GEBZE TECHNICAL UNIVERSITY

ELECTRONIC ENGINEERING

ELM335

Microprocessors Laboratory

LAB 5 Experiment Report

1) 210102002037 – Beyzanur Cam	
2) 210102002026 – Efe Bayrakçeken	
3) 200102002027 – Erhan Gök	

Introduction

In this assigments first part, two leds brightness is adjustted with the potentiometers output. An analog value is read from an potentiomater and converted to a digital signal. With the reading digital signal from potentiometer, leds duty cycles are changed. In the second part a knock counter was implemented with a digital microphone module. When a knock is done a counter is incremented and shown in the display, then counter is cleared with an external button.

Problem 1.

In this problem, light dimmer was implemented. A potentiometer was connected to STM32, and read its value with ADC of the middle pin of potentiometer. One green and one red led were connected to the board with series resistors. By changing the potentiometer, two leds will light up in opposite directions. In order to drive the leds, PWM was used as it was previous labs.

```
#include "stm32g0xx.h"

void PWM_Init();
void ADC_Init();
void SysTickInit();
void SysTick_Handler();
void setDutyCycle(uint16_t dutyCycle, uint8_t channel);
uint32_t readADC();

volatile uint32_t millis = 0;
uint32_t dutyCycle = 0;
```

At the beginning of our code all fucntions and variables that are used are called.

```
int main(){
    SysTickInit();
    PWM_Init();
    ADC_Init();

    setDutyCycle(10, 2);
    setDutyCycle(10, 3);

    while(1)
    {
      dutyCycle = (uint16_t) readADC()*100/4095;
      setDutyCycle(dutyCycle, 2);
      setDutyCycle(100-dutyCycle, 3);
    }
    return 0;
}
```

```
void SysTickInit() {
    SysTick->CTRL |= SysTick_CTRL_ENABLE_Msk;    // Enable SysTick
    SysTick->LOAD = 16000;                       // Load 16000 for 1ms tick
    SysTick->VAL = 0;                            // Reset SysTick
    SysTick->CTRL |= SysTick_CTRL_TICKINT_Msk;   // SysTick Enable Interrupt
    SysTick->CTRL |= SysTick_CTRL_CLKSOURCE_Msk; // SysTick clock source = AHB
    NVIC_EnableIRQ(SysTick_IRQn);
    NVIC_SetPriority(SysTick_IRQn, 0);
}
void SysTick_Handler(void) {
    millis++; // Increment millis value
}
```

In SysTickInit function our systick timer is defined and its features are given in needed values. The systick initialization code is the same as all the previous labs. This code was explained in detail in the lab 3 and at lab 4.

```
void setDutyCycle(uint16_t dutyCycle, uint8_t channel)
{
	if (channel==2){
		TIM1->CCR2 = dutyCycle;
	}
	else if (channel==3){
		TIM1->CCR3 = dutyCycle;
	}
}
```

```
void PWM_Init() {
    RCC->IOPENR |= RCC_IOPENR_GPIOAEN;   // Enable GPIOA clock
    RCC->APBENR2 |= RCC_APBENR2_TIM1EN;  // Enable TIM1 clock

    GPIOA->MODER &= ~GPIO_MODER_MODE10;   // Clear mode bits for PA10
    GPIOA->MODER &= ~GPIO_MODER_MODE9;   // Clear mode bits for PA09

    GPIOA->MODER |= GPIO_MODER_MODE10_1;  // Set PA10 to Alternate Function mode
    GPIOA->MODER |= GPIO_MODER_MODE9_1;  // Set PA9 to Alternate Function mode


    GPIOA->AFR[1] &= ~(0xF << ((10 - 8) * 4));  // Clear the current AF setting
for PA10
    GPIOA->AFR[1] |= (2 << ((10 - 8) * 4));    // Set the AF (AF2) for TIM1_CH3
for PA10

    GPIOA->AFR[1] &= ~(0xF << ((9 - 8) * 4));  // Clear the current AF setting
for PA9
    GPIOA->AFR[1] |= (2 << ((9 - 8) * 4));    // Set the AF (AF2) for TIM1_CH2
for PA9

    TIM1->PSC = 1600 - 1;  // Prescaler for 1kHz PWM frequency
    TIM1->ARR = 100;       // Auto-reload value for 100 steps, this is for 1%
increments

    TIM1->CCMR2 |= TIM_CCMR2_OC3M_1 | TIM_CCMR2_OC3M_2; // PWM mode 1 on Channel
3
    TIM1->CCMR1 |= TIM_CCMR1_OC2M_1 | TIM_CCMR1_OC2M_2; // PWM mode 1 on Channel
2

    TIM1->CCER |= TIM_CCER_CC3E;        // Enable capture/compare for channel 3
    TIM1->CCER |= TIM_CCER_CC2E;        // Enable capture/compare for channel 3
    TIM1->BDTR |= TIM_BDTR_MOE;          // Main output enable (needed for TIM1)
    TIM1->CR1 |= TIM_CR1_CEN;            // Enable timer
    setDutyCycle(0, 2); //set initial duty cycle to 0
    setDutyCycle(0, 3); //set initial duty cycle to 0
}
```

Before set/clear registers that are needed for PWM control are set, the registers that are needed to setup the GPIO’s (that will output the PWM signal) are configured. The pin that is connected to PA10 was used for the PWM. In the relevant parts of the reference manual and the datasheet, it can be seen that the PA10 is connected to timer 1 channel 3. Since the PA10 is on AFR high registers, that is used. Since the prescaler counts from 0, 1 was subtracted from 1600 to get the desired frequency for the PWM output. (16MHz/1600=10kHz). After 100 clock cycles, the clock is reset, so ARR is 100. This subsequently means the PWM frequency is 100Hz. Then the Capture/Compare Mode Register 2 can be set. When the timers count matches the value in the CCR, the output pin is set. Lastly, the timer can be enabled. After the timer is enabled, a timer interrupt is set to set the duty cycle independent of the main program.

```
void ADC_Init()
{
        //PA7
        RCC->IOPENR |= RCC_IOPENR_GPIOAEN;
        RCC->APBENR2 |= RCC_APBENR2_ADCEN;
        GPIOA->MODER |= GPIO_MODER_MODE7; //Set PA7 as analog

        //ADC1->CFGR1 &= ~ADC_CFGR1_CHSELRMOD;
        ADC1->CFGR1 |= ADC_CFGR1_CONT; //Enable continuous conversion mode
        ADC1->SMPR |= ADC_SMPR_SMPSEL7;
        ADC1->SMPR |= 0b111;
        ADC1->CHSELR |= ADC_CHSELR_CHSEL7;

        ADC1->CR |= ADC_CR_ADEN; //ADC enable
        ADC1->CR |= ADC_CR_ADSTART; //ADC start
}
```

ADC_Init function is used to convert analog signal to digital signal. In order to do that first a pin is selected which is PA7. For that pin clock and peripheral bus is activated and its mode is setted as analaog with the GPIO_MODER register. After that its configuration register is chosed as continous conversion mode because value needs to be read continously. Later than with the ADC_SMPR register time is sampled to 39.5 ADC clock cycles. Then with the ADC_CHSELR register channel is selected to 7 because adc is read from PA7 pin. And lastly ADC is enabled and started with the ADC_CR register.

```
uint32_t readADC()
{
        while(!(ADC1->ISR & ADC_ISR_EOC))
        {
        }
        return ADC1->DR;
}
```

Because converting a value from analog to digital takes more than a duty cycle it can only be read when the conversion's done. To understand the end of conversion ADC_ISR_EOC register is used because this register is raises an end of conversion flag. In while loop it's value checked with and logic operation. İf it's zero it will go in to a loop that does nothing else it will return the converted data.

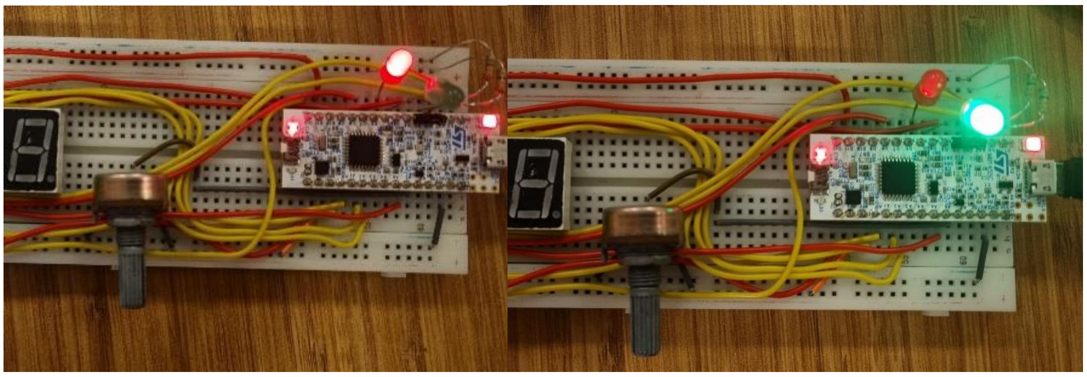

Figure 1 Light Dimmer Circuit in Max Values

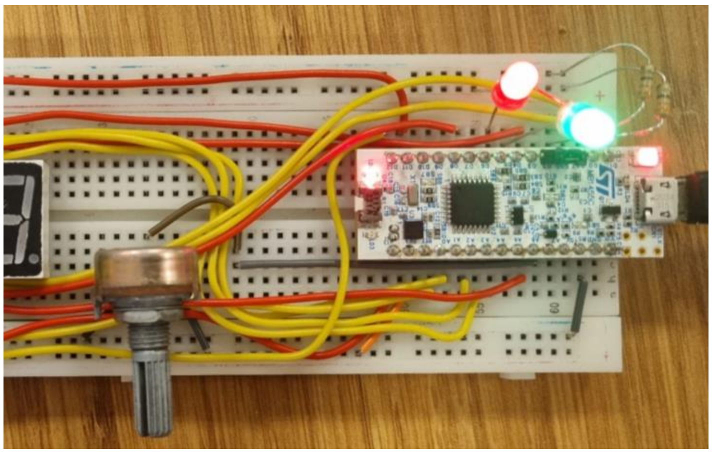

Figure 2 Light Dimmer Circuit in Mean Values

Problem 2.

In this problem, we implemened a knock counter. In order to count the knocs we connected a microphone module that gives a digital output whenever it detects a sound that is higher than its threshold value. The microphone module's threshold value can be adjusted with the potentiometer on the module. An external button was connected to the board in order to reset the counter and a four digit seven segment display connected to show the counter value.

In the beginning of the code all variable and function declarations were made. An array of numbers and letters was defined to show on the display.

```
#include "stm32g0xx.h"

int D1;
int D2;
int D3;
int D4;
volatile uint32_t millis = 0;

void initMic();
void SysTickInit();
void SysTick_Handler();
void ButtonInit();
void setPanel();
void PanelInit();
void setDigit(uint32_t mask,int digit);
void I2C_Init();
void I2C_Read();
void delay_ms(uint32_t delay);

static const uint8_t digitPins[16] = {
    0x3F, // 0
    0x06, // 1
    0x5B, // 2
    0x4F, // 3
    0x66, // 4
    0x6D, // 5
    0x7D, // 6
    0x07, // 7
    0x7F, // 8
    0x6F, // 9
    0x77, // A
    0x7C, // B
    0x39, // C
    0x5E, // D
    0x79, // E
    0x71  // F
};
```

Inside the main function of the code contains the initializations of all the peripherals, systic timer and external interrupts. The while loop updates the display a single digit at a time (needed since the digits in the display are connected). For each digit, a ‘d’ variable was declared, that filters out the required digit to be displayed. Since the displaying of the numbers are not time critical, no interrupts were used for this operation.

```
int main(void) {
        ButtonInit();
        PanelInit();
        SysTickInit();
        initMic();
        setDigit(GPIO_ODR_OD6,0);
        I2C_Init();
    while(1) {
        D1 = counter/1000;
        D2 = (counter%1000)/100;
        D3 = (counter%100)/10;
        D4 = counter%10;

        setDigit(GPIO_ODR_OD6,D4);
        delay_ms(1);
        setDigit(GPIO_ODR_OD5,D3);
        delay_ms(1);
        setDigit(GPIO_ODR_OD4,D2);
        delay_ms(1);
        setDigit(GPIO_ODR_OD1,D1);
        delay_ms(1);
    }
    return 0;
}
```

The systick initialization code is the same as all the previous labs. This code was explained in detail in the lab 3 and at lab 4.

```
void SysTickInit() {
    SysTick->CTRL |= SysTick_CTRL_ENABLE_Msk;    // Enable SysTick
    SysTick->LOAD = 16000;                       // Load 16000 for 1ms tick
    SysTick->VAL = 0;                            // Reset SysTick
    SysTick->CTRL |= SysTick_CTRL_TICKINT_Msk;   // SysTick Enable Interrupt
    SysTick->CTRL |= SysTick_CTRL_CLKSOURCE_Msk; // SysTick clock source = AHB

    NVIC_EnableIRQ(SysTick_IRQn);
    NVIC_SetPriority(SysTick_IRQn, 0);
}

void delay_ms(uint32_t delay) {
    millis = 0;
    while (millis < delay)
    {
        // Wait for the specified duration
    }
}
void SysTick_Handler(void) {
    millis++; // Increment millis value
}
```

The set digit function sets a certain digit of the seven segment displays digit. This turns all the output data registers off, then turns the related mask's odr on.

```
void setDigit(uint32_t mask,int digit)
{
    //Disable D1,D2,D3,D4
    GPIOA->ODR |= GPIO_ODR_OD1;
      GPIOA->ODR |= GPIO_ODR_OD4;
      GPIOA->ODR |= GPIO_ODR_OD5;
      GPIOA->ODR |= GPIO_ODR_OD6;

    GPIOA->ODR &= ~mask; //Enable mask2
    GPIOB->ODR |= digitPins[digit];
    GPIOB->ODR &= digitPins[digit];
}
```

The button init function initializes the button and its related interrupts. Here, the button is connected to GPIOA0.

```
void ButtonInit()
{
      RCC->IOPENR |= RCC_IOPENR_GPIOAEN_Msk;
      GPIOA->MODER &= ~GPIO_MODER_MODE0_Msk;
      GPIOA->PUPDR &= ~GPIO_PUPDR_PUPD0_Msk;
      GPIOA->PUPDR |=  (1U<<0);

      RCC->APBENR2 |= (1U<<0);
      EXTI->RTSR1 |= (1U<<0);
      EXTI->IMR1 |= (1U<<0);

     EXTI->FTSR1 |= (1U<<0);        // Enable EXTI on Falling edge
      EXTI->RTSR1 &= ~(1U<<0);       // Disable EXTI on Rising edge

      NVIC_SetPriority(EXTI0_1_IRQn,0);
      NVIC_EnableIRQ(EXTI0_1_IRQn);
}
```

The interrupt handler functions are defined as below. The 0_1 interrupt is for reseting the counter, thus setting the counter to 0, and the interrupt handler 4_15 is for incrementing the counter and then a delay function preventing recounts.

```
void EXTI0_1_IRQHandler(void){
      counter = 0;
      GPIOC->ODR &= ~(1U << 6);
      EXTI->FPR1 |= (1<<0); //EXTI falling edge pending register 1 (EXTI_FPR1)
}

void EXTI4_15_IRQHandler(){

      counter++;
      delay_ms(100); // Delay for debouncing
      EXTI->RPR1 = EXTI_RPR1_RPIF9;  // Clear the pending bit for EXTI line 2

}
```

The panel init function initializes the seven segment display. The ssd is connected to the GPIOB register in order.

```
void PanelInit()
{
	RCC->IOPENR |= RCC_IOPENR_GPIOBEN_Msk;
    GPIOB->MODER &= ~0xFFFF;
    GPIOB->MODER |= 0x5555;
    GPIOB->ODR |= digitPins[0];


    GPIOA->MODER &= ~GPIO_MODER_MODE1_Msk;
    GPIOA->MODER |= GPIO_MODER_MODE1_0;

    GPIOA->MODER &= ~GPIO_MODER_MODE4_Msk;
    GPIOA->MODER |= GPIO_MODER_MODE4_0;

    GPIOA->MODER &= ~GPIO_MODER_MODE5_Msk;
    GPIOA->MODER |= GPIO_MODER_MODE5_0;

    GPIOA->MODER &= ~GPIO_MODER_MODE6_Msk;
    GPIOA->MODER |= GPIO_MODER_MODE6_0; //pa1 pa4 pa5 pa6

    //Disable D1,D2,D3,D4
    GPIOA->ODR |= GPIO_ODR_OD1;
	GPIOA->ODR |= GPIO_ODR_OD4;
	GPIOA->ODR |= GPIO_ODR_OD5;
	GPIOA->ODR |= GPIO_ODR_OD6;
}
```

The mic initialization function is the same as the button initialization code. Since our mic module provides a digital output, it can be read like a button. When a 1 from the mic is detected, it enters the interrupt routine described above.

```
void initMic(){
	    // Enable the clock for GPIO port B
	        RCC->IOPENR |= RCC_IOPENR_GPIOBEN;

	        // Configure PB9 as input
	        GPIOB->MODER &= ~GPIO_MODER_MODE9_Msk;  // Clear mode bits for PB9


	          EXTI->EXTICR[2]|= (1U << 8*1);

	        // Enable SYSCFG clock
	        RCC->APBENR2 |= RCC_APBENR2_SYSCFGEN;

	        EXTI->IMR1 |= (1U<<9);
	        EXTI->RTSR1 |= (1U << 9);

	        NVIC_SetPriority(EXTI4_15_IRQn, 2);
	        NVIC_EnableIRQ(EXTI4_15_IRQn);
}
```

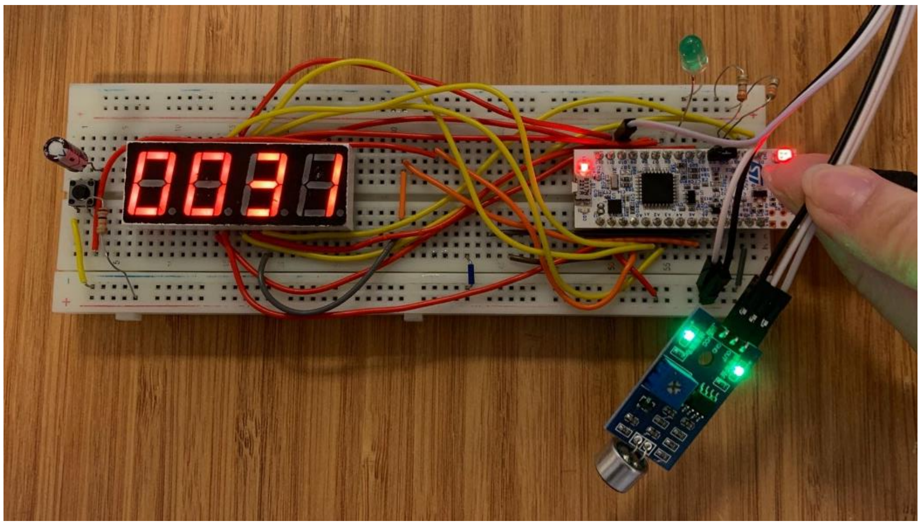

Figure 3 Knock Counter on Board

Conclusion:

In conclusion, in this lab, a light dimmer was implemented wich reacts the voltage value of the output of a potentimeter which the value is read with the ADC. Since the potentiometer was not high quality, the potentiometer never reached to 0 ohm value, so neither of the leds were completely off. In the knock counter problem, with the help of a digital microphone module a knock counter was implemented. Since an IMU sensor was not connected, the counter continues to increment even though a knock was not made. The reason is that the microphone module reacts to sound.

References:

https://developer.arm.com/documentation/ddi0337/e/BABBCJII

https://github.com/fcayci/stm32g0

STM32G0x1 advanced Arm®-based 32-bit MCUs - Reference manual

STM32G0 Nucleo-32 board - User manual

cortexM0 referance manual.pdf