GEBZE TECHNICAL UNIVERSITY

ELECTRONIC ENGINEERING

ELM334

Microprocessors Laboratory

LAB 3 Experiment Report

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

Introduction

In this experiment we have gained extensive knowledge about interrups, 7 segment display, watchdog timer, Systick timer and exceptions. Also for the whole software part C language has been used. Problem 1 focuses on creating an accurate delay function using the SysTick exception, with an emphasis on software and hardware methods for measuring its precision. Problem 2 involves working with general-purpose timers to control an on-board LED, dynamically adjusting its blinking speed with an external button, all managed within interrupts. In Problem 3, we connect a seven-segment display to implement a count-up timer with specific constraints, emphasizing interrupt-driven functionality and synchronizing digit increments. Problem 4 delves into the world of watchdog timers, setting up either window or independent timers, calculating reset times, and implementing handler routines. Finally, Problem 5 challenges us to incorporate the watchdog timer developed in Problem 4 into the count-up timer system from Problem 3, requiring strategic integration and thorough explanations of the solution and implementation to cover all possible scenarios.

Problem 1. In this problem, you will work on creating an accurate delay function using “SysTick” exception. Create a “SysTick” exception with 1 millisecond interrupt intervals. Then create a “delay_ms(..)” function that will accurately wait for (blocking) given number of milliseconds.

• How would you measure the accuracy of your delay using software methods?

• How would you measure the accuracy of your delay using hardware methods? Explain each case, and (if possible) implement your solution.

In this part of the question we obtain a system tick counter (Systick) which is basically a simple timer that is part of the NVIC controller in the Cortex-M microprocessor. In **`LedInit()`** function we activated the on-board led like we did in the previous assigments. And we define a counter and a unsigned 32 bit millis as zero for our starting point. In the beginning of our code there is a macro named LEDDELAY and it was assigned to 1000 which means our on-board led is being delayed in for 1000 miliseconds.

```
#define LEDDELAY    1000

volatile int counter = 0;
volatile uint32_t millis = 0;

void delay_ms(uint32_t delay);
void SysTickInit();
void LedInit();

int main(void) {
    LedInit();
    SysTickInit();

    while (1) {
        delay_ms(LEDDELAY);
        /* Toggle LED */
        GPIOC->ODR ^= (1U << 6);
    }
    return 0;
}
void LedInit() {
    /* Enable GPIOC clock */
    RCC->IOPENR |= (1U << 2);
```

```
    /* Setup PC6 as output */
    GPIOC->MODER &= ~(3U << 2 * 6);
    GPIOC->MODER |= (1U << 2 * 6);

    /* Turn on LED */
    GPIOC->ODR |= (1U << 6);
}
```

SysTickInit() function is for initialization of SysTick timer. Systick_CNTRL register is used to enable SysTick timer and to do that we masked the register.

The LOAD register specifies the start value to load into the STK_VAL register when the counter is enabled.

Systick_VAL register is current counter value which is set to 0 and can be read and written.

When we set Systick_VAL register to 0 it resets the SysTick timer.

```
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);
}
```

After setting up initializing SysTick Timer, a few functions were implemented. “SysTick_Handler” is implemented in order to catch SysTick Timer interrupts and when STM32 is interrupted it increases the millis counter. “delay_ms” function was implemented in order to delay the loop. It sets the “millis” variable that keeps the tick count of the SysTick counter. After setting to 0, it starts wait until SysTick counter to count till it reaches to wanted delay value in miliseconds.

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

void SysTick_Handler(void) {
    millis++; // Increment millis value
}
```

For software methods in order to measure the accuracy of our timer we can use a stopwatch and measure the difference between the stopwatch and the Systick timer we implemented.

For hardware methods in order to measure the accuracy, we can make the SysTick timer interrupt to toggle an pin in each interval and connect this pin to an oscilloscope or an logic analyzer which is a lot

cheaper and easier to use in logic applications compared to oscilloscopes. Here is an example of a logic analyzer:

Figure 1 logic analyzer

Problem 2. In this problem, you will work with general purpose timers. Set up a timer with lowest priority that will be used to toggle on-board LED at 1 second intervals. Change the blinking speed using an external button. Each button press should increase the blinking speed by 1 second up to a maximum of 10 seconds. Next button press after 10 should revert it back to 1 second. All the functionality should be inside your interrupts.

In order to toggle the led we implemented a "ToggleLed" function which manupulates GPIOC_ODR register by making XOR operation to bit 6.

In "TimerInit" function we initialized TM2 which is a general purpose timer. First we applied `RCC_APBENR1_TIM2EN_Msk mask` which comes with CMSIS libraries `to RCC_APBENR1 register.` After that TIM2_CNT register was set to 0 in order to the timer to start counting from 0. The prescaler was set to 0. The timer calculations was made according to formula below:

$$timer\ delay\ (seconds) = \frac{(TIMx\_ARR)}{counter\ frequency}$$

$$counter\ frequency = \frac{timer\ input\ frequency}{TIMx\_PSC + 1}$$

Figure 2 equations used to calculate time delay

When TIMX_ARR is 16000000 and TIMX_PSC is 0, timer delay becomes 1 second. After that Interrupt Generation Enable Bit (TIM2->DIER[0]) was set to 1 which enables the timer to generate interrupts. In order to handle TIM2 interrupts, we enabled IRQ from NVIC with NVIC_EnableIRQ and set its priority to lowest priority which is 3. TIM2_SR register which is interrupt pending flag set to 0 to get interrupts. Finally EGR register set to 1 to reset timer and CR1 to enable timer.

```
void ToggleLed()
{
	GPIOC->ODR ^= (1U << 6);
}
```

```
void TimerInit()
{
	RCC->APBENR1 |= RCC_APBENR1_TIM2EN_Msk;

	TIM2->CNT = 0;
	TIM2->PSC = 0;
	TIM2->ARR = (uint32_t) 16000000;
	TIM2->DIER |= (1U<<0);

	NVIC_EnableIRQ(TIM2_IRQn);
	NVIC_SetPriority(TIM2_IRQn,3);

	TIM2->SR &= ~(1U<<0);

	TIM2->EGR |= (1U<<0); // Reset timer
	TIM2->CR1 |= (1U<<0); // Enable timer
}
```

"void TIM2_IRQHandler(void)" function is implemented to handle timer interrupts. In the handler function we toggled the led and cleared the status register. Since we are using external interrupts to get button status, in "ButtonInit" function we initialized EXTI. The clock of the pin is enabled, set to OUTPUT mode and set as pull-up. SYSCFG was enabled from RCC_APBENR2 register, enabled EXTI_RTSR1 and EXTI->IMR1. Set the EXTI to falling edge since it is pull-up register. Finally the button set to 0 priority.

```
void TIM2_IRQHandler(void){
	TIM2->SR &= ~(1<<0); // Clear UIF update interrupt flag
	ToggleLed();
}

void ButtonInit()
{
   RCC->IOPENR |= RCC_IOPENR_GPIOAEN_Msk;
   GPIOA->MODER &= ~GPIO_MODER_MODE0_Msk;
   GPIOA->PUPDR &= ~GPIO_PUPDR_PUPD0_Msk;
   GPIOA->PUPDR |= (1U<<0); //GPIO port pull-up/pull-down register


   RCC->APBENR2 |= (1U<<0); //APB peripheral clock enable register
   EXTI->RTSR1 |= (1U<<0); //EXTI rising trigger selection register
   EXTI->IMR1 |= (1U<<0); //EXTI CPU wakeup with interrupt mask register


   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);
}
```

In EXTI handler function we implemented the code that changes the led blinking speed by setting the prescaler. When the prescaler goes up the blinking speed lowers, and it changes direction when the counter hits 9 or 0.

```
void EXTI0_1_IRQHandler(void){
```

```
	EXTI->FPR1 |= (1<<0);
	if(incrementing)
	{
		counter++;
		TIM2->PSC++;
		if(counter>=9)
		{
			incrementing = 0;
		}}
	else
	{
		counter--;
		TIM2->PSC--;
		if(counter <= 0)
		{
			incrementing = 1;
		}}
	delay = counter + 1;
}
```

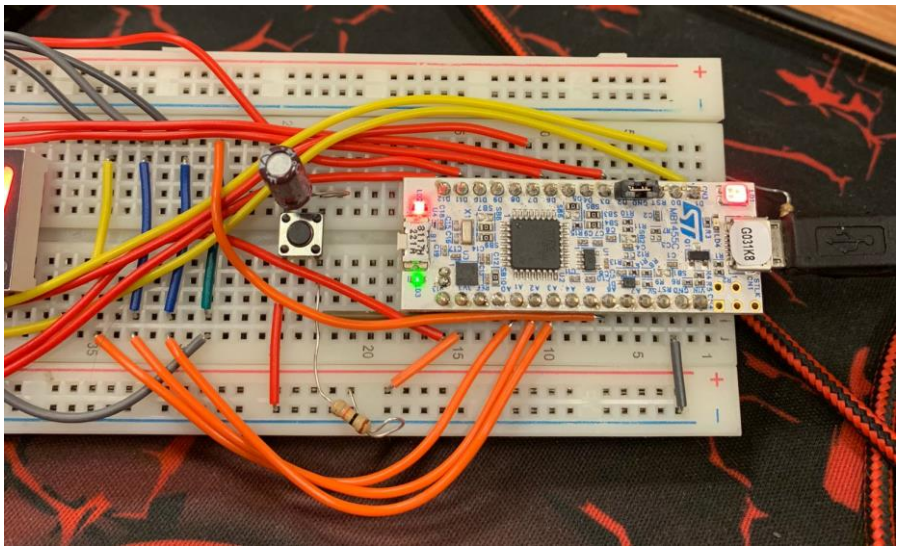

Figure 3 Problem2 on circuit

Problem 3. In this problem, connect your seven-segment display and implement a count up timer. It should sit at zero, once the external button is pressed, it should count up to the max value (i.e. 9999) then once it overflows to 0, it should stop, and light up an LED (on-board or external). Pressing the button again should clear the LED and count again. For this problem, your main loop should not have anything. All the functionality should be inside your interrupts. Note: Arrange it so that the LSD of the number increments in 1 second intervals. (i.e it should take 10 seconds to go from 0000 to 9999)

In this problem, in order to set the 4 digit 7 segment display to any value we used an array of hex codes that represents bit values. In the main function led, timer, button and the panel was initialized and was set to 0 for beginning.

```
uint16_t counter = 0;
int D1;
int D2;
int D3;
```

```
int D4;
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
};

int main(void) {
      LedInit();

      Timer2Init();

      ButtonInit();
      PanelInit();
      setDigit(GPIO_ODR_OD6,0);
    while(1) {
    }
    return 0;
}
```

The timer "Timer2Init" is the same as the problem before. However, the TIM2_PSC was set to 160 and TIM2_ARR to 25. And this time we implemented EnableTimer and DisableTimer functions that takes TIM_TypeDef type of TIM value.

```
void EnableTimer(TIM_TypeDef* TIM)
{
      TIM->CR1 |= (1U<<0);
}
void DisableTimer(TIM_TypeDef* TIM)
{
      TIM->CR1 &= ~(1U<<0);
}
```

In timer handler, it increments a counter till it becomes 40000. If it reaches that, it means the value in the display becomes 9999 and it disables the timer, sets the counter to 0 and lights up the on-board led. Till it becomes 40000, it calculates the digits of display seperately with mod operations and it sets the pins to show these digits.

```
void TIM2_IRQHandler(void){
TIM2->SR &= ~(1<<0); // Clear UIF update interrupt flag
counter++;

if(counter>=40000)
```

```
{
DisableTimer(TIM2);
GPIOC->ODR ^= (1U << 6);
counter = 0;
}
int number_counter = counter/4;
D1 = number_counter/1000;
D2 = (number_counter%1000)/100;
D3 = (number_counter%100)/10;
D4 = number_counter%10;
if (counter%4 == 0){
setDigit(GPIO_ODR_OD6,D4);
}else if(counter%3 == 0){
setDigit(GPIO_ODR_OD5,D3);
}else if(counter%2 == 0){
setDigit(GPIO_ODR_OD4,D2);
}else if(counter%1 == 0){
setDigit(GPIO_ODR_OD1,D1);
}
}
```

The “setDigit” function takes two parameters which are a mask to apply to ODR register to enable the necessary bits and digit to show on the display.

```
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];
}
```

In the EXTI handler we set the counter to zero and the clear the bit of on-board led, then enabled the timer.

```
void EXTI0_1_IRQHandler(void){
	EXTI->FPR1 |= (1<<0); //EXTI falling edge pending register 1 (EXTI_FPR1)
	counter = 0;
	GPIOC->ODR &= ~(1U << 6);
	EnableTimer(TIM2);
}
```

Since we connected d1,d2,d3 and d4 pins of 7 segment to line B and A,B,C,D,G,E we enabled line A and B. Then set the pins d1,d2,d3,d4 to high which means the displays are off.

```
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;
                                          }
```

Expression	Type	Value
counter	uint	1345
time	int	50
D1	int	1
D2	int	3
D3	int	4
D4	int	5
Add new expre		

Figure 4 value read for each segment

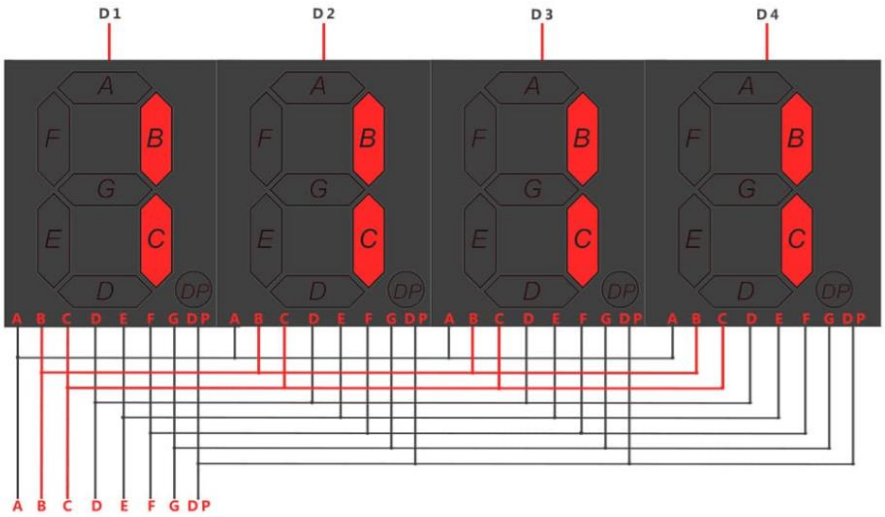

Figure 5 seven-segment led display pinout

Problem 4. In this problem, you will work with watchdog timers. Setup either window or independent watchdog timer and observe its behavior in the simple blinky example from the repo. Calculate the appropriate reset time and implement it. Add the necessary handler routine for resetting the device.

A watchdog timer (WDT) is a timer that monitors microcontroller (MCU) programs to see if they are out of control or have stopped operating. It acts as a “watchdog” watching over MCU operation. A microcontroller (MCU) is a compact processor for controlling electronic devices. A watchdog timer (WDT) is a timer that monitors microcontroller (MCU) programs to see if they are out of control or have stopped operating. It acts as a “watchdog” watching over MCU operation. A microcontroller (MCU) is a compact processor for controlling electronic devices. There are two type of WDT timers external and on-board and there are four feature of MCU WDT. They are Interrups, Window, Security, Fail-safe activation.

In this question we prefered to use independent WDT. From the previous question our led was binkling in 1 second intervals. And from the IWDG time-base equation(1.1) we wanted our timeout to have 2 values one is smaller than 1 second and the other is bigger than 1 second(2.02s). So we can see if WDT works properly.

$$t_{IWDG} = t_{LSI} x 4 x 2^{prescaler} x (RL + 1) \qquad (1.1)$$

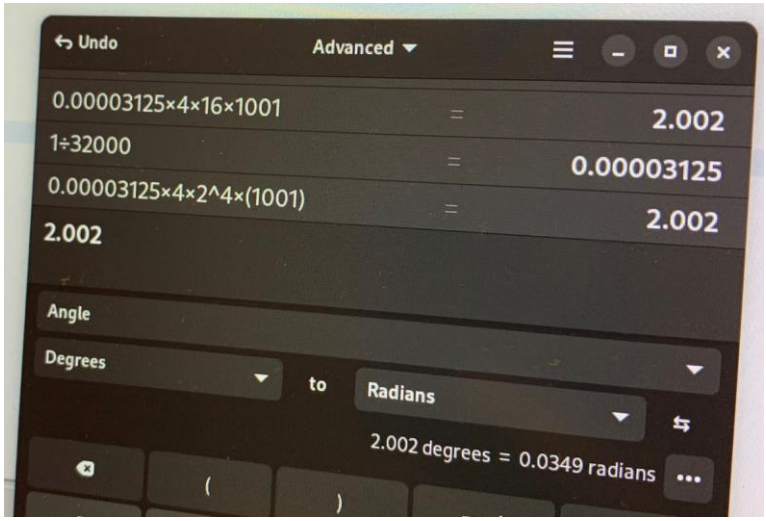

Figure 6 time calculation

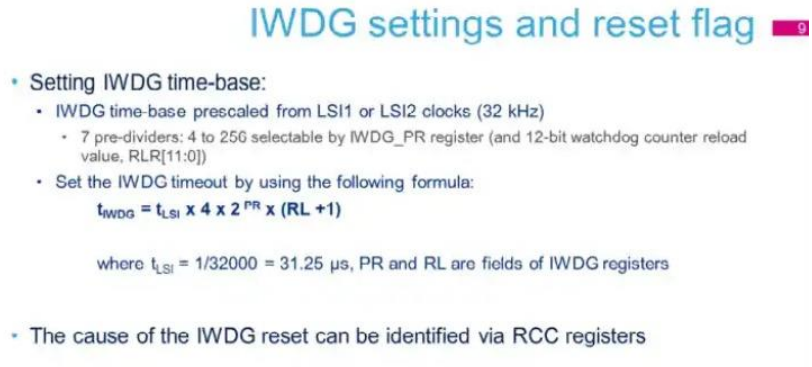

Figure 7 time base settingd

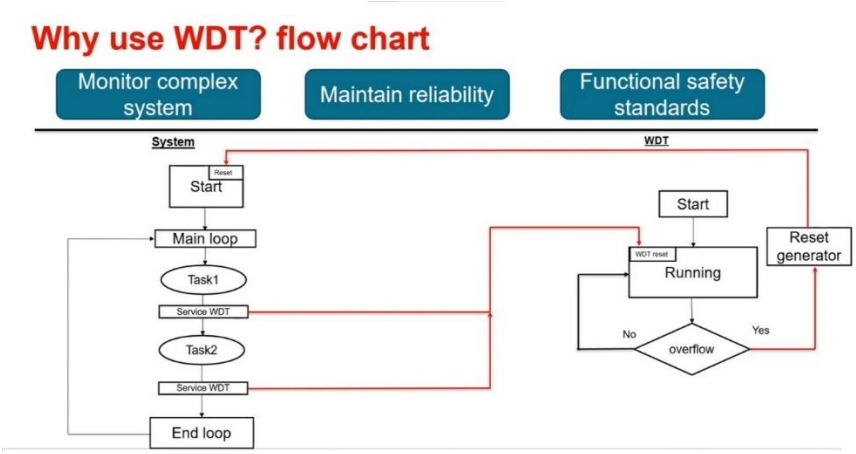

Figure 8 WDT flowchart

```
#define LEDDELAY    1000
#define WDG_RELOAD_COMMAND 0xAAAA
#define WDG_RELOAD_VAL 1000 //1000de calısıyor, 100'de reset atıyor

int main(void) {
    LedInit();
    SysTickInit();
    IWDGInit();

    while (1) {
        delay_ms(LEDDELAY);
        /* Toggle LED */
        GPIOC->ODR ^= (1U << 6);
        IWDG_Refresh();
    }
    return 0;
}
```

IWDG_KR(IWDG key register) Writing the key value 0x5555 to enable access to the IWDG_PR, IWDG_RLR and IWDG_WINR registers then with IWDG prescaler register (IWDG_PR) we select the counter clock divider. The watchdog counter counts down from this value(IWDG reload register). And this value is set as 1000. The timeout period is a function of this value and the clock prescaler.

IWDG_Refresh() function helps us to reset watchdog counter to zero which stands for 0xAAAA value (from user manual).

```
void IWDGInit() {

    IWDG->KR = 0x5555; // Erişim açılıyor.
    IWDG->PR = 4;    // Prescaler: 4
    IWDG->RLR = WDG_RELOAD_VAL; // Reload degeri: 1000

    // Watchdog counterı yenile
    IWDG_Refresh();

    // Sayıcı saymaya başlıyor.
    IWDG->KR = 0xCCCC;
}
```

```
void IWDG_Refresh(void) {
    // Watchdog counterı yenile
    IWDG->KR = WDG_RELOAD_COMMAND;
}
```

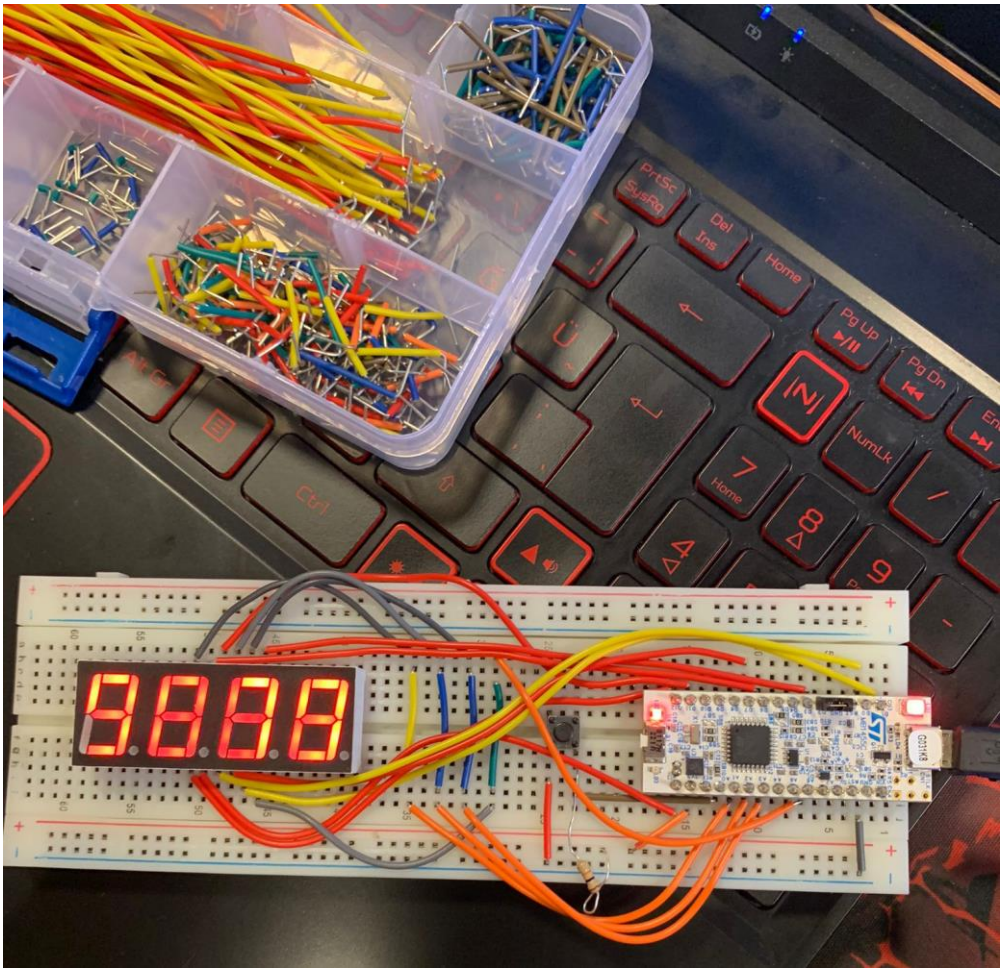

Figure 9 Problem 4 in circuit

Problem 5. In this problem, you will implement your watchdog timer in Problem 3. Figure out a way to properly incorporate it to your code when it all works with timers. Explain your solution and implementation and make sure you covered all possible scenarios.

In this problem everything is the same as the problem 3, except the lines below and the watchdog timer parts.

For calculating the real time in WDT we tried to use same equation (1.1). But somehow it didn't work. So we improvised and gave exceptable numbers to reload value, one of them is worked.

```
#define WDG_RELOAD_COMMAND 0xAAAA
#define WDG_RELOAD_VAL 10000 //PSC=7
#define IWDG_PR 7
//For 16 seconds(To observe it resets if the timer does not stop after 10
seconds
```

In order to observe the effect of the watchdog timer on problem 3, we removed the “9999” limit of the timer counter. After timer reaches “9999”, timer does not stop and continues incrementing counter and shows it on the display. And since there is no limit it also means that we can observe characters like A,B,C,D,E,F since we implemented as before. After taking more than 10 seconds, in the 16th second watchdog timer resets the MCU and the display shows “0” like it has just powered on.

```
void TIM2_IRQHandler(){
	TIM2->SR &= ~(1<<0); // Clear UIF update interrupt flag
	counter++;

	/*
	if(counter>=40000)
	{
		DisableTimer(TIM2);
		GPIOC->ODR ^= (1U << 6);
		counter = 0;
	}
	*/

	int number_counter = counter/4;
	D1 = number_counter/1000;
	D2 = (number_counter%1000)/100;
	D3 = (number_counter%100)/10;
	D4 = number_counter%10;
	if (counter%4 == 0){
		setDigit(GPIO_ODR_OD5,GPIO_ODR_OD6,D4);
	}else if(counter%3 == 0){
		setDigit(GPIO_ODR_OD4,GPIO_ODR_OD5,D3);
	}else if(counter%2 == 0){
		setDigit(GPIO_ODR_OD1,GPIO_ODR_OD4,D2);
	}else if(counter%1 == 0){
		setDigit(GPIO_ODR_OD6,GPIO_ODR_OD1,D1);
	}
}
```

Watchdog timer functions are the same as before except IWDG_Start command was added in order the Watchdog to start when the button is pressed. In “**EXTI0_1_IRQHandler**” function IWDG_Start and IWDG_Refresh commands are added.

```
void IWDGInit() {
	// Erişim açılıyor.
    IWDG->KR = 0x5555;

    IWDG->PR = IWDG_PR;    // Prescaler: 4
    IWDG->RLR = WDG_RELOAD_VAL;

    // Watchdog counterı yenile
    IWDG_Refresh();
}

// Sayıcı saymaya başlıyor.
void IWDG_Start()
{
    IWDG->KR = 0xCCCC;
}
```

```
void IWDG_Refresh(void) {
    // Watchdog counterı yenile
    IWDG->KR = WDG_RELOAD_COMMAND;
}

void EXTI0_1_IRQHandler(void){
EXTI->FPR1 |= (1<<0);
counter = 0;
GPIOC->ODR &= ~(1U << 6);
EnableTimer(TIM2);
IWDG_Start();
IWDG_Refresh();
}
```

We can observe the character “B” and “A” in the photos below since it continues to increment counter after reaching “9999”.

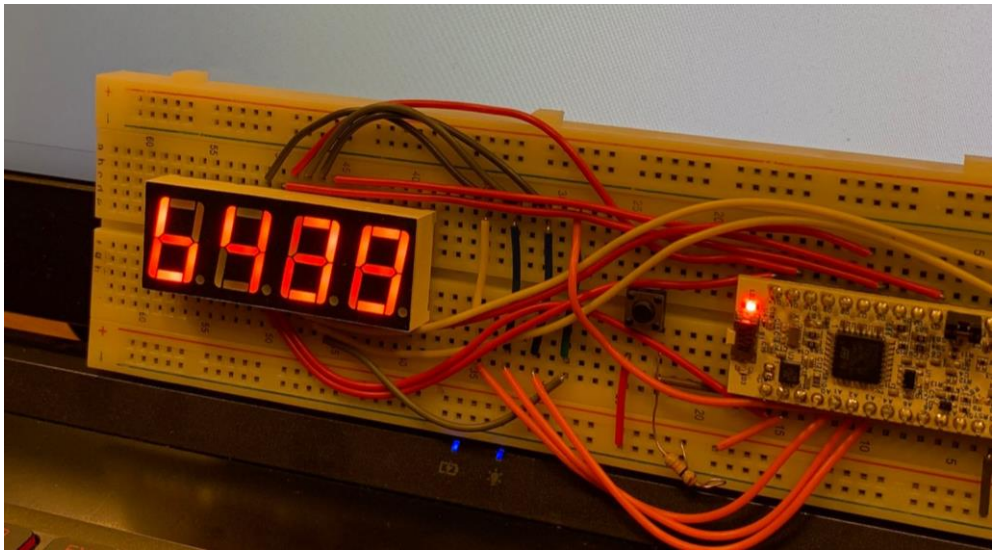

Figure 10 Problem 5 on board-1

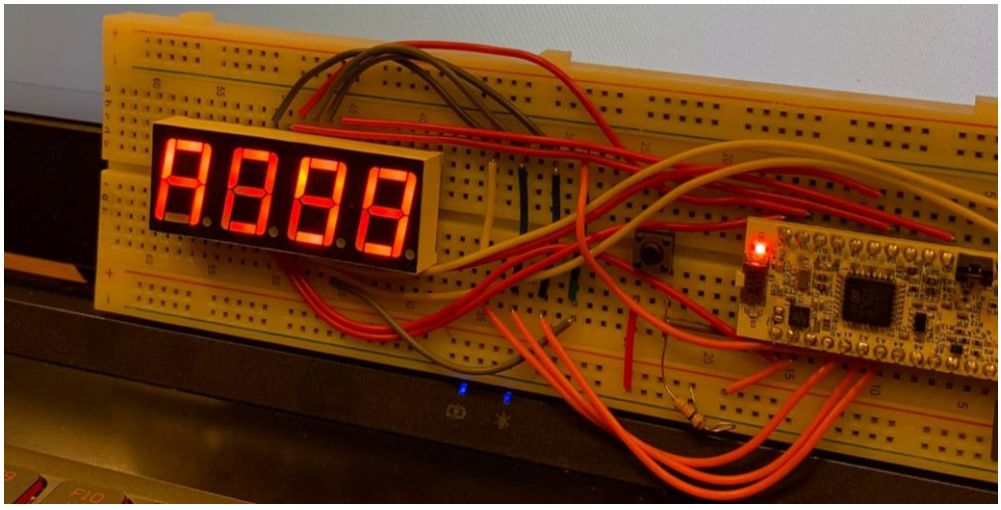

Figure 11 Problem 5 on board-2

References:

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

https://github.com/fcayci/stm32g0

Referance Manual.pdf

STM32G0 Nucleo-32 board - User manual

https://www.mcu-turkey.com/stm8s-iwdgindependent-watchdog-modulu-kullanimi/#codesyntax_3

https://www.ti.com/video/6313371139112

https://kunalsalvi63.medium.com/stm32-window-watchdog-94bc0406ea51

https://www.youtube.com/watch?v=AelNsnpfbcM&ab_channel=ControllersTech

https://ozgurayik.com/2020/12/16/stm32-programlama-systick-timer-ile-delay/