1. INRODUCTION:

The objective of this lab is to give you a thorough understanding of timers. You will use C language for the problems unless some parts require inline assembly. Use blinky project from stm32g0 repo as the starting point for your problems.

2. PROBLEMS:

2.1. 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.

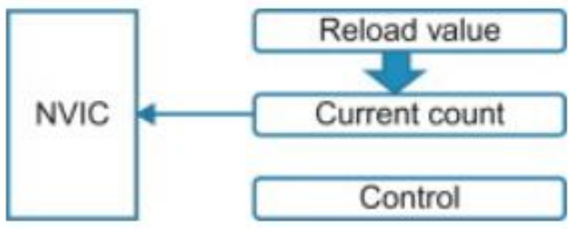
- How would you measure the accuracy of your delay using software methods?
- How would you measure the accuracy of your delay using hardware methods?

Explain each case, and (if possible) implement your solution.

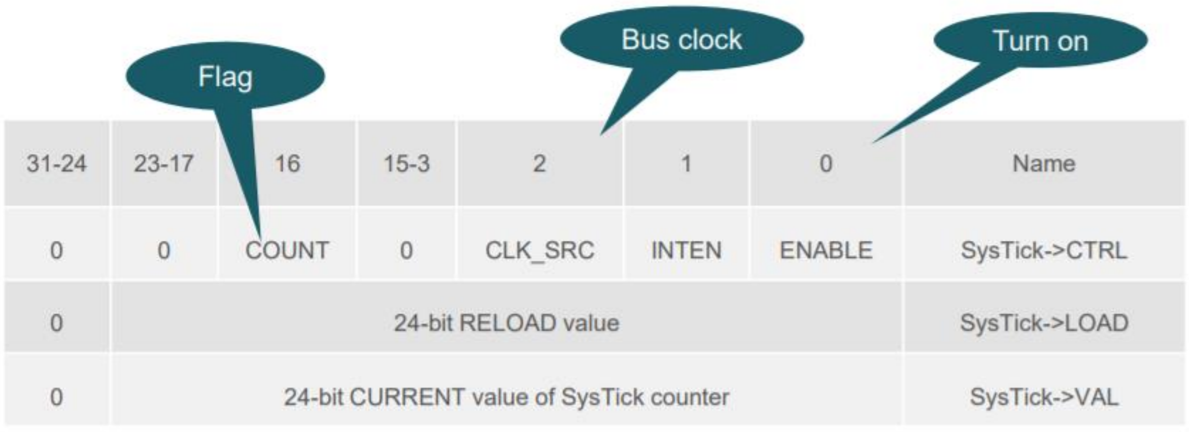
2.1.1. Theoretical Research:

Systick Timer

All of the Cortex-M processors also contain a standard timer. This is called the systick timer and is a 24-bit countdown timer with auto reload. Once started the systick timer will count down from its initial value. Once it reaches zero it will raise an interrupt and a new count value will be loaded from the reload register. The main purpose of this timer is to generate a periodic interrupt for an RTOS or other event-driven software. If you are not running an OS, you can also use it as a simple timer peripheral.



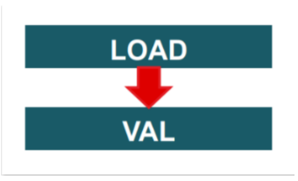
The default clock source for the systick timer is the Cortex-M CPU clock. It may be possible to switch to another clock source, but this will vary depending on the actual microcontroller you are using. While the systick timer is common to all the Cortex-M processors, its registers occupy the same memory locations within the Cortex-M3 and Cortex-M4. In the Cortex-M0 and Cortex-M0+, the systick registers are located in the SCB and have different symbolic names to avoid confusion. The systick timer interrupt line and all of the microcontroller peripheral lines are connected to the NVIC.


Flag
Bus clock
Turn on

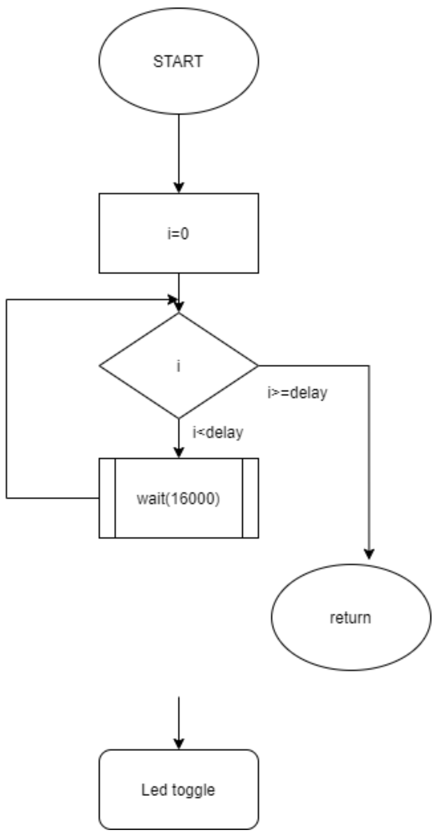
31-24	23-17	16	15-3	2	1	0	Name
0	0	COUNT	0	CLK_SRC	INTEN	ENABLE	SysTick->CTRL
0	24-bit RELOAD value						SysTick->LOAD
0	24-bit CURRENT value of SysTick counter						SysTick->VAL

How it Works

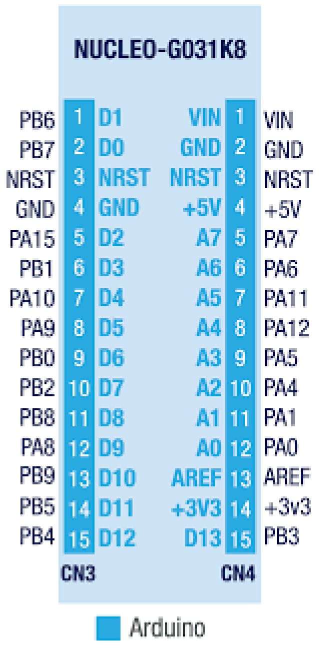
- ✓ 24-bit down counter decrements at bus clock frequency
- ✓ With a 48 MHz bus clock, decrements every 20.83 ns
- ✓ Software sets a 24-bit LOAD value of n
- ✓ The counter, VAL, goes from n → 0
  - Sequence: n, n-1, n-2, n-3… 2, 1, 0, n, n-1…
- ✓ SysTick is a modulo n+1 counter:
- ✓ VAL = (VAL - 1) mod (n+1)



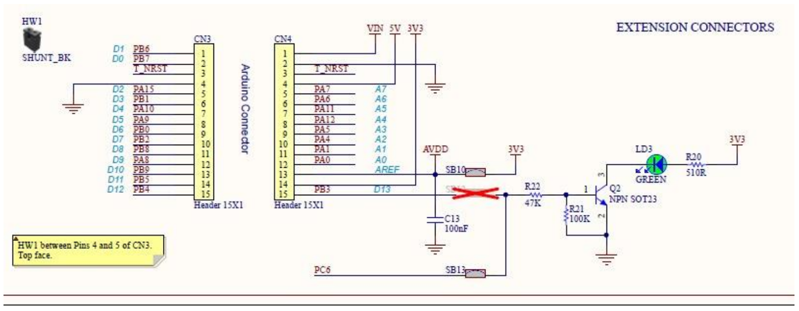
2.1.2 Flowchart:



2.1.3 Block diagram:



2.1.4 Schematic:



2.1.5 Code:

```c
/*
 * main.c
 *
 * author: Mehmet Akif GÜMÜŞ
 *
 *                  description:I work on creating an accurate
delay function using SysTick exception. I 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.
 *
 */

#include "stm32g0xx.h"

void delay_ms(volatile unsigned int);
int main(void) {
```

```
int main(void) {

    /* 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);
    int Start = SysTick->VAL;       //Systic VAL ile o systick in o
anki değeri alınır
      delay_ms(100000);
    int Stop = SysTick->VAL;        //Systic VAL ile o systick in o
anki değeri alınır
    unsigned int Delta = 0x00FFFFFF&(Start-Stop);    // Start eksi
stop ile arada geçen zamanı bulup deltaya atadım

    SystemCoreClockUpdate();  //contains the system frequency

    while(1) {
        delay_ms(100000);           //delay fonksiyonu ile while
döngüsü içerisinde led in toggle etmesi sağlandı
        /* LED Toggle */
        GPIOC->ODR ^= (1U << 6);
    }
    return 0;
}

void delay_ms(volatile unsigned int s){

      for(int i=s; i>0; i--){
       SysTick_Config(SystemCoreClock / 1000); // 16 MHz / 1000 ile 1
ms elde edildi.
       }
`
```

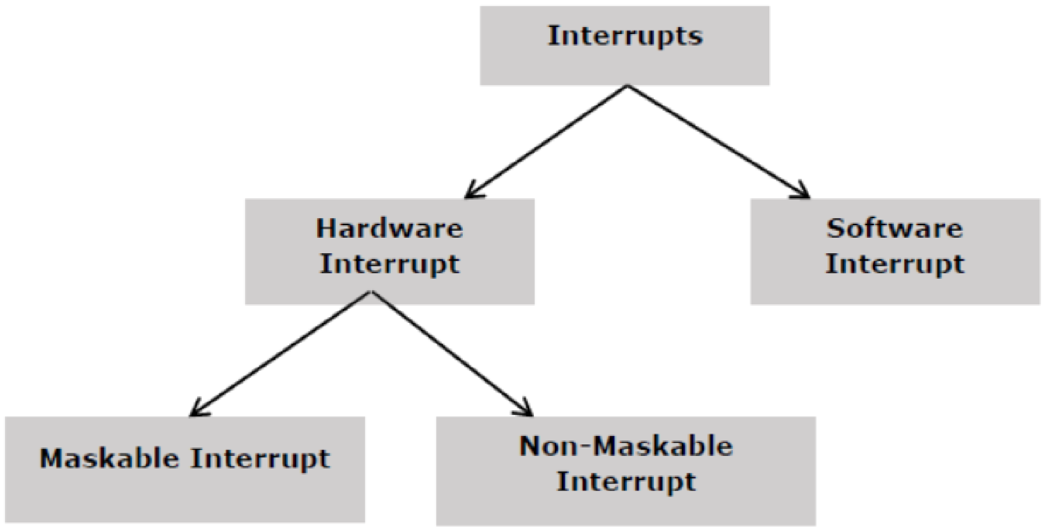
2.2 PROBLEM-2

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 hould 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.

2.2.1 Theoretical Research:

Microprocessor - 8086 Interrupts

Interrupt is the method of creating a temporary halt during program execution and allows peripheral devices to access the microprocessor. The microprocessor responds to that interrupt with an **ISR** (Interrupt Service Routine), which is a short program to instruct the microprocessor on how to handle the interrupt.



Hardware Interrupts

Hardware interrupt is caused by any peripheral device by sending a signal through a specified pin to the microprocessor. The 8086 has two hardware interrupt pins, i.e. NMI and INTR. NMI is a non-maskable interrupt and INTR is a maskable interrupt having lower priority. One more interrupt pin associated is INTA called interrupt acknowledge.

NMI

It is a single non-maskable interrupt pin (NMI) having higher priority than the maskable interrupt request pin (INTR)and it is of type 2 interrupt.

When this interrupt is activated, these actions take place −

- Completes the current instruction that is in progress.
- Pushes the Flag register values on to the stack.
- Pushes the CS (code segment) value and IP (instruction pointer) value of the return address on to the stack.
- IP is loaded from the contents of the word location 00008H.
- CS is loaded from the contents of the next word location 0000AH.
- Interrupt flag and trap flag are reset to 0.

INTR

The INTR is a maskable interrupt because the microprocessor will be interrupted only if interrupts are enabled using set interrupt flag instruction. It should not be enabled using clear interrupt Flag instruction.

The INTR interrupt is activated by an I/O port. If the interrupt is enabled and NMI is disabled, then the microprocessor first completes the current execution and sends ‘0’ on INTA pin twice. The first ‘0’ means INTA informs the external device to get ready and during the second ‘0’ the microprocessor receives the 8 bit, say X, from the programmable interrupt controller.

These actions are taken by the microprocessor –

- First completes the current instruction.
- Activates INTA output and receives the interrupt type, say X.
- Flag register value, CS value of the return address and IP value of the return address are pushed on to the stack.
- IP value is loaded from the contents of word location X × 4
- CS is loaded from the contents of the next word location.
- Interrupt flag and trap flag is reset to 0

Software Interrupts

Some instructions are inserted at the desired position into the program to create interrupts. These interrupt instructions can be used to test the working of various interrupt handlers.

2.2.2 Code:

```
/*
 * main.c
 *
 * author: Mehmet Akif GÜMÜŞ
 *
 * description: I 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.
 *
 */

#include "stm32g0xx.h"

volatile int k=1;       //k yı başlangıç için volatile 1
tanımladık.

void EXTI0_1_IRQHandler(void)
{

      if (EXTI->RPR1 & (1U << 1))              //pinimiz exti1
üzerinde bazı hatalar yol açmamak için pending ii kontrol
ediyoruz
      {
            k++;
            EXTI->RPR1 |= (1U << 1);
            if(k>10)          //k 10a ulaşınca 10dan sonra tekrar
k va 1 atadık.
```

```
            {
                  k=1;
            }
      }
}

void TIM1_BRK_UP_TRG_COM_IRQHandler(void)
{
      TIM1->SR &= ~(1U<<0); //clear update status register

      static int t = 1;
      if(t==k)                      //buton interruptı ile harekete
geçen k sayısını kullanarak kurulan algoritma ile ledin yanma
sıklığını ayarlama
      {
              /* blinky LED */
            GPIOC->ODR ^= (1U << 6);
            t = 1;
      }
      else if (t<k)
      {
            t++;
      }
      else t=1;
}

void init_timer1(){

      RCC->APBENR2 |= (1U<< 11);// enable time1 module clock

      TIM1->CR1=0;// zero out the control register just in case
      TIM1->CR1 |= (1<<7);    // ARPE
      TIM1->CNT=0;// zero out counter

      /*1 second interrupt     */

      TIM1->PSC=999;
      TIM1->ARR=16000;

      TIM1->DIER |= (1 << 0);// update interrupt enable
      TIM1->CR1 |= (1 << 0);//      tım1 enable

      NVIC_SetPriority(TIM1_BRK_UP_TRG_COM_IRQn,3); //
TIM1_BRK_UP_TRG_COM_IRQn fonksiyonuna butondan daha düşük olması için
priority si 3 atandı
      NVIC_EnableIRQ(TIM1_BRK_UP_TRG_COM_IRQn); //timer interrupt ı
nvic içerisinde enable edildi

}

int main(void) {

    /* Enable GPIOC and GPIOA clock */
    RCC->IOPENR |= (1U << 2);
    RCC->IOPENR |= (1U << 0);

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

    /* Setup PA1 as input */
    GPIOA->MODER &= ~(3U << 2*1);
    GPIOA->PUPDR |= (2U << 2*1); // Pull-down mode

    /*setup interrrupts for inputs*/
    EXTI->EXTICR[0] |=(0U << 8*1);//PA1   EXTI1 mux ta PA1 için
EXTICR0'ın 9.biti 0 yapıldı

    /* MASK*/
    EXTI->IMR1 |= (1U << 1);

    /*rising edge*/
    EXTI->RTSR1 |= (1U << 1);

    /*NVIC*/
    NVIC_SetPriority(EXTI0_1_IRQn,0);    // buton interruptı PA1
için EXTI1 in içerisinde olduğundan EXTI0_1_IRQn kullanıldı.
    NVIC_EnableIRQ(EXTI0_1_IRQn);   //buton interrupt ı nvic
içerisinde enable edildi

    init_timer1();

    while(1) {

    }

    return 0;
}
```

2.3 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)

2.4 PROBLEM-4

In this problem, you will work with watchdog timers. Setup either window or indepdentent 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 reseting the device.

2.4.1 Theoretical Research:

Watchdog Timer:

A watchdog timer (sometimes called a *computer operating properly* or *COP* timer, or simply a *watchdog*) is an electronic or software timer that is used to detect and recover from computer malfunctions. During normal operation, the computer regularly resets the watchdog timer to prevent it from elapsing, or "timing out". If, due to a hardware fault or program error, the computer fails to reset the watchdog, the timer will elapse and generate a timeout signal. The timeout signal is used to initiate corrective actions. The corrective actions typically include placing the computer system in a safe state and restoring normal system operation.

Watchdog timers are commonly found in embedded systems and other computer-controlled equipment where humans cannot easily access the equipment or would be unable to react to faults in a timely manner. In such systems, the computer cannot depend on a human to invoke a reboot if it hangs; it must be self-reliant. For example, remote embedded systems such as space probes are not physically accessible to human operators; these could become permanently disabled if they were unable to autonomously recover from faults. A watchdog timer is usually employed in cases like these. Watchdog timers may also be used when running untrusted code in a sandbox, to limit the CPU time available to the code and thus prevent some types of denial-of-service attacks.[1]

Watchdog timers are also used in operating systems where certain high priority operations are required to complete in a specific time interval. If the timer expires before the operation completes then the OS responds to the timer interrupt by recording error data and terminating the operation. A system may have both types of watchdog timer.

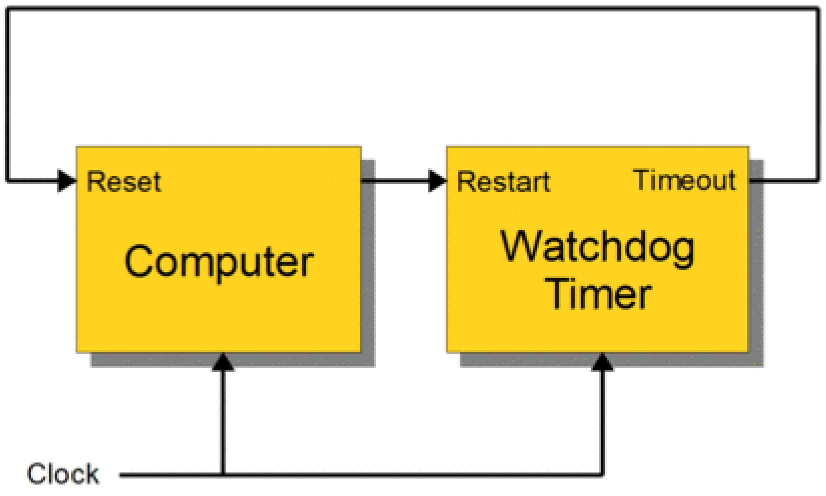
Watchdog restart

The act of restarting a watchdog timer, commonly referred to as "kicking" the watchdog[2][3], is typically done by writing to a watchdog control port. Alternatively, in microcontrollers that have an integrated watchdog timer, the watchdog is sometimes kicked by executing a special machine language instruction or setting a specific bit in a register. An example of this is the CLRWDT (clear watchdog timer) instruction found in the instruction set of some PIC microcontrollers.

In computers that are running operating systems, watchdog resets are usually invoked through a device driver. For example, in the Linux operating system, a user space program will kick the watchdog by interacting with the watchdog device driver, typically by writing a zero character to /dev/watchdog. The device driver, which serves to abstract the watchdog hardware from user space programs, is also used to configure the time-out period and start and stop the timer.

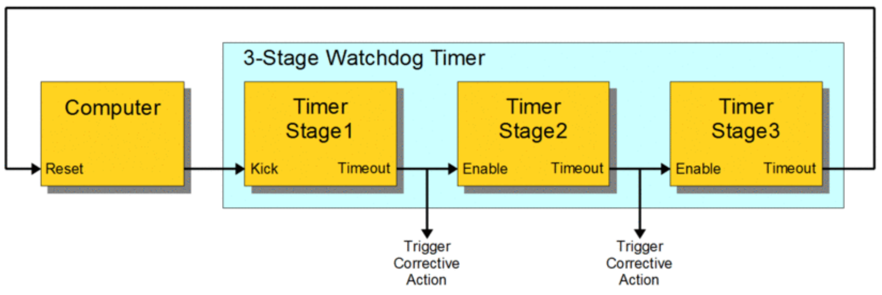
Single-stage watchdog

Watchdog timers come in many configurations, and many allow their configurations to be altered. Microcontrollers often include an integrated, on-chip watchdog. In other computers the watchdog may reside in a nearby chip that connects directly to the CPU, or it may be located on an external expansion card in the computer's chassis. The watchdog and CPU may share a common clock signal, as shown in the block diagram below, or they may have independent clock signals.



Multistage watchdog

Two or more timers are sometimes cascaded to form a multistage watchdog timer, where each timer is referred to as a timer stage, or simply a stage. For example, the block diagram below shows a three-stage watchdog. In a multistage watchdog, only the first stage is kicked by the processor. Upon first stage timeout, a corrective action is initiated and the next stage in the cascade is started. As each subsequent stage times out, it triggers a corrective action and starts the next stage. Upon final stage timeout, a corrective action is initiated, but no other stage is started because the end of the cascade has been reached. Typically, single-stage watchdog timers are used to simply restart the computer, whereas multistage watchdog timers will sequentially trigger a series of corrective actions, with the final stage triggering a computer restart.[3]



2.4.2 Code:

```c
/*
 * main.c
 *
 * author: Mehmet Akif GÜMÜŞ
 *
 * description: I will work with watchdog timers. Setup either window
or indepdentent watchdog timer and
 * observe its behavior in the simple blinky example from the repo.
Calculate the appropriate reset time and implement
 * it.
 */

#include "stm32g0xx.h"

#define LEDDELAY    160000
#define delayled  16000000
int main(void);
void turn_off(void);
void init_wd(void);

void delay(volatile uint32_t);

void EXTI0_1_IRQHandler(void){
      if (EXTI->RPR1 & (1U << 1)){
             while(1) {
                     delay(LEDDELAY);
                     /* Toggle LED */
                     GPIOC->ODR ^= (1U << 6);
                  }
            EXTI->RPR1 |= (1U << 1);
            }
}
 void NonMaskableInt_IRQHandler(void){
      turn_off();
}


int main(void) {

    /* Enable GPIOC clock */
    RCC->IOPENR |= (1U << 2);
    RCC->IOPENR |= (1U << 0);

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

    /* Setup PA1 as input */
    GPIOA->MODER &= ~(3U << 2*1);
    GPIOA->PUPDR |= (2U << 2*1); // Pull-down mode

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

       /*setup interrrupts for inputs*/
          EXTI->EXTICR[0] |=(0U << 8*1);//PA1

          /* MASK*/
          /* MASK*/
          EXTI->IMR1 |= (1U << 1);

          /*rising edge*/
          EXTI->RTSR1 |= (1U << 1);



          /*NVIC*/
          NVIC_SetPriority(EXTI0_1_IRQn, 0);
          NVIC_EnableIRQ(EXTI0_1_IRQn);

          init_wd();

    while(1) {

    }

    return 0;
}


void init_wd(void){
      RCC->CSR |= (3U << 0);
    IWDG->KR = 0xAAAA;

    // Wait until the IWDG is ready to accept the new parameters
    while (IWDG->SR != 0) { }

      IWDG->KR = 0x5555; // enable access to the e IWDG_PR, IWDG_RLR

      IWDG->PR = 6;

      //IWDG->SR = (1U << 2*0); // enable WVU and RVU
      IWDG->RLR = 0x0AA; //watchdog counter each time the value

      IWDG->WINR= 0x0AA; // access protected
      IWDG->KR = 0xAAAA;
      IWDG->KR = 0xCCCC; //Starts if wasn't running yet
    /*NVIC*/
    NVIC_SetPriority(NonMaskableInt_IRQn, 3);
    NVIC_EnableIRQ(NonMaskableInt_IRQn);
}



void turn_off(void){

    /* Turn off LED */
    GPIOC->ODR |= (0U << 6);
    delay(delayled);
    main();
}

void delay(volatile uint32_t s) {
    for(; s>0; s--);
}
```

2.5 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.

3. CONCLUSION

The purpose of this experiment is to understand the timers and write them as C code. Creating a delay function in problem 1. SysTick exception was used when creating this function. SysTick Exception is set to 1 ms. Then, the delay function was written and kept waiting for the specified millisecond. With SystemCoreClock, we divide the number of clocks of the system by 1000 and get 1 millisecond. Then, in the for loop, the desired number of SysTick functions were returned and the desired millisecond values were reached. In problem 2, external button interrupt was generated. And when this button is pressed, the frequency of the LED being on decreased. This frequency ranged from 1 second to 10 seconds. When it reached 10 seconds, when the button was pressed, it decreased to 1 second. Problem 3 and problem 5 could not be done due to busy schedule. Watchdog timers were focused in Problem 4. The code was written based on the research. But it was not successful. What is intended to be done in this problem; When the code starts running the led will be on. Then when the button is pressed, the led will toogle in an unlimited while loop. Will reset because wd timer cannot return to reload value. And it will reach zero, then reset the system. Because the priority of the reset is high, it will exit the while loop. Many problems were successfully concluded, but other problems could not be completed because there was not enough time and the self-teach technique was used in this lesson.

4. References

- https://www.ti.com/lit/ml/swrp171/swrp171.pdf?ts=1607348284680&ref_url=https%253A%252F%252Fwww.google.com%252F
- https://en.wikipedia.org/wiki/Watchdog_timer