Cheatsheet for Sensor System Laboratory Exam

Components

Light sensor: PA0
Potentiometer: PA1
Usart TX: PA2
Usart RX: PA3
Green LED (LD2 + SCK SPI): PA5
MISO SPI: PA6
MOSI SPI: PA7
microphone: PA8
speaker: PA9
USART1_TX (IR receiver): PA9
USART1_RX (IR receiver): PA10
SS SPI: PB6
I2C1_SCL (external temperature sensor and accelerometer): PB8
I2C1_SDA (external temperature sensor and accelerometer): PB9
TIM2_CH3 (IR transmitter): PB10
TIM3_CH1 (encoder): PC6
TIM3_CH2 (encoder): PC7
Blue push button: PC13

EXTI peripheral

The EXTI (External Interrupt) on an STM32 microcontroller is a peripheral used to handle interrupts triggered by external events, such as changes in GPIO pins.
To enable it, we should set the PIN to GPIO_EXTIx and then in the NVIC, we should flag the field EXTI line [9, 5] interrupts or EXTI line [15, 10] interrupts based on the GPIO_EXTIx (x has to be in the interval of the interrupts of the line).
Then in the System Core/GPIO we should select the GPIO mode, for example Exernal Interrupt in Rising Edge trigger detection.
Then we should write the callback function:

```
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin) {
        switch(GPIO_Pin) {
        case GPIO_PIN_13:

        }
}
```

and in the body of the function put a switch to be sure that the right GPIO_PIN has generated the interrupt.

Read a Pin:

```
GPIO_PinState HAL_GPIO_ReadPin(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin)
```

Write a Pin:

void HAL_GPIO_WritePin(GPIO_TypeDef GPIOx, uint16_t GPIO_Pin, GPIO_PinState PinState)*
Invert the state of a Pin:
void HAL_GPIO_TogglePin(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin)

Timer setup

PWM frequency: $$f_{PWM} = \frac{\overset{84\ \text{MHz}}{f_{TIM}}}{\underbrace{(ARR+1)}_{10'000-1}\cdot\underbrace{(PSC+1)}_{8400-1}} = 1\ \text{Hz}$$

1 second values: Prescaler: 8400-1
Counter period: 10000-1
4ms values: Prescaler: 8400-1
Counter period: 40-1
1ms values: Prescaler: 8400-1
Counter period: 10-1

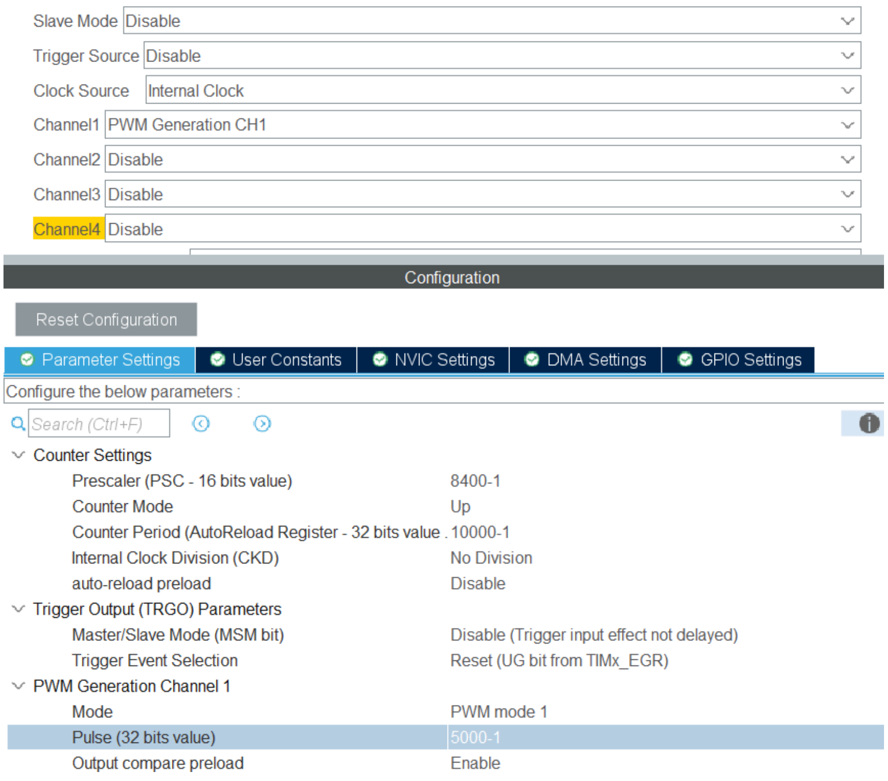
Generate a PWM with duty cycle 50%

ftim = 84 MHz

For the frequency of the timer: $f_{pwm} = \frac{f_{tim}}{(ARR+1)*(PSC+1)}$

For the duty cycle: $DC = \frac{CCRx+1}{ARR+1}$

In our case the CCRx is the Pulse


Slave Mode: Disable
Trigger Source: Disable
Clock Source: Internal Clock
Channel1: PWM Generation CH1
Channel2: Disable
Channel3: Disable
Channel4: Disable

Configuration

Counter Settings

Prescaler (PSC - 16 bits value)	8400-1
Counter Mode	Up
Counter Period (AutoReload Register - 32 bits value .	10000-1
Internal Clock Division (CKD)	No Division
auto-reload preload	Disable

Trigger Output (TRGO) Parameters

Master/Slave Mode (MSM bit)	Disable (Trigger input effect not delayed)
Trigger Event Selection	Reset (UG bit from TIMx_EGR)

PWM Generation Channel 1

Mode	PWM mode 1
Pulse (32 bits value)	5000-1
Output compare preload	Enable

Then in the code to start the timer in PWM mode, we should write:

*HAL_TIM_PWM_Start(TIM_HandleTypeDef *htim, uint32_t Channel)*

Instead to stop it:

*HAL_TIM_PWM_Stop(TIM_HandleTypeDef *htim, uint32_t Channel)*

Usage example

In the lab 1c we have to configure the LED to blink at 1 Hz with duty cycle of 50%.
To do it we have to link a timer with the LED, since the LED is linked to the port PA5, we click on it in the ioc file and we see that the only timer attached to it is the TIM2 Channel 1, so we configure the timer and the channel to the proper set up to blink at 1 Hz.
In the software then we use the HAL_TIM_PWM_Start(...)

Generate a timer interrupt

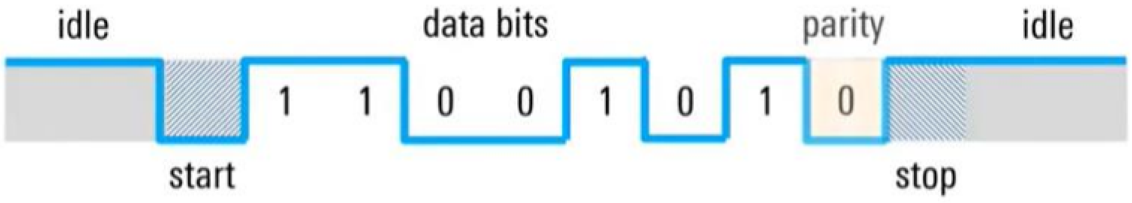
To generate an interrupt every 1 second, we have to use the same formula of the frequency, but in the NVIC we have to flag the TIMx global interrupt. (And in the GUI choose “output compare CHx” instead of “PWM Generation CHx”)
Then, to start the timer we should write:
*HAL_TIM_Base_Start_IT(TIM_HandleTypeDef * htim)*
Instead to stop it:
*HAL_TIM_Base_Stop_IT(TIM_HandleTypeDef * htim)*

USART communication

Introduction

The USART communication is a type of a serial interface device that can be programmed to communicate asynchronously or synchronously.
We will usually use the UART communication that is a type of asynchronous communication in which the sender and the receiver must agree on the baud rate.
Data is transmitted as "frames":

- Start/stop bits
- Data bits (5 to 9, usually 7 or 8, sent LSB first)
- Parity bit (optional) useful for error detection



On our board

In the board the USART2 is enabled by default.
To make it work also with the DMA: enable DMA and global interrupt, moreover in the string sent add at the end \r and \n.
The default baud rate is 115200 bit/s, but for the IR communication set it to: 2400 bit/s
Be sure to match the BaudRate between the board and MATLAB.

LCD

To use the LCD we need to:

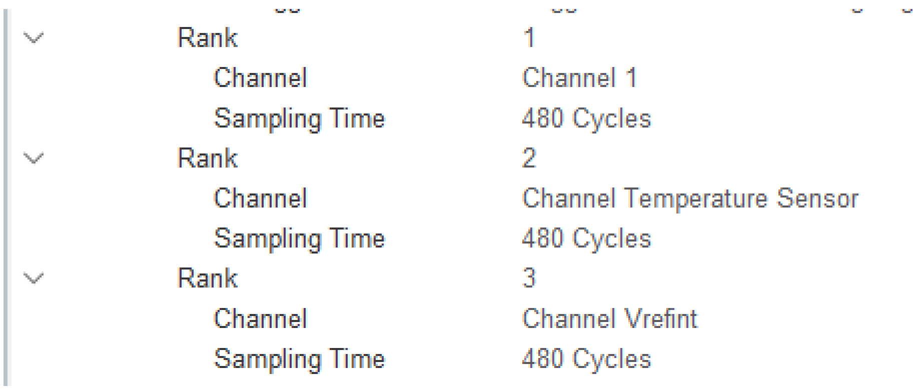
- Set the pins: PA4, PB1, PB2, PB12, PB13, PB14, PB15 to GPIO_Output
- Copy the file PMBD16_LCD.h in project/Core/Inc and the file PMBD16_LCD.c in the project/Core/Src folder
- Write the functions that we need

Before using it, initialize with *lcd_initialize()* and then turn on the backlight with *lcd_backlight_ON()*.
To print a string use *lcd_println(char string[], uint8_t row)*: it prints a string on the top (row = 0) or bottom (row = 1) row of the LCD. Remember the maximum length of the string is 16 characters.
To print a bar use *lcd_drawBar(int value)*: it prints a bargraph on the bottom row of the LCD controller. Value range: 0 to 80.

ADC

The set correctly the ADC:

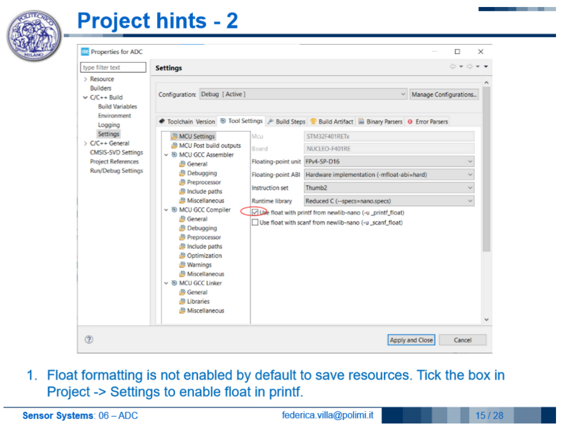
- Set the input channels
- Set the clock prescaler to PCLK2 divided by 4 (the resultant frequency is 84Mhz / 4 = 21 Mhz)
- Optionally select some configuration to enable DMA or continuous scan
- In the rank table select the channel and the sampling time (we usually modify only the sampling time to 480)
- If reading also other sensors such as Vref and Temp sensor: set ranks accordingly and the number of conversion must be modified, in the example below the number of conversions must be set to 3



Rank	1
Channel	Channel 1
Sampling Time	480 Cycles
Rank	2
Channel	Channel Temperature Sensor
Sampling Time	480 Cycles
Rank	3
Channel	Channel Vrefint
Sampling Time	480 Cycles

- Polling: it does not need to be initialized, but before getting the value of the ADC we have to use the *HAL_ADC_Start(&hadc1)* function. Then we need to use the *HAL_ADC_PollForConversion(&hadc1, int timeout)* function to poll the conversion and to get the value we can use the *HAL_ADC_GetValue(&hadc1)* function.
- Interrupt: it does need to flag the ADC1 global interrupt and to initialize the ADC with *HAL_ADC_Start_IT(&hadc1)* function, then to retrieve the value use *HAL_ADC_GetValue(&hadc1)* function.
- Interrupt via hardware: remember that the ADC can be triggered via hardware using a timer. We have done it in the project 2b of the LAB 6.

If there is an error in the floating point, enable it in the settings:



Project hints - 2

Properties for ADC

Settings

Configuration: Debug [Active]

Mcu	STM32F401RETx
Board	NUCLEO-F401RE
Floating-point unit	FPv4-SP-D16
Floating-point ABI	Hardware implementation (-mfloat-abi=hard)
Instruction set	Thumb2
Runtime library	Reduced C (--specs=nano.specs)

☑ Use float with printf from newlib-nano (-u _printf_float)
☐ Use float with scanf from newlib-nano (-u _scanf_float)

1. Float formatting is not enabled by default to save resources. Tick the box in Project -> Settings to enable float in printf.

Sensor Systems: 06 – ADC   federica.villa@polimi.it   15 / 28

Potentiometer settings

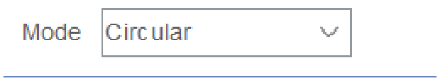
In the ioc file we have to set the PA1 to GPIO_AnalogADC1_IN1, then in the ADC we have to set the IN1 channel.

Light sensor settings

In the ioc file we have to set the PA0 to GPIO_AnalogADC1_IN0, then in the ADC we have to set the IN0 channel.

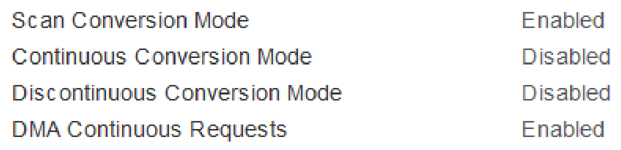
Circular mode setting

To enable the DMA circular mode:



Mode	Circular

it operates in scan conversion mode, with DMA continuous request



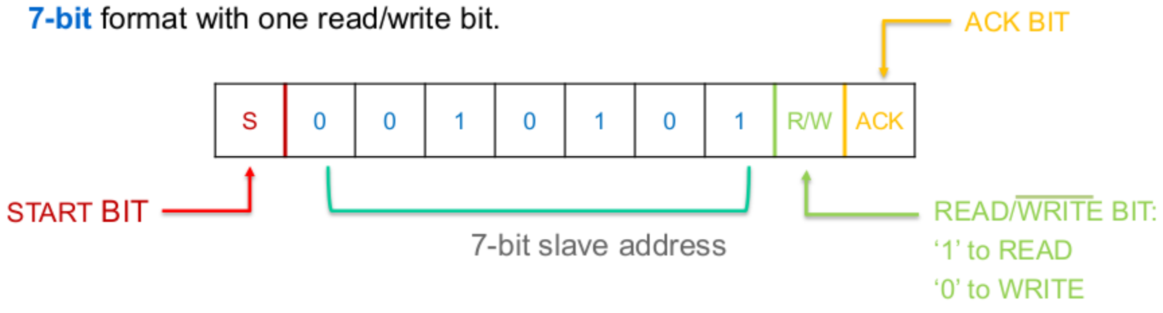
Scan Conversion Mode	Enabled
Continuous Conversion Mode	Disabled
Discontinuous Conversion Mode	Disabled
DMA Continuous Requests	Enabled

I2C

Introduction

The I2C is a multi-master, multi-slave protocol in which many devices are connected to the same bus to send or retrieve data.
It is a 2 wires connection: SDA (data), SCL (clock) and one common ground. Both SDA and SCL are bidirectional.
Initially SCL and SDA are kept “high”, to acquire the channel the master must start a transition from high to low of SDA, while SCL is kept high (START condition).
To terminate the communication the master should start a transition from low to high of SDA while SCL is kept high.
Each device connected to the I2C bus has a unique address and it could be a 7 bit address (used in the board) or a 10 bit address.

- **7-bit** format with one read/write bit.



The software engineers should take care about the baud rate of the communication (we select the standard mode), since the low to high transitions are limited by the RC circuit (pull up resistor + bus capacitance), so if the clock is too fast, the peripheral cannot acquire the transitions from low to high.

HAL functions

To transmit:
*HAL_I2C_Master_Transmit(I2C_HandleTypeDef *hi2c, uint16_t DevAddress, uint8_t *pData, uint16_t Size, uint32_t Timeout)*
where DevAddress is the address of the peripheral: remember that usually the addresses are 7 bits + 1 bit to read or write (read 1, write 0).
It is used to write some registers in the peripheral connected to the I2C bus.

To read:
*HAL_I2C_Master_Receive(I2C_HandleTypeDef *hi2c, uint16_t DevAddress, uint8_t *pData, uint16_t Size, uint32_t Timeout)*
where DevAddress is the address of the peripheral: remember that usually the addresses are 7 bits + 1 bit to read or write (read 1, write 0).
It is used to read some registers in the peripheral connected to the I2C bus.

Temperature sensor (LM75/LM75B)

In the ioc file, we should activate the I2C1 in standard mode with 100_000 Hz as clock speed, then we set PB8 as I2C1_SCL and PB9 as I2C1_SDA.
We would like to acquire the temperature of the sensor: initially we have to know its address, that is 0b1001000 = 0x48 = 72, to this address we must concatenate 1 bit to read or write.
We initially have to set the temperature sensor in a mode that when we read, it has to give us the temperature, so we must write in the sensor register what we would like to read: in our case we would like to read the temperature register, so we have to write 0x00 = 0.
In summary:

- We have to write in the sensor register using the *HAL_I2C_Master_Transmit()* where in the DevAddress we must put 0b10010000 = 0x90 = 144 and as the pData we must put a single value that contains 0 (temperature register).
- We have to read the temperature register using the *HAL_I2C_Master_Receive()* where in the DevAddress we must put 0b10010001 = 0x91 = 145 and as the pData we must put a single value that will be overwritten with the temperature

Accelerometer (LIS2DE)

The only difference with respect to the temperature sensor is that if we want to write in some control register, we should put in the DevAddress the address concatenated with the write bit and then in the data we should put first the number of the control register and then the value to put in the register.
Another particularity is the autoincrement address, so instead of reading 1 register at a time, we can set the autoincrement and each time a register is read, the sensor will increment a counter and the next time the sensor will give the value of the next register.

SPI

Introduction

SPI is a synchronous serial communication interface used for short-distance communication, moreover it is full-duplex (bidirectional) and uses a single master and multiple slaves.
Slaves are selected individually using a signal called slave select (SS) or chip select (CS) asserted by the master.
Transmissions normally involve two shift registers of some given word-size, such as eight bits. During each clock cycle, the master sends a bit on the MOSI line and the slave reads it. At the same time, the slave outputs a bit on the MISO line and the master reads it.
There are 2 different configurations:

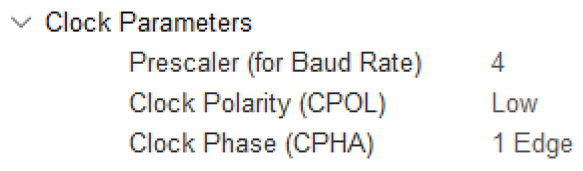
- Independent slaves: every slave has an independent SS line.
- Daisy chain: a single SS line is shared by all devices. Communication is handled like a shift register.

HAL functions

*HAL_StatusTypeDef HAL_SPI_Transmit (SPI_HandleTypeDef *hspi, uint8_t * pData, uint16_t Size, uint32_t Timeout)*
Usual function used to transmit.
*HAL_StatusTypeDef HAL_SPI_Transmit_DMA (SPI_HandleTypeDef *hspi, uint8_t * pData, uint16_t Size)*
Usual function to transmit using the DMA.

LED matrix

To make it work, we have to set the pins in the correct way: PA5 SP1_SCK, PA6 SPI1_MISO, PA7 SPI1_MOSI, PB6 GPIO_Output (SS).
After that set the parameters as shown in the figure (MSB first)



Clock Parameters

Prescaler (for Baud Rate)	4
Clock Polarity (CPOL)	Low
Clock Phase (CPHA)	1 Edge

After that we have to put in a SET state the SS to not disturb the signal stored in the registers of the LED matrix.
To send data, the SS must be put in a RESET state then send the data with the *HAL_SPI_Transmit_DMA(&hspi1, coord[j][i], 2)*.
Then put the SS in a SET state when the DMA has finished, so we have to set it in the callback of the DMA.

Keyboard

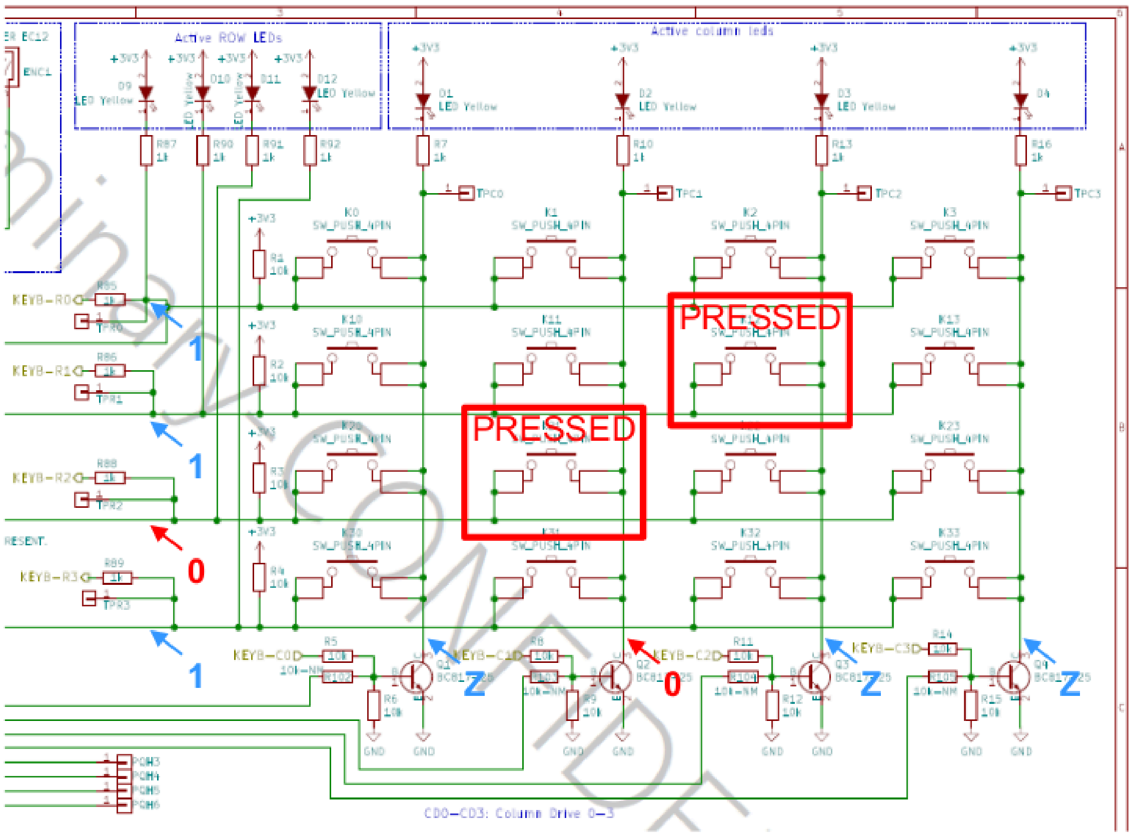
Initially we have to setup the ioc file:

- set PC2, PC3, PC12, PC13 to GPIO_Input (rows)

- set PC8, PC9, PC10, PC11 to GPIO_Output (columns)

To detect the press of a key, we have to do this:

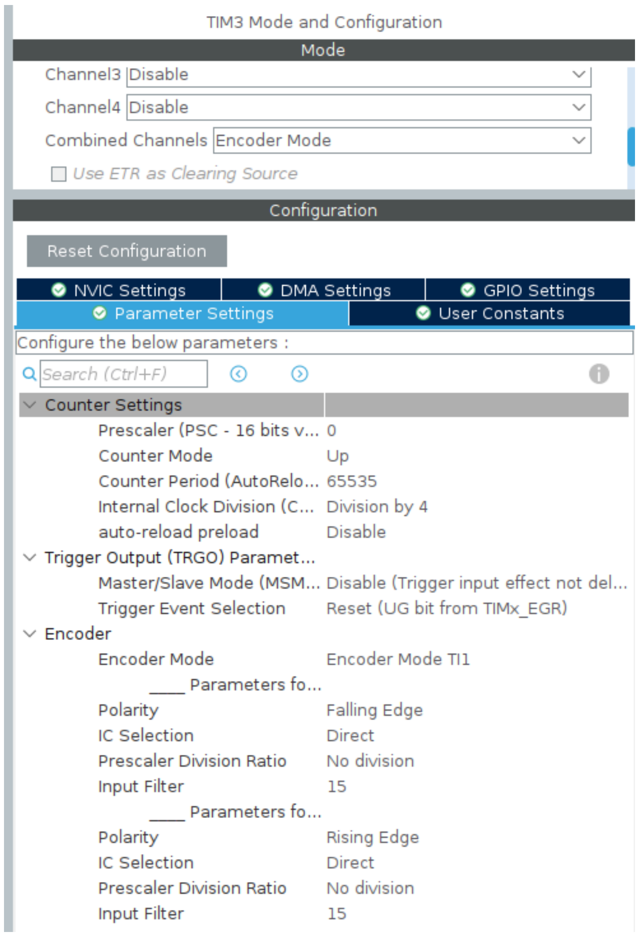
- Scan through the columns setting one column at a time (the others are in a RESET state)
- Read the rows, if the row is in a SET state the key is not pressed, otherwise if the row is in a RESET state the key is pressed.



REMEMBER: to solve the blue pushbutton issue, we put the next column in a SET state during the reading of the current column, in this way there is enough time to propagate the signal. Moreover we added a software debouncing mechanism.
The timer 2 is set to 1000 Hz (1 ms).

Encoder

We have to set the ioc file, set the PC6 to TIM3_CH1, PC7 to TIM3_CH2. Then in the TIM3 panel we have to do this.


TIM3 Mode and Configuration

Mode

Channel3 Disable
Channel4 Disable
Combined Channels Encoder Mode
☐ *Use ETR as Clearing Source*

Configuration

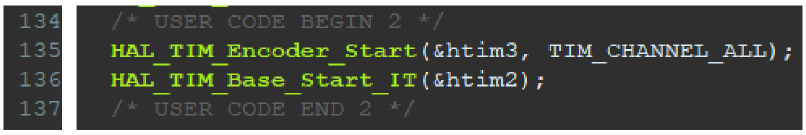
Reset Configuration

NVIC Settings | DMA Settings | GPIO Settings | Parameter Settings | User Constants

Configure the below parameters :

Counter Settings	
Prescaler (PSC - 16 bits v...	0
Counter Mode	Up
Counter Period (AutoRelo...	65535
Internal Clock Division (C...	Division by 4
auto-reload preload	Disable
Trigger Output (TRGO) Paramet...	
Master/Slave Mode (MSM...	Disable (Trigger input effect not del...
Trigger Event Selection	Reset (UG bit from TIMx_EGR)
Encoder	
Encoder Mode	Encoder Mode TI1
____ Parameters fo...	
Polarity	Falling Edge
IC Selection	Direct
Prescaler Division Ratio	No division
Input Filter	15
____ Parameters fo...	
Polarity	Rising Edge
IC Selection	Direct
Prescaler Division Ratio	No division
Input Filter	15

To detect the RPM of the encoder we have to initialize the TIM3 to be wired to the encoder (it is the only timer connected to it). Moreover with these settings, there is a digital filtering that avoids bouncing of the hardware. How? The filtering is made of an event counter in which N consecutive events are needed to validate the transition on the output. Using 15 as input filter means that N is equal to 8 and the frequency of sampling is equal to f_dts / 32 where f_dts = f_ck_int / CKD (in the case of the figure CKD = 4), CKD is the internal clock division.


```
/* USER CODE BEGIN 2 */
HAL_TIM_Encoder_Start(&htim3, TIM_CHANNEL_ALL);
HAL_TIM_Base_Start_IT(&htim2);
/* USER CODE END 2 */
```

Then we can calculate the RPM every 1 second using the timer interrupt of the TIM2, so we can get the counter of the TIM3 and by subtraction we can calculate the RPM.
We must pay attention to the overflow or underflow checking a flag and make the right computation, then we can clear it.

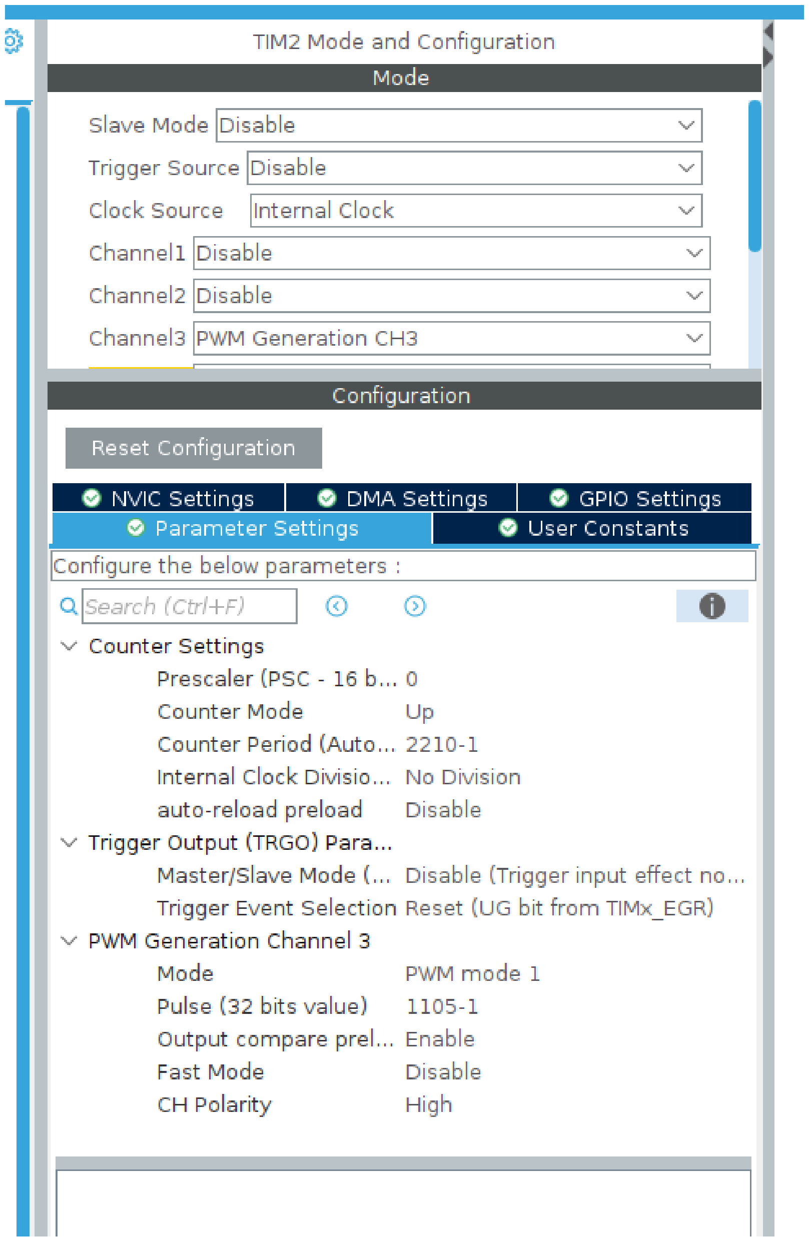
IR communication

To receive ones and zeroes, we use a receiver that consists of: a photodiode, an amplifier with automatic gain control (to amplify the signal), a bandpass filter and demodulator (38 kHz). So when modulated light (IR light) is received the output goes to 0.

To transmit ones and zeroes, we can use PWM to drive the LED at 50% duty cycle, 38 kHz, to stimulate the receiver.
To transmit a 1 the LED must stay off, to transmit a 0 the LED must blink at a frequency of 38 kHz, for how much time? 1 / baud_rate, we use a baud_rate of 2400 bits.

Transmitter

To set up the transmitter we have to set the PB10 to TIM2_CH3, this will command the IR LED transmitter.
So the TIM2 panel should be set up like this



TIM2 Mode and Configuration

Mode

Slave Mode: Disable
Trigger Source: Disable
Clock Source: Internal Clock
Channel1: Disable
Channel2: Disable
Channel3: PWM Generation CH3

Configuration

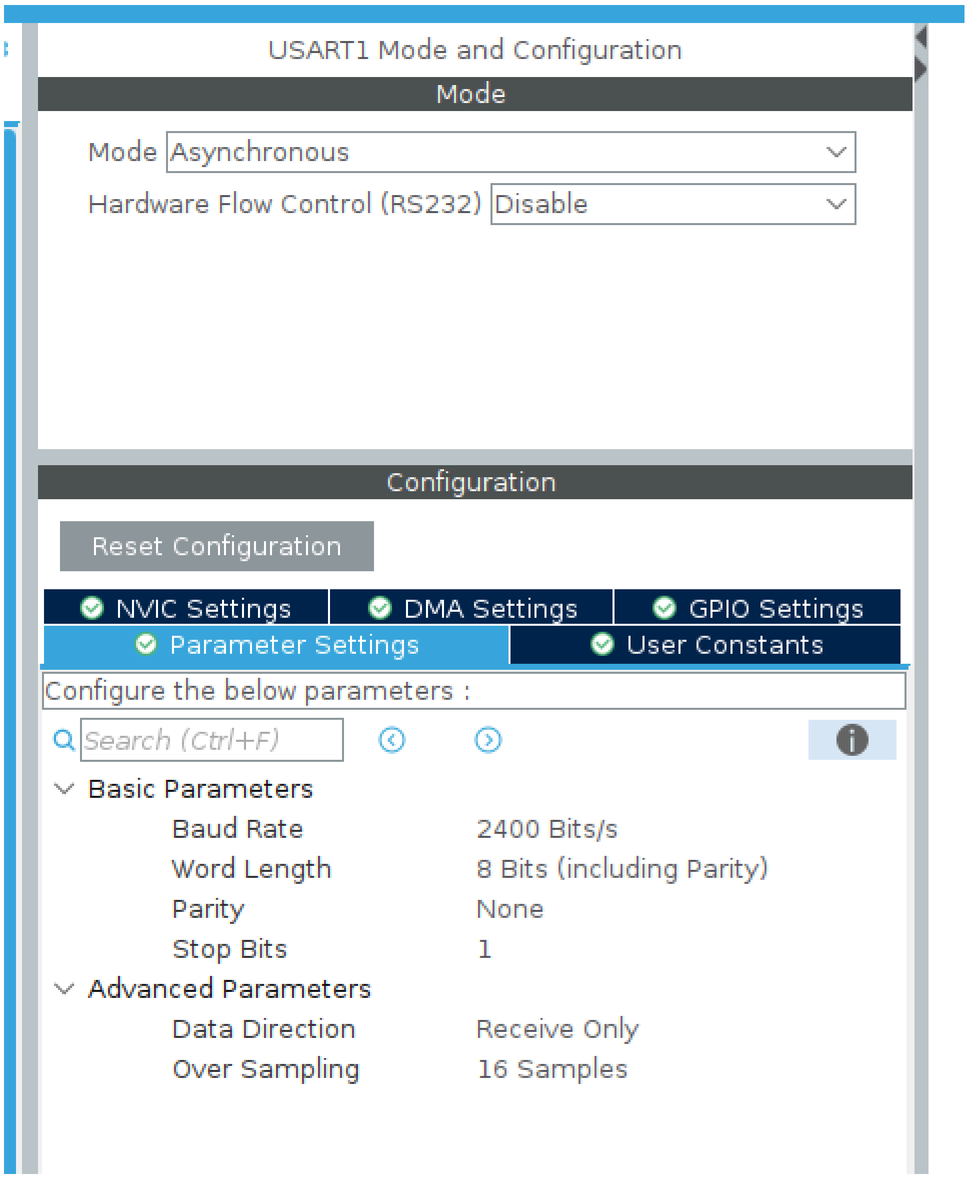
Configure the below parameters :

```
Counter Settings
    Prescaler (PSC - 16 b...     0
    Counter Mode                 Up
    Counter Period (Auto...      2210-1
    Internal Clock Divisio...    No Division
    auto-reload preload          Disable
Trigger Output (TRGO) Para...
    Master/Slave Mode (...       Disable (Trigger input effect no...
    Trigger Event Selection      Reset (UG bit from TIMx_EGR)
PWM Generation Channel 3
    Mode                         PWM mode 1
    Pulse (32 bits value)        1105-1
    Output compare prel...       Enable
    Fast Mode                    Disable
    CH Polarity                  High
```

Receiver

The receiver has the same behaviour of the UART; the IR receiver is connected to the PA10, so we have to set up the PA10 to USART1_RX and PA9 to USART1_TX.

Then set up the UART in this way enabling also the interrupt in the NVIC:



USART1 Mode and Configuration

Mode

Mode: Asynchronous

Hardware Flow Control (RS232): Disable

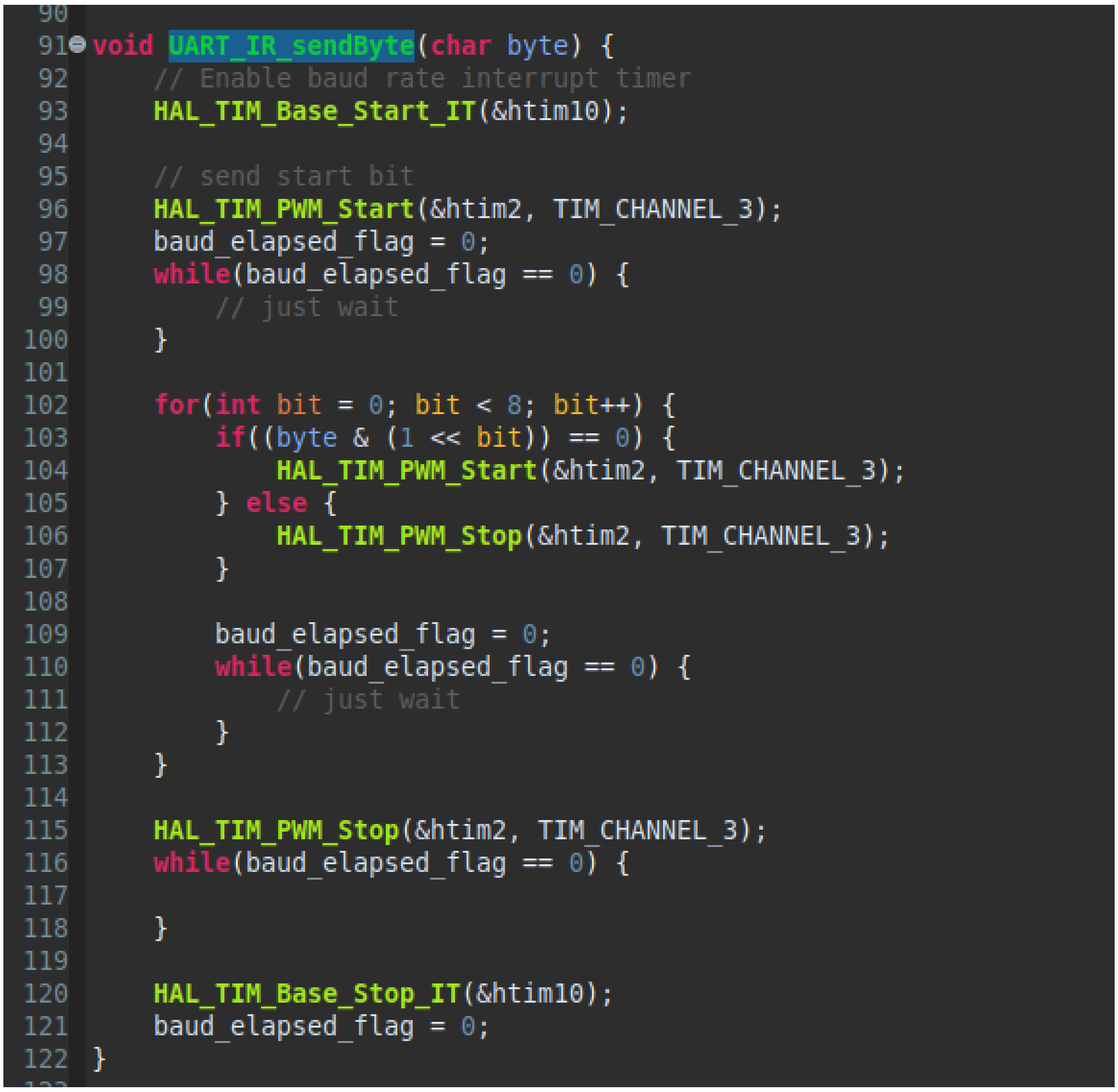
Configuration

Configure the below parameters :

Basic Parameters	
Baud Rate	2400 Bits/s
Word Length	8 Bits (including Parity)
Parity	None
Stop Bits	1
Advanced Parameters	
Data Direction	Receive Only
Over Sampling	16 Samples

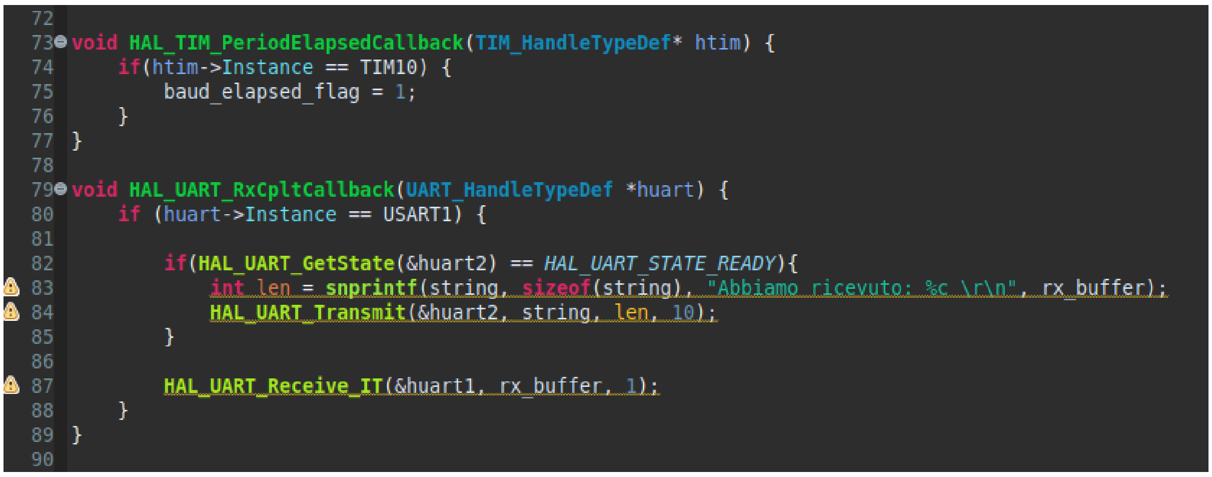
Software

To send a simple letter ‘A’,we initialize the receiver with *HAL_UART_Receive_IT(&huart1, &rx_buffer, 1)* then we put in the *while(1)* block a function called *UART_IR_sendByte(‘A’)* then we put an *HAL_Delay(1000)*, but this one can be substituted with a Timer.



```c
void UART_IR_sendByte(char byte) {
    // Enable baud rate interrupt timer
    HAL_TIM_Base_Start_IT(&htim10);

    // send start bit
    HAL_TIM_PWM_Start(&htim2, TIM_CHANNEL_3);
    baud_elapsed_flag = 0;
    while(baud_elapsed_flag == 0) {
        // just wait
    }

    for(int bit = 0; bit < 8; bit++) {
        if((byte & (1 << bit)) == 0) {
            HAL_TIM_PWM_Start(&htim2, TIM_CHANNEL_3);
        } else {
            HAL_TIM_PWM_Stop(&htim2, TIM_CHANNEL_3);
        }

        baud_elapsed_flag = 0;
        while(baud_elapsed_flag == 0) {
            // just wait
        }
    }

    HAL_TIM_PWM_Stop(&htim2, TIM_CHANNEL_3);
    while(baud_elapsed_flag == 0) {

    }

    HAL_TIM_Base_Stop_IT(&htim10);
    baud_elapsed_flag = 0;
}
```

The TIM10 is used to count the time, it provides an interrupt every 1 / 2400 seconds (2400 Hz), and in the callback the baud_elapsed_flag is set to 1.



```c
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef* htim) {
    if(htim->Instance == TIM10) {
        baud_elapsed_flag = 1;
    }
}

void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart) {
    if (huart->Instance == USART1) {

        if(HAL_UART_GetState(&huart2) == HAL_UART_STATE_READY){
            int len = snprintf(string, sizeof(string), "Abbiamo ricevuto: %c \r\n", rx_buffer);
            HAL_UART_Transmit(&huart2, string, len, 10);
        }

        HAL_UART_Receive_IT(&huart1, rx_buffer, 1);
    }
}
```

Moreover we need a callback of the receiver of the UART1 to receive the byte and manage it (in this case we send the byte to the remote terminal).