Lab 4: Timers and PWM
Due Date: October 27th, 2023 at 5pm

Prerequisites:

- Understand the concept of frequency and how it relates to timers
    - This was a lecture topic
- Understand how to program multiple interrupts
    - This includes configuring interrupt priority
- How to closely read the reference manual
    - Seriously, if you do not know how to read and understand what you need and what you don't from the reference manual, this lab may be extremely difficult
- Solid understanding of C
    - Strong understanding of structs and the fact you can have nested structs
- The ability to use compiler switches
- The ability to use deductive reasoning to implement tasks with limited information
- Read and understand the prelab

Background information:

This lab will be used to implement the knowledge of timers, interrupts, and code logic. There will be room in this lab for you, the student, to implement things differently if you choose to. This lab, similar to later labs, will only provide essential information. Some instructions may *seem* incomplete, but they are not, as you should have the knowledge and understanding of embedded systems and coding to develop the proper code and/or procedures for a given instruction. This will be the last lab where we create a driver from the ground up, so look at this as a way to assess your understanding of developing your own HAL (Hardware Abstraction Layer) driver.

Lab Instruction:

1. Verify you understand the prelab.
2. Read the "General-Purpose timers (TIM2 and TIM5)" section of the reference manual
    a. This can be a "skim" read, but make sure to glance at the main features and registers
3. Verify you have an archived copy of Lab 3 in a safe location
4. Either copy or rename Lab 3
    a. Follow the naming guidelines per usual
5. Create the following fileset
    a. Timer_Driver.*
6. Create the following prototypes:
    a. In Timer_Driver.h:
        i. *There is no need to pre-append 'Timer' to every function but verify that each function/prototype name has the word 'Timer' in it.*
        ii. *All prototypes, but one, should not return anything. I will explicitly state which prototype will return something.*
        iii. Create a prototype that will be used to initialize the timer, this will take in an argument of type 'Timer_Handle_t'

iv. Create a prototype that will initialize the Channel 1 of a specified timer. This will also take in an argument of ‘Timer_Handle_t’
v. Create a prototype that will control the clock. This will take in an argument of pointer type of ‘GPTIMR_RegDef_t’ and a uint8_t that will dictate if we are enabling or disabling the clock
vi. Create three prototypes one for starting the timer, one for stopping the timer, and one for resetting the timer, and they will all take an argument of pointer type 'GPTIMR_RegDef_t'
vii. Create a prototype that will return the timer value, this will return a 'uint32_t' and take in an argument of pointer type 'GPTIMR_RegDef_t'
viii. Create a prototype responsible for enabling/disabling the timer interrupt. This will take an argument of pointer type 'GPTIMR_RegDef_t' and a 'uint8_t' that will signify if we are enabling or disabling the interrupt
ix. Create a prototype that will change the priority of the timer interrupt, this will take an argument of pointer type 'GPTIMR_RegDef_t' and a 'uint8_t' to represent the new IRQ priority

b. In LED_Driver.h:
    i. You must create a set of prototypes/functions for each of the two timers used. In this lab, we will be using timer 2 and timer 5. Both timers should have the following prototypes. All prototypes within these sets do not return or take in any argument, These prototypes should have 'LED' prepended to them:
        1. A prototype to initialize the timer
        2. A prototype to start the timer
        3. A prototype to stop the timer
        4. A prototype to reset the timer
        5. A prototype to elevate the timer's IRQ priority
        6. A prototype to restore the timer's IRQ priority to default
        7. *You should have 12 new prototypes, six for Timer 2 and six for Timer 5*

c. In interruptControl.h:
    i. If you haven't already (from previous labs potentially), create a prototype that will configure the priority of a certain interrupt, this will take in two uint8_t arguments, one for the IRQ number and another for the IRQ priority

7. Add microcontroller-specific details
    a. If you have not already, in prior labs, add the first Interrupt priorities register and its address for the ARM Cortex
        i. This should be very similar to how we programmed the other interrupt registers for the ARM cortex
        ii. We are only doing the base register because there are technically more than 40+ registers for the interrupt priority
    b. Add the Base Address Macros for TIM2 and TIM5
    c. Create the register typedef struct GPTIMR_RegDef_t

d. There should be 21 members of the struct.. don't worry; your driver will not need to access all of them 🙂
e. Create the Macros for TIM2 and TIM5 that will allow us to access their respective registers
f. Create Macros for TIM2 and TIM5 that will represent their clock offset
g. Create a SINGLE macro (“function-like”) that will take in the offset and enable the clock associated with the offset
h. Create a SINGLE macro (“function-like”) that will take an offset and disable the clock associated with the offset

8. Add the appropriate code to the Timer Driver
    a. *Note - Timer 2 will be used interchangeably with TIM2. Timer 5 will be used interchangeably with TIM5.*
    b. In the header file
        i. Create a typedef struct for channel configuration. Each member of this struct should represent the following:
            1. Channel mode selection
            2. Output compare selection
            3. Output compare fast enablement
            4. Output compare preload enablement
            5. Output compare clear enablement
            6. Output channel interrupt enablement
            7. Capture Compare Value
            8. *These can be of whatever type you feel is necessary. Some may HAVE to be a certain type (like a uint32_t, for example). It may be convenient to make some of them as type boolean. They can also be named whatever you want but make sure it makes sense* 🙂
        ii. Create a typedef struct for the general-purpose timer configuration named ‘GPTImer_Config_t’. The member. Each member of this struct should represent the following:
            1. Auto Reload Value
            2. Master Mode Selection
            3. Clock Division Value
            4. Prescaler Value
            5. Center Aligned Mode Selection
            6. Auto reload buffer enablement
            7. Timer count-down mode enablement
            8. Interrupt update enablement
            9. Disable update event
            10. One pulse mode enablement
            11. Channel 1 configuration (this should be the same type you created above, that is, the timer channel configuration type)
            12. *These can also be of whatever type you feel is necessary. Some may NEED to be a certain type (like a uint32_t, for example). It may be convenient to make some of them as type boolean. They*

can also be named whatever you want but make sure it makes sense 🙂

iii. Create a typedef struct named ‘Timer_Handle_t’ that has two members:
    1. The timer to be configured. This will be of pointer type ‘GPTIMR_RegDef_t’
    2. Timer configuration details of type ‘GPTImer_Config_t’

iv. Create some macros that will be used for timer configuration, if there are only two macros used to configure a setting (a high and low value), it would be worth it to make that member in whichever struct a boolean. Therefore you won’t have to make a macro for it
    1. *Remember, when configuring a peripheral, we use these macros to help us determine which bit(s) we need to flip in a specific register(s). This information is in reference manual*
    2. Remember, “magic numbers” should not be used when configuring the driver, so create a macro(s) to prevent using magic numbers by having a macro take its place.

c. In the source file
    i. In the function responsible for initializing the timer:
        1. Like other initialization functions in prior labs, you will use the configuration options from the Timer_Handle_t struct to set the appropriate bits in the correct timer register(s). This concept is not new.
        2. Clear and then set only the clock division bit field in the appropriate register
        3. Clear and then set the centered aligned mode selection in the appropriate register
        4. Configure the appropriate bit for if down counting mode should be enabled or not
        5. Configure the appropriate bit for if auto-reload buffer should be enabled or not
        6. Configure the appropriate bit for if one pulse mode should be enabled or not
        7. Configure the appropriate bit for if we should be disabling update events or not
        8. Configure the appropriate bit for if the Interrupt update functionality should be enabled or not
            a. *This may be kind of confusing. Read the documentation to determine what counts as an “update event” if you are curious ;)*
        9. Store the prescaler value in the appropriate register
        10. Store the auto-reload value in the appropriate register
        11. Call the Channel 1 Init function and pass the handle struct into it.
            a. We have not created this function yet, but you can still add the function call

12. Determine if we need to enable the timer's interrupt
    a. You can do this by doing some logical operation of both struct members that determine if the interrupt mode should be enabled
13. Depending on if the timer's interrupt needs to be enabled or not, enable or disable interrupt by calling the appropriate timer function

ii. In the function responsible for initializing channel 1 for the timer:
1. Clear and set the appropriate bits in the CCMR1 register for the output compare mode
2. Clear and set the appropriate bits in the CCMR1 register for the capture/compare selection mode
3. Configure the appropriate bit in the CCMR1 register if output compare clear enable should be enabled or not
4. Configure the appropriate bit in the CCMR1 register if output compare preload enable should be enabled or not
5. Configure the appropriate bit in the CCMR1 register if output compare fast enable should be enabled or not
6. Configure the appropriate bit in the DIER register if the Capture/Compare 1 interrupt should be enabled or not
7. Store the Capture Control value in the CCR1 register

iii. In the function responsible for the timer clock control
1. Enable or disable the appropriate clocks, TIM2 or TIM5, based on the input arguments

iv. In the function responsible for starting a timer
1. Set the appropriate bit in the CR register to enable the timer specified by the input argument

v. In the function responsible for stopping a timer
1. Clear the appropriate bit in the CR register to disable the timer specified by the input argument

vi. In the function responsible for resetting the timer
1. Set the CNT value to zero for the timer specified by the input argument

vii. In the function responsible for getting the timer
1. Return the CNT value for the timer specified by the input argument
2. *The application code will not use this function, but it is good to have for debugging and potentially later use cases*

viii. In the function responsible for configuring the timer interrupt
1. Enable or disable the proper interrupt given the input arguments
    a. Calls to the functions in the InterruptControl fileset should be made here

ix. In the function responsible for configuring the timer interrupt priority
1. Configure the appropriate interrupt priority using the input arguments

        a. Calls to the function in the InterruptControl fileset should be made here
9. Add the appropriate code to the Interrupt Control fileset
    a. In the header file
        i. Create macros for the IRQ numbers for Timer 2 and Timer 5
    b. In the source file
        i. In the function responsible for configuring interrupt priority
            1. Create a variable that will be used to select the proper register, this will be the IRQ number divided by 4
            2. Create a variable that will be used to select the Bit field this will be the IRQ number divided by 4
            3. Create a third variable that will hold the shift amount this will be the bitfield multiplied by the amount of bits in each priority offset
                a. Read the documentation to understand this number
            4. Using pointer arithmetic, dereference the sum of the base address and the register select variable to get the register we want to access, and then clear the appropriate bits using a bit mask and the shift amount variable created above
            5. Then we will do the same logic, set the bits we want
10. If you haven't decided yet, decide which Timer you want on for which LED. For example, the Red LED uses TIM2, and the Green LED uses TIM5.
11. Add appropriate code to the LED driver
    a. Create three macros, one will have TIM2 default IRQ priority, one will have TIM5 default priority, and one will be the elevated priority; this can be anything but shouldn't be greater than the button's priority but should be greater than both timer's default priority
        i. Refer to the reference manual if needed, IRQ priority and IRQ number are two different things
    b. In the function responsible for initializing timer 2
        i. Create and configure a config variable of type 'Timer_Handle_t' to give timer 2 the proper configuration. Read and understand the reference manual regarding timers to configure them properly. Things to consider:
            1. Regarding the auto-reload and channel capture/compare values:
                a. These can be any number(s) you want as long as you meet the acceptance criteria, but it may save you time to properly figure out how to determine this value rather than just trying random values
            2. Hint #1: A lot of these modes can be kept as their default value
            3. Hint #2: Pay attention to what classifies what an "event" entails
        ii. Enable the clock for the appropriate timer
        iii. Call the initialization function with the proper input argument
    c. In the functions responsible for starting, stopping, and resetting timer 2:
        i. Call the appropriate functions from the timer driver with the appropriate command argument

d. In the functions responsible for elevating and restoring timer 2's IRQ priority
    i. Call the appropriate function from the Interrupt Control fileset with the appropriate arguments
e. Repeat all these steps for Timer 5 (TIM5)
    i. Most configuration options can be the same but the prescaler, auto-reload value, clock division value, and the channel capture compare value MAY need to be different than Timer 2 (TIM2).

12. Add appropriate code to the Application Code
    a. In the source file
        i. Create a macro that will be used for the compile switch, name this something along the lines of 'USE_LIMITED_RESOURCES' and set it equal to zero
            1. The term 'when using limited resources' (or similar) will be used later on. This term means when the macro you just created is equal to 1
        ii. Create some static variables to keep track of the status of each, LED
        iii. You may also, if you haven't already, add macros that will explain if a LED is off or not
            1. These values will be assigned to the static variables you just created
        iv. In the function responsible for initializing the application code
            1. Deactivate both LEDs
            2. Set both static variables that describe their state appropriately
            3. Initalize both timers
            4. Start both timers
            5. Verify the delay function event is still scheduled
        v. Adjust the Button IRQ handler
            1. If we are using limited resources, we should deactivate both LEDs and update their statuses accordingly
            2. If we are not using limited resources, we should just reset both timers
        vi. Create and populate Timer 2's interrupt handler
            1. Disable the interrupt by calling appropriate functions with appropriate input arguments
            2. Determine which interrupt for Timer 2 is pending
            3. If the correct interrupt flag is high, do the following:
                a. If using limited resources
                    i. Toggle the appropriate LED associated with this timer, but implement logic to ensure this LED cannot be on if the other LED is on
                    ii. Also, if this timer's LED is on, it (the timer) should have elevated IRQ priority. If the LED is off, it should have its default priority

        b. If not using limited resources, just toggle the appropriate LED
        c. Be sure to clear the appropriate “pending” flag
      4. Clear the pending interrupt in the NVIC by calling the appropriate function and input argument(s)
        a. Yes this is different from step 3c.
      5. Enable the interrupt by calling the appropriate interrupt function with appropriate input arguments
    vii. Create and populate Timer 5’s interrupt handler
      1. The same process as Timer 2, but this time make sure you toggle the correct LED and use the correct IRQ priority functions
13. Download the code to your board, and debug it if necessary
    a. If your button appears to be not working, odds are, it is because of a potential bug with the STM hardware
      i. I say potential because I cannot say for sure it is THEIR (STM’s) fault, but it doesn’t appear to be ours ;)
      ii. Hints on finding and working around this bug:
        1. Set two breakpoints, one after enabling the interrupt for the button and one just inside the Timer2 Interrupt (so the code stops before disabling the interrupt)
        2. Step through and monitor the NVIC registers accordingly..
        3. Ultimately, there is an unwanted action happening when writing to one of the registers. It is doing more than what we are telling it to do..
        4. Once you figure out the bug, add a workaround.
          a. This workaround should add two lines, one in the TIM2 IRQ handler and the Button’s IRQ handler
          b. These lines will “undo” the unwanted behavior that one of the functions causes
14. Verify that all acceptance criteria are met
    a. Sometimes the button needs to be pressed multiple times (or at a particular time) to get things back in “sync”
15. Export your project appropriately and turn it into Canvas

Acceptance Criteria:

- When not using “limited resources:
  - Both LEDs flash at a fixed rate and overlap when they are off and on (20 points).
- When using “limited resources” (30 total points) :
  - Both LEDs flash at a fixed rate, BUT they are never on simultaneously (15 points).
  - An LED should not turn on and off three consecutive times without the opposite LED turning on and off (10 points).
  - The button should cause both LEDs to go low (5 points).

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 (-3 points for each warning)
    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 (50 points)
    a. Each Acceptance criteria point will be worth their specified amount
4. Project Exporting (5 points)
    a. Was the project exported right with the appropriate naming?
5. Lab Attendance (5 points)
    a. Did the student attend lab sessions appropriately and consistently?