GEBZE TECHNICAL UNIVERSITY

ELECTRONIC ENGINEERING

ELM334

Microprocessors Laboratory

LAB 4 Experiment Report

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

Introduction

The purpose of this lab assignment is to explore and implement key concepts in embedded systems, focusing on UART communication and PWM signal generation. In this assignment, we will delve into three distinct problems, each designed to enhance our understanding of hardware communication protocols and signal modulation.

Problem 1.

In this problem, you will connect your board to the PC using UART protocol. For this you will need to create an initialization routine for UART and create send and receive functions. You should not use interrupts for this assignment. List the possible transmit and receive bit rates for UART serial communication according to the EISA RS-232 communication Standard.

UART_Init() function is called inside the main and uart_tx(uart_rx()) is written like this to transmit what is recivied in while loop.

```
int main()
{
	UART_Init();
	while(1)
	{
		uart_tx(uart_rx());
}}
```

Inside UART_Init function, clocks that are related to UART2 and line A is enabled. PA2 and PA3 are setted as alternate function because these are the ones related to USART_TX and USART_RX.

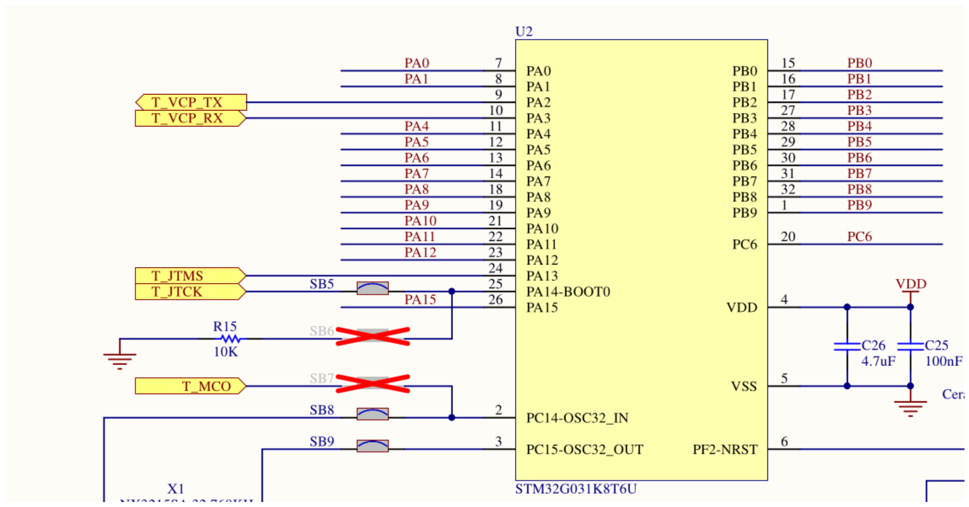
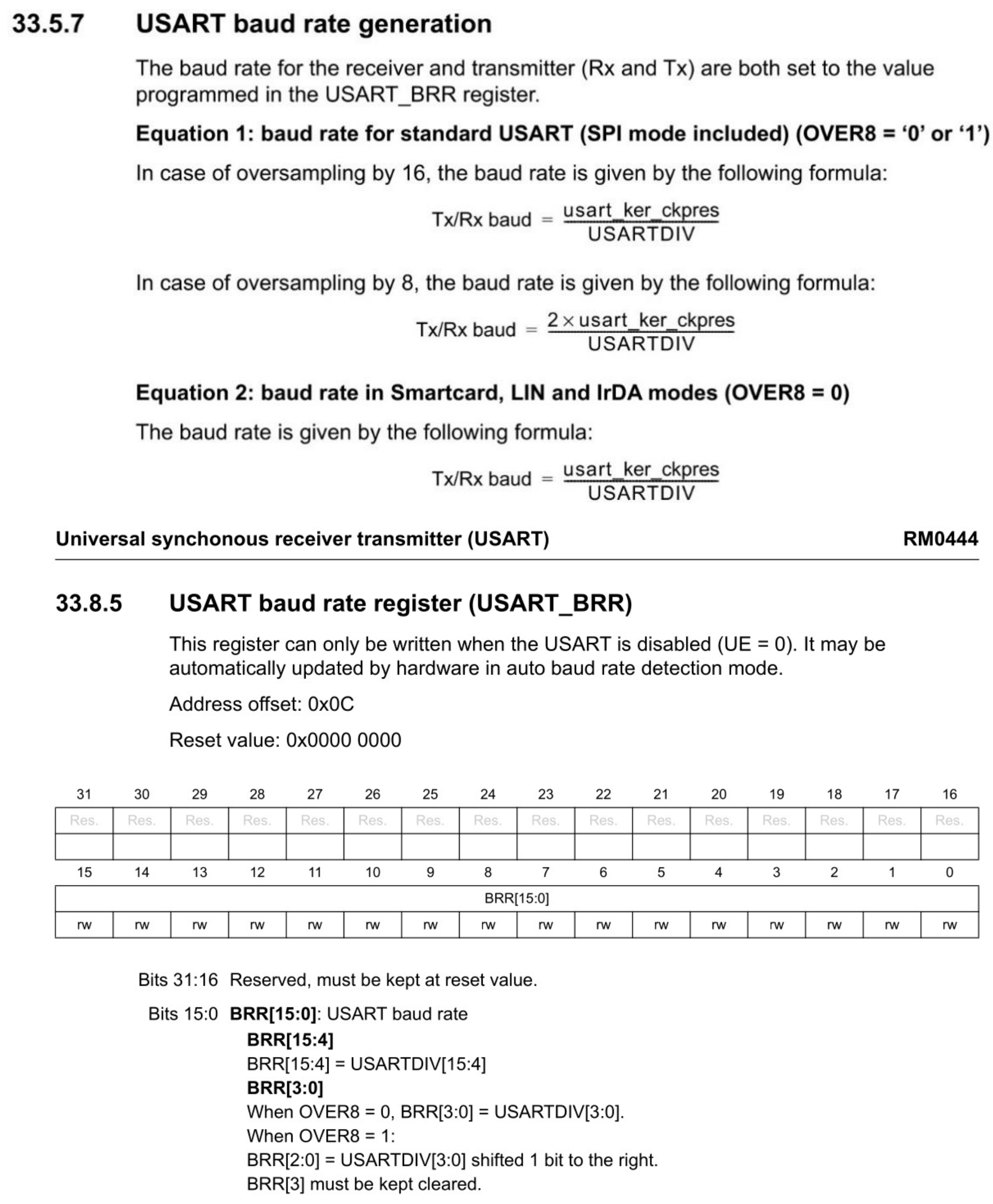


Figure 1 Pinout of UART modules

After that with the GPIOA alternate function low register (GPIOx_AFRL) PA2 and PA3's ports are selected.

Baud rate value is calculated and setted by using USART baud rate register (USART_BRR) with figure below. [16M/(9600*16)]



33.5.7 USART baud rate generation

The baud rate for the receiver and transmitter (Rx and Tx) are both set to the value programmed in the USART_BRR register.

Equation 1: baud rate for standard USART (SPI mode included) (OVER8 = '0' or '1')

In case of oversampling by 16, the baud rate is given by the following formula:

$$\text{Tx/Rx baud} = \frac{\text{usart\_ker\_ckpres}}{\text{USARTDIV}}$$

In case of oversampling by 8, the baud rate is given by the following formula:

$$\text{Tx/Rx baud} = \frac{2 \times \text{usart\_ker\_ckpres}}{\text{USARTDIV}}$$

Equation 2: baud rate in Smartcard, LIN and IrDA modes (OVER8 = 0)

The baud rate is given by the following formula:

$$\text{Tx/Rx baud} = \frac{\text{usart\_ker\_ckpres}}{\text{USARTDIV}}$$

Universal synchonous receiver transmitter (USART) **RM0444**

33.8.5 USART baud rate register (USART_BRR)

This register can only be written when the USART is disabled (UE = 0). It may be automatically updated by hardware in auto baud rate detection mode.

Address offset: 0x0C

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
BRR[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 31:16 Reserved, must be kept at reset value.

Bits 15:0 **BRR[15:0]**: USART baud rate

BRR[15:4]
BRR[15:4] = USARTDIV[15:4]
BRR[3:0]
When OVER8 = 0, BRR[3:0] = USARTDIV[3:0].
When OVER8 = 1:
BRR[2:0] = USARTDIV[3:0] shifted 1 bit to the right.
BRR[3] must be kept cleared.

Figure 2 Baud rate calculation references

Then by using USART control register 1 [alternate] (USART_CR1) it's setted to receive, transmit and enable uart.

```c
void UART_Init()
{
	RCC->IOPENR |= RCC_IOPENR_GPIOAEN; //Enable clock for line A
	RCC->APBENR1 |= RCC_APBENR1_USART2EN; //Enable clock for UART2

	GPIOA->MODER &= ~GPIO_MODER_MODE2_Msk; //Clear MODE2
	GPIOA->MODER |= GPIO_MODER_MODE2_1; //Set PA2 as alternate function

	GPIOA->MODER &= ~GPIO_MODER_MODE3_Msk; //Clear MODE3
	GPIOA->MODER |= GPIO_MODER_MODE3_1; //Set PA3 as alternate function

	GPIOA->AFR[0] |= GPIO_AFRL_AFSEL2_0 | GPIO_AFRL_AFSEL3_0;

	USART2->BRR = (104<<4) | (3<<0);//(104<<4) | (3<<0); //0x683
	USART2->CR1 |= USART_CR1_RE; //Enable receive
	USART2->CR1 |= USART_CR1_TE; //Enable transmit
	USART2->CR1 |= USART_CR1_UE; //Enable uart
}
```

“printChar” function is implemented as said in Appendix part. It takes an unsigned 8 bit integer named c and assign UART transmit data register to it. By assigning it, the data transmits over UART, after that inside a while loop if the transmission completed is controlled.

```
void printChar(uint8_t c)
{
	USART2->TDR = c;
	while(!(USART2->ISR & USART_ISR_TC))
	{
	}
}
```

“_print” function takes f as integer, char pointer and length as integer. The aim of this function is to return the length of the char and transmit each char with the help of **“printChar”** function.

```
int _print(int f,char *ptr, int len)
{
	(void)f;
	for(int i=0; i<len; i++)
	{
		printChar(ptr[i]);
	}
	return len; // return length
}
```

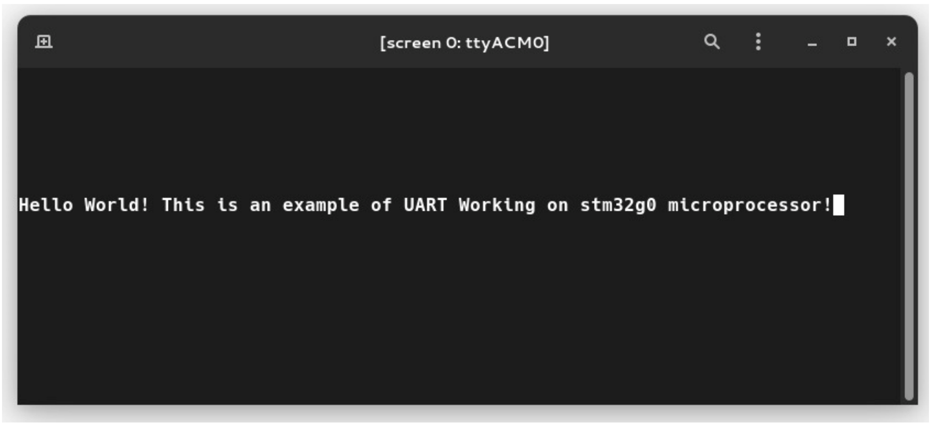
The objective of “**print**” function is to take a char pointer named as “s” and counts the length of the char until a null terminator ('\0') is encountered. Then, it calls **“_print“** with parameters 0, the input string (s), and the calculated length.

```
void print(char *s)
{
	// count number of characters in s string until a null byte comes `\0`
	int length = 0;
	while(s[length]!='\0')
	{
		length++;
	}
	_print(0, s, length);
}
```

‘uart_tx’ function takes an 8-bit unsigned integer as an argument. It calls **‘printChar(c)’** to transmit the character c through the UART.

‘uart_rx’ function receives a single byte through the UART. It enters a loop that waits for the Receive Data Register Not Empty (RXNE) flag in the USART2 Status Register (USART2->ISR) to be set. This loop essentially waits until there is data available to be read from the UART. Once data is available (the RXNE flag is set), it returns the received byte by reading from the Receive Data Register (RDR) of USART2 (return (uint8_t) USART2->RDR).

```
void uart_tx(uint8_t c)
{
	printChar(c);
}
uint8_t uart_rx(void)
{
	while(!(USART2->ISR & USART_ISR_RXNE_RXFNE))
	{
	}
	return (uint8_t) USART2->RDR;
}
```



```
Hello World! This is an example of UART Working on stm32g0 microprocessor!
```

Figure 3 First output from stm32g0

Problem 2.

In this problem, you will implement a PWM signal and drive an external LED using varying duty cycles. Your LED should display a triangular pattern, meaning the brightness should linearly increase and decrease in 1 second. Bonus - After you setup and get your PWM working with the LED, replace the LED with a speaker and see what happens. (Make sure to keep the series resistor)

For this question, the main loop is simple. Since the PWM is updated with timer interrupts, the main loop is left barren. There is a simple while loop to forever with just the initialization code. Above the main loop, all the variables needed for the question are declared and defined. These will be explained in the relevant parts.

```
#include "stm32g0xx.h"

void PWM_Init();
void setDutyCycle(uint16_t dutyCycle);

volatile int dutyCycle = 0;
volatile int direction = 1;

int main(void) {
    PWM_Init();

    while (1) {

    }
}
```

Before we start to set/clear the registers that are needed for PWM control, 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 we want to reset the clock, 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 PWM_Init() {
    // Enable clock for GPIOA and TIM1
    RCC->IOPENR |= RCC_IOPENR_GPIOAEN;
    RCC->APBENR2 |= RCC_APBENR2_TIM1EN;

    // Set PA10 to Alternate Function mode for TIM1_CH3
    GPIOA->MODER &= ~GPIO_MODER_MODE10;
    GPIOA->MODER |= GPIO_MODER_MODE10_1;

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

    // Configure TIM1 for PWM
    TIM1->PSC = 1600 - 1;  // Prescaler for 1kHz PWM frequency
    TIM1->ARR = 100;       // Auto-reload value for 100 steps
    TIM1->CCMR2 |= TIM_CCMR2_OC3M_1 | TIM_CCMR2_OC3M_2//PWM mode 1 on Channel 3
    TIM1->CCER |= TIM_CCER_CC3E;    // Enable capture/compare for channel 3
    TIM1->BDTR |= TIM_BDTR_MOE;     // Main output enable (needed for TIM1)
    TIM1->CR1 |= TIM_CR1_CEN;       // Enable timer

    //for interrupts
    TIM1->DIER |= TIM_DIER_CC1IE;

    // Enable TIM1 interrupt in NVIC
    NVIC_EnableIRQ(TIM1_CC_IRQn);
    NVIC_SetPriority(TIM1_CC_IRQn, 0);
}
```

For fading the led with a bounce, interrupt mentioned above can be written like below. First, it checks if the PWM interrupt is triggered. If it is, the interrupt is cleared and the logic for the fading can be coded. First, the duty cycle is incremented by the direction. Since the duty cycle goes from 0 to 100 then back down to 0, the incrementor can first add by 1 and add by –1, effectively switching directions of the counting.

Lasly our duty cycle can be set to the desired value with **‘setDutyCycle(uint16_t dutyCycle)’** function. It takes an 16 bit integer and sets it to TIM1 channel 3 CCR(capture compare) register.

```
void TIM1_CC_IRQHandler(void) {
    if (TIM1->SR & TIM_SR_CC1IF) { // Check capture/compare 1 interrupt flag
        TIM1->SR &= ~TIM_SR_CC1IF; // Clear the interrupt flag

        // Update the duty cycle
        dutyCycle += direction;
        if (dutyCycle >= 100) {
            dutyCycle = 100;
            direction = -1;
        } else if (dutyCycle <= 0) {
            dutyCycle = 0;
            direction = 1;
        }

        setDutyCycle(dutyCycle); // Update TIM1 channel 3 duty cycle
    }
}

void setDutyCycle(uint16_t dutyCycle) {
    TIM1->CCR3 = dutyCycle;  // Set duty cycle for TIM1 channel 3
}
```

After the code is run, a logic analizer can be used to see the PWM signals.

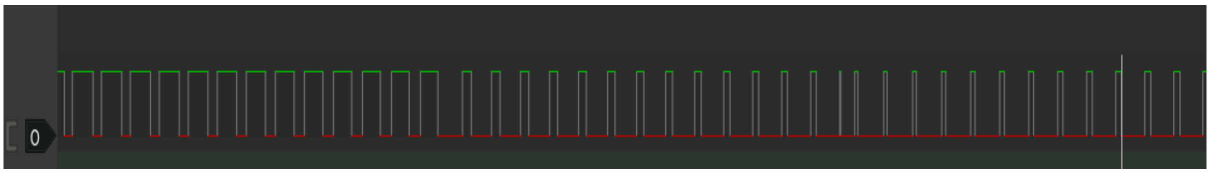


Figure 4 PWM signals with logic analizer

As can be seen from the logic analizer graph above, the pulses gradually get longer and shorter. A led or buzzer connected to the related pin can also be seen to pulse.

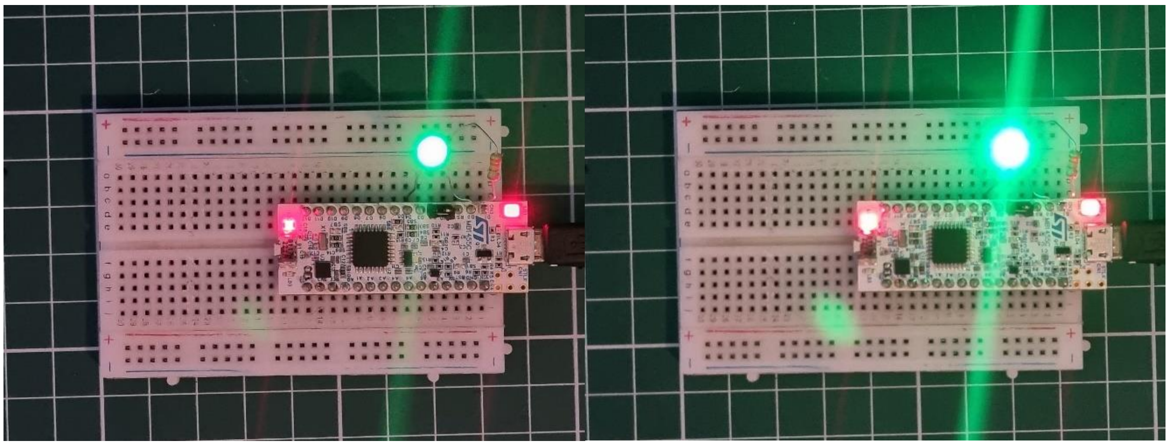


Figure 5 Led brightness in different PWM signals

Problem 3.

In this problem, you will implement a PWM signal and drive an LED at di_erent speeds. You will use keypad to set the duty cycle. Pressing 10# will set the duty cycle to 10% and 90# will set the duty cycle to 90%. Any unusal presses should be ignored. (i.e. 9F#)

Unlike the previous questions, this problem is more complex. It requires the use of multiple interrupts, delays, PWM, UART and more. Because of this the starting declarations of variables have more finesse. With all the usual function declarations, there was a counter variable also defined. This is just for debugging and shows how many times the keypress interrupts are triggered.

More importantly, a matrix containing all the locations for the keypad was declared. For the non-number characters, an "id" was declared. This id's will be important in the following parts of the code. A, B, C, D in the keypad was declared to be some arbitary large number just to differentiate them. Then the dutycycle_raw is used for holding the temporary value for the keypad and the dutycycle variable is used for the current PWM duty cycle. All the declared functions will be explained in their relative sections.

```
#include "stm32g0xx.h"
#include "stdlib.h"

volatile uint8_t counter = 0;
volatile uint32_t millis = 0;

void delay_ms(uint32_t delay);

void SysTickInit();

// PWM function declarations
void PWM_Init();
void setDutyCycle(uint16_t dutyCycle);

// UART initialization and functions
void UART_Init();
void printChar(uint8_t c);
int _print(int f,char *ptr, int len);
void print(char *s);
uint8_t uart_rx(void);
void uart_tx(uint8_t c);

// Constants for keypad
#define A 50
#define B A
#define C A
#define D A
#define clear 10 // "*" sign is for clearing the keypad
#define send 11 // "#" confirms the entered value.

// If this does not exist, counter gets discarded while optimizing
uint8_t temp_counter;

// 4x4 matrix for keypad
int matrix[4][4] = {{1,2,3,A},
                                {4,5,6,B},
                                {7,8,9,C},
                                {clear,0,send,D}};
```

```
uint8_t dutycycle_raw = 0; // Raw duty cycle value
uint8_t dutycycle;         // Processed duty cycle value
uint8_t updateStatus();    // Function to update status of GPIO


void init_keypad();

// Function to scan the keypad
void scan_keypad(uint8_t row);

void EXTI0_1_IRQHandler(void);
void EXTI2_3_IRQHandler(void);
void EXTI4_15_IRQHandler(void);
```

The main function just initializes the relevant peripherals, then prints the duty cycle in to UART every 2 seconds. **'utoa()'** function is for converting the integer value (dutycycle) to a printable string value. This is the main and the only reason that the **'stdlib.h'** was imported.

```
int main(){
    // Initialize UART, keypad, SysTick, and PWM
        UART_Init();
        init_keypad();
        SysTickInit();
         PWM_Init();

        char str_dutycycle[4]; // String to hold duty cycle

        while(1){
                utoa(dutycycle,str_dutycycle, 10); // Convert duty cycle to string
                print("Current Duty Cycle: %");
                for(int i=0;i<4;i++)
                {
                        //Transmit duty cycle character by character
                        uart_tx(str_dutycycle[i]);
                }
                print("\n\r"); //Create new line, go to the beginning of the line

                delay_ms(2000);
                temp_counter = counter; // Counter gets discarded if not used.

        }
        return 0;
}
```

After the main loop is writtern, the keypad initialization code can also be written. This part of the code enables all the relevant Pins (which pins are shown below. Green pins are defined as inputs, and red are defined as outputs). All the output pins are held high. When a button is pressed, an output pin pulls an input pin to high. To not get held by another operations when a key is pressed, the input pins are connected to external interrupts. This is part is identical to what was done in lab3, but here there is 4 pins to trigger external interrputs. Since these pins (PA9, PB0, PB2, PB8) are in external interrupt lines 0-1, 2-3 and 4-15, all of these interrupt handlers have to be defined and enabled.

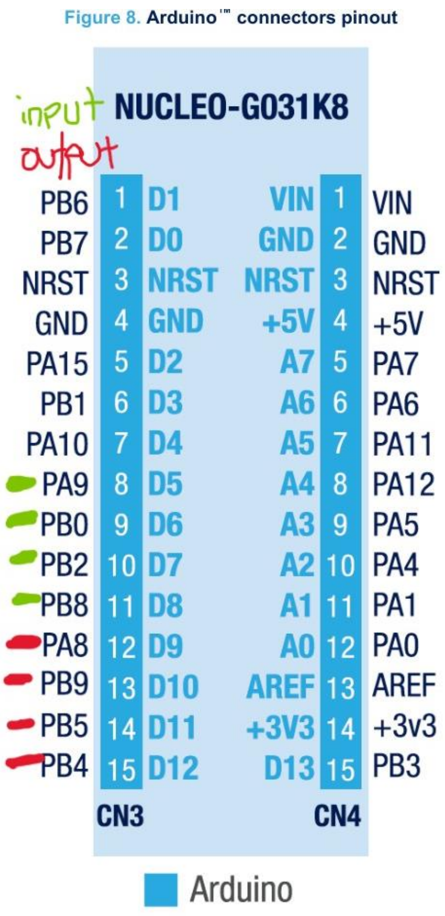


Figure 6 Pinout shematic

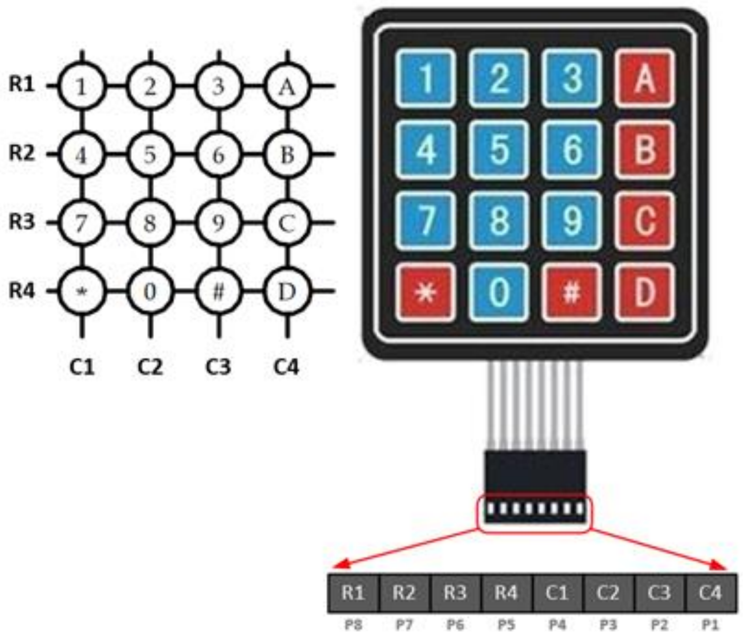


Figure 7 Keypad diagram

```
// Initialize all the inputs and interrupts needed for keypad
void init_keypad(){
    // Enable clock for GPIOA and GPIOB
    RCC->IOPENR |= RCC_IOPENR_GPIOAEN | RCC_IOPENR_GPIOBEN;

    // Configure GPIO pins for keypad
    GPIOA->MODER &= ~(GPIO_MODER_MODE9_Msk | GPIO_MODER_MODE8_Msk);
    GPIOA->MODER |= GPIO_MODER_MODE8_0;

    GPIOB->MODER &= ~(GPIO_MODER_MODE0_Msk |
                      GPIO_MODER_MODE2_Msk |
                      GPIO_MODER_MODE4_Msk |
                      GPIO_MODER_MODE5_Msk |
                      GPIO_MODER_MODE8_Msk |
                      GPIO_MODER_MODE9_Msk);

    GPIOB->MODER |= GPIO_MODER_MODE4_0 |
                    GPIO_MODER_MODE5_0 |
                    GPIO_MODER_MODE9_0;

    // Set initial output levels for GPIO pins to HIGH
    GPIOA->ODR |= GPIO_ODR_OD8;
    GPIOB->ODR |= GPIO_ODR_OD9 | GPIO_ODR_OD5 | GPIO_ODR_OD4;

    // Setup pins as Pullup
    GPIOA->PUPDR &= ~GPIO_PUPDR_PUPD9_Msk;
    GPIOB->PUPDR &= ~GPIO_PUPDR_PUPD0_Msk | GPIO_PUPDR_PUPD2_Msk |
GPIO_PUPDR_PUPD8_Msk;

    GPIOA->PUPDR |= GPIO_PUPDR_PUPD9_1;
    GPIOB->PUPDR |= GPIO_PUPDR_PUPD0_1 | GPIO_PUPDR_PUPD2_1|GPIO_PUPDR_PUPD8_1;

    // Enable APB peripheral clock
    RCC->APBENR2 |= (1U<<0);

    // Configure external interrupt lines for keypad
    EXTI->EXTICR[2] |= (0U << 8*1);
    EXTI->EXTICR[0] |= (1U << 8*0);
    EXTI->EXTICR[0] |= (1U << 8*2);
    EXTI->EXTICR[2] |= (1U << 8*0);

    // Configure rising edge trigger for external interrupts
    EXTI->RTSR1 |= (1U << 9);
    EXTI->RTSR1 |= (1U << 0);
    EXTI->RTSR1 |= (1U << 2);
    EXTI->RTSR1 |= (1U << 8);

    // Enable external interrupts
    EXTI->IMR1 |= (1U<<9);
    EXTI->IMR1 |= (1U<<0);
    EXTI->IMR1 |= (1U<<2);
    EXTI->IMR1 |= (1U<<8);

    // Enable NVIC interrupts
    NVIC_SetPriority(EXTI0_1_IRQn, 2);
    NVIC_EnableIRQ(EXTI0_1_IRQn);

    NVIC_SetPriority(EXTI2_3_IRQn, 2);
    NVIC_EnableIRQ(EXTI2_3_IRQn);
    NVIC_SetPriority(EXTI4_15_IRQn, 2);
    NVIC_EnableIRQ(EXTI4_15_IRQn);
}
```

Then the main keypad scanning logic can be written. When an external interrupt gets triggered, this function is called. This function takes which row that caused the interrupt as an input, this is used to look up the keypad map.

The main logic of this function depends on turning on and off of the outputs (columns) individually to see which column is responsible for pulling a row (input that caused the interrupt) high. It does this by turning off the column, see if any of the rows turned off and if it did, set the current column variable to that row. Then it turns on the row again to reset the column for the next press.

After the column is determined, a small delay for debouncing was done. If this was not done, when any of the keys was pressed, interrupt was triggered rapidly and without control.

When the debouncing wait is over, now determining what to do with the coulmn and row variables can begin. First, the corresponding id/number is looked in the button matrix. If the button id equals "#" (aka send), it sets the PWM duty cycle to our desired value, resets the raw duty cycle value (for the next input, otherwise pressing the clear key is required after each send). Clear key just clears the current input. When a key that corresponds to a digit, the function shifts the current entered value by a decimal digit. This can be accomplished by multiplying the the existing value by 10. Then just adding the currently pressed digit gets our desired value. This is the same as adding the currently pressed digit to the right of the previosly pressed value, but this way does not require the conversion of int to str and then back to int. If the operations above caused a greater value than 100, it resets the value to 100, because that's probably what was intended in the first place.

$$5$$

$$5 * 10 = 50$$

$$50+6=56$$

Example of pressing 5 and then 6 was shown above.

The row reading and updating function just checks if any of the rows are high. It returns the value 1 if there is a row that is held high. It checks all the rows at once since if there cant be more than one row that can be held high at normal intended use case, and this makes the coding simpler with the trade off with speed.

```
// This function is for the logic for when keypad is pressed
void scan_keypad(uint8_t row)
{
    delay_ms(20); // Delay for debouncing
    uint8_t column;

    // Check each column to identify the pressed key
    // This turns off each row and checks if any of the rows turned off.
    GPIOA->ODR &= ~GPIO_ODR_OD8;
    if (!updateStatus()){
        column = 0;
    }
    GPIOA->ODR |= GPIO_ODR_OD8; //then turns them on again.

    GPIOB->ODR &= ~GPIO_ODR_OD9;
    if (!updateStatus()){
        column = 1;
    }
    GPIOB->ODR |= GPIO_ODR_OD9;

    GPIOB->ODR &= ~GPIO_ODR_OD5;
    if (!updateStatus()){
        column = 2;
    }
    GPIOB->ODR |= GPIO_ODR_OD5;

    GPIOB->ODR &= ~GPIO_ODR_OD4;
    if (!updateStatus()){
        column = 3;
    }
    GPIOB->ODR |= GPIO_ODR_OD4;

    // Lookup the matrix for what value is in that position
    uint8_t current_num = matrix[row][column];
    if(current_num == send)
    {
        setDutyCycle(dutycycle_raw); // Set PWM duty cycle
        dutycycle = dutycycle_raw;   // Update duty cycle since raw duty cycle
gets reset.
        dutycycle_raw = 0;           // Reset raw duty cycle
    }
    else if(current_num == clear)// if clear is pressed, reset raw duty cycle
    {
        dutycycle_raw = 0;
    }
    else if(current_num < 10) //if a normal number is pressed, update the raw
duty cycle
    {
        dutycycle_raw = 10*dutycycle_raw + current_num; // Update raw duty cycle
        if(dutycycle_raw > 100)
        {
            dutycycle_raw=100; // Limit duty cycle to 100
        }
    }
}
```

As said previously, external interrpts just call the scan keypad function. The important part here is that all the pins can be differentiated, and since each pin is responsible for for a single column, the argument of which row is pressed can be just passed to the function. For the inputs that are not on the same row, the logic is just simple. For the rows that share the same interrupt, the status register that is related to that interrupt needs to be checked. The status register holds which line caused the interrupt to occur. After the register is checked, the scan_keypad function can be called with the relevant row information.

```
// Read the status of the rows
uint8_t updateStatus()
{
    return (GPIOA->IDR & GPIO_IDR_ID9) ||
           (GPIOB->IDR & GPIO_IDR_ID0) ||
           (GPIOB->IDR & GPIO_IDR_ID2) ||
           (GPIOB->IDR & GPIO_IDR_ID8);
}

// Interrupt handler for EXTI0_1
void EXTI0_1_IRQHandler(void) {
    counter++; //to count how many times the interrupts are called

    scan_keypad(1); // Scan keypad for row 2
    EXTI->RPR1 = EXTI_RPR1_RPIF0;  // Clear the pending bit for EXTI line 0
}

// Interrupt handler for EXTI2_3
void EXTI2_3_IRQHandler(void) {
    counter++;//to count how many times the interrupts are called

    scan_keypad(2); // Scan keypad for row 3
    EXTI->RPR1 = EXTI_RPR1_RPIF2;  // Clear the pending bit for EXTI line 2
}

// Interrupt handler for EXTI4_15
void EXTI4_15_IRQHandler(void) {
    counter++;//to count how many times the interrupts are called

    // Determine which key was pressed and scan accordingly
    if((EXTI->RPR1&EXTI_RPR1_RPIF8))
    {
        scan_keypad(3); // Scan keypad for row 4
    }
    if((EXTI->RPR1&EXTI_RPR1_RPIF9))
    {
        scan_keypad(0); // Scan keypad for row 1
    }
    EXTI->RPR1 |= EXTI_RPR1_RPIF8; // Clear the pending bit for EXTI line 8
    EXTI->RPR1 |= EXTI_RPR1_RPIF9; // Clear the pending bit for EXTI line 9
}
```

After the interrupts are defined, the code is *almost* identical to to to what was done in the previous questions. The only difference is that this questions PWM implementation does not update the PWM duty cycle based on the timer value. This forgoes the need for the timer interrupt in the question.

```
void setDutyCycle(uint16_t dutyCycle)
{
	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
	// Set PA10 to Alternate Function mode
	GPIOA->MODER |= GPIO_MODER_MODE10_1;
			// Clear the current AF setting for PA10
	GPIOA->AFR[1] &= ~(0xF << ((10 - 8) * 4));
	GPIOA->AFR[1] |= (2 << ((10 - 8) * 4)); // Set the AF (AF2) for TIM1_CH3
for PA10

	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->CCER |= TIM_CCER_CC3E; // 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); //set initial duty cycle to 0
}
```

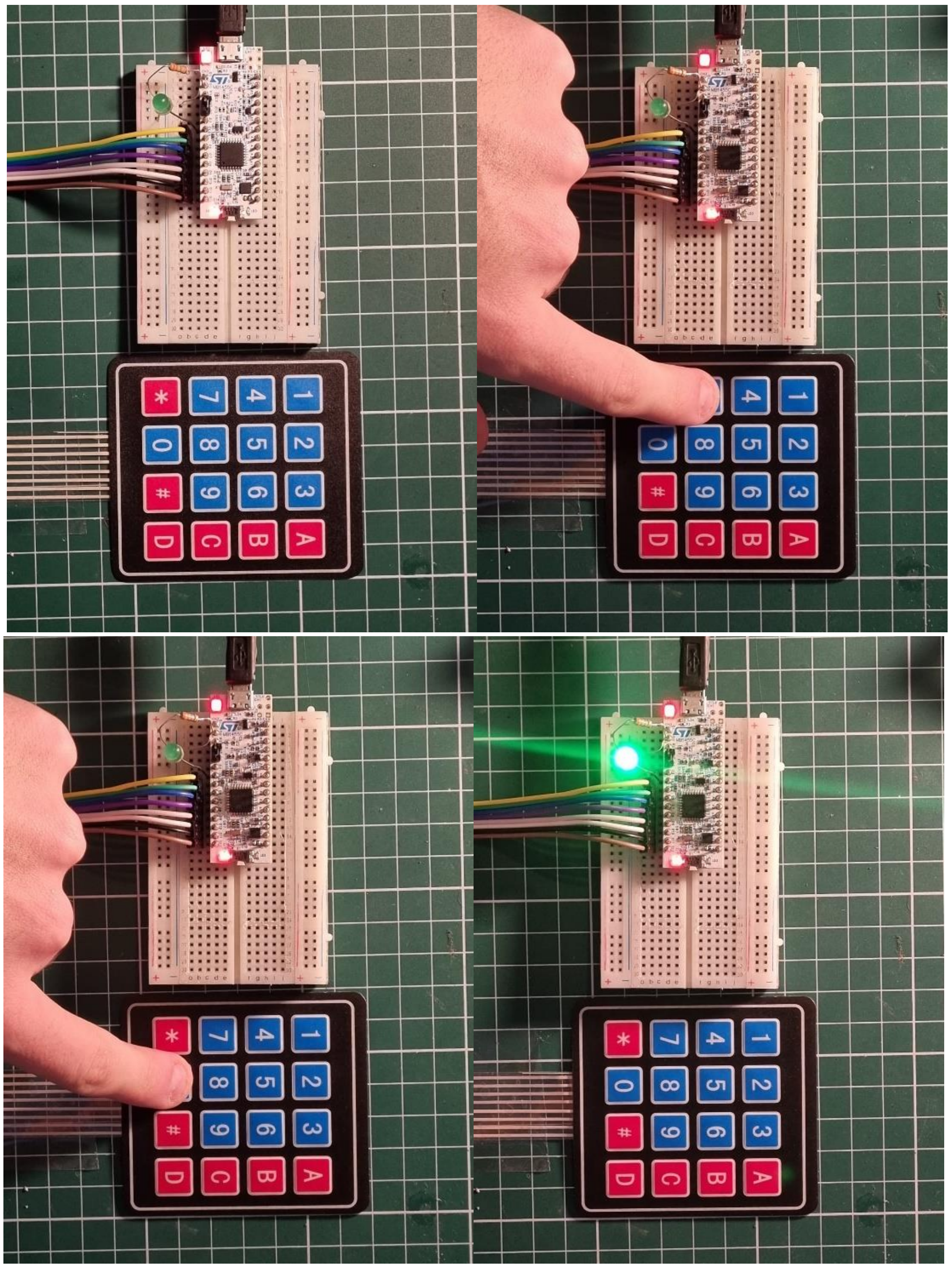
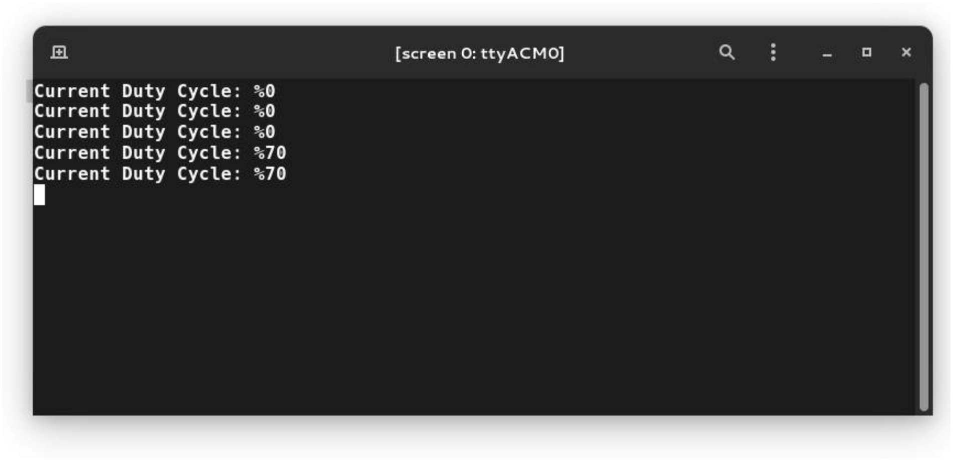


Figure 8 Led with keypad connected to it.



```
Current Duty Cycle: %0
Current Duty Cycle: %0
Current Duty Cycle: %0
Current Duty Cycle: %70
Current Duty Cycle: %70
```

Figure 9 Duty Cycle change when button is pressed.

Conclusion:

In conclusion, this lab assignment has provided a comprehensive exploration of fundamental concepts in embedded systems, with a focus on UART communication and PWM signal generation. Through the three distinct problems tackled in this assignment, we have gained valuable hands-on experience in configuring hardware interfaces, implementing communication protocols, and modulating signals for effective control of external devices.

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

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

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

cortexM0 referance manual.pdf

https://www.youtube.com/watch?v=weu9J-mKkuE&list=PLiWDuW_1eKN5IpCAaeE9ncaPtxK61YFf-&index=28&ab_channel=FurkanCayci