Lab 4: Timers
Due Date: October 25th, 2024 @ 11:59 PM
Estimated Complexity: High
Estimated Time to Complete: 3 weeks
START EARLY!!

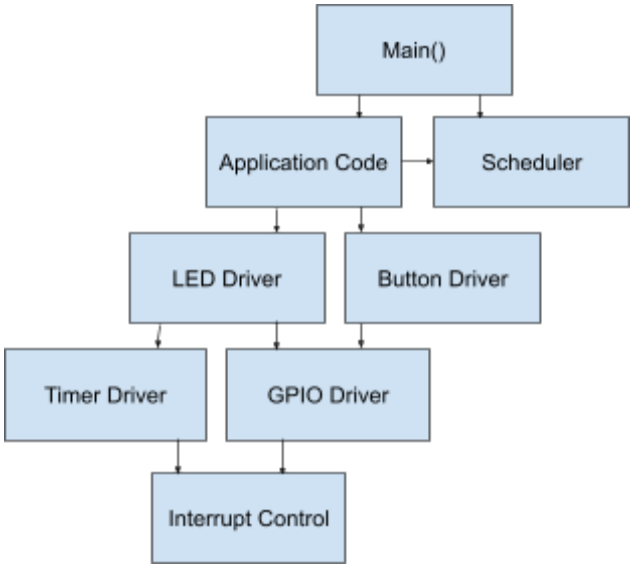
(Hypothetical) Engineering Request: You have been tasked by a security company to make a program that will assist with their silent alarm system. There are two parts to this system. The first part is that an LED must flash at a fixed rate to indicate the alarm is on. This should be easily changeable in the firmware and based on real-time intervals. For example, it should be on for 3 seconds and turn off for 3 seconds. The second part of the program will need to show how long the user holds the button, display the LED for that time, and then turn it off for that time. In other words, we control the rate at which the LED flashes without changing the firmware. Using the LED to tell us how long the button has been pushed is just for prototyping in the field. That PWM/frequency will be used by the user of the silent alarm to relay information to the 24/7 security response center. However, each part needs to be executed independently.

Prerequisites:

- Understand the concept of frequency and how it relates to timers
    - This was a lecture topic
- Understand how to program multiple interrupts
- How to get information to and from within an interrupt
- 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
    - Knowledge of writing and implementing your own code logic
- 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.

Coding Hierarchy:



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, except 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 specified timer, this will take in two arguments, one of ‘GPTIMR_RegDef_t’ type and one of ‘GPTimer_Config_t’ type
        iv. 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
        v. 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'
        vi. 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'

vii. Create a prototype that is 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

viii. Create a prototype that will return a timers auto-reload value given the argument of pointer type ‘GPTIMR_RegDef_t’

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. *You should have 8 new prototypes, 4 for Timer 2 and 4 for Timer 5*

ii. You will create and refer to the following prototypes later, but these will only interact with Timer 5 and should be named well.

1. A prototype to return the current AutoReload Value
2. A prototype to return the current Count Value
3. A prototype to reconfigure the auto-reload value. This should take in a uint32_t perimeter that will be the new value to be reconfigured
4. A prototype to start Timer 5 from a specific starting count. This should take in a uint32_t perimeter that will be the new start value

7. Add microcontroller-specific details
   a. Add the Base Address Macros for TIM2 and TIM5
   b. Create the register typedef struct GPTIMR_RegDef_t
   c. There should be 21 members of the struct.. don't worry; your driver will not need to access all of them 🙂
   d. Create the Macros for TIM2 and TIM5 that will allow us to access their respective registers
   e. Create Macros for TIM2 and TIM5 that will represent their clock offset
   f. Create a SINGLE macro (“function-like”) that will take in an offset and enable the clock associated with that offset
   g. 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 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 Selection
    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. *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* 🙂
ii. 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 the 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 GPTimer_Config_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. 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 the timer clock control
    1. Enable or disable the appropriate clocks, TIM2 or TIM5, based on the input arguments

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

iv. In the function responsible for stopping a timer
    1. Set or clear the appropriate bit in the CR register to disable the timer specified by the input argument

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

vi. In the function responsible for returning the auto-reload value
    1. Return the contents in the auto-reload register of the appropriate timer

vii. In the function responsible for getting the timer's count
    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

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
10. Add appropriate code to the LED driver
    a. In the function responsible for initializing timer 2
        i. Create and configure a config variable of type 'GPTimer_Config_t' to give timer 2 the proper configuration and then call the appropriate function(s) to initialize it. Read and understand the reference manual regarding timers to configure them properly. Things to consider:
            1. Refer to the prelab for the useful formula

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

b. 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(s)
c. Repeat all these steps for Timer 5 (TIM5)
   i. Be mindful that your configuration *may* be different from TIM2.
d. Populate the functions that were created to interact with TIM5 specifically
   i. Our naming of these functions should illustrate exactly what we want these functions to accomplish

11. Add appropriate code to the Application Code
    a. *The following instructions will be very high-level and ambiguous. This is by design; you, the engineer, will require a reasonable amount of thinking and experimenting.*
       i. There are multiple ways to get to the desired “solution”
    b. A timer function can be directly called within the IRQ handlers but nowhere else in the application code.
    c. When populating the code for the interrupt handlers, remember the following:
       i. You should disable the interrupt upon entering the handler
       ii. You should then lower the appropriate flag(s)
           1. Always clear the flag in the NVIC
           2. There *may* be another flag you need to clear within the peripheral
       iii. You should then re-enable the interrupt just prior to exiting
    d. In the Header file
       i. Create a macro that will be used for the compile switch, name this something along the lines of ‘DAUL_TIMER_USAGE’ and set it equal to zero
           1. This will be used to switch between Timer 2’s and Timer 5’s usage, as they are used to accomplish two different things.
    e. In the source file
       i. In the function responsible for initializing the application code
           1. Initialize and Deactivate both LEDs
           2. If ‘DAUL_TIMER_USAGE’ is 1, we should call the button interrupt initialization function and the timer 5 initialization function
           3. If ‘‘DAUL_TIMER_USAGE’ is not 1, we should call the initialization function for timer 2 and start timer 2
           4. Verify the delay function event is still scheduled
       ii. Create and populate Timer 2’s interrupt handler
           1. This IRQ will only be compiled in and used when ‘DAUL_TIMER_USAGE’ is NOT 1
           2. Below aren’t instructions per se but hints and things to think about
               a. Verify you know what you want this IRQ to accomplish
               b. Understand exactly what triggers this IRQ
       iii. Adjust the Button IRQ handler

1. This IRQ will only be compiled in and used when ‘DAUL_TIMER_USAGE’ is 1
2. Below aren’t instructions per se but hints and things to think about
    a. You need to be mindful if the interrupt was triggered by a rising or falling edge
    b. You may need to refer if the timer has overlapped or not
    c. You may want to start and stop the appropriate timer within this IRQ
    d. Refer to some of the prototypes that were created in LED_Driver.h as they may be of use here

iv. Create and populate Timer 5’s interrupt handler
1. This IRQ will only be compiled in and used when ‘DAUL_TIMER_USAGE’ is 1
2. Below aren’t instructions per se but hints and things to think about
    a. You will need to keep track of whether you are timing the button input or timing the LED output toggle
    b. You may need to keep track of how many times the timer overlaps
    c. Refer to some of the prototypes that were created in LED_Driver.h as they may be of use here

v. Verify the appropriate IRQ handlers are only being compiled when ‘DAUL_TIMER_USAGE’ is the appropriate value

12. Download the code to your board and debug it if necessary
13. Verify that all acceptance criteria are met
14. Export your project appropriately and turn it into Canvas

Acceptance Criteria:

- When ‘DAUL_TIMER_USAGE’ is 0, The RED LED turns on for 5 seconds every 5 seconds. (15 points)
- When ‘DUAL_TIMER_USAGE’ is 1, the GREEN LED matches the frequency that the user indicated by holding down the button (40 points)
    - For example, If the user holds it for 6 seconds, the GREEN LED should turn on for 5 seconds and turn off for 6 seconds
    - For example, If the user holds it for half a second, the GREEN LED should turn on for half a second and turn off for half a second
        - Visually, this *may* make it seem like the LED stays on. I would advocate for setting breakpoints in specific places to verify.
    - **Other times should also be supported, not just 6 seconds and the half-second**
    - You should only have to match the input frequency once during the code execution
        - I.e. if you want to match another frequency, the user will have to reset the board

- A compile switch appropriately separates the functionality associated with the RED LED and the GREEN LED (5 points)

Additional Notes:

- 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 (10 points)
    a. Full credit - Code Compiles with 0 errors and 0 code warnings
    b. Partial Credit - Code contains warnings (-5 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 (60 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?
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)