Mark	/11

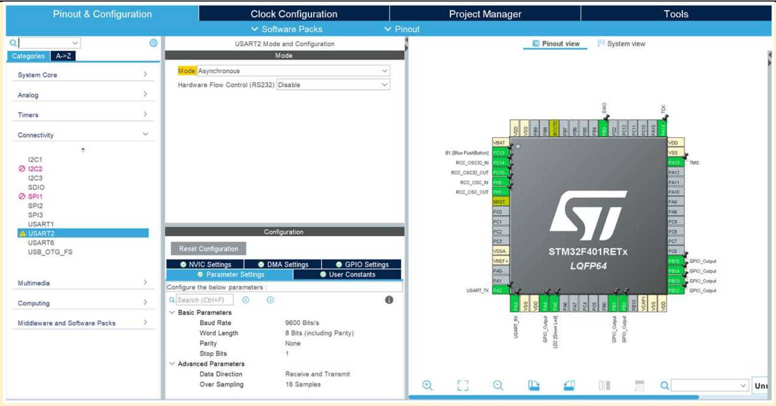
Team name:	*A14*		
Homework number:	*HOMEWORK 05*		
Due date:	19/10/25		
Contribution	NO	Partial	Full
Mattia Di Mauro			*x*
Francesca Biondi			*x*
Lorenzo Castelli			*x*
Pietro Albrigi			*x*
Notes: none			

Project name	HOMEWORK 5		
Not done	Partially done (major problems)	Partially done (minor problems)	Completed
			x

Part 1:

The aim of the project was: "Try sending from the PC via UART a string of variable length that is displayed on the LCD. Hint: send one letter per time. If you are curious, there are also smarter solutions which are not so immediate.".

To send and receive data between the PC and the microcontroller using the USART interface, we first verified that the TX and RX pins were correctly assigned.
USART2 was configured in the Pinout & Configuration tab under the Connectivity section.
In the Parameter Settings, the Baud Rate was set to 9600 bits/s, while the USART2 global interrupt was enabled in the NVIC Settings to allow asynchronous reception via interrupts. To control the LCD display, the pins PA4, PB1, PB2, PB12, PB13, PB14, and PB15 were configured as GPIO output pins, as shown below.


USART2 Mode and Configuration

Mode

Mode: Asynchronous

Hardware Flow Control (RS232): Disable

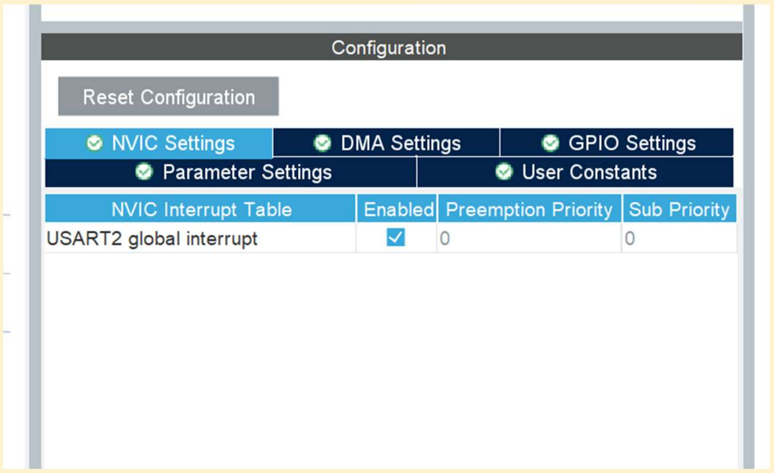
Configuration

Basic Parameters

Baud Rate	9600 Bits/s
Word Length	8 Bits (including Parity)
Parity	None
Stop Bits	1

Advanced Parameters

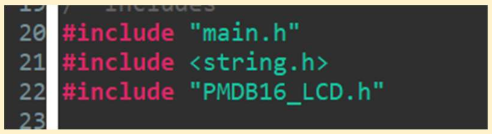
Data Direction	Receive and Transmit
Over Sampling	16 Samples


Configuration

NVIC Interrupt Table	Enabled	Preemption Priority	Sub Priority
USART2 global interrupt	✓	0	0

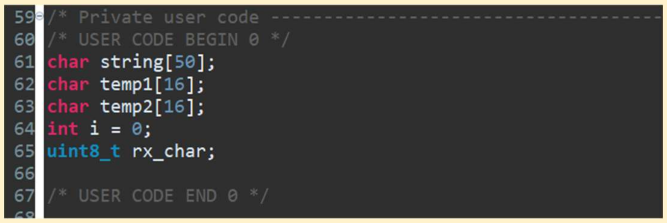
After this setup, the initialization code was generated.

For the LCD management, the "PMDB16_LCD" library was imported into the STM32CubeIDE project.
The source file PMDB16_LCD.c was placed in the Core/Src directory, while the header file PMDB16_LCD.h was added to Core/Inc.
In the header section of main.c, the following include directives were added:



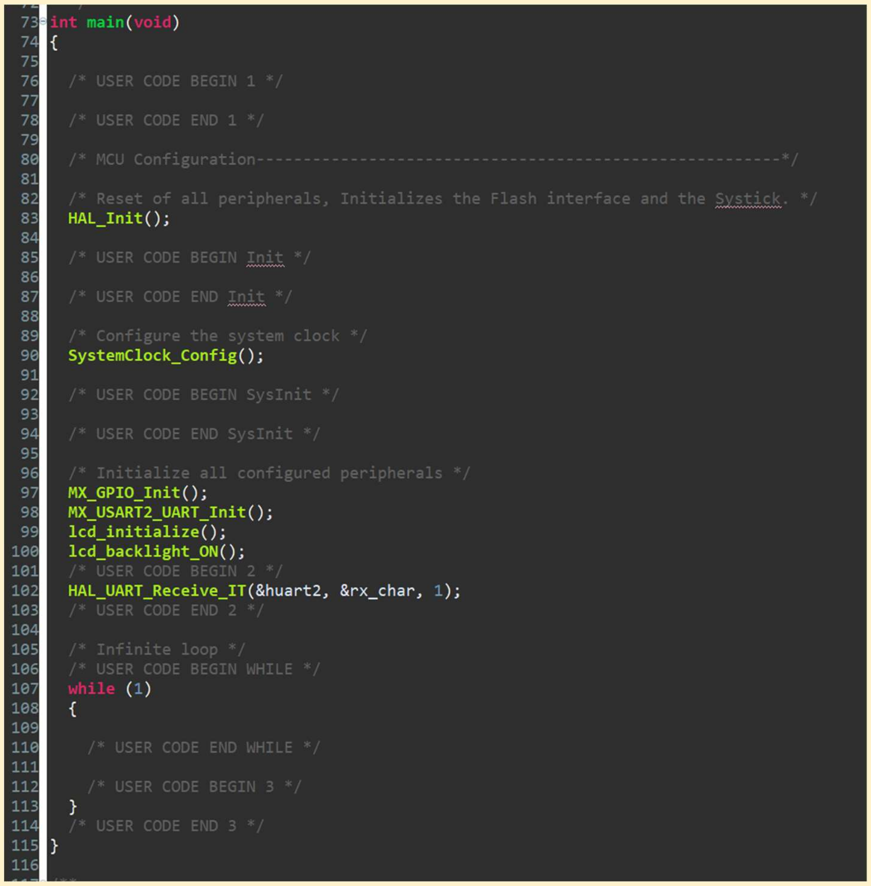
```c
#include "main.h"
#include <string.h>
#include "PMDB16_LCD.h"
```

A global character buffer was defined as char string[50], along with two temporary buffers used to handle strings longer than 16 characters (the width of one LCD line).
The variable rx_char stores each received character from the UART interface.



```c
/* Private user code ---------------------------------------------------------*/
/* USER CODE BEGIN 0 */
char string[50];
char temp1[16];
char temp2[16];
int i = 0;
uint8_t rx_char;

/* USER CODE END 0 */
```

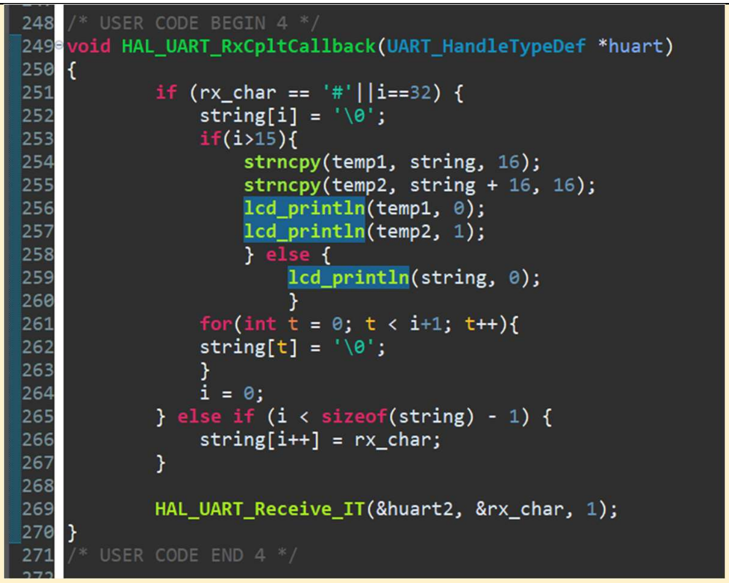
We inserted the initialization calls for the LCD controller and for enabling the display backlight, then the reception via UART interrupt was then enabled with HAL_UART_Receive_IT(&huart2, &rx_char, 1);



```c
int main(void)
{

  /* USER CODE BEGIN 1 */

  /* USER CODE END 1 */

  /* MCU Configuration--------------------------------------------------------*/

  /* Reset of all peripherals, Initializes the Flash interface and the Systick. */
  HAL_Init();

  /* USER CODE BEGIN Init */

  /* USER CODE END Init */

  /* Configure the system clock */
  SystemClock_Config();

  /* USER CODE BEGIN SysInit */

  /* USER CODE END SysInit */

  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  MX_USART2_UART_Init();
  lcd_initialize();
  lcd_backlight_ON();
  /* USER CODE BEGIN 2 */
  HAL_UART_Receive_IT(&huart2, &rx_char, 1);
  /* USER CODE END 2 */

  /* Infinite loop */
  /* USER CODE BEGIN WHILE */
  while (1)
  {

    /* USER CODE END WHILE */

    /* USER CODE BEGIN 3 */
  }
  /* USER CODE END 3 */
}
```

This function allows the microcontroller to receive one character at a time asynchronously.

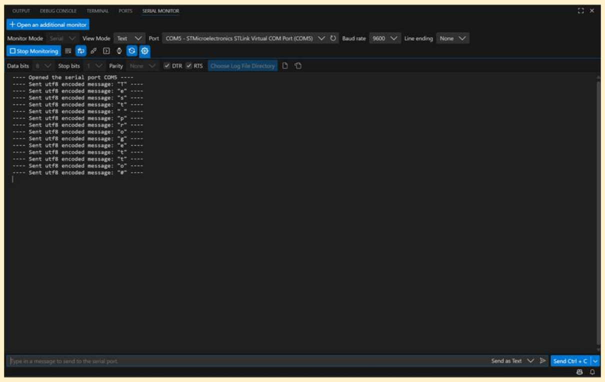
In the callback function HAL_UART_RxCpltCallback(), each received character is appended to the buffer string[i++].
When the user sends the special character '#' or when more than 32 characters are received, the LCD prints the entire string.
If the string length exceeds 16 characters, it is split into two parts: temp1 for the first 16 characters and temp2 for the following 16 characters. These are then displayed on the first and second lines of the LCD.



```c
/* USER CODE BEGIN 4 */
void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart)
{
        if (rx_char == '#'||i==32) {
            string[i] = '\0';
            if(i>15){
                strncpy(temp1, string, 16);
                strncpy(temp2, string + 16, 16);
                lcd_println(temp1, 0);
                lcd_println(temp2, 1);
                } else {
                    lcd_println(string, 0);
                    }
            for(int t = 0; t < i+1; t++){
            string[t] = '\0';
            }
            i = 0;
        } else if (i < sizeof(string) - 1) {
            string[i++] = rx_char;
        }

        HAL_UART_Receive_IT(&huart2, &rx_char, 1);
}
/* USER CODE END 4 */
```

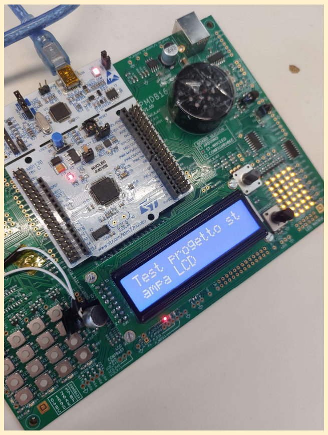
At the end of the callback, the instruction HAL_UART_Receive_IT(&huart2, &rx_char, 1) is called again to re-enable the interrupt and allow continuous character reception.

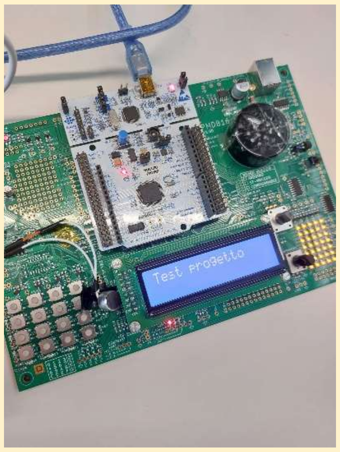
Finally, we used Visual Studio Code with the Serial Monitor extension to send data to the STM32 board. The correct serial port associated with the board was selected, and the baud rate was set to 9600 bits/s, matching the USART2 configuration.
The successful reception and display of the transmitted string on the LCD confirmed that our code was functioning as intended.



Monitor Mode: Serial · View Mode: Text · Port: COM5 - STMicroelectronics STLink Virtual COM Port (COM5) · Baud rate: 9600 · Line ending: None
Data bits: 8 · Stop bits: 1 · Parity: None · DTR ✓ · RTS ✓

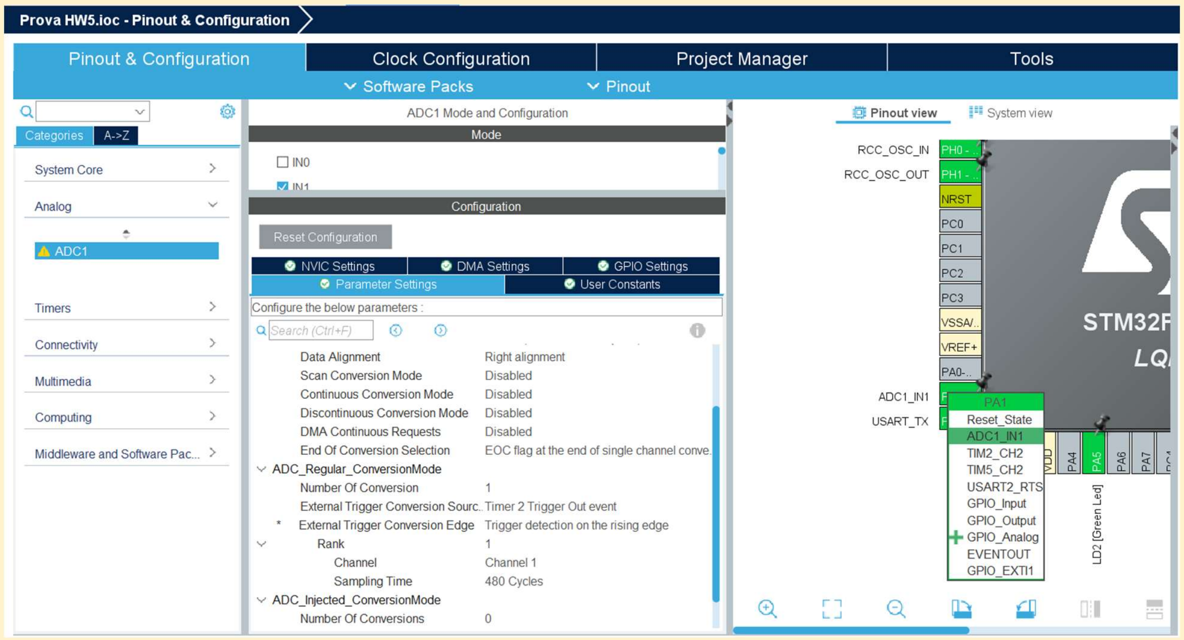
```
---- Opened the serial port COM5 ----
---- Sent utf8 encoded message: "T" ----
---- Sent utf8 encoded message: "e" ----
---- Sent utf8 encoded message: "s" ----
---- Sent utf8 encoded message: "t" ----
---- Sent utf8 encoded message: " " ----
---- Sent utf8 encoded message: "p" ----
---- Sent utf8 encoded message: "r" ----
---- Sent utf8 encoded message: "o" ----
---- Sent utf8 encoded message: "g" ----
---- Sent utf8 encoded message: "e" ----
---- Sent utf8 encoded message: "t" ----
---- Sent utf8 encoded message: "t" ----
---- Sent utf8 encoded message: "o" ----
---- Sent utf8 encoded message: "#" ----
```





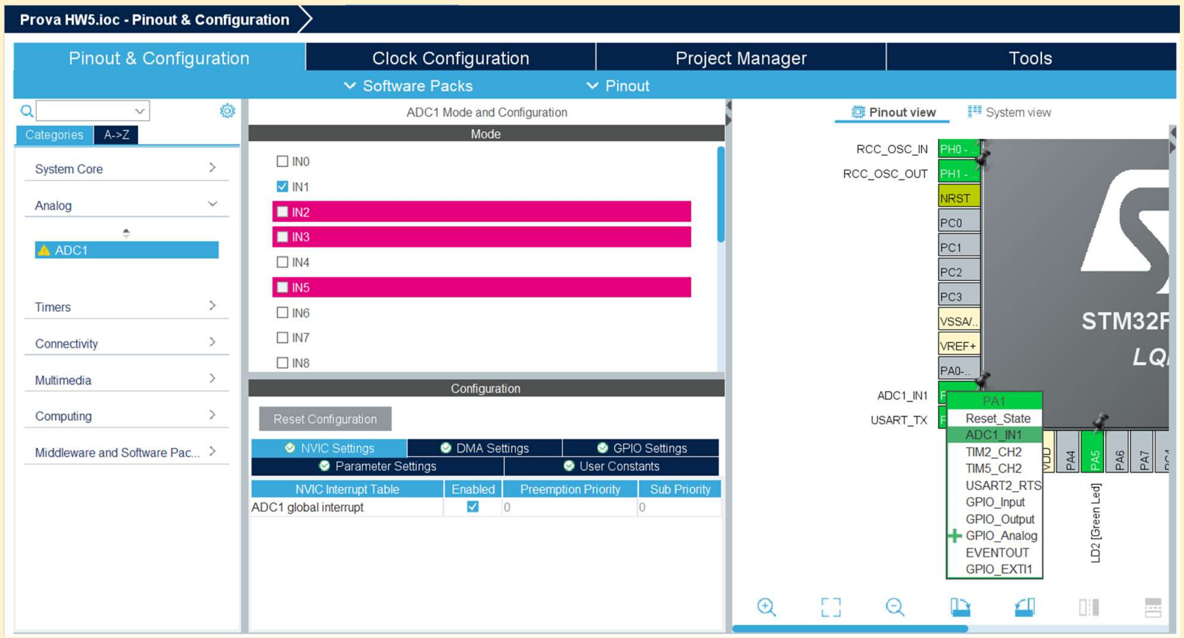
Part 2: Project 2b

First, we identified the PA1 pin as the one connected to the potentiometer, by examining the Hands-on Lab Schematics file. In the .ioc configuration, we set PA1 as ADC1_IN1, enabling it as the analog input channel for the potentiometer. We then configured the ADC with a sampling time of 480 cycles.
After that, we set the ADC to be triggered by Timer 2, by configuring the external trigger conversion source as Timer 2 Trigger Out Event (TRGO), with trigger detection on the rising edge:



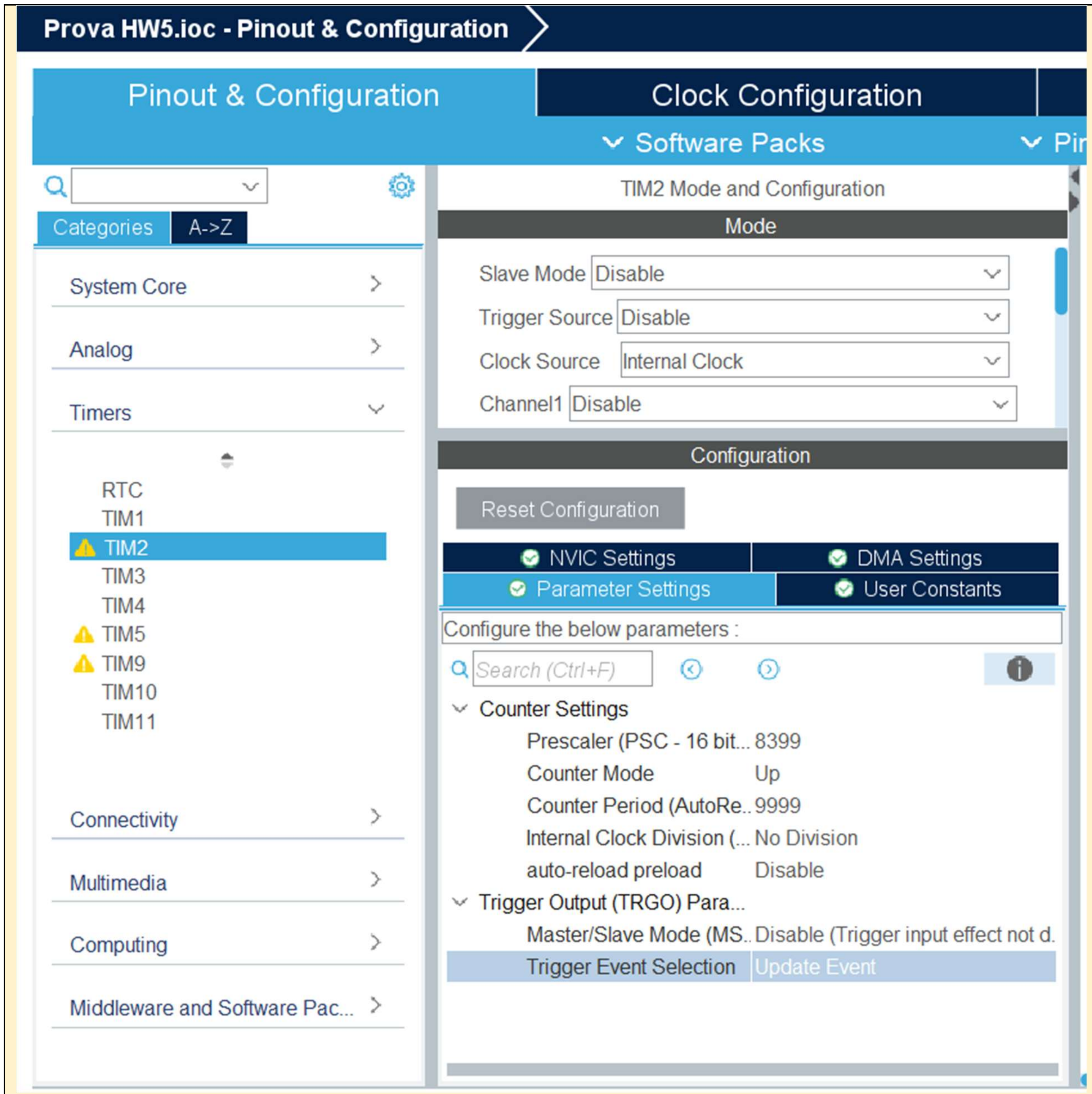
Data Alignment	Right alignment
Scan Conversion Mode	Disabled
Continuous Conversion Mode	Disabled
Discontinuous Conversion Mode	Disabled
DMA Continuous Requests	Disabled
End Of Conversion Selection	EOC flag at the end of single channel conve
ADC_Regular_ConversionMode	
Number Of Conversion	1
External Trigger Conversion Sourc...	Timer 2 Trigger Out event
External Trigger Conversion Edge	Trigger detection on the rising edge
Rank	1
Channel	Channel 1
Sampling Time	480 Cycles
ADC_Injected_ConversionMode	
Number Of Conversions	0

The ADC global interrupt was enabled to allow the converter to operate in interrupt mode, triggering a callback once each conversion is complete:



NVIC Interrupt Table	Enabled	Preemption Priority	Sub Priority
ADC1 global interrupt	✓	0	0

Timer 2 was configured to use the internal clock source, with a prescaler value of 8399 and an ARR value of 9999. The trigger event selection was set to Update Event:


Prova HW5.ioc - Pinout & Configuration

Pinout & Configuration | Clock Configuration

Software Packs | Pi

Categories | A->Z

System Core
Analog
Timers
RTC
TIM1
TIM2
TIM3
TIM4
TIM5
TIM9
TIM10
TIM11
Connectivity
Multimedia
Computing
Middleware and Software Pac...

TIM2 Mode and Configuration

Mode

Slave Mode Disable
Trigger Source Disable
Clock Source Internal Clock
Channel1 Disable

Configuration

Reset Configuration

NVIC Settings | DMA Settings | Parameter Settings | User Constants

Configure the below parameters :

Search (Ctrl+F)

Counter Settings

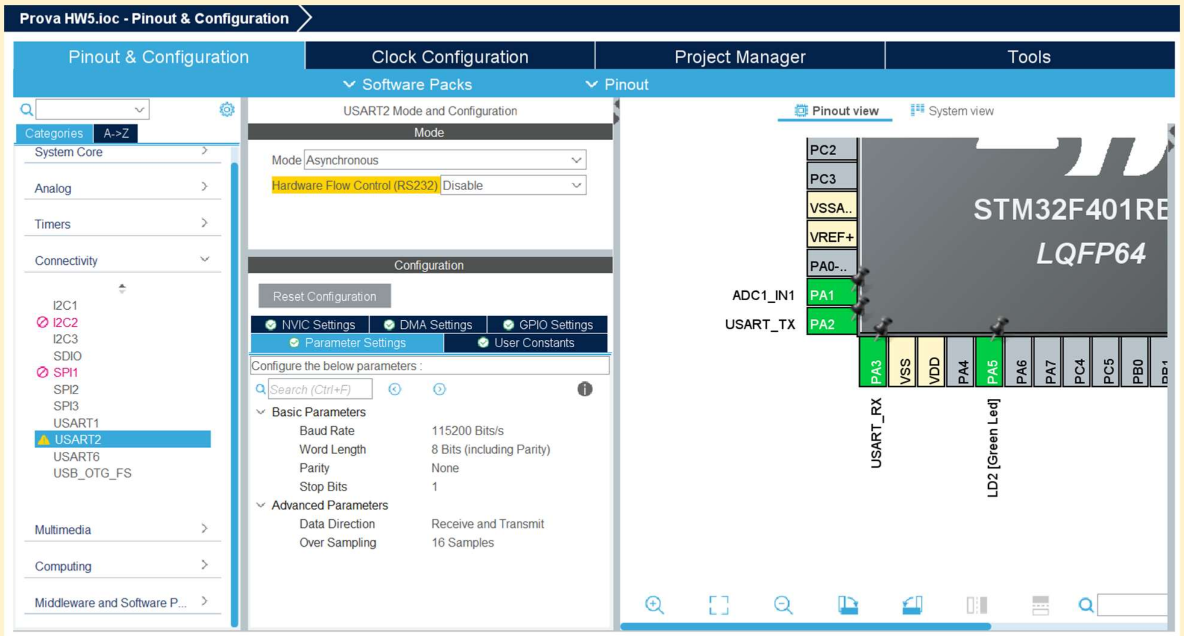
Prescaler (PSC - 16 bit...	8399
Counter Mode	Up
Counter Period (AutoRe..	9999
Internal Clock Division (...	No Division
auto-reload preload	Disable

Trigger Output (TRGO) Para...

Master/Slave Mode (MS..	Disable (Trigger input effect not d.
Trigger Event Selection	Update Event

As a result of this configuration, Timer 2 generates a TRGO signal each time an update event occurs, so everytime the timer counter overflows and resets. This TRGO signal serves as the trigger for a new ADC conversion. With the chosen prescaler and ARR values, the timer overflows once every 1 second, leading to a regular conversion rate of 1 Hz.

We also verified that the USART pins were correctly configured and set the baud rate to 115200 bps:



USART2 Mode and Configuration

Mode

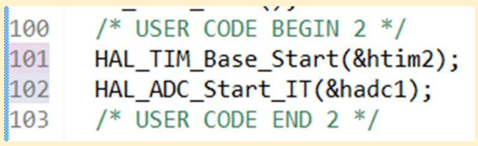
Mode: Asynchronous
Hardware Flow Control (RS232): Disable

Configuration

Configure the below parameters :

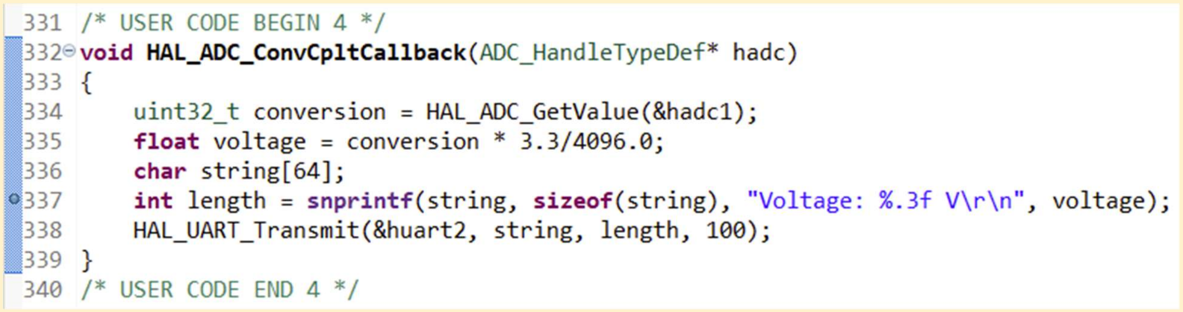
Basic Parameters	
Baud Rate	115200 Bits/s
Word Length	8 Bits (including Parity)
Parity	None
Stop Bits	1
Advanced Parameters	
Data Direction	Receive and Transmit
Over Sampling	16 Samples

In the main function, we called HAL_TIM_Base_Start() and HAL_ADC_Start_IT() to start the timer and the ADC (in interrupt mode), respectively.



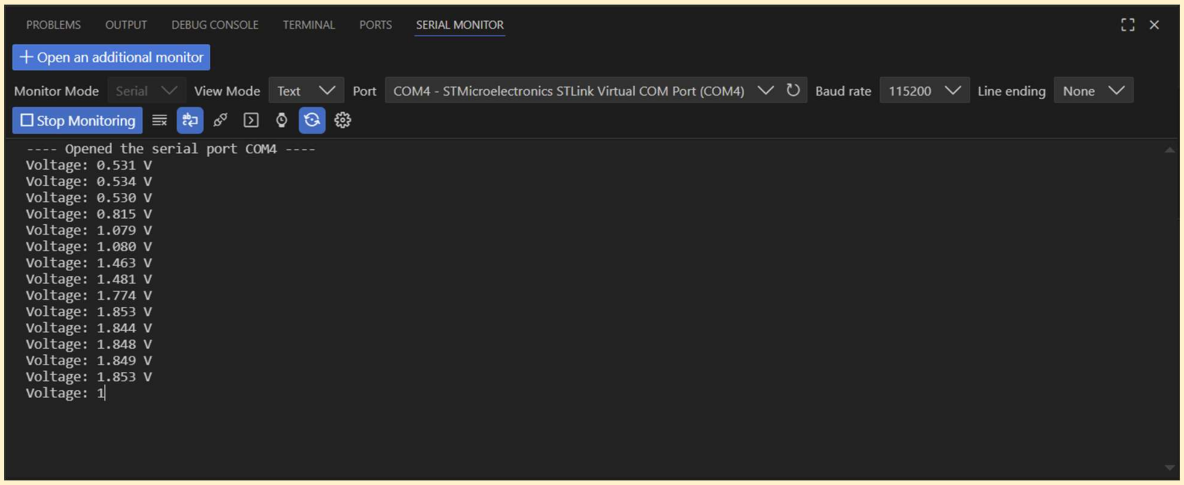
```c
  /* USER CODE BEGIN 2 */
  HAL_TIM_Base_Start(&htim2);
  HAL_ADC_Start_IT(&hadc1);
  /* USER CODE END 2 */
```

Outside the main function, a callback function is executed each time the ADC completes a conversion. This function converts the ADC reading into a voltage and transmits it to an external serial monitor.



```c
/* USER CODE BEGIN 4 */
void HAL_ADC_ConvCpltCallback(ADC_HandleTypeDef* hadc)
{
    uint32_t conversion = HAL_ADC_GetValue(&hadc1);
    float voltage = conversion * 3.3/4096.0;
    char string[64];
    int length = snprintf(string, sizeof(string), "Voltage: %.3f V\r\n", voltage);
    HAL_UART_Transmit(&huart2, string, length, 100);
}
/* USER CODE END 4 */
```

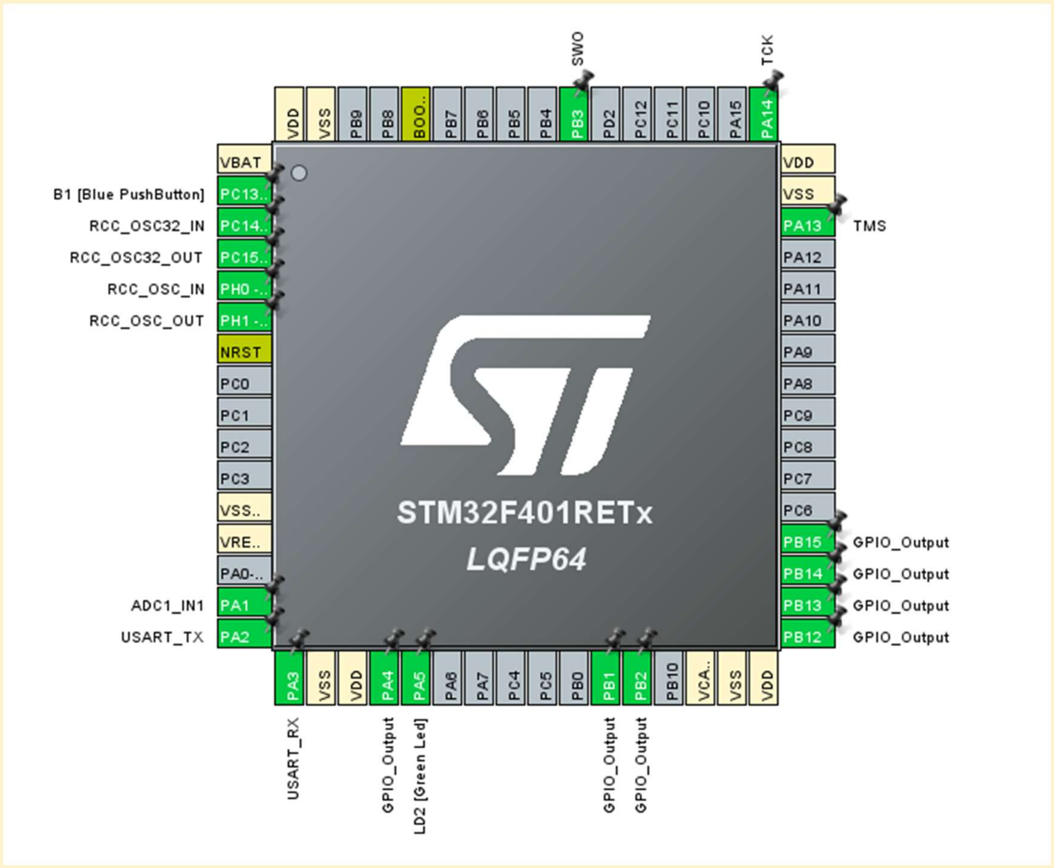
The code was then tested on the board and verified to work as intended, displaying the potentiometer voltage on the serial monitor once every second, using the same baud rate and COM port as previously configured:



Monitor Mode Serial · View Mode Text · Port COM4 - STMicroelectronics STLink Virtual COM Port (COM4) · Baud rate 115200 · Line ending None

```
---- Opened the serial port COM4 ----
Voltage: 0.531 V
Voltage: 0.534 V
Voltage: 0.530 V
Voltage: 0.815 V
Voltage: 1.079 V
Voltage: 1.080 V
Voltage: 1.463 V
Voltage: 1.481 V
Voltage: 1.774 V
Voltage: 1.853 V
Voltage: 1.844 V
Voltage: 1.848 V
Voltage: 1.849 V
Voltage: 1.853 V
Voltage: 1
```

Project 2c:
To display the potentiometer values on the LCD, we kept the same configuration as in the previous project. In addition, we configured the pins PA4, PB1, PB2, PB12, PB13, PB14, and PB15, which are required for controlling the LCD display, as GPIO output pins.



After importing and including the “PMDB16_LCD” library, we added the initialization functions for the LCD controller and to enable the display backlight in the main function:

```c
/* USER CODE BEGIN 2 */
lcd_initialize();
lcd_backlight_ON();
HAL_TIM_Base_Start(&htim2);
HAL_ADC_Start_IT(&hadc1);
/* USER CODE END 2 */
```

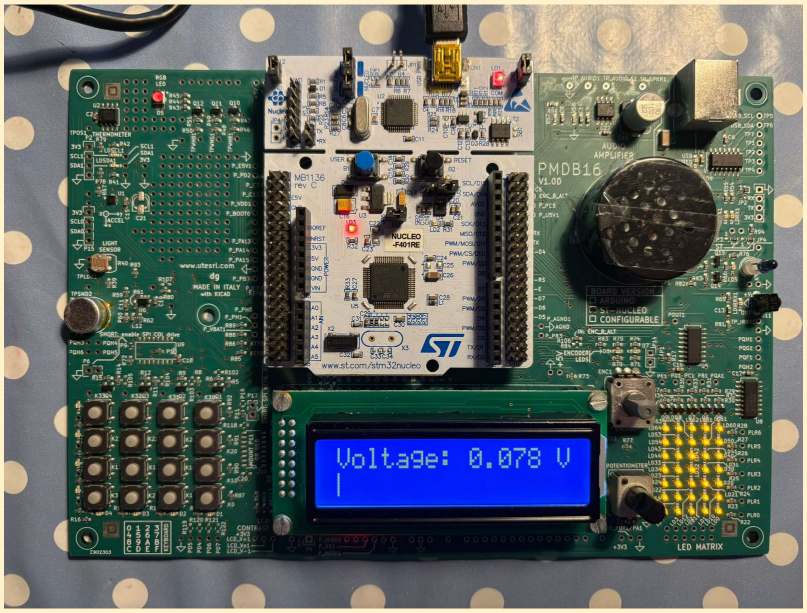
Then we modified the ADC's callback function as follows:

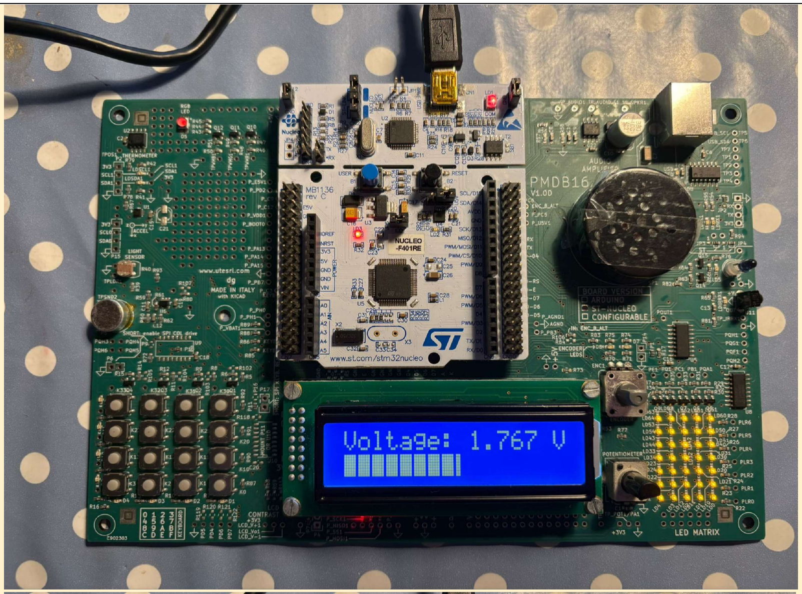
```c
/* USER CODE BEGIN 4 */
void HAL_ADC_ConvCpltCallback(ADC_HandleTypeDef* hadc)
{
    uint32_t conversion = HAL_ADC_GetValue(&hadc1);
    float voltage = conversion * 3.3/4096.0;
    char string[17];
    snprintf(string, sizeof(string), "Voltage: %.3f V", voltage);
    lcd_println(string, 0);
    lcd_drawBar(voltage*80/3.3);
}
/* USER CODE END 4 */
```

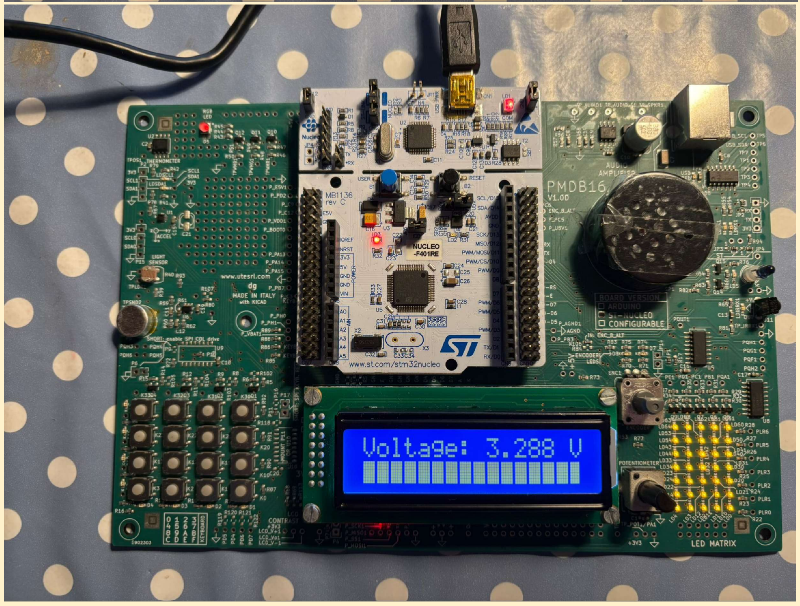
The top row shows the voltage as a numeric value. Since each row of the display can hold up to 16 characters, we defined the buffer string slightly larger to account for the null terminator character.

The bottom row displays a bar graph to represent the voltage. The bar graph has a range from 0 to 80, so the measured voltage (from 0 to 3.3 V) is scaled accordingly to fit this range.

The code was tested on the board and was confirmed to work correctly, displaying both the numeric voltage and the bar graph on the LCD as expected:







Professor comments: