UNIVERSITY of WISCONSIN
UWMILWAUKEE

LABORATORY REPORT

Colin Young
EE 340 Embedded Systems

PROJECT TITLE: Timers & ADCs

LABORATORY REPORT :

INTRODUCTION:

For those of you who have been following my blog posts you are no doubt aware of my last adventure in engineering where I used a 25k flat flex sensor to understand ADC input. This week I will be expanding on the use of ADC input. Instead of just reading in values and printing them to the terminal screen, we will take those inputs and use them to switch between one of three different built-in timers on the NUCLEO-L5522ZE-Q board. The switch will be decided by reading which way the flex sensor is being bent. Be sure to check out the video submitted along with this report to see the final project in action.

REQUIREMENTS:

1) Set up the three timers.

2) Set up the interrupt to trigger the change of timers based on adc input.

3) Configure ADC sensor to read in values.

VERIFICATION:

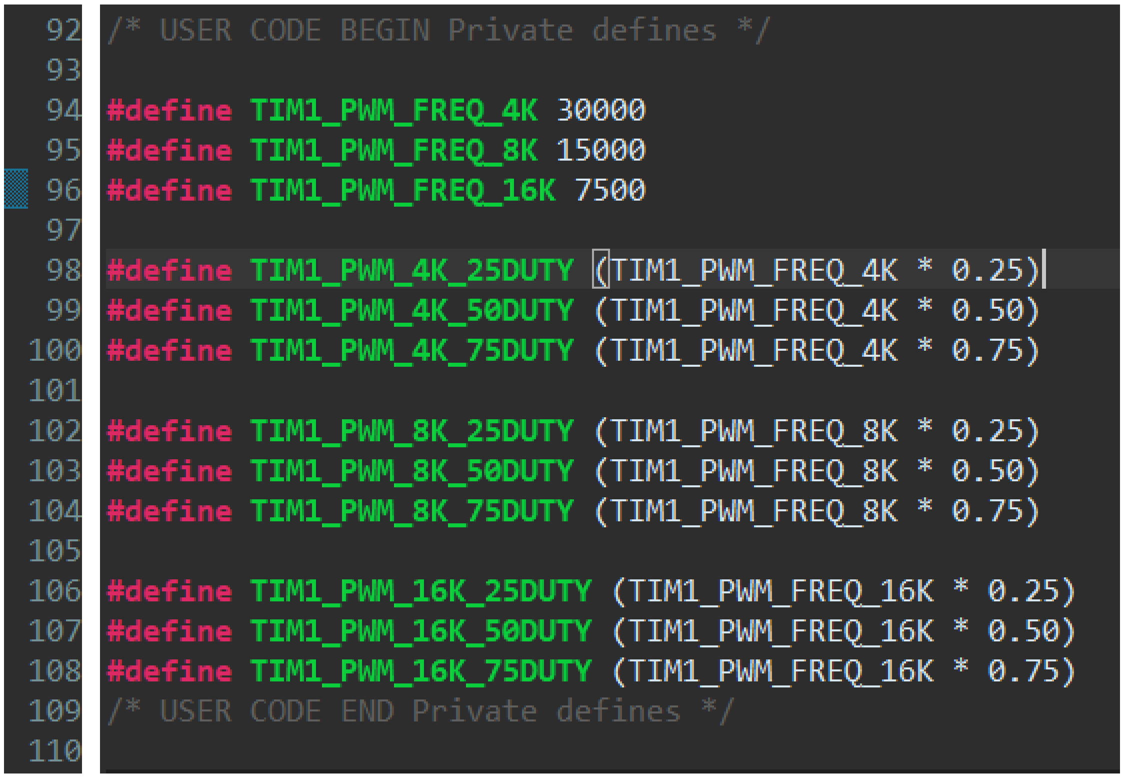
1) For this project we will be utilizing three timers. To properly set up these timers we will need to activate channels one, two, and three on TIM1, changing their settings from disabled to PWM Generation CH1 CH1N in the ioc file. Now that the timers are turned on, they must be given the correct parameters to work correctly. Under the parameter settings, counter settings, the counter mode should be set to center aligned mode1, the period must be set to 30,000, and the auto reload preload should be set to enable. Next we must configure each PMW generation channel pulse. Channel 1 will be set to 7,500, channel 2 will be set to 15,000, and channel 3 will be set to 22,500. All other settings can be left in their default state. In order to use the pins we just set up we must set them to a pin on the nucleo board. In GPIO settings, we set TIM1_CH1N, CH2N, and CH3N to pins A7, B0, and B1, and TIM_CH1, CH2, and CH3, to pins E9, E11, AND E13 respectively. We then move to the main header file where we define each timer, as well as 25%, 50%, and 75% duty cycles for each timer. The code snippet for setting this up can be seen in figure 1 of the technical details section. Finally in the main c file, we can set the initial timer value to 4k, then start each of the timer channels. The code snippet for this can be seen in figure 2 of the technical details section.

2) To set up the interrupt we first need to head back to the IOC file and make sure the TIM1 update interrupt enabled box is checked. We can then set up a function called HAL_TIM_PeriodElapsedCallback function. This function will have no return and have one input parameter TIM_HandleTypeDEF *htim. Inside of the function we set up a simple if statement with four branches. The branches check to see which value the timer is currently set to then use the HAL_TIM_SET_AUTORELOAD and HAL_TIM_SET_COMPARE functions to dynamically switch between the three channels without having to call another time init or “ConfigChannel” function. The code for this function can be seen in figure 3 in the technical details section of this report. At this point, in order to make sure everything is set up properly, we can attach an oscilloscope to each of the timer pins. A picture of the oscilloscope readings is seen in figure 4 of the technical details section.

3) For this step we will be utilizing the ADC code used in our last project. We must first import our EE340 header and source file into the project. The class diagram and flow chart of this file can be seen in figure 5, figure 6, figure 7, and figure 8 in the technical details section of this lab report. We can then set up the flat sensor exactly the same way we did last time, but instead of sending the value to the terminal we save it to a uint16_t variable called pwm. From here we have to set which values we want each clock to be activated at. For this project I chose the cut off values of 2250 and 2500. If the ADC reads in anything below 2250 we would set the clock to the lowest frequency, if it reads in above 2500 it sets to the highest frequency, and in between it is defaulted to the middle frequency.

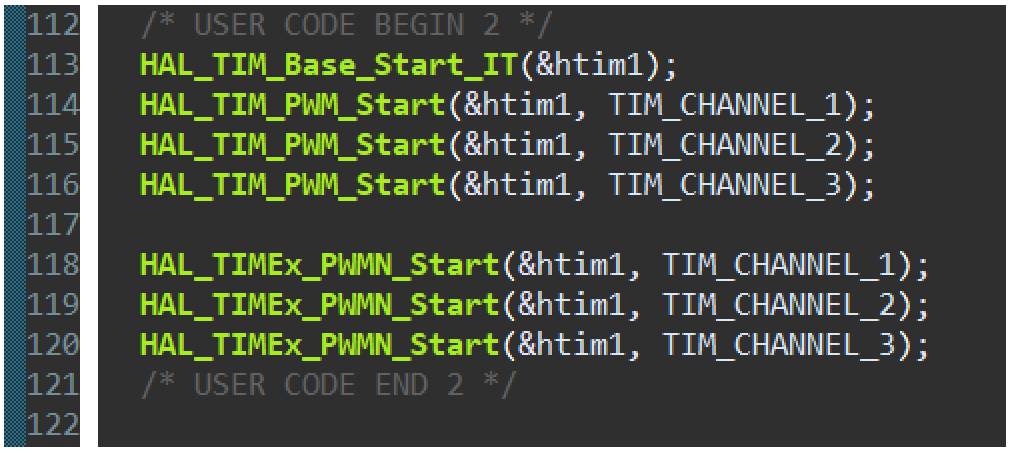
TECHNICAL DETAILS:

Figure 1: Code snippet of time definitions and duty cycles.



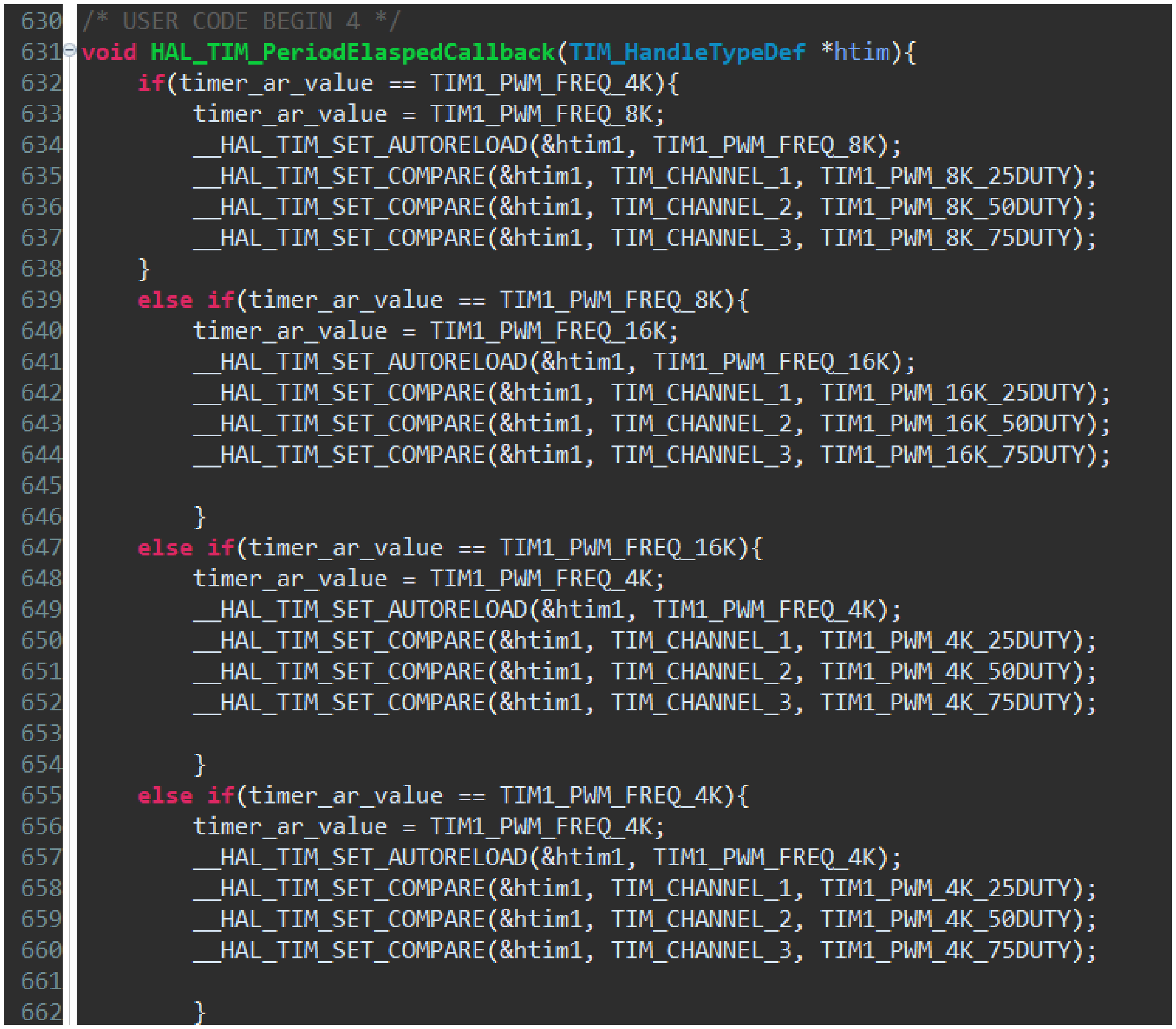
```
/* USER CODE BEGIN Private defines */

#define TIM1_PWM_FREQ_4K 30000
#define TIM1_PWM_FREQ_8K 15000
#define TIM1_PWM_FREQ_16K 7500

#define TIM1_PWM_4K_25DUTY (TIM1_PWM_FREQ_4K * 0.25)
#define TIM1_PWM_4K_50DUTY (TIM1_PWM_FREQ_4K * 0.50)
#define TIM1_PWM_4K_75DUTY (TIM1_PWM_FREQ_4K * 0.75)

#define TIM1_PWM_8K_25DUTY (TIM1_PWM_FREQ_8K * 0.25)
#define TIM1_PWM_8K_50DUTY (TIM1_PWM_FREQ_8K * 0.50)
#define TIM1_PWM_8K_75DUTY (TIM1_PWM_FREQ_8K * 0.75)

#define TIM1_PWM_16K_25DUTY (TIM1_PWM_FREQ_16K * 0.25)
#define TIM1_PWM_16K_50DUTY (TIM1_PWM_FREQ_16K * 0.50)
#define TIM1_PWM_16K_75DUTY (TIM1_PWM_FREQ_16K * 0.75)
/* USER CODE END Private defines */
```

Figure 2: Code snippet of timer initiations.



```
/* USER CODE BEGIN 2 */
HAL_TIM_Base_Start_IT(&htim1);
HAL_TIM_PWM_Start(&htim1, TIM_CHANNEL_1);
HAL_TIM_PWM_Start(&htim1, TIM_CHANNEL_2);
HAL_TIM_PWM_Start(&htim1, TIM_CHANNEL_3);

HAL_TIMEx_PWMN_Start(&htim1, TIM_CHANNEL_1);
HAL_TIMEx_PWMN_Start(&htim1, TIM_CHANNEL_2);
HAL_TIMEx_PWMN_Start(&htim1, TIM_CHANNEL_3);
/* USER CODE END 2 */

```

Figure 3: Code for interrupt.



```
/* USER CODE BEGIN 4 */
void HAL_TIM_PeriodElaspedCallback(TIM_HandleTypeDef *htim){
    if(timer_ar_value == TIM1_PWM_FREQ_4K){
        timer_ar_value = TIM1_PWM_FREQ_8K;
        __HAL_TIM_SET_AUTORELOAD(&htim1, TIM1_PWM_FREQ_8K);
        __HAL_TIM_SET_COMPARE(&htim1, TIM_CHANNEL_1, TIM1_PWM_8K_25DUTY);
        __HAL_TIM_SET_COMPARE(&htim1, TIM_CHANNEL_2, TIM1_PWM_8K_50DUTY);
        __HAL_TIM_SET_COMPARE(&htim1, TIM_CHANNEL_3, TIM1_PWM_8K_75DUTY);
    }
    else if(timer_ar_value == TIM1_PWM_FREQ_8K){
        timer_ar_value = TIM1_PWM_FREQ_16K;
        __HAL_TIM_SET_AUTORELOAD(&htim1, TIM1_PWM_FREQ_16K);
        __HAL_TIM_SET_COMPARE(&htim1, TIM_CHANNEL_1, TIM1_PWM_16K_25DUTY);
        __HAL_TIM_SET_COMPARE(&htim1, TIM_CHANNEL_2, TIM1_PWM_16K_50DUTY);
        __HAL_TIM_SET_COMPARE(&htim1, TIM_CHANNEL_3, TIM1_PWM_16K_75DUTY);

        }
    else if(timer_ar_value == TIM1_PWM_FREQ_16K){
        timer_ar_value = TIM1_PWM_FREQ_4K;
        __HAL_TIM_SET_AUTORELOAD(&htim1, TIM1_PWM_FREQ_4K);
        __HAL_TIM_SET_COMPARE(&htim1, TIM_CHANNEL_1, TIM1_PWM_4K_25DUTY);
        __HAL_TIM_SET_COMPARE(&htim1, TIM_CHANNEL_2, TIM1_PWM_4K_50DUTY);
        __HAL_TIM_SET_COMPARE(&htim1, TIM_CHANNEL_3, TIM1_PWM_4K_75DUTY);

        }
    else if(timer_ar_value == TIM1_PWM_FREQ_4K){
        timer_ar_value = TIM1_PWM_FREQ_4K;
        __HAL_TIM_SET_AUTORELOAD(&htim1, TIM1_PWM_FREQ_4K);
        __HAL_TIM_SET_COMPARE(&htim1, TIM_CHANNEL_1, TIM1_PWM_4K_25DUTY);
        __HAL_TIM_SET_COMPARE(&htim1, TIM_CHANNEL_2, TIM1_PWM_4K_50DUTY);
        __HAL_TIM_SET_COMPARE(&htim1, TIM_CHANNEL_3, TIM1_PWM_4K_75DUTY);

        }
```

Figure 4: oscilloscope pictures of timers

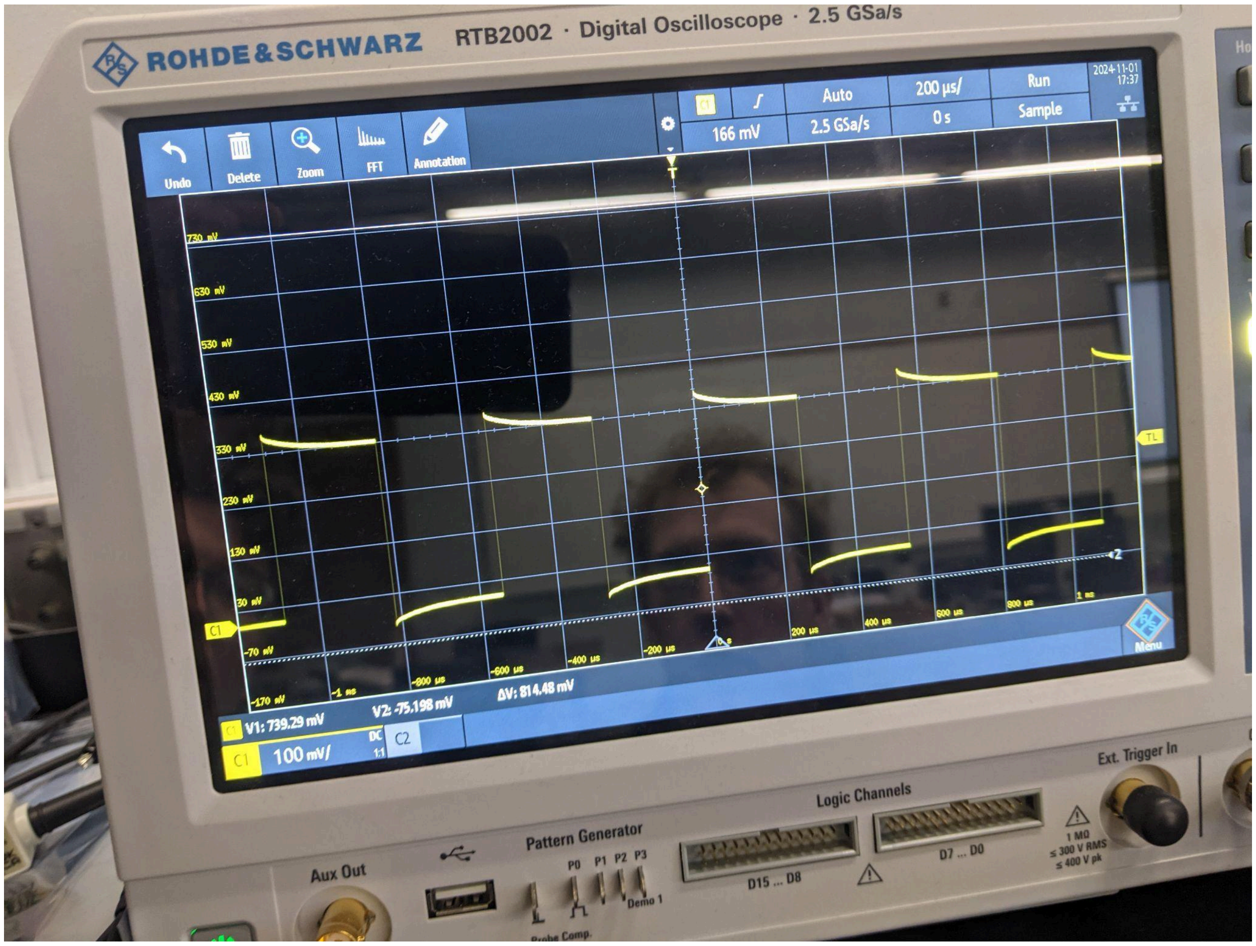


Figure 5: EE340 Class Diagram.

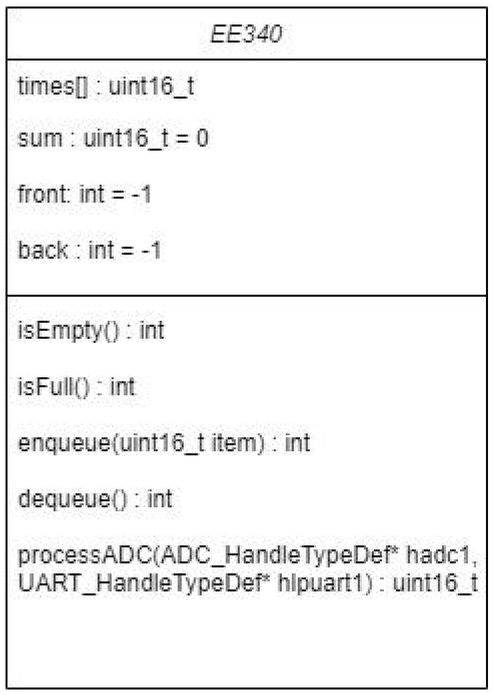


Figure 6: EE340 Flow Chart.

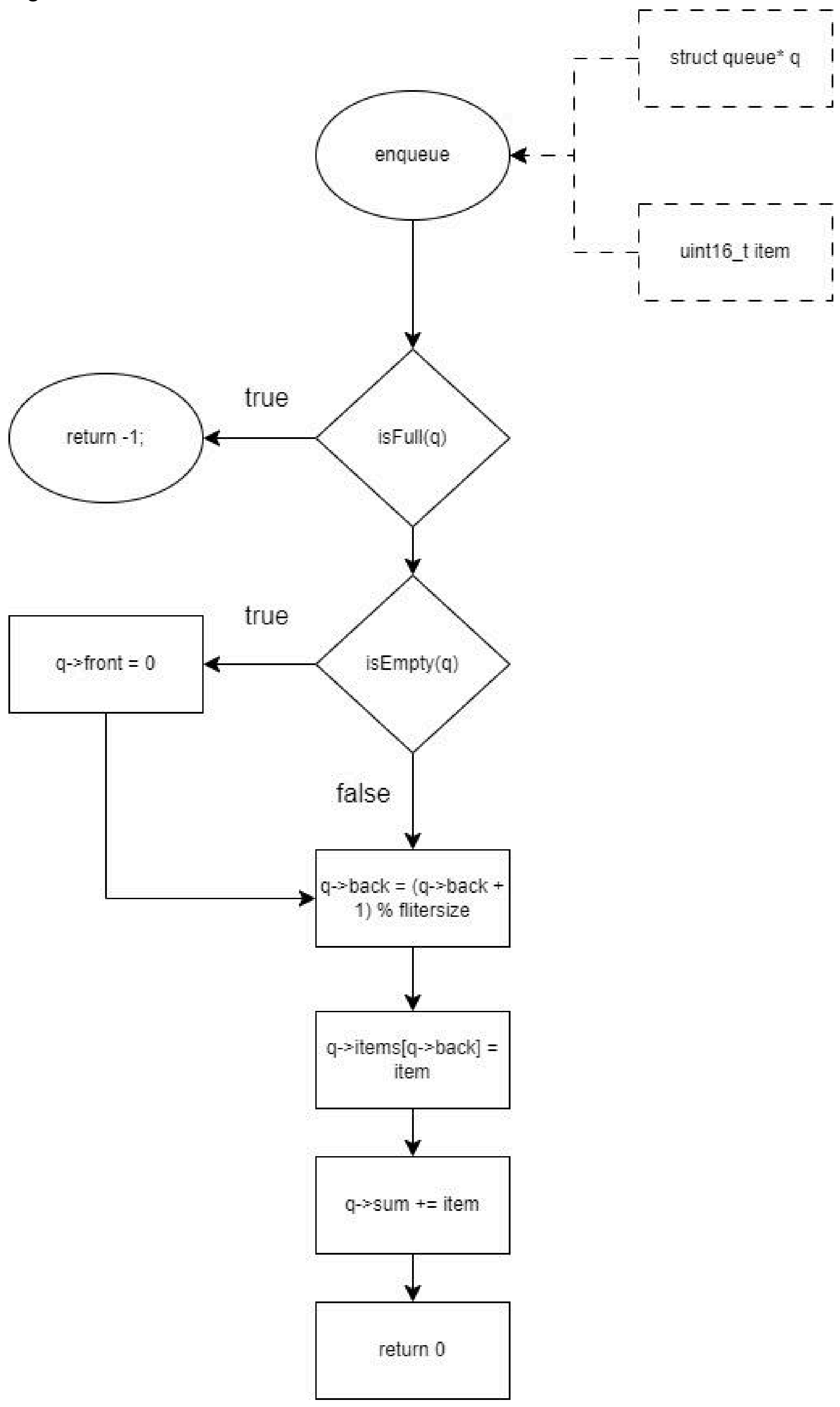


Figure 7: isEmpty function flow chart

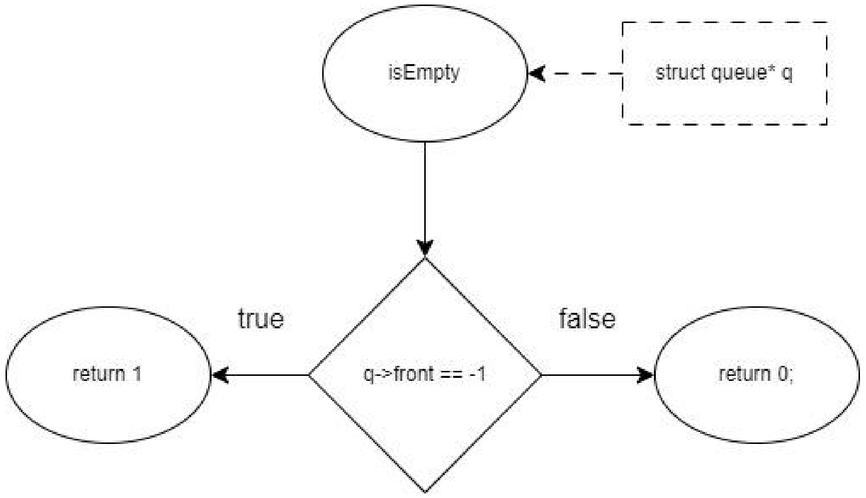
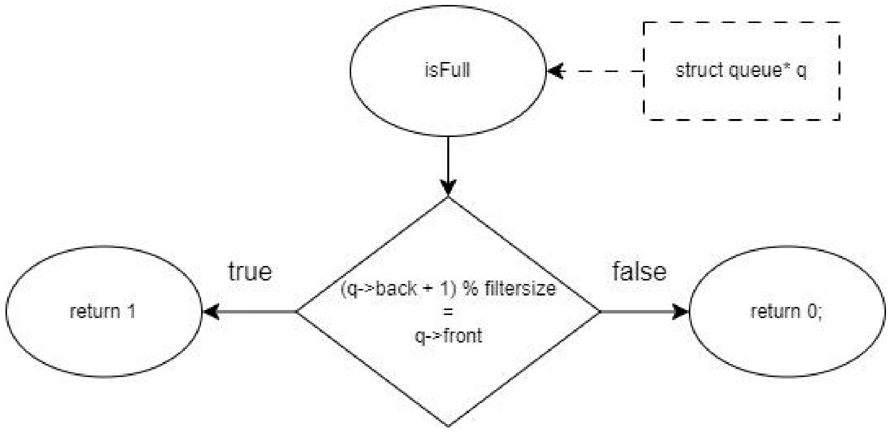


Figure 8:



CONCLUSION:

Throughout the course of this lab we were effectively able to swap between different timers based on ADC input from the flex sensor. If we were concerned about the flex sensor swapping back and forth between two timers while the flex sensor was put in a position causing input right on the fence of the two PWM frequencies we could implement a Schmitt trigger of sorts, giving the PWM ranges a dynamic threshold. For example, if the sensor goes above 2500 and triggers the highest frequency, in order to drop back to the middle frequency it would need to fall below a value of say 2400, instead of the normal 2500. This would limit isolation between timers when the ADC input is close to the current thresholds. But that's a project for a different time. For now, our engineers are happy to have something to show for the work they have put in this week.