. Pinumbers will be the pin number of the appropriate LED
    b. We will be using the output mode for pin mode
    c. You can use whichever speed you want
    d. Use the push/pull for the output type
    e. No pull-up or pull-down control
    f. *Refer to the lecture slides if you don't remember what each of these configurations means*
2. Call the function responsible for initialization of the GPIO and pass in the targeted port and the pin configuration struct

iii. Do the same thing for the other LED

c. In the toggle LED function
    i. Create a switch statement that takes the input argument and determines which LED needs to be toggled
    ii. Toggle the appropriate LED
        1. Call the GPIO function needed to toggle, pass in the port, and pin number
    iii. Do the same thing for the other LED

d. In the disable LED function
    i. Create a switch statement that takes the input argument and determines which LED needs to be disabled
    ii. Disable the appropriate LED
        1. Call the GPIO function responsible for writing to a specified output pin pass in the port, pin, and what we want to write to it.
    iii. Do the same thing for the other LED

e. In the enable LED function
    i. Create a switch statement that takes the input argument and determines which LED needs to be enabled
    ii. Enable the appropriate LED
        1. Call the GPIO function responsible for writing to a specified output pin pass in the port, pin, and what we want to write to it.
    iii. Do the same thing for the other LED

8. Create the scheduler code:
    a. We need to create two macros to represent events in the header file.
        i. Create a bit mask to access the 0th bit. This will be for the LED Toggle event. That is:

```
#define LED_TOGGLE_EVENT    (1 << 0)
```

        ii. Create a bit mask to access the 1st bit. This will be for the Delay event
        iii. *We will continue to shift the bit 1 left for each new event we add in later labs*
    b. In the source file, we must do the following:

        i. Create a **static** variable of uint32_t that will contain the events that need to be run
            1. Name this <u>‘scheduledEvents’</u>
        ii. We will then need to add functionality that will add, remove, and return the contents of ‘scheduledEvents’
            1. In the function to add a scheduled event:
                a. Given the input argument, that should be a valid bit of a 32-bit value. Make sure to **set** that bit the same way you would set a bit in a register
            2. In the function to remove an event:
                a. Given the input argument, that should be a valid bit of a 32-bit value. Make sure to **clear** that bit the same way you would set a bit in a register
            3. In the function responsible for returning the scheduled event, return the contents of ‘scheduledEvents’

9. Create the Application **source** code
    a. Write code for the three application LED initialization functions
        i. For two of the three functions, call the appropriate function from the LED driver.
            1. One function call to the LED driver should be responsible for initializing a given LED.
        ii. For the third function, you’ll need to call either two LED driver functions OR two of the application LED init functions. The choice is yours!
    b. Write code for toggling each LED
        i. A toggle function should be created for both the Red LED and Green LED
            1. *One of the functions won’t function properly if you only configure one of the LEDs. We still expect the function to still be there.*
        ii. These functions should call the function responsible for toggling in the LED driver
    c. Write code for activating each LED
        i. An activate function should be created for both LEDs
        ii. These functions should just call the function responsible for enabling/activating from the LED driver
    d. Write code for deactivating each LED
        i. A deactivate function should be created for both LEDs
        ii. These functions should just call the function responsible for disabling/deactivating from  the LED driver
    e. Write the application delay function
        i. Create a local array with your name
            1. When defining the size, use the macro you defined in the ApplicationCode header file
        ii. Create a destination array with the same size as your name
        iii. Create a nested for-loop

        1. The outer for loop will iterate through the “time to delay” (the macro you created in the ApplicationCode header file
        2. The inner loop will iterate through each letter of your name array and store the letter in the destination array

    f. Write code for application initialization function
        i. Initialize the RED LED
            1. Use one of the functions you created above
        ii. We also want to add two events to the scheduler: one to toggle the LED and one to execute the delay
            1. Call the function that adds scheduled events to add the LED toggle event
            2. Call the function that adds scheduled events to add the Delay event

10. Integrate the functionality for main.c
    a. Suppress warning regarding the FPU by doing the same process we did in Lab 1
    b. Include necessary files
        i. *Remember the coding hierarchy...*
    c. Populate main()
        i. Call your application code init function
        ii. Create a local variable that will dictate the events to run
            1. You could name it ‘eventsToRun’ if you’d like
        iii. Populate your forever loop
            1. Get the events to be run, and store that value into the local variable you made above
            2. Toggle the LED if the LED toggle event is scheduled
                a. Using the macro for the LED toggle event, use a bitwise operation to compare it with the events to be run
                b. Call your toggle LED function (if the condition above is met)
            3. Execute the delay if the delay event is scheduled
                a. Using the macro for the delay event, use a bitwise operation to compare it with the events to be run
                b. Call your delay function (if the condition above is met)
11. Debug your code (if necessary)
    a. You may need to set breakpoints, step through functions, and/or view registers
12. Verify the LED is flashing!
    a. If you are still having issues, continue to debug.
13. Verify there are no warnings
    a. If you have a warning for "passing argument 1…. from incompatible pointer type [-Wincompatible-pointer-types]"
        i. You may be passing the argument wrong. Maybe it needs the variable's address, or maybe it just needs the variable.
        ii. *This warning may be due to something related to the keyword static; what may be happening here?*

   b. If you have a warning regarding an unused variable in the delay function
      i. Add the attribute [[maybe_unused]] to the array that is “unused”
         1. This attribute will be placed just before (on the same line) the data type of the array
         2. *This tells the compiler that we know this may not be used, so it doesn’t need to annoy or complain about it.*
         3. *This has its issue because the IDE will highlight it as an error, but your code compiles with this not being a warning or error….*
            a. *Why is this? What do you think causes this?*

14. Export your project
15. Turn in the project (.zip file) into Canvas by the due date.

Acceptance Criteria:

- Code compiles and can be downloaded to the board
- The **RED LED** flashes at a **consistent** rate
- Printf should not be used nor should the OCD debug probe be used
  - If used, 30 points will be deducted from the final score
- Use of the generated HAL code is prohibited, if you use the HAL, you will forfeit ALL points for this assignment.

Grading rubric:
This lab will be worth 150 points. The breakdown of grading will be given below.

1. Code Compilation (20 points)
   a. Full credit - Code Compiles with 0 errors and 0 code warnings
   b. Partial Credit - Code contains warnings (-10 points for each code-related warning from the 20 points possible)
   c. No credit - Code does not compile (Student has at least one error)
2. Code Standards and Hierarchy (20 points)
   a. Proper naming of functions/files
   b. Proper layering of files
   c. For each violation, 5 points will get subtracted from the 20 points possible
3. Code functionality (40 points)
   a. Does the **RED LED** flash at a consistent rate?
4. Project Exporting (10 points)
   a. Was the project exported right with the appropriate naming?
5. Lab Attendance (10 points)
   a. Did the student attend lab sessions appropriately and consistently?
6. Interview Grading (50 points)
   a. Does the student understand a basic concept utilized in their code? (15 Points)
   b. Does the student understand a semi-complex concept utilized in their code? (25 points)
   c. Does the student understand or partially understand a semi-advanced concept utilized in their code? (10 points)