Lab 2: Flashing an LED
Due Date: September 22, 2023, at 5 p.m.
START EARLY!

Prerequisites:

- Completion and full understanding of the Lab 2 Prelab
    - **This is VERY important**. Concepts and instructions of this lab will be very confusing if the prelab is not understood fully
    - The instructional staff will be instructed not to answer questions that can easily be found in the prelab! You can ask questions about the prelab if things are confusing.
- 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
- Pick a color: Red or Green

High-Level Overview:
This lab will configure the GPIO (General Purpose Input-Output) pins to toggle an LED after a fixed delay. This will result in the LED flashing at a fixed rate. The rate 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. For this lab, we will have two events, 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 <u>underlined</u>, we expect the data type to be named that. You WILL lose points if it is not named that. This is a part of the coding guidelines.

Refer to the prelab for Coding Hierarchy.

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 source in the ‘src’ directory:
    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 as directed in the specified files:
    a. In “ApplicationCode.h”:
        i. 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. One prototype that will initialize all of the application code, more specifically, initialize the appropriate hardware for the given application. No input arguments are needed.
            1. It is recommended you name this “Application_Init()” or something similar
            2. Unlike other driver files, you do not have to prepend “App” or “Application” to prototype/function names but you can if you want to
        iii. Three prototypes that initialize an LED. All of them do not have input arguments
            1. One for the Green LED
            2. One for the Red LED
            3. One for both LEDs
                a. *Note - Although this lab will only require one LED to flash, we will still have the prototypes for the other LED, as we will eventually need them in later labs*

iv. Two prototypes that toggle the LEDs. All of them do not have input arguments
    1. One for the Green LED
    2. One for the Red LED
v. Two prototypes that activate (turn on) an LED. All of them do not have input arguments
    1. One for the Green LED
    2. One for the Red LED
vi. Two prototypes that deactivate (turn off) an LED. All of them do not have input arguments
    1. One for the Green LED
    2. One for the Red LED
vii. One prototype to act as a delay function
    1. This will take a uint32_t argument that will dictate the time to be delayed

b. In “Scheduler.h”:
    i. Create the following prototypes **verbatim** (ie name the prototypes how they are named below):
        1. getScheduledEvents - this will return the scheduled events, which will be of type uint32_t
        2. addSchedulerEvent - this takes in a uint32_t value that will be the event to be scheduled.
        3. removeSchedulerEvent - this will also take a uint32_t value, and that will be an event to be removed

c. In “GPIO_Driver.h”:
    i. Note - All functions within GPIO Driver should have ‘GPIO_’ prefixed to the function, failure to abide by this will lead to a significant point deduction.
    ii. One prototype that will initialize a GPIO configuration
        1. This function will take in a **pointer** argument of type ‘GPIO_Handle_t’ (what this is will be explained later)
    iii. One prototype that will enable or disable a clock for a given GPIO port.
        1. This will need to take two arguments, one will be a pointer argument of type ‘GPIO_RegDef_t’ and an argument of type uint8_t that will be used to dictate if we are enabling or disabling a clock
    iv. One prototype to toggle an input pin of a given port
        1. This will have the input arguments of pointer type ‘GPIO_RegDef_t’ that will dictate which port and a uint_8 that will dictate the pin number
    v. One prototype to write a value to an input pin of a given port
        1. This will have the same argument set of the toggle function AND will have an additional argument of type uint8_t, and this will be the value to be written to the pin

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. For this lab, all prototypes in this driver will have the same input argument
        1. That is, of type uint8_t and will dictate which LED will be operated on
    iii. One prototype that will initialize a specific LED
    iv. One prototype that enables the clock for a given LED
    v. One prototype that enables (activates) an LED
    vi. One prototype that disables (deactivates) an LED
    vii. One prototype that toggles an LED

5. Create microcontroller-specific macros and register maps
    a. We will be using the reference manual, more specifically, the memory map sections (Section 2.3).
    b. We care about two components for this lab; one will be for 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 you want (Green 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 what peripheral or bus it is 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. Make macros of the **base address** of the following:
            1. The entire peripheral address space
            2. The bus that the two peripherals we will be interacting with is on
            3. The base addresses of both peripherals we will be interacting with. RCC and GPIOG.
                a. Note the value the macro takes should not be hardcoded; it should use the bus peripheral address and add an offset.
                    i. For example, (XXX_BASE_ADDR + 0x1800)
            4. Create the typedef struct ‘GPIO_RegDef_t’ that will represent the register map for GPIO
                a. There should be 9 members of the struct. Note the alternate function registers should be combined into a single array of size 2.
            5. Create the typedef stuct ‘RCC_RegDef_t’ that will represent the register map for RCC

        a. This will be a lot of typing and careful coordination, there will be numerous reserved sections which you will also need to include
            i. Although they cannot be “accessed”, they still have a role in the register memory map
        b. 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
    6. Create a macro for GPIOG that will act as a pointer of type GPIO_RegDef_t to the GPIOG base address.
    7. Create a macro for the RCC that will act as a pointer type of ‘RCC_RegDef_t’ to the RCC base address.
    8. Create two macros, one to enable and one to disable, the clock for GPIOG
        a. 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 *(refer to the prelab if needed)*
    9. 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, 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;

typedef struct
{
   GPIO_RegDef_t* pGPIOx;           // GPIO port
   GPIO_PinConfig_t GPIO_PinConfig; // The pin configuraitons
```

```
}GPIO_Handle_t;
```

   i. *GPIO_PinConfig_t is used to act as the configuration struct that clients will use when trying to configure a GPIO. When developing the LED code, we will go more in-depth on the purpose of these struct members*
   ii. *GPIO_Handle_t contains information used to initialize the GPIO and set the mode appropriately. This will be referred to as the handle struct.*
   iii. Create a set of macros for each the GPIO pin numbers
      1. There should be a total of 16 Macros in this set
   iv. Create a set of macros for all GPIO pin mode configurations
      1. There should be 4 modes
   v. Create a pair of macros for GPIO output type (OPType) configurations
   vi. Create a set of macros for the GPIO pin speed configurations
   vii. Create a set of macros for the GPIO pin push-up/pull-down control configurations
   viii. 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)
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 properly set/clear the proper bit
      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. *Remember, the handle struct will refer to this function's input argument, which is of type GPIO_Handle_t*
   ii. Create a local variable of type uint32_t that will be used as a ‘temporary” (temp) variable
   iii. Configure the port mode (MODER) register with the appropriate mode from the handle struct
      1. Assign the temp variable the pin mode shifted by the pin number times 2.

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

    a. *Note - pGPIOHandle 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*
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

```
pGPIOHandle->pGPIOx->MODER &= ~(0x3 << (2 * pGPIOHandle->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

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

iv. Configure the output speed register
    1. Same process as configuring the port mode register, but instead using the pin speed configuration provided in the handle struct
v. 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 handle struct
vi. Configure the output type register
    1. Same process as configuring the port mode register however, use the output type member of the handle struct **AND** there are only two possible options for configuration which means there is only 1 bit that 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.
vii. 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 due to 8 being 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 due to us referring 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 handle 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 from the one you selected, but the code will not need to be populated.*
    b. Create two static structs of type GPIO_Handle_t, one to configure the red LED and one to configure the blue LED
    c. 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 handle struct elements with the appropriate configurations
                a. Use the appropriate port macro for the port member of the handle struct
                b. For the configuration portion of the struct,
                    i. Pinumbers will be the pin number of the appropriate LED
                    ii. We will be using the output mode for pin mode
                    iii. You can use whichever speed you want
                    iv. Use the push/pull for output type
                    v. No pull-up or pull-down control
                    vi. 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 struct by reference
        iii. Do the same thing for the other LED
    d. 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
                a. No magic numbers here. Use the values stored in members of your struct

iii. Do the same thing for the other LED

e. 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.
         a. No magic numbers here. Use the values stored in the members of your struct and utilize macro(s)
   iii. Do the same thing for the other LED
f. 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.
         a. No magic numbers here. Use the values stored in the members of your struct and utilize macro(s)
   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
      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
      iv. Be sure to include whichever library is needed for uintx_t support
   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 ‘scheduledEvents’
      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 with iterate through each letter of your name array and store the letter into the destination array
        3. *If you are struggling with grasping this, please ask for help and refer to the C refresher.*
        4. *It is to note this loop is an AWFUL programming technique. There is no worse way to create a delay than this way. We will touch on better ways to do this in a later lab.*
f. Write code for application initialization function
    i. Initialize a LED (green or red, depending on which one you choose).
        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. Why isn't the LED flashing?
        i. Using the debugger, verify that you are reaching the code you expect to reach
            1. You can achieve this by stepping through the code or setting breakpoints
            2. Verify that the events are correctly scheduled, and the code that is supposed to execute executes
        ii. I left out an important step when initializing things……
            1. Refer to the prelab if you're stuck. There is something we MUST ALWAYS do if we want a peripheral to work…..
12. Squash the bug (If needed)
    a. Hint: *Tick. Tock. Tick. Tock. Tick. Tock*
    b. This line or lines of code can be put in two different places, the upper-level driver (LED Driver) initialize function, or the lower driver (GPIO Driver) initialize function. The choice is yours!
        i. *In industry, we would ask, do we want to client of the GPIO Driver (LED driver) to be responsible for handling this or do we want to handle it?*
13. Verify the LED is flashing!
    a. If you are still having issues, continue to debug.
    b. Refer to the debugging techniques documentation if needed
14. 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 address of the variable, 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 us and 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?
15. Export your project
16. Turn in the project (.zip file) into Canvas by the due date.

Acceptance Criteria:

- Code compiles and can be downloaded to the board
- One LED flashes at a **consistent** rate

Grading rubric:

This lab will be worth 100 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 (-5 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 one 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?