WRIGHT STATE UNIVERSITY

LAB-3

Understanding Interrupts and Different ADC Modes

CEG 7360 Embedded Systems

Name : - Keerthi Patil
UID : - U01110229
Email : - patil.115@wright.edu

Name : - Dhamini Vootkuri
UID : - U01111434
Email : - vootkuri.3@wright.edu

Lab 3

STM32 Configuration and Programming

(Understanding Interrupts and Different ADC Modes)

Introduction:

This report outlines a series of experiments aimed at enhancing the understanding of interrupts and Analog-to-Digital Converter (ADC) configurations using the STM32 microcontroller.

The first experiment involved setting up an external interrupt on pin PC13, renamed pwBtn , which toggled an LED (LD2) when the button was pressed. This application illustrated how external events can prompt specific actions in the microcontroller, demonstrating the effectiveness of interrupt-driven programming compared to traditional polling techniques.

The following experiments delved into various ADC functionalities. The second experiment configured multiple ADC channels (A0, A1, A2) in scan mode to facilitate simultaneous readings, with results sent via UART. The third experiment incorporated Direct Memory Access (DMA) for improved data handling, where an interrupt was triggered once the ADC buffer was filled. Lastly, the fourth experiment explored continuous mode with DMA, enabling continuous data acquisition and highlighting the microcontroller's real-time capabilities. Collectively, these experiments provided valuable insights into interrupt management and ADC applications in embedded systems.

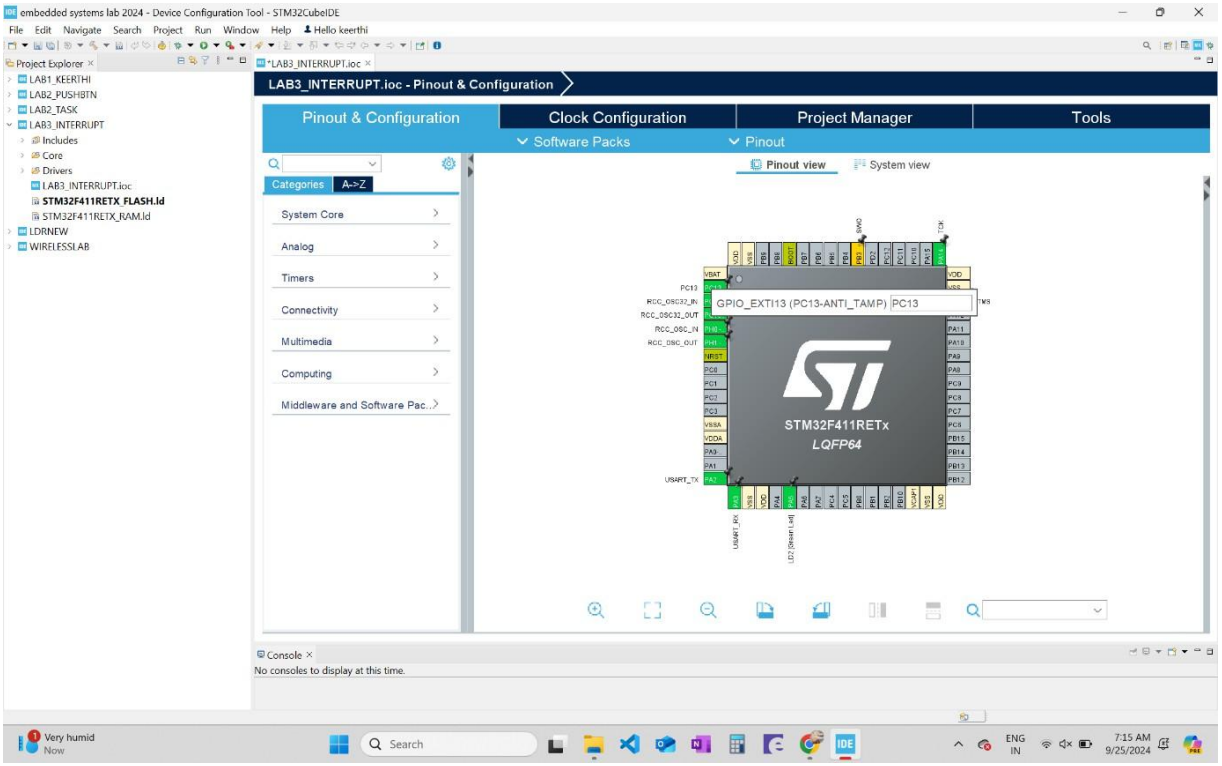
Materials:

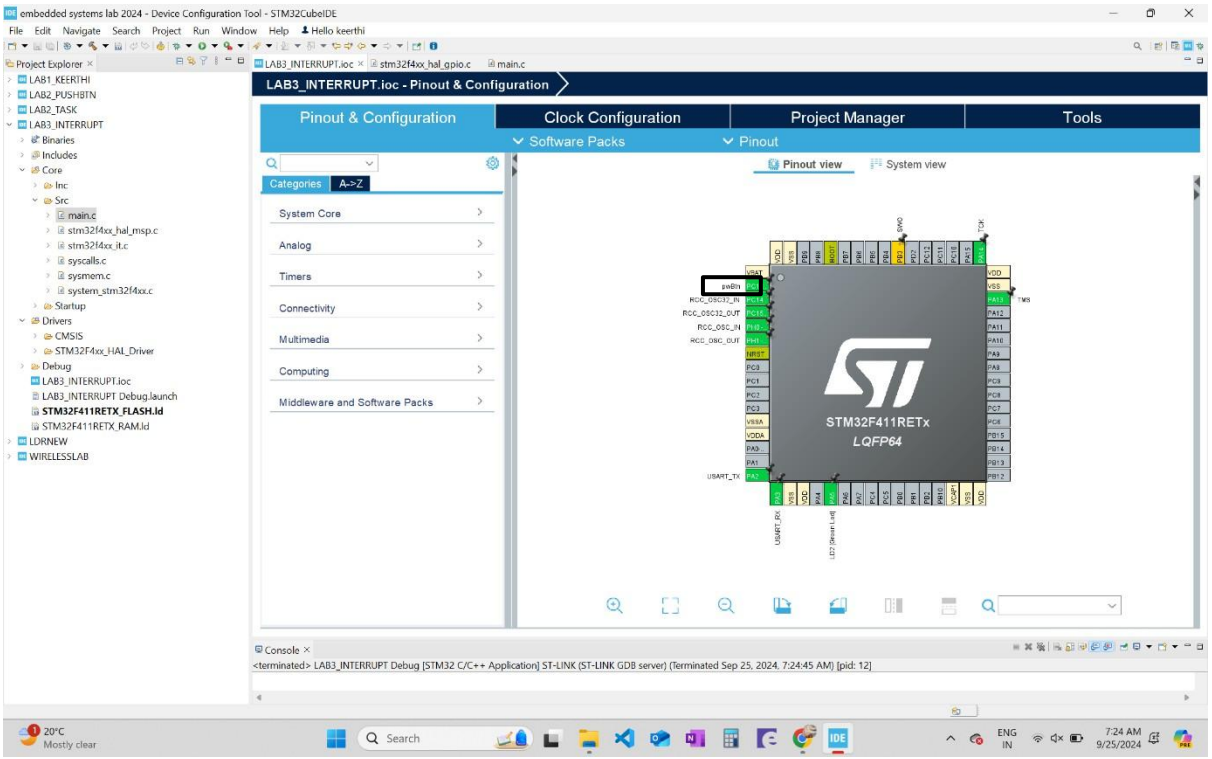
- Software tools: STM32CubeIDE
- Hardware tools: STM32 Nucleo Board, 3 resistors, 3 LDR’s (Light-dependent resistor) , Breadboard, jumper wires, USB Cable.

Procedure:

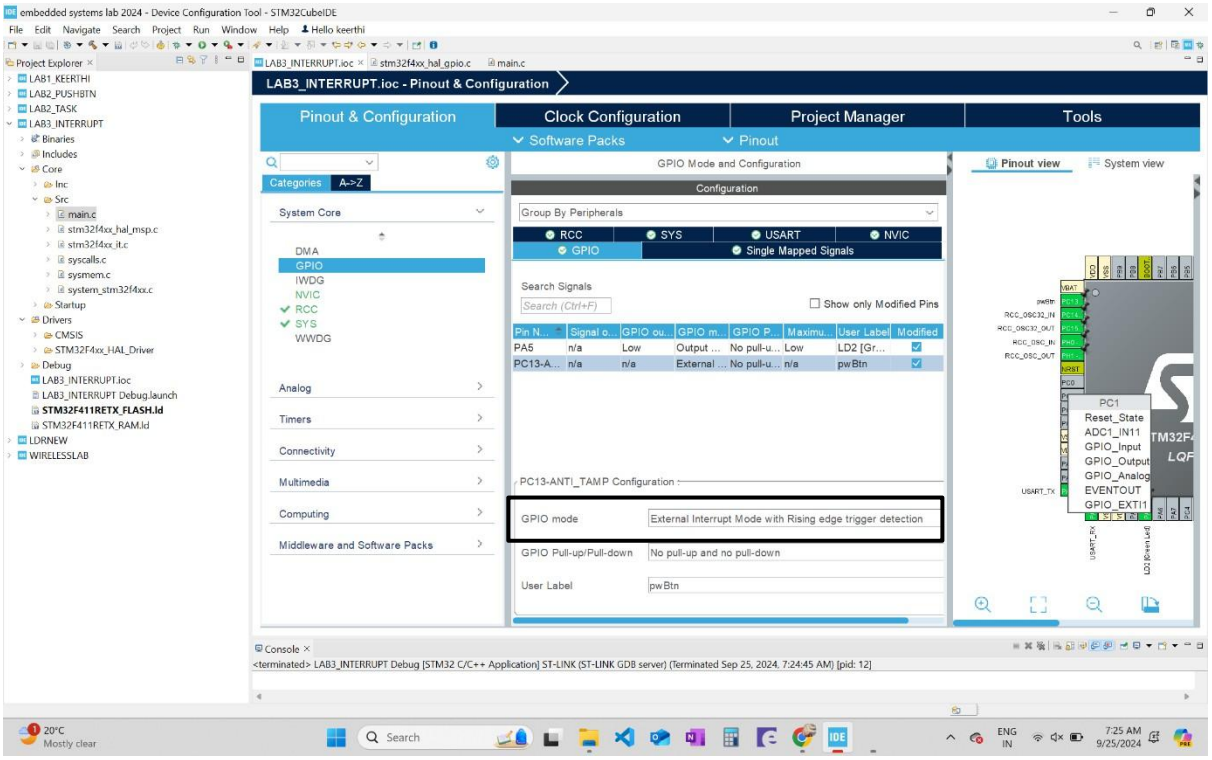
Part 1. GPIO External Interrupt (EXTI) Configuration

- To configure the GPIO External Interrupt, I have set the **PC13** pin to **GPIO_EXTI13** and named it as **pwBtn.**

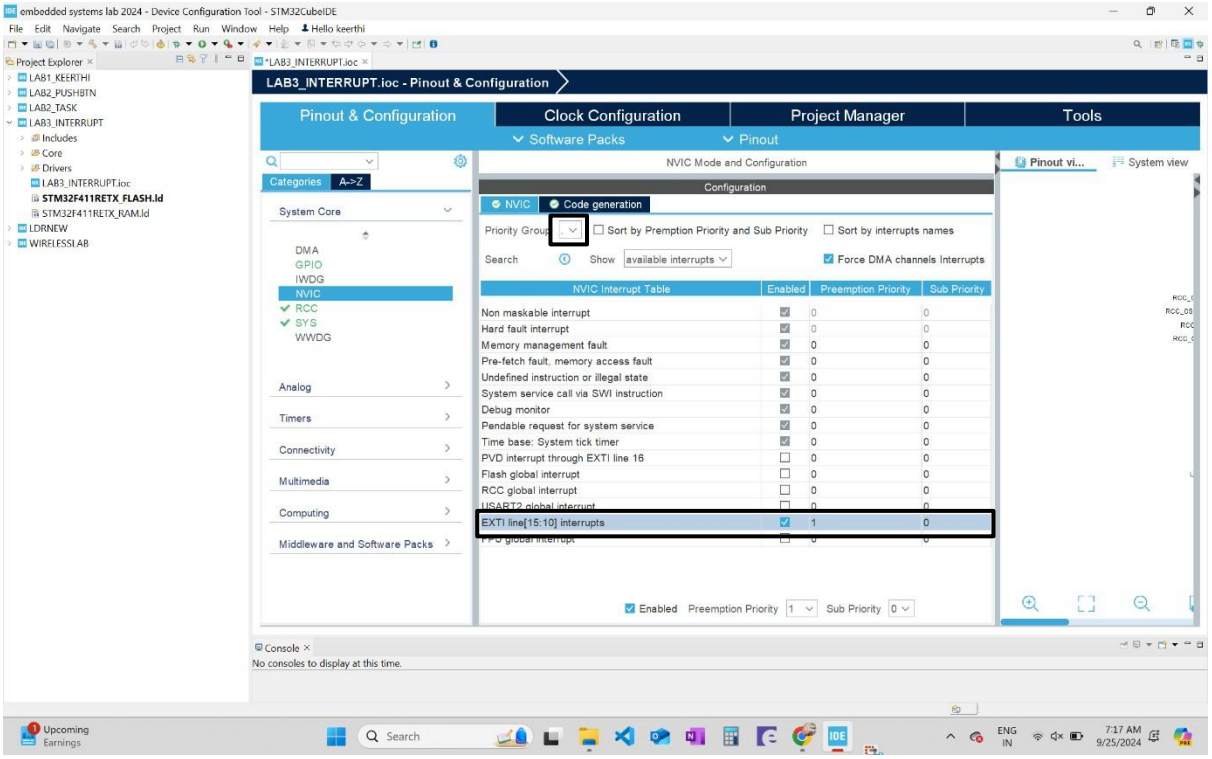




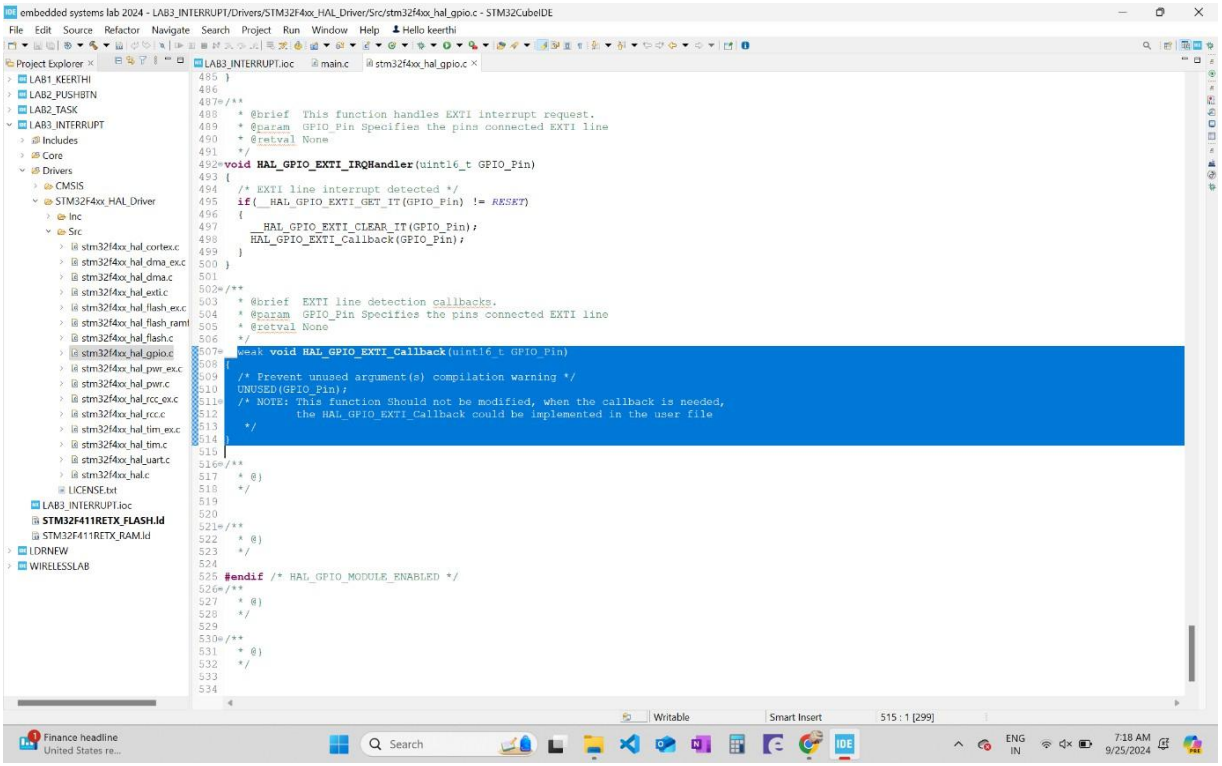
- In **System Core**, Under **GPIO** I have set the Pin **PC13** mode to be **Rising edge trigger detection**.



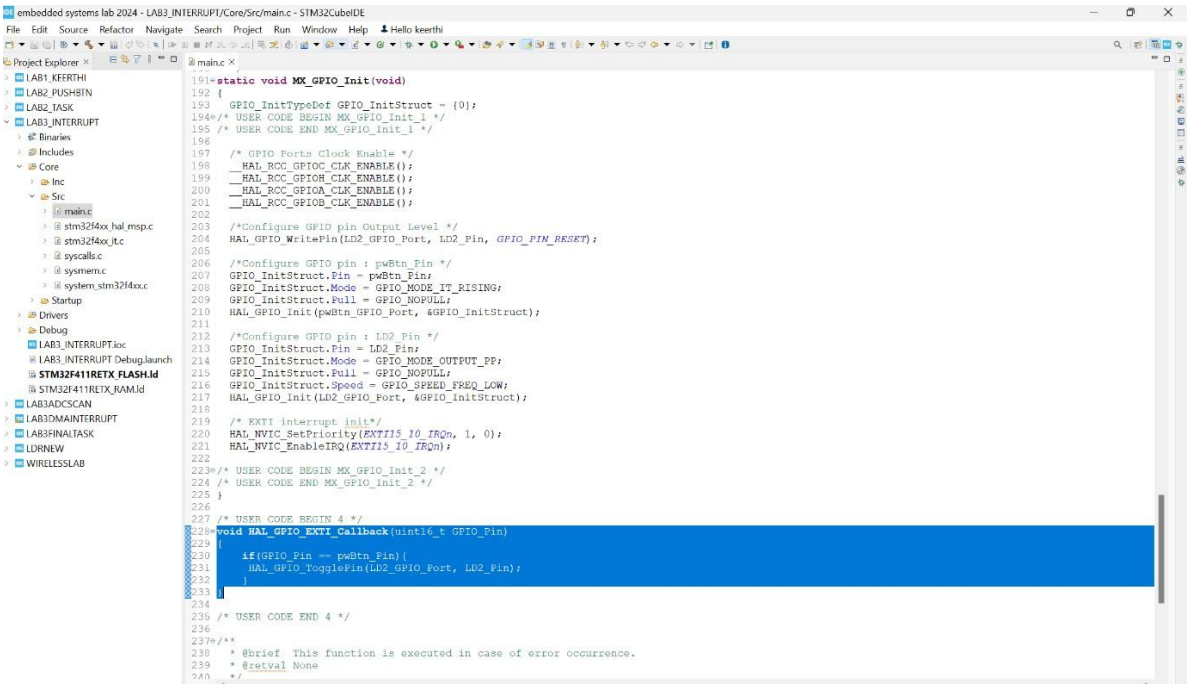
- In **System Core**, Under **NVIC** Set the **priority group** as **4- bit.**
- After selecting EXTI line[15:10] interrupts, I have Set the Preemption priority to 1.



- After saving, Under **Drivers** in **STM32F4xx_HAL_Driver** in **stm32f4xx_hal_gpio.c** I have copied the code from the line 507-514 as highlighted below to program the interrupt.

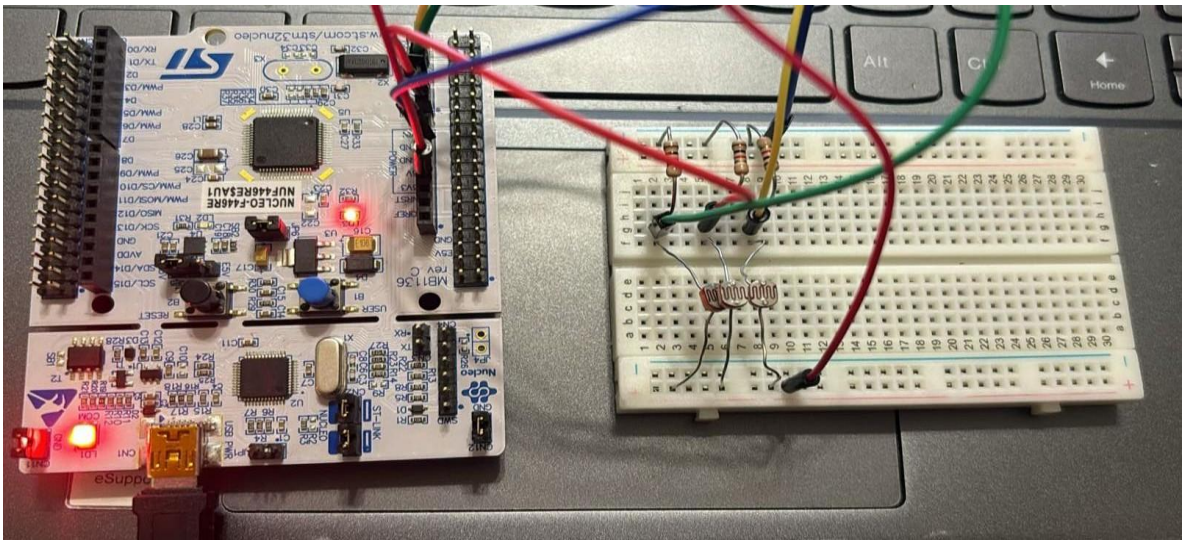


- The copied code from **stm32f4xx_hal_gpio.c** is pasted in **main.c** in the line 228 under USER CODE BEGIN 4 and I have made few changes and added a code in while loop.



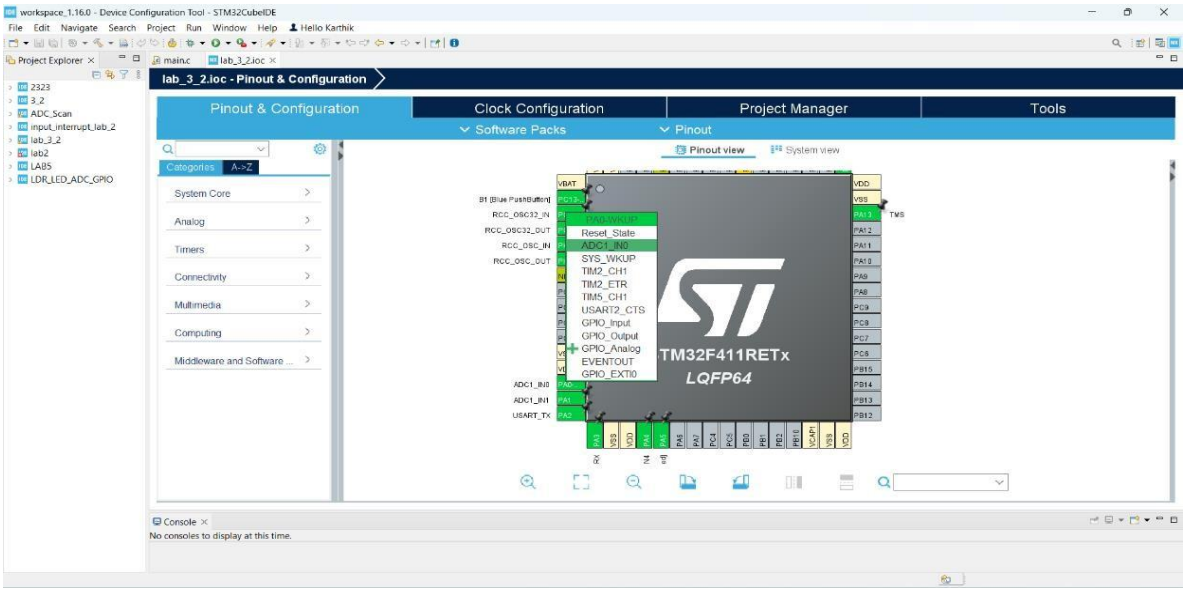
Experimental Setup:

- Here the board is directly connected to the system using a USB cable and the blue switch on the board acts as push button.

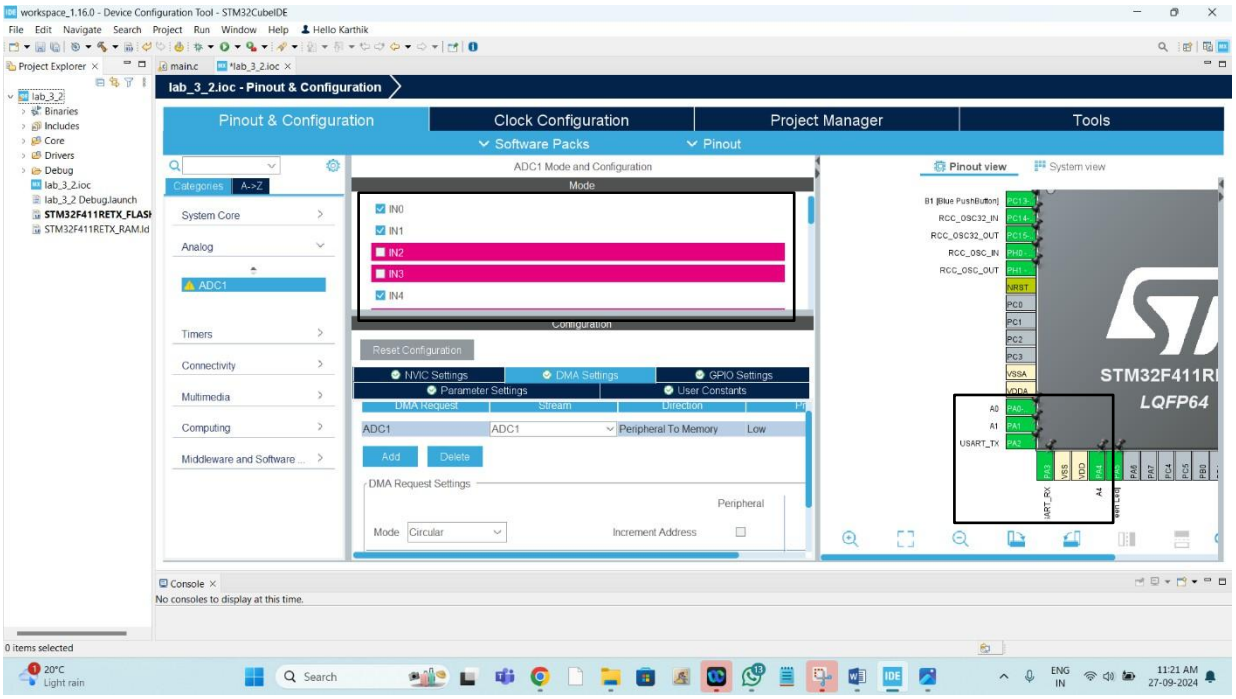


Part 2. ADC Scan Mode Configuration

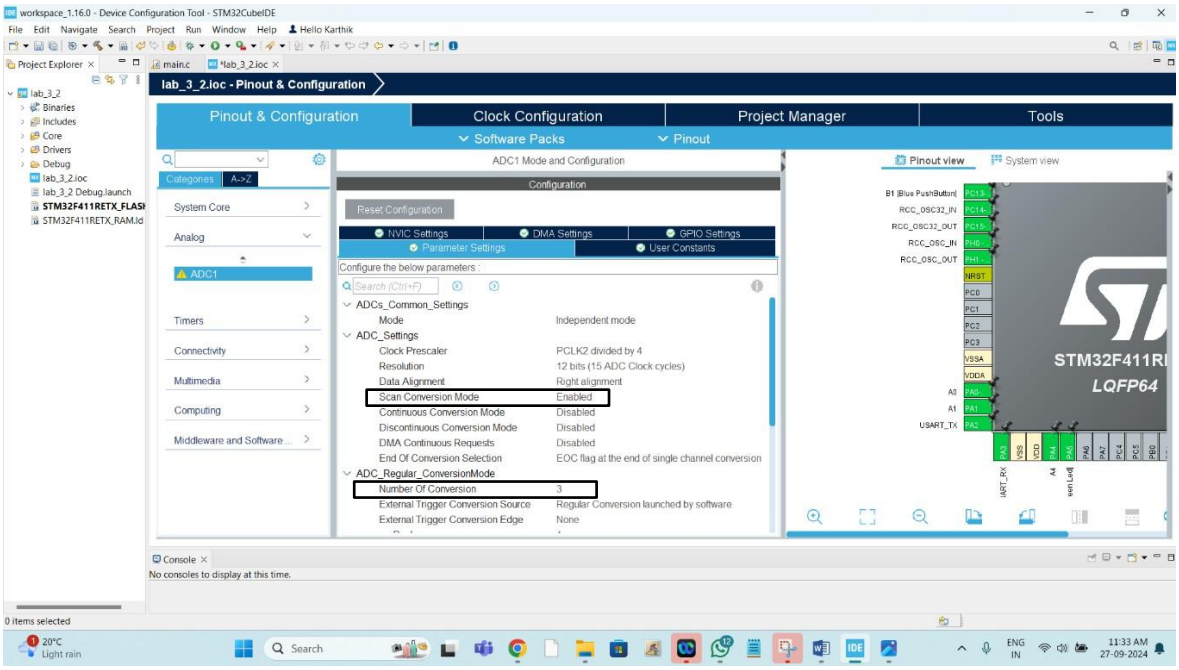
- The aim of the part 2 is to create 3 channels which can read the analog data and send it to the micro controller
- For this operation we have selected **PA0, PA1 and PA4 as ADC1_IN0, ADC1_IN1 and ADC1_IN2**



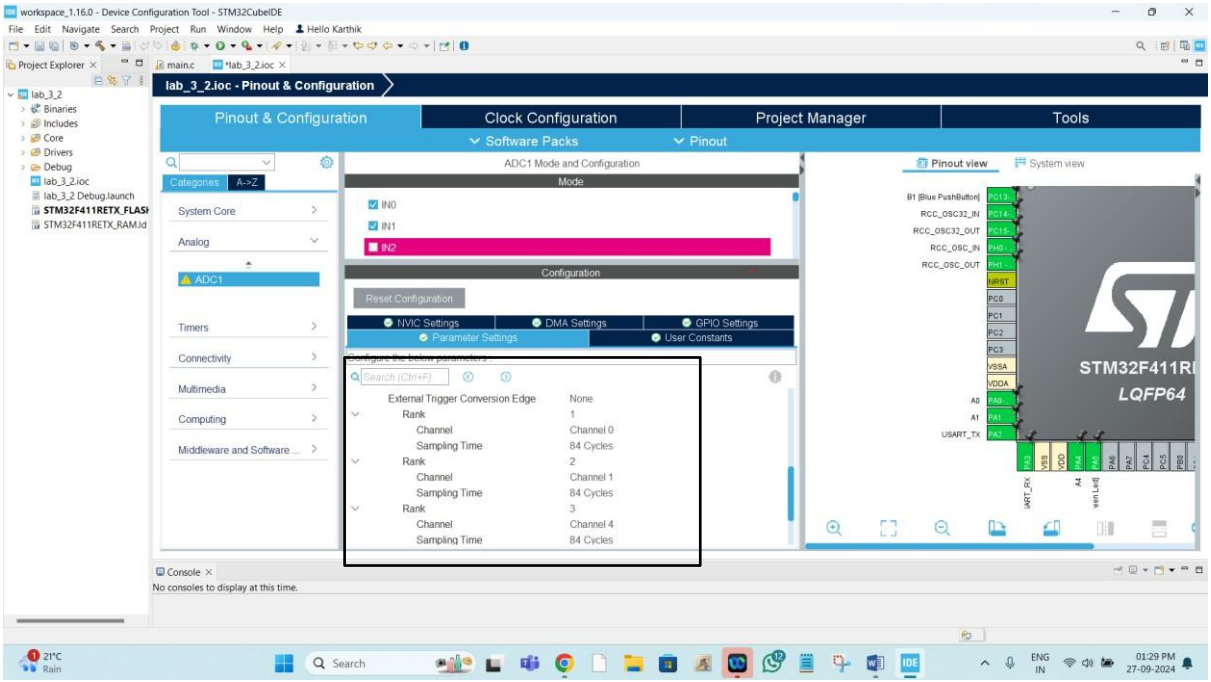
- Rename each ADC converter pin as A0, A1 and A4
- Select ADC1 mode and configuration as IN0, IN1 and IN4



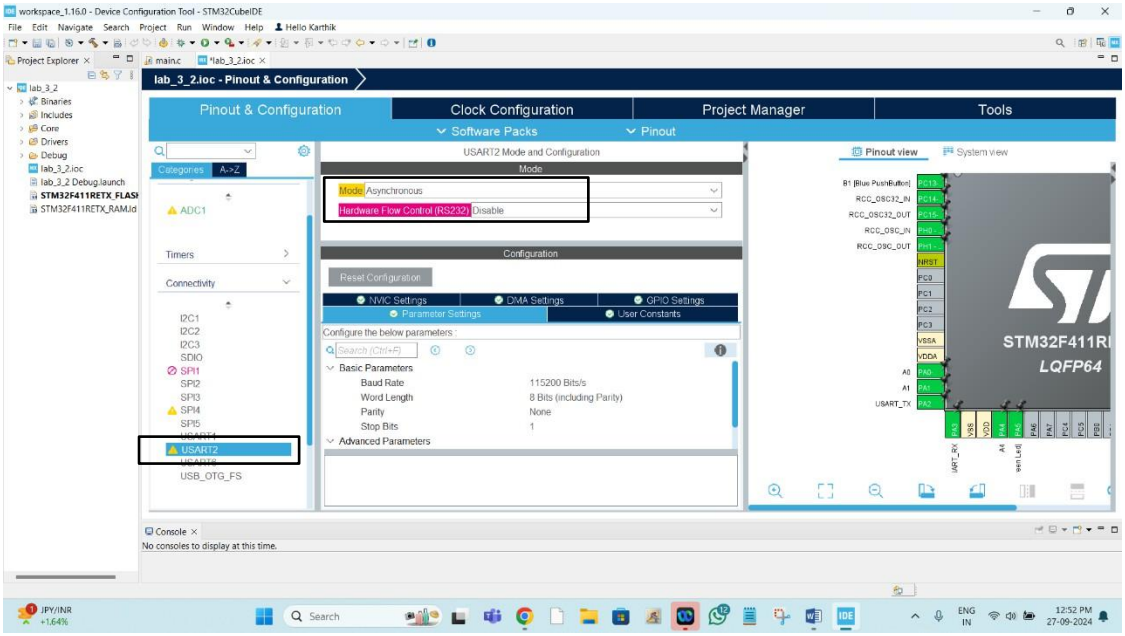
In system core under Analog in **ADC1** , I have **Enabled** the **scan conversion mode** and increased the **Number of conversions** to **3**



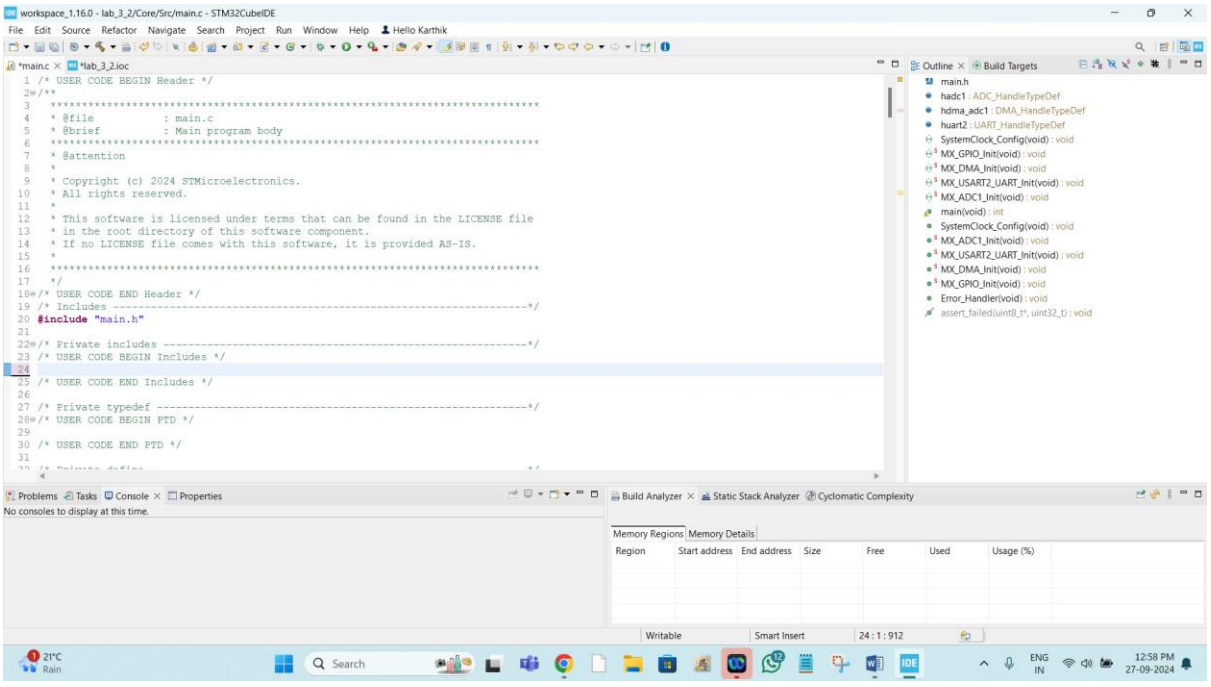
- I have made channel 0, channel 1 and channel 4 to **84 Cycles**.



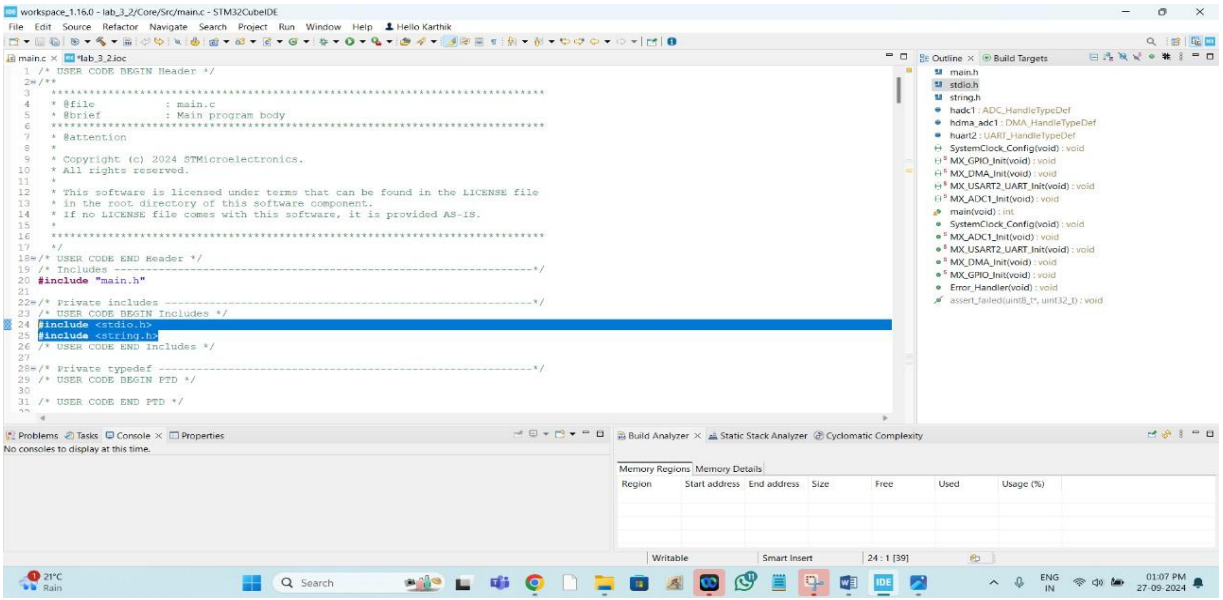
- I have made sure that **USART2** is in **Asynchronous mode**.



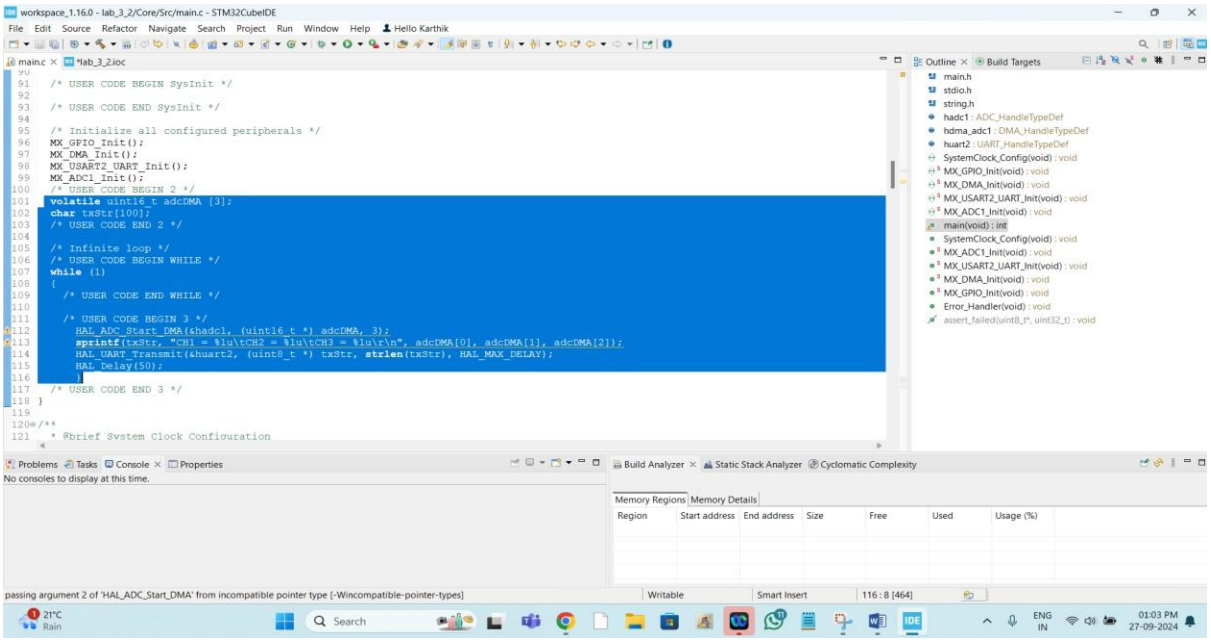
- After saving configuration the code is generated



- I made some changes to the code adding the library in main.c



- Added the code which prints the digital value of the resistance(interruption) after processing it in the micro controller

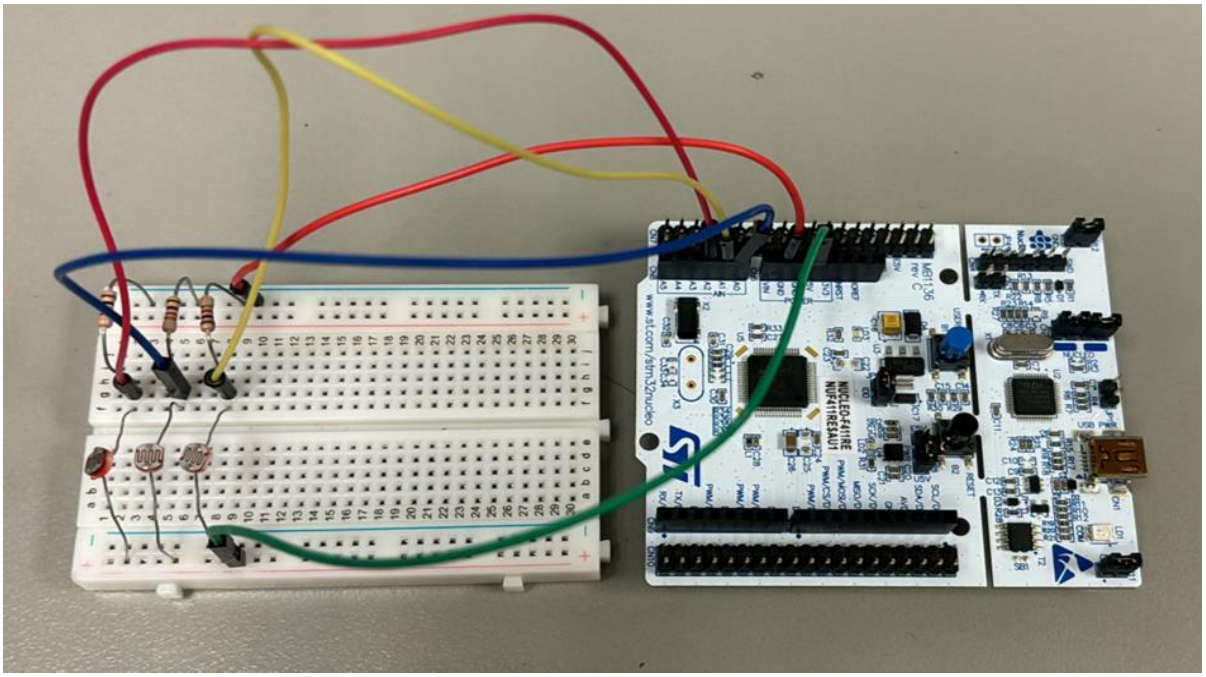


Experimental Setup:

The below image depicts the hardware setup for the experiment to perform. The connections are made as follows

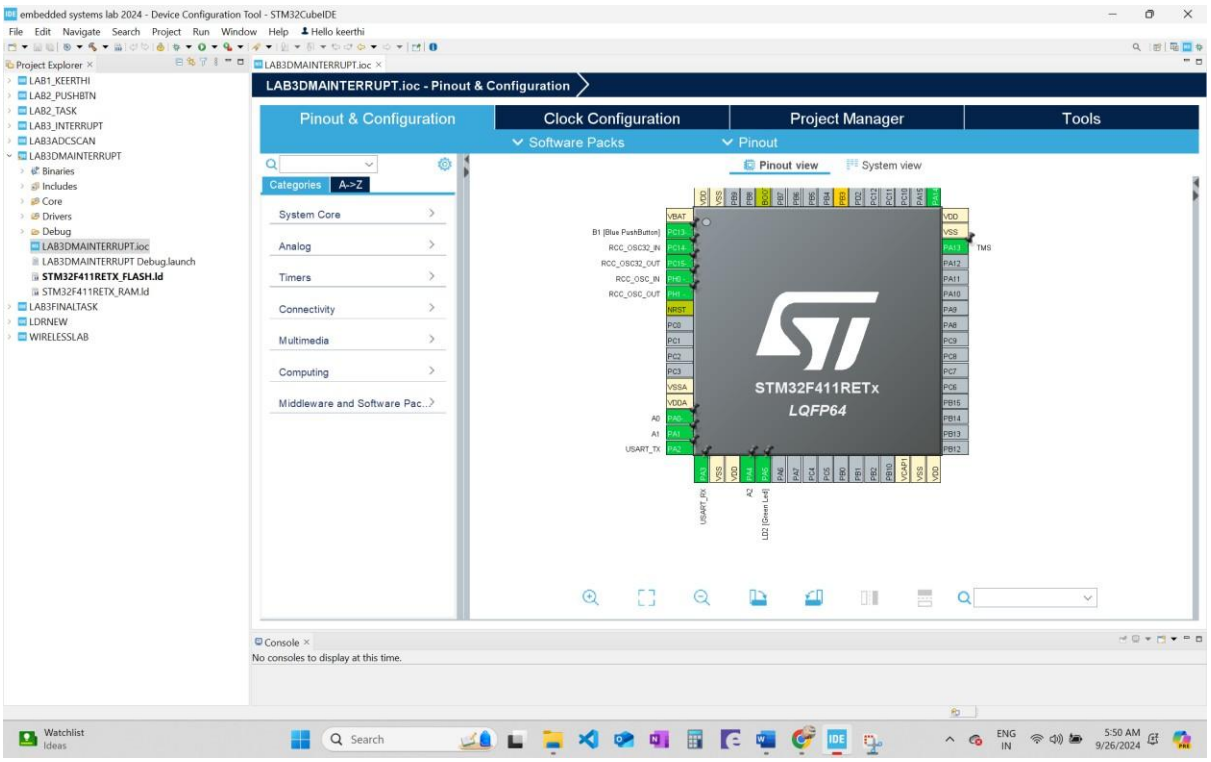
1.LDR: I have used 3 LDR's to perform the experiment, in which One of the terminal of all the 3 LDR's is connected to 5v on the board through a wire and the other terminal of first LDR is connected to A0 (because we configured the PA0 pin to ADC1_IN0) and second LDR is connected to A1 and similarly the third LDR is connected to A2.

2. Resistor: I have used 3 resistors to perform the experiment in which three of the resistor terminals are connected in parallel to the LDR terminals that is connected to A0, A1, A2 pin. Other terminal of all the resistors is connected to the ground using a wire to the board.

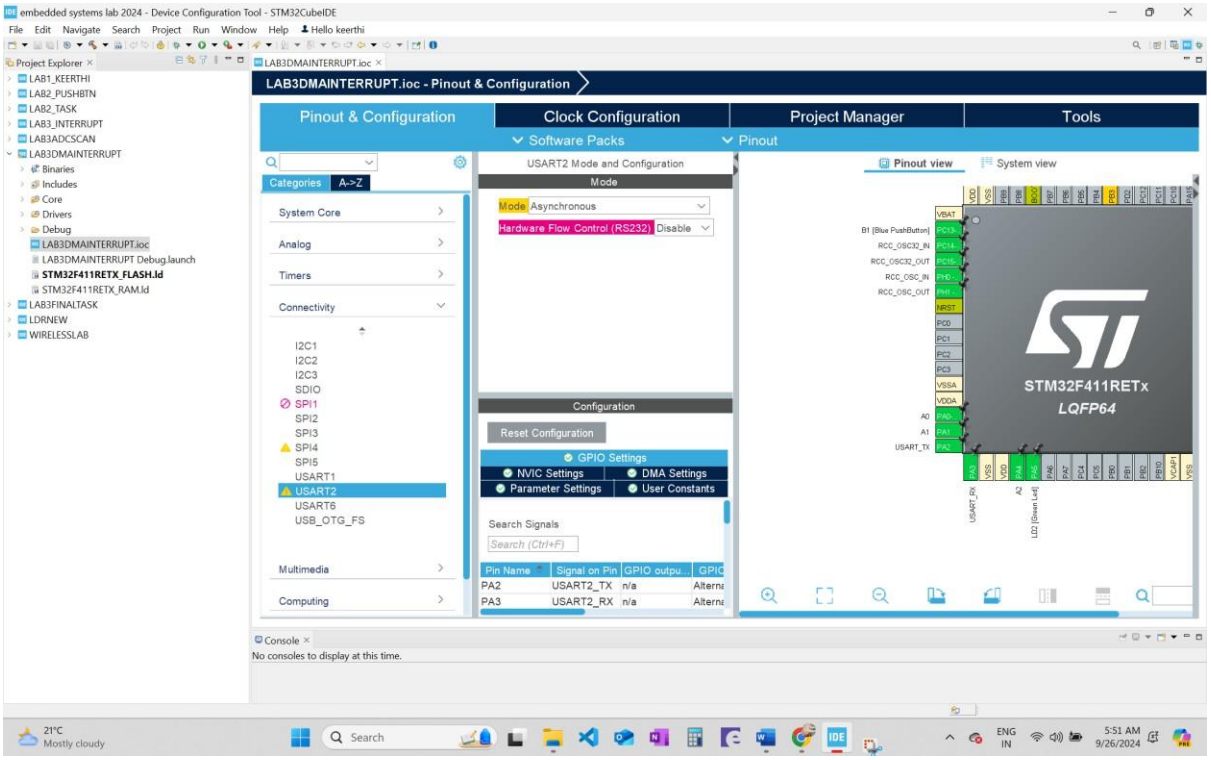


Part 3. ADC Scan Mode with DMA (& Interrupt)

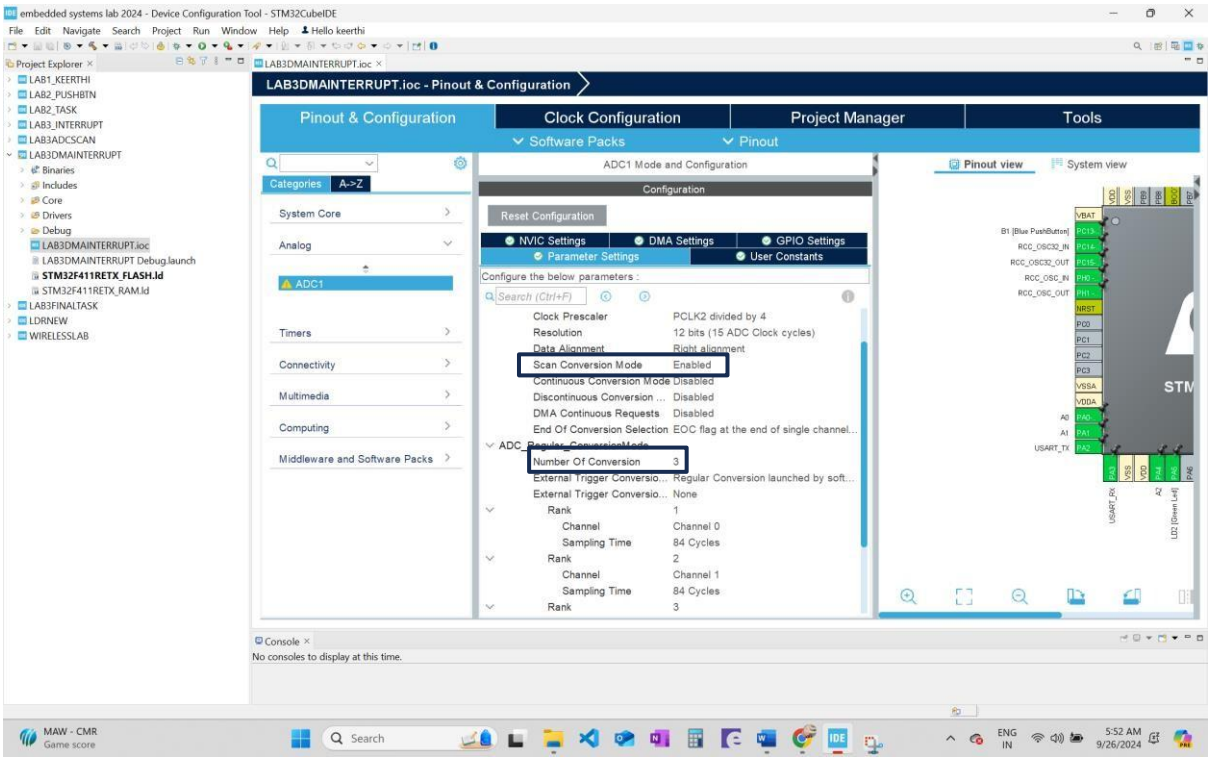
- I have selected the pin **PA0**, **PA1** and **PA4** as **ADC1 inputs** and renamed as **A0, A1** and **A2**.

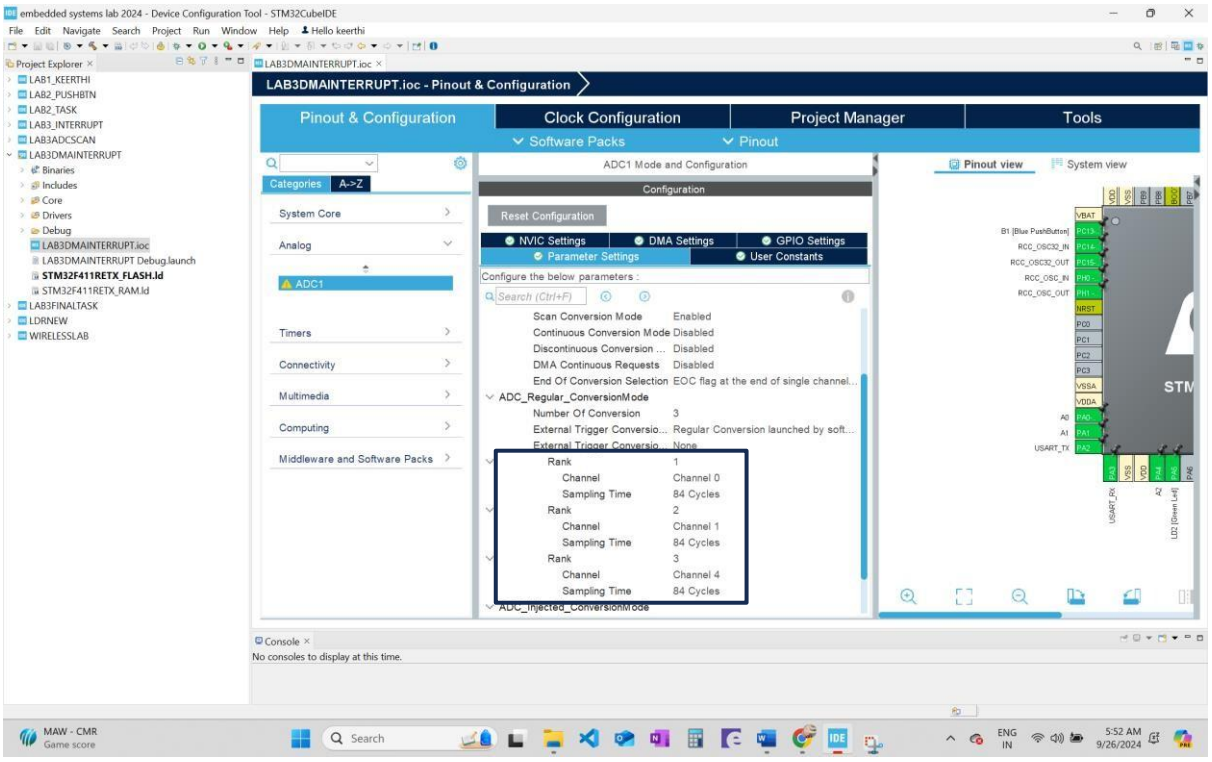


- I have made sure that **USART2** is in **Asynchronous mode**.

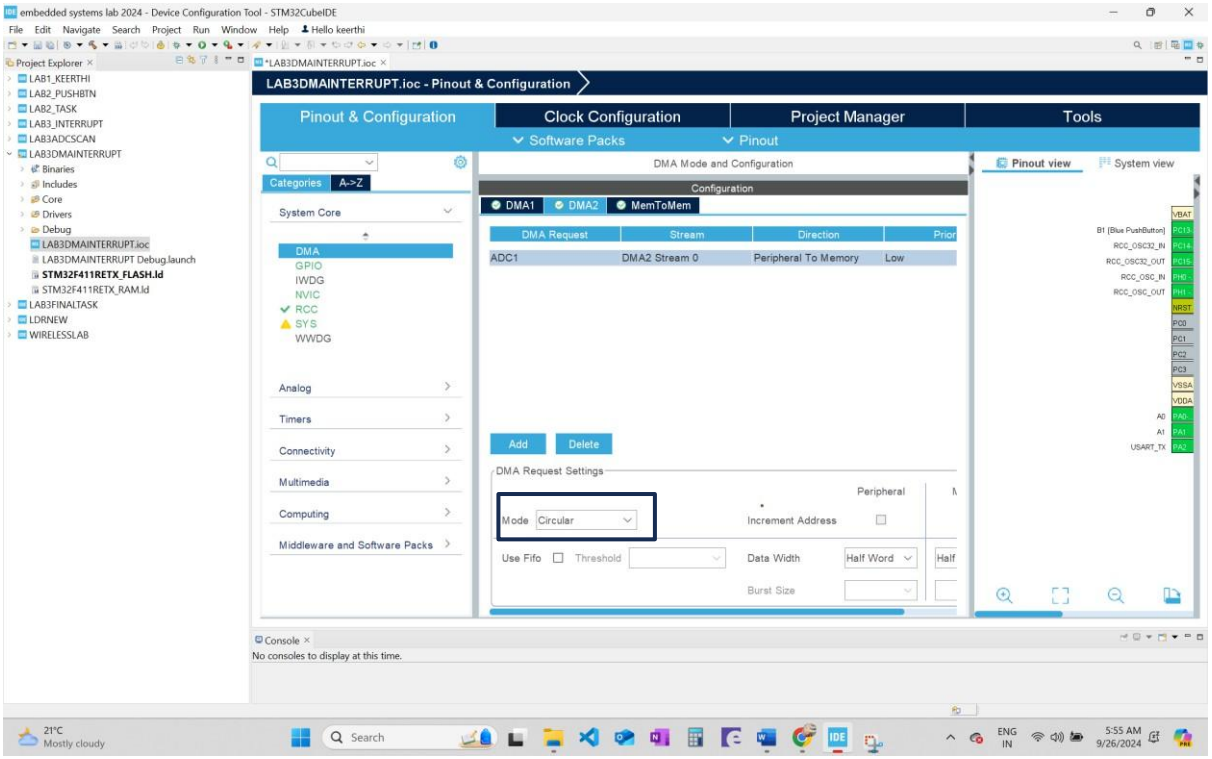


- In system core under Analog in **ADC1** , I have **Enabled** the **scan conversion mode** and increased the **Number of conversions** to **3** and I have made channel 0, channel 1 and channel 4 to **84 Cycles**.

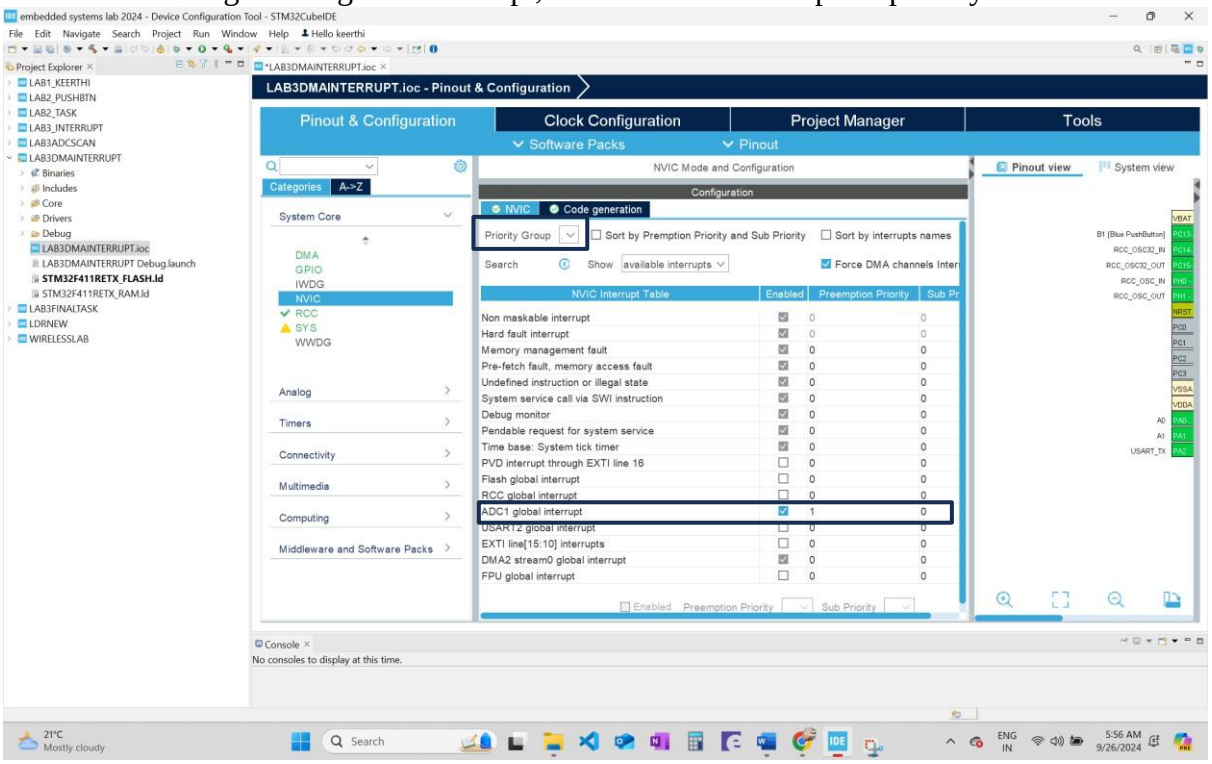




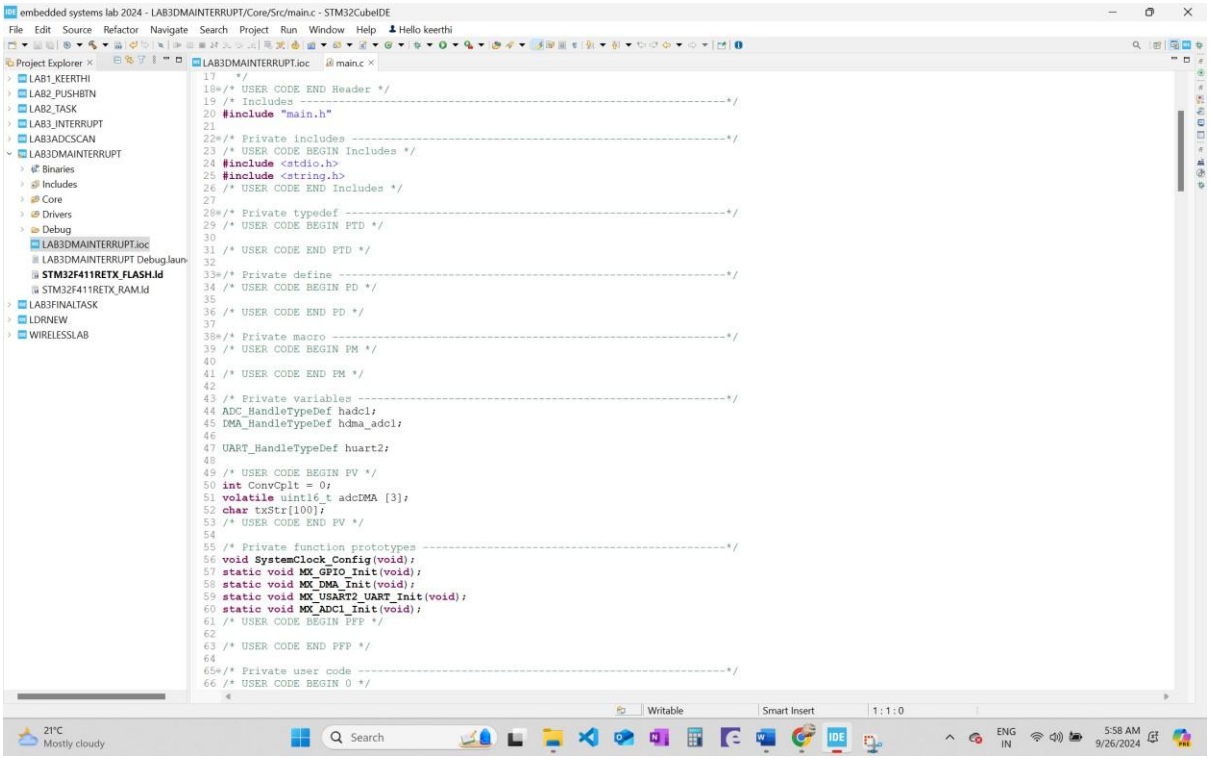
- In System Core, Under **DMA** I have added the **ADC1** and made the **mode** to be **Circular**.

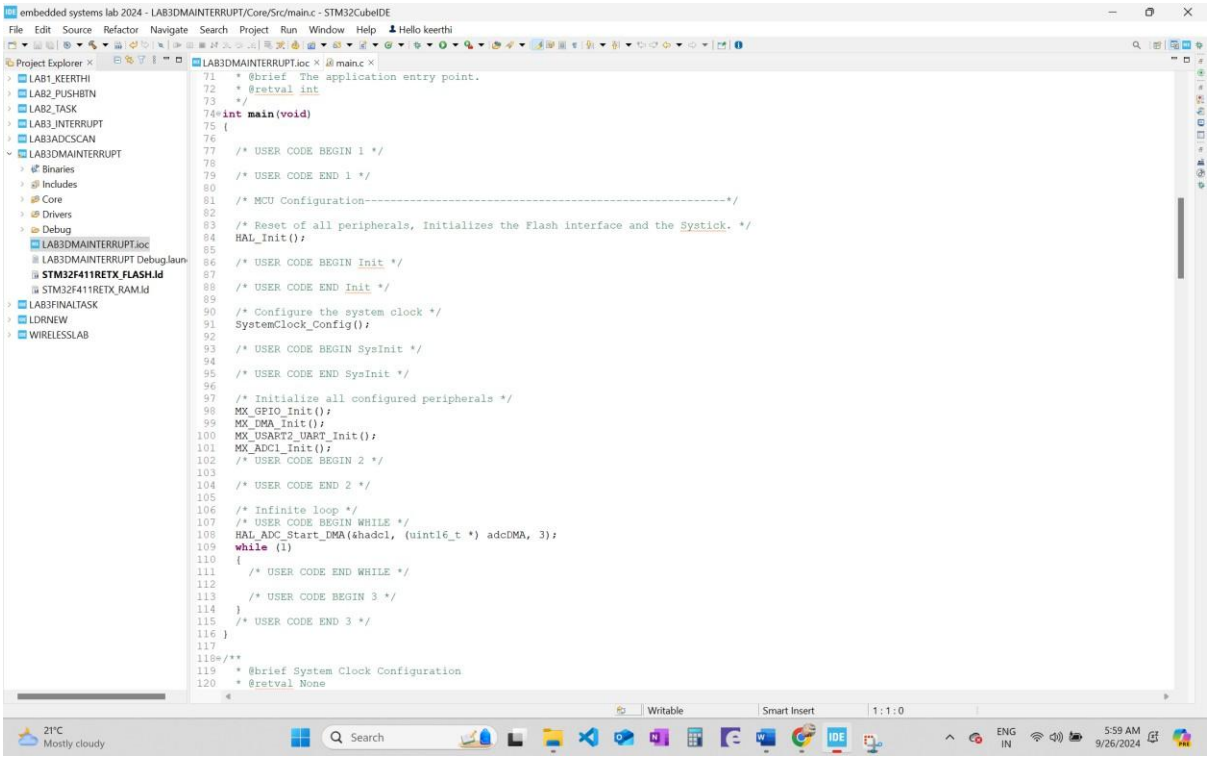


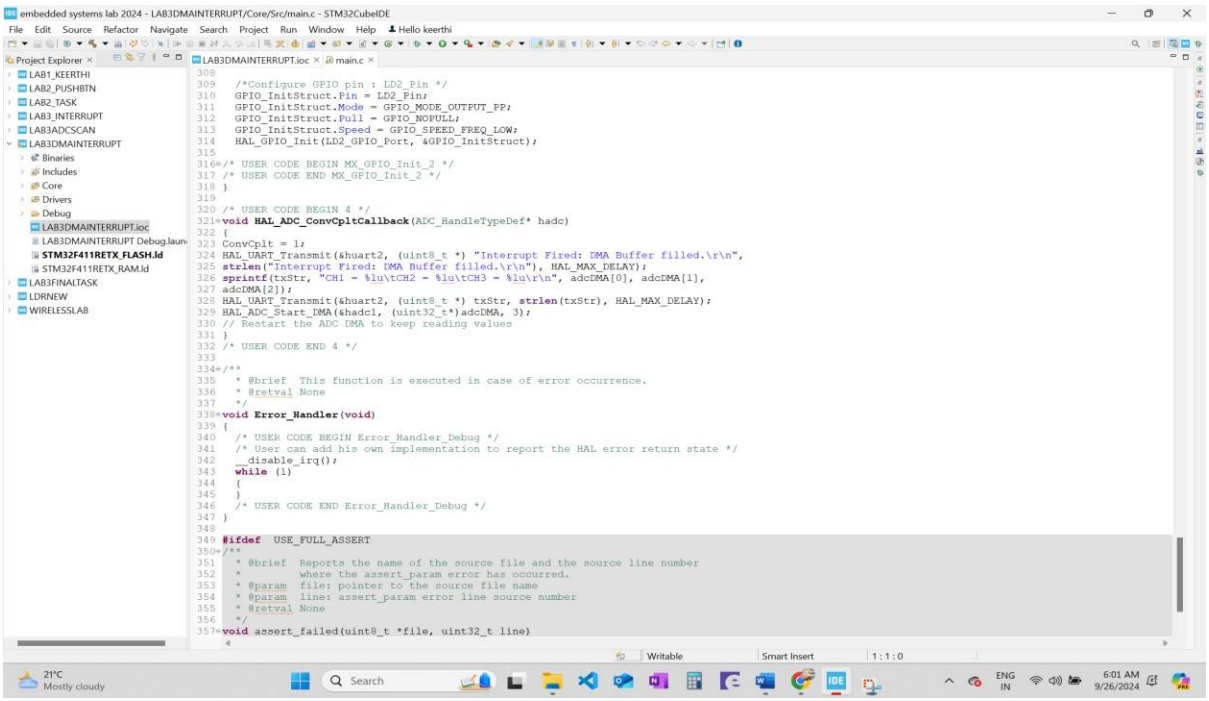
- In **System Core**, Under **NVIC** Set the **priority group** as **4- bit.**
- After selecting ADC1 global interrupt, I have Set the Preemption priority to 1.



- After saving the file, I have added the code in the main.c.

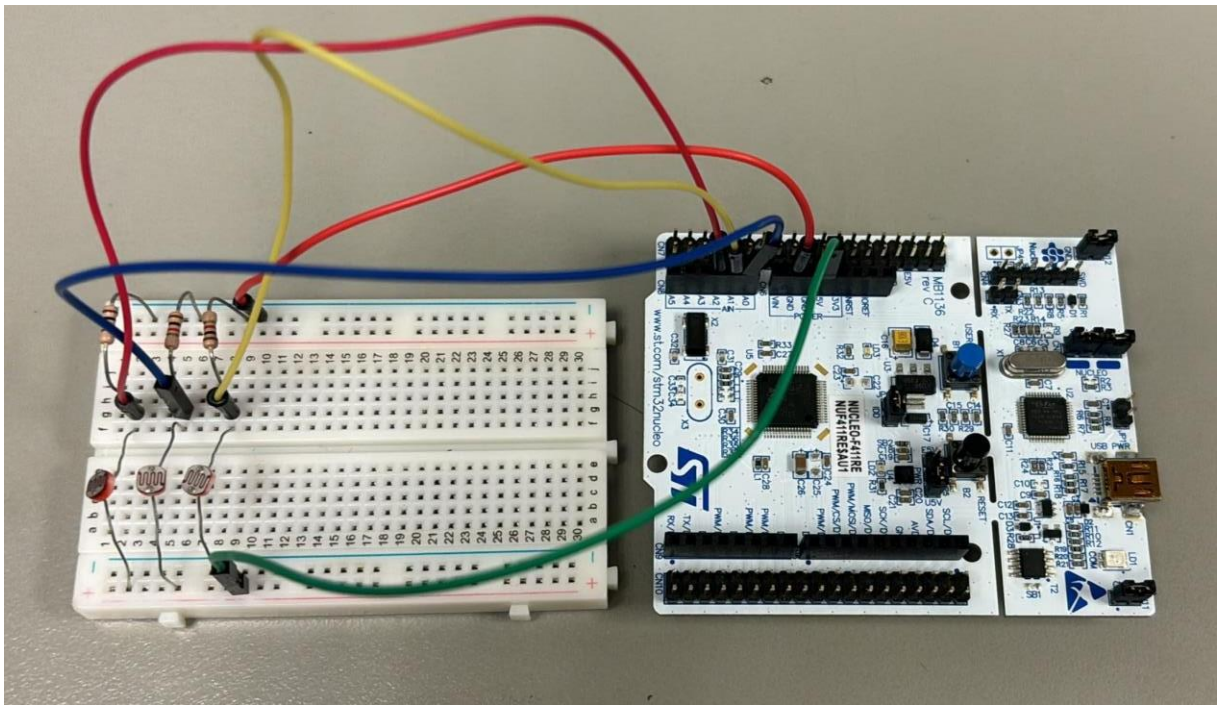






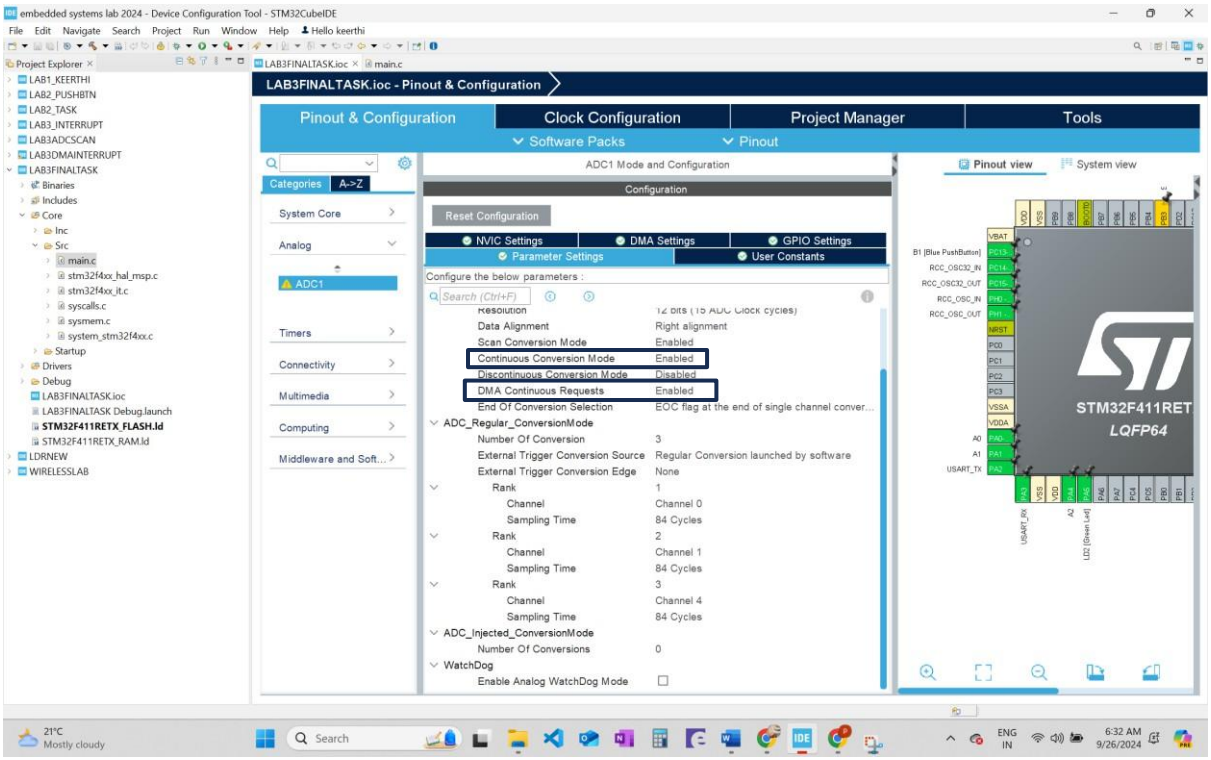
Experimental Setup:

- This experiment is conducted using the same setup as the ADC Scan Mode Configuration experiment.

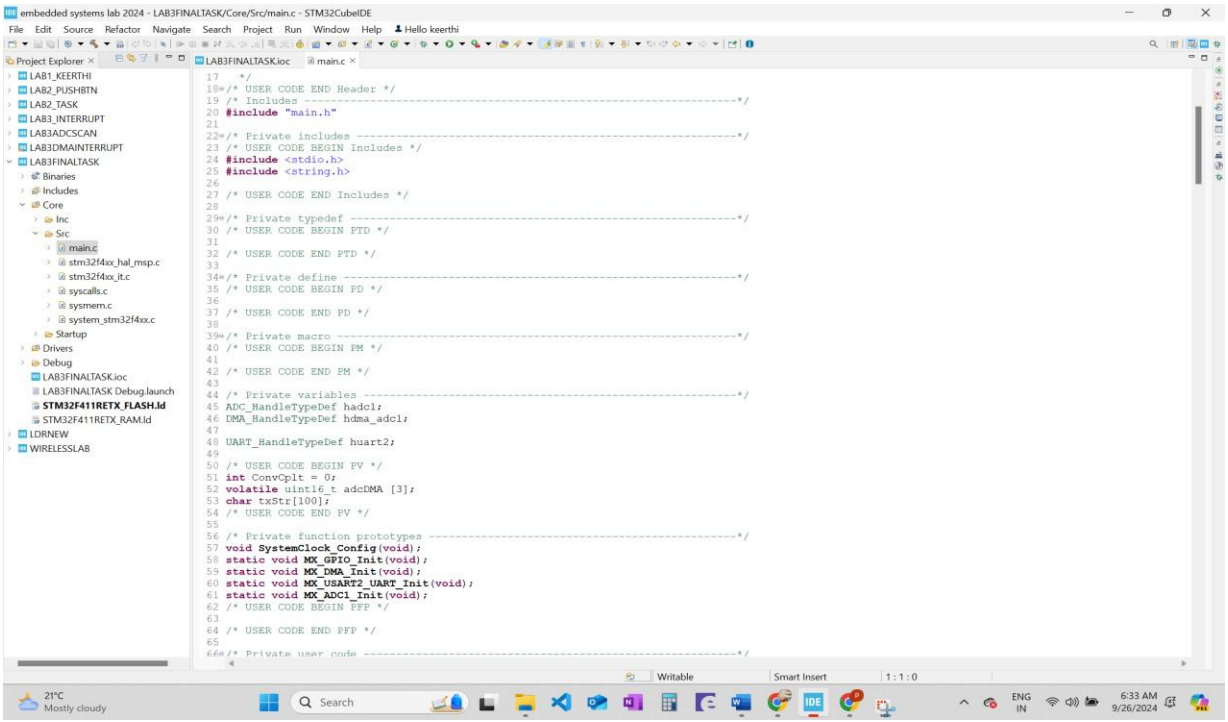


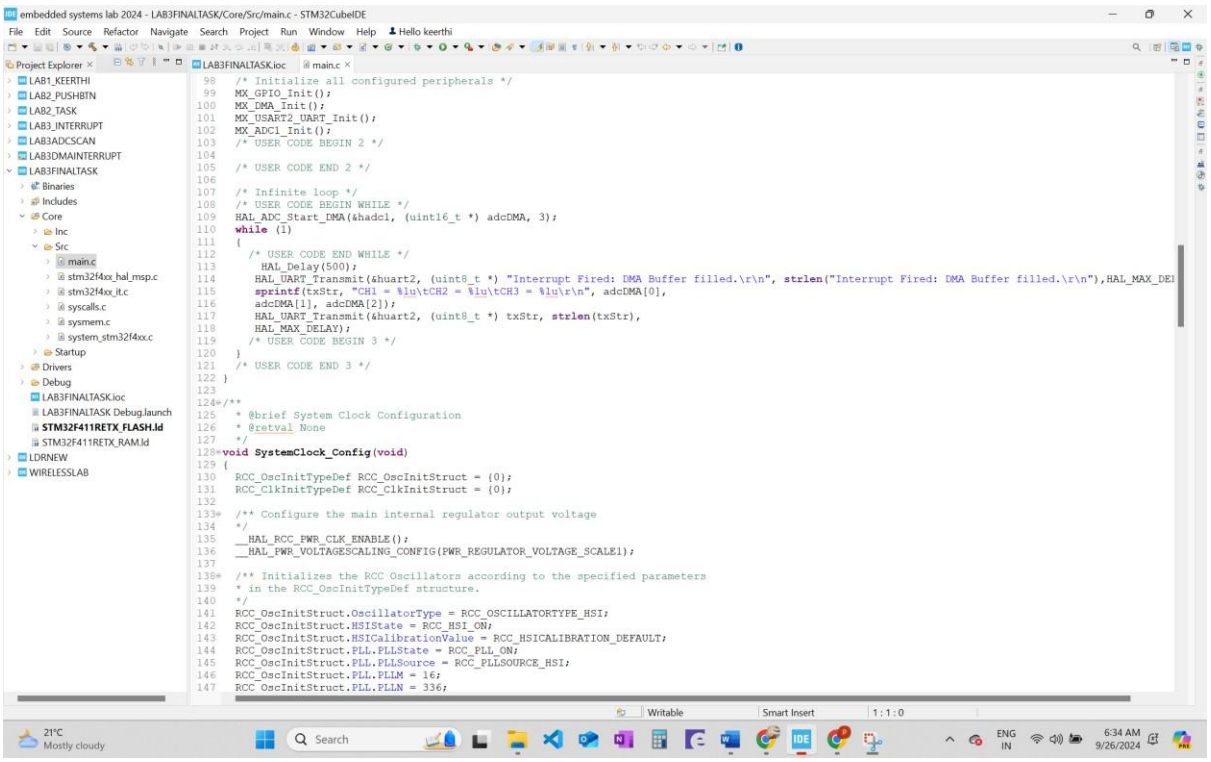
Part 4. ADC Continuous Mode with DMA (& Interrupt)

- This procedure follows exactly the same steps as the ADC Scan Mode with DMA (& Interrupt) software configuration.
- Additionally in this experiment the **Continuous Conversion Mode** and **DMA Continuous Mode** is **Enabled**.



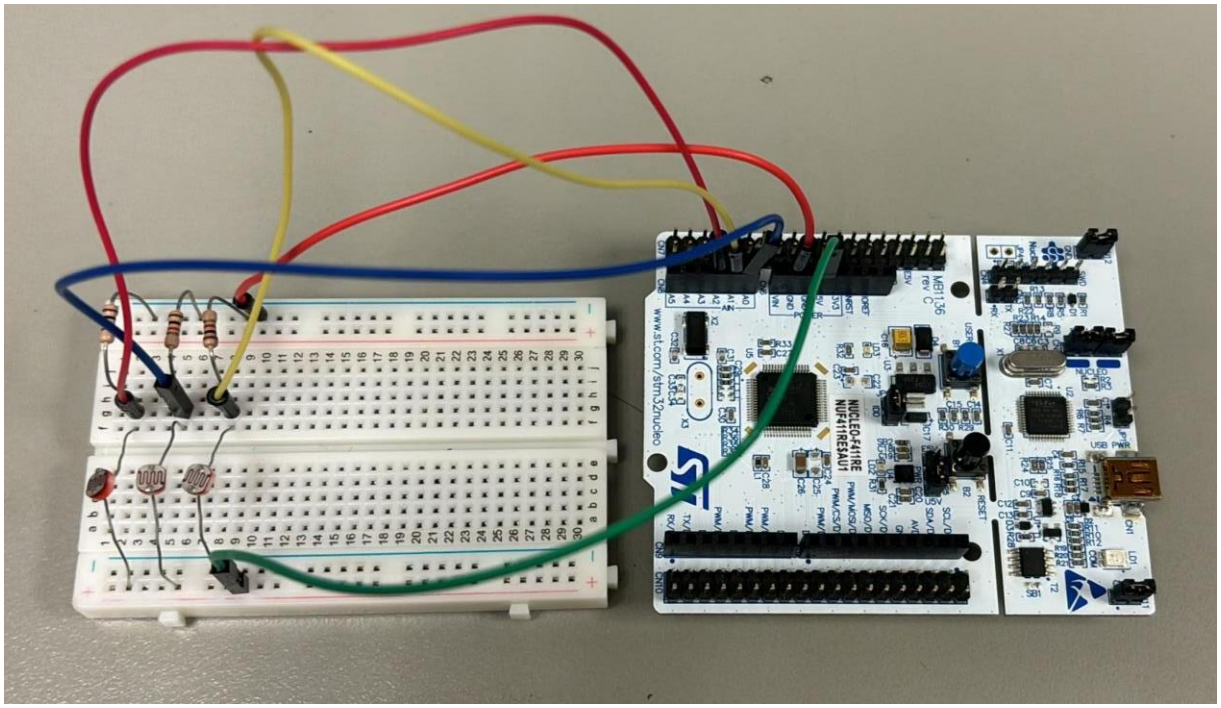
- After saving the file, I have added the code in the main.c.





Experimental Setup:

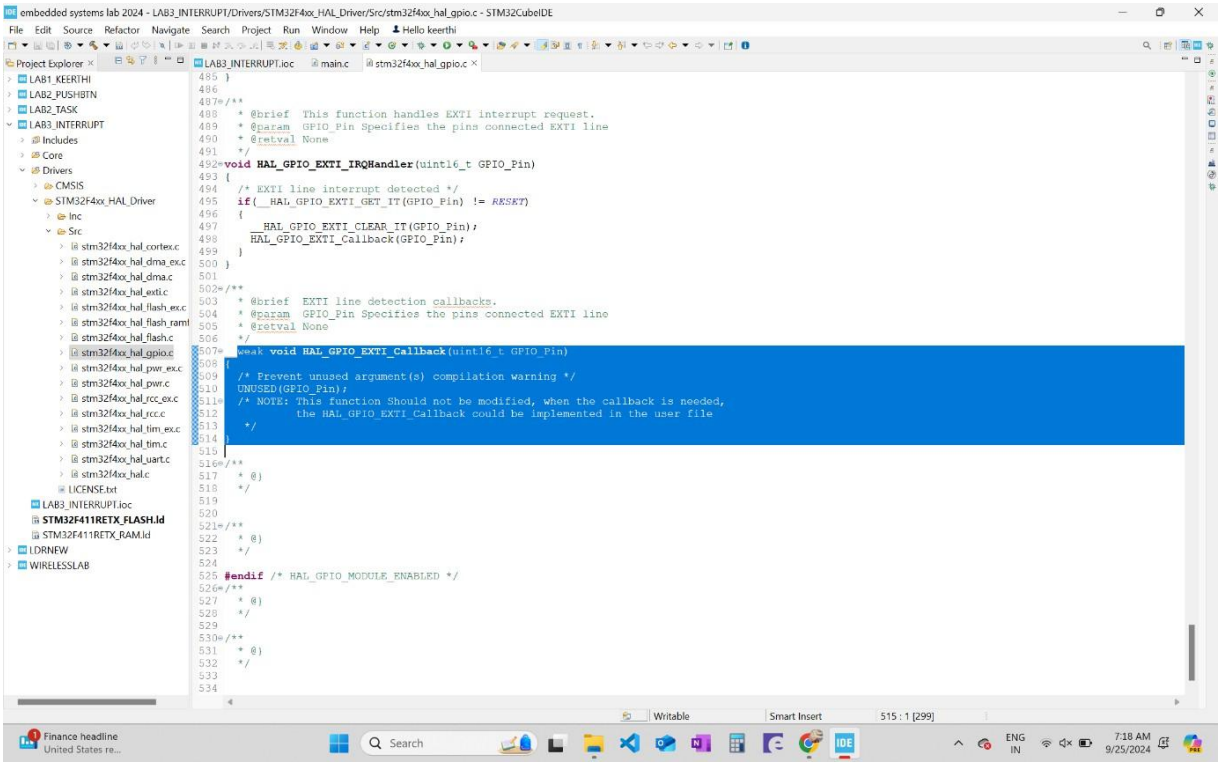
- This experiment is conducted using the same setup as the ADC Scan Mode Configuration experiment.

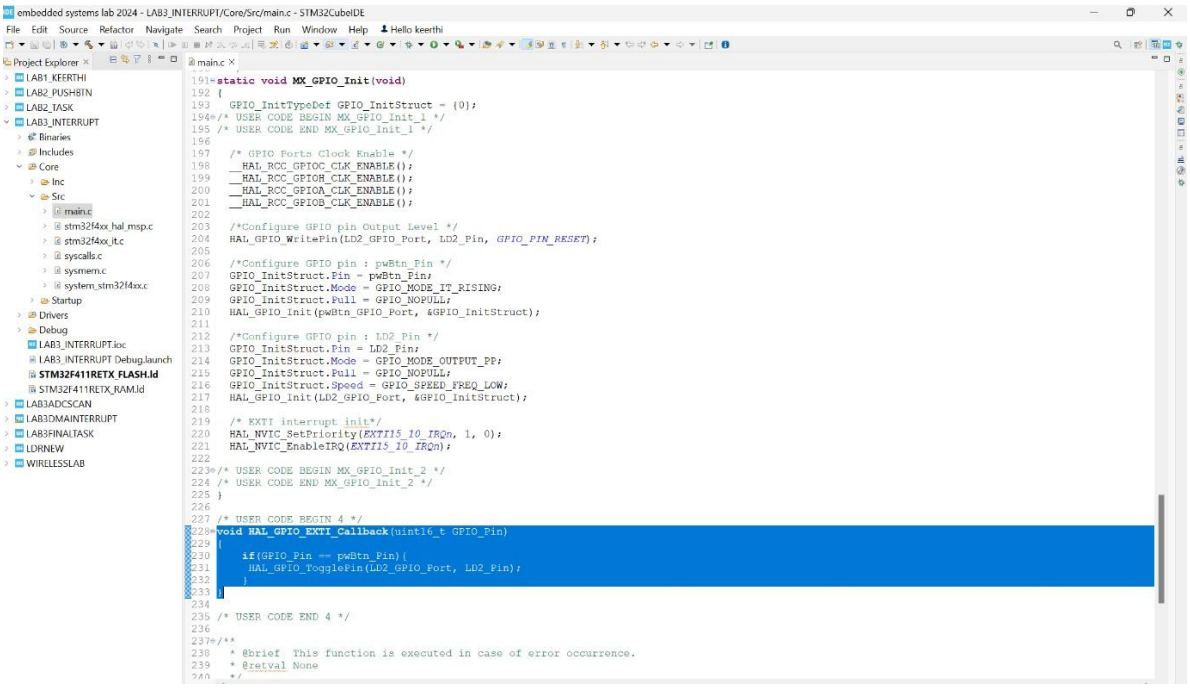


Code Review:

Part 1. GPIO External Interrupt (EXTI) Configuration

- This code executes to flip the LED (LD2) when the button (pwBtn) is pressed, enabling a prompt response to user input without requiring the main program loop to continuously monitor the button state. As a result, the system is more effective and event-responsive.





Code:

```
/* USER CODE BEGIN 4 */
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
if(GPIO_Pin == pwBtn_Pin){
HAL_GPIO_TogglePin(LD2_GPIO_Port, LD2_Pin);
}
}
/* USER CODE END 4 */
```

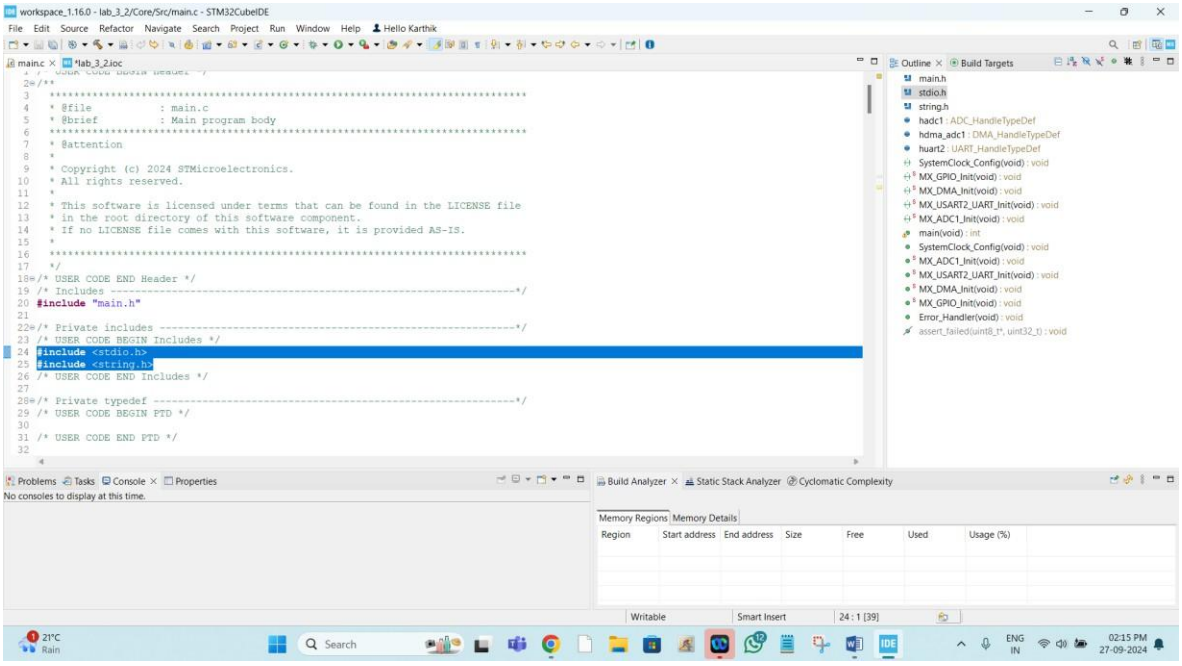
This function is a callback that the STM32 HAL library calls when an external interrupt occurs. The GPIO_Pin parameter tells you which pin triggered the interrupt.

This conditional statement checks if the GPIO pin that triggered the interrupt is the same as the pwBtn_Pin. If it matches, it means the push button connected to pwBtn_Pin has been pressed or released.

If the condition in step 2 is true, this line of code toggles the state of the LED connected to the LD2_Pin. This means that if the LED was on, it will be turned off, and vice versa.

Part 2. ADC Scan Mode Configuration

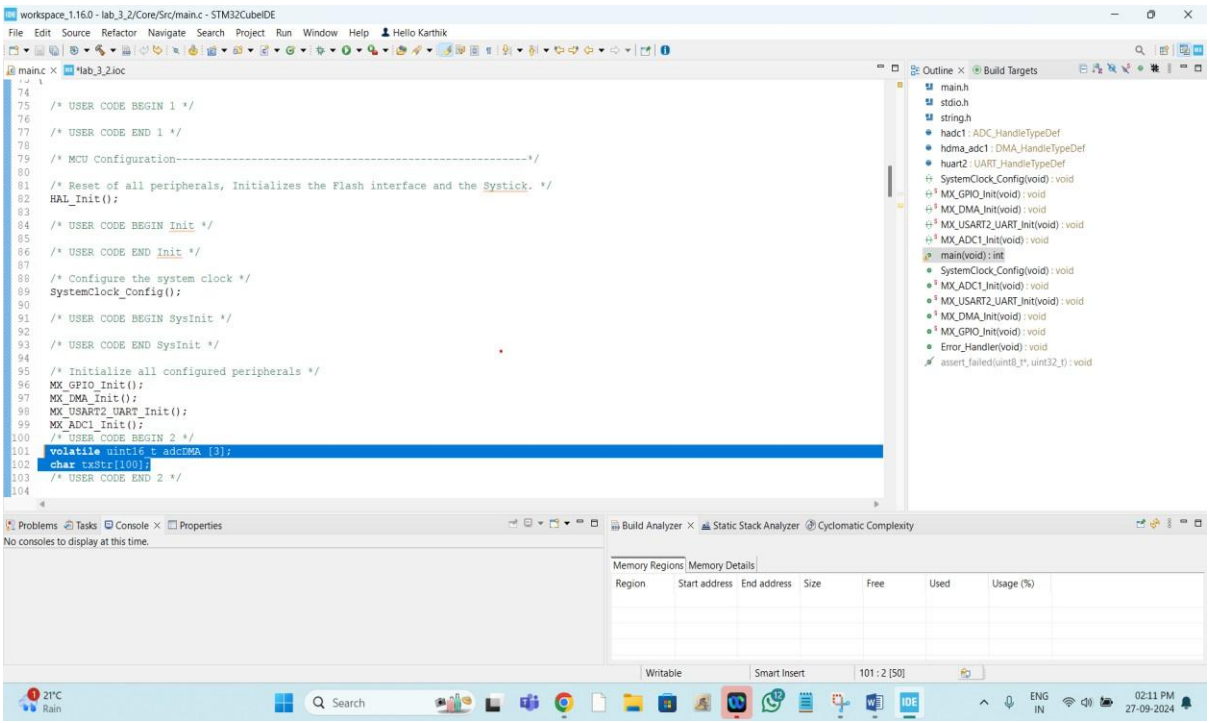
- After generating the code we made some changes in the main.c



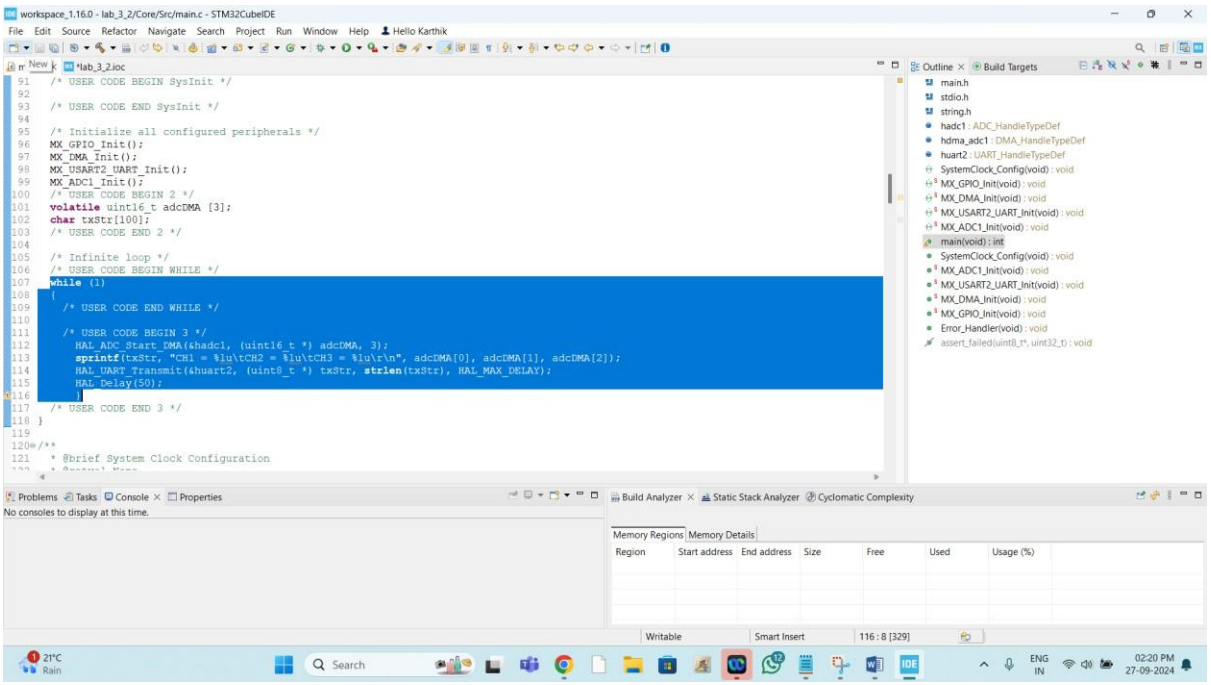
Code

```
/* USER CODE BEGIN Includes */
#include <stdio.h>
#include <string.h>
/* USER CODE END Includes */
```

These headers provide functions for standard input/output operations and string manipulation.



- adcDMA[3]: An array of 3 uint16_t elements to store the ADC conversion results.
- txStr[100]: A character array to store the formatted string for UART transmission.



Code explanation

```
/* USER CODE BEGIN 2 */
volatile uint16_t adcDMA [3];
char txStr[100];
/* USER CODE END 2 */
/* Infinite loop */
/* USER CODE BEGIN WHILE */
while (1)
{
/* USER CODE END WHILE */
/* USER CODE BEGIN 3 */
HAL_ADC_Start_DMA(&hadc1, (uint16_t *) adcDMA, 3);
sprintf(txStr, "CH1 = %lu\tCH2 = %lu\tCH3 = %lu\r\n", adcDMA[0], adcDMA[1], adcDMA[2]);
HAL_UART_Transmit(&huart2, (uint8_t *) txStr, strlen(txStr), HAL_MAX_DELAY);
HAL_Delay(50);
}
/* USER CODE END 3 */
```

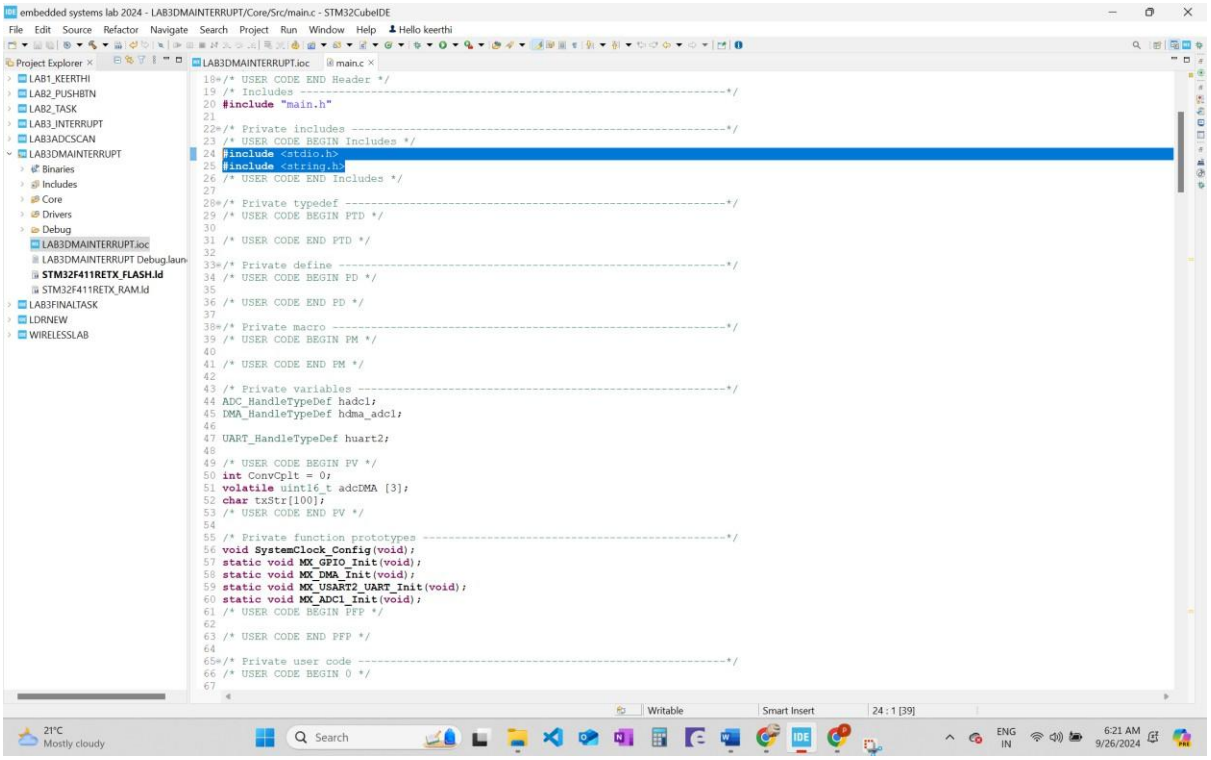
This function starts the ADC to read three channels and store the results in the adc DMA array using DMA (Direct Memory Access).

Formats the ADC conversion results into a string for transmission.

Sends the formatted string containing ADC values over UART.

Part 3. ADC Scan Mode with DMA (& Interrupt)

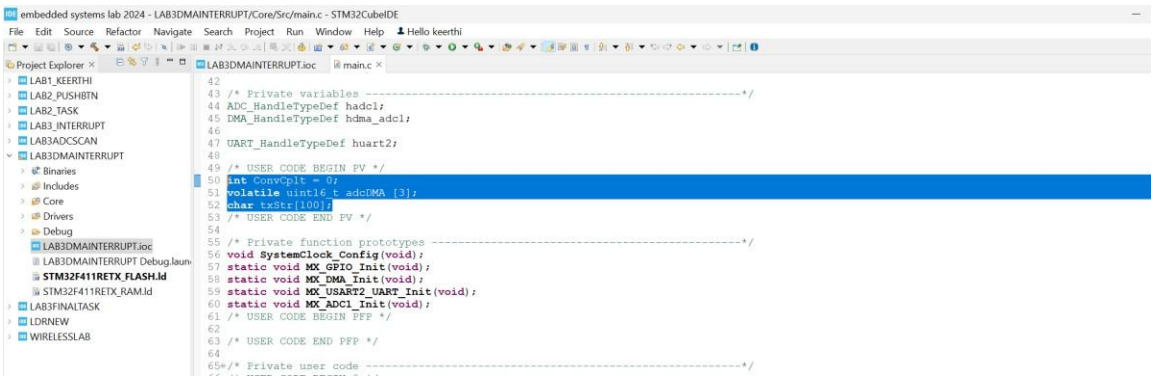
- The code initializes the ADC in DMA mode, which allows the ADC to continuously convert analog inputs and store the results in memory without requiring constant CPU intervention.
- The callback function is triggered whenever an ADC conversion is complete. It processes the conversion results, sends them to the UART for display, and restarts the ADC DMA to continue the conversion process.
- This approach is efficient for applications that require continuous ADC measurements without significant CPU overhead.



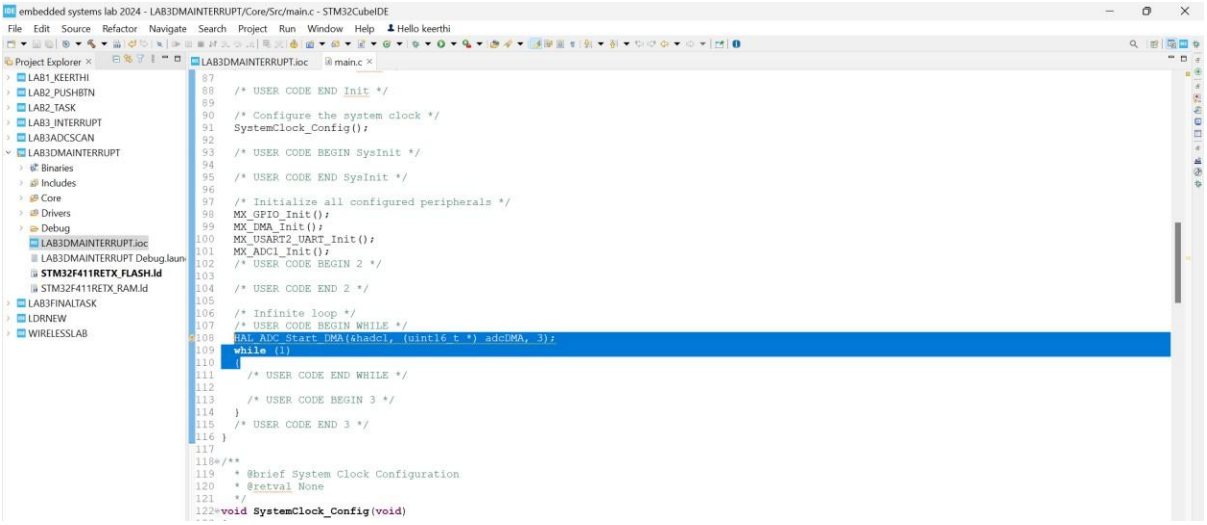
Code:

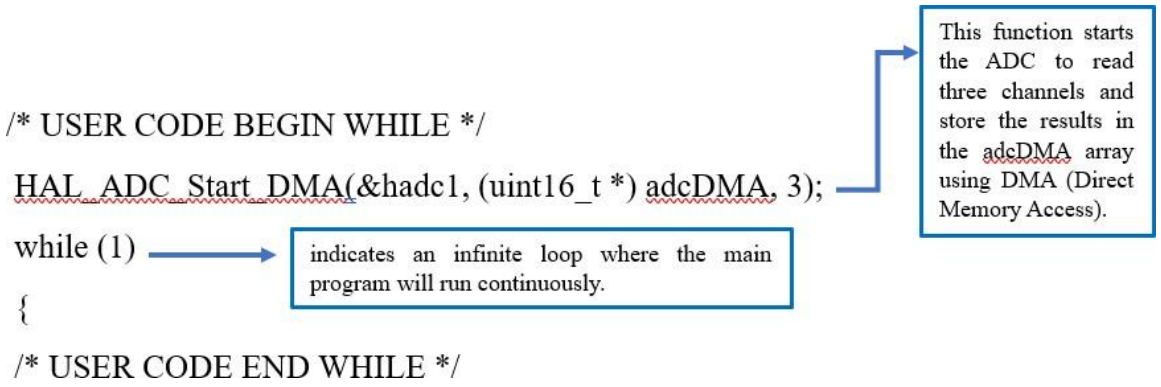
```
/* USER CODE BEGIN Includes */
#include <stdio.h>
#include <string.h>
/* USER CODE END Includes */
```

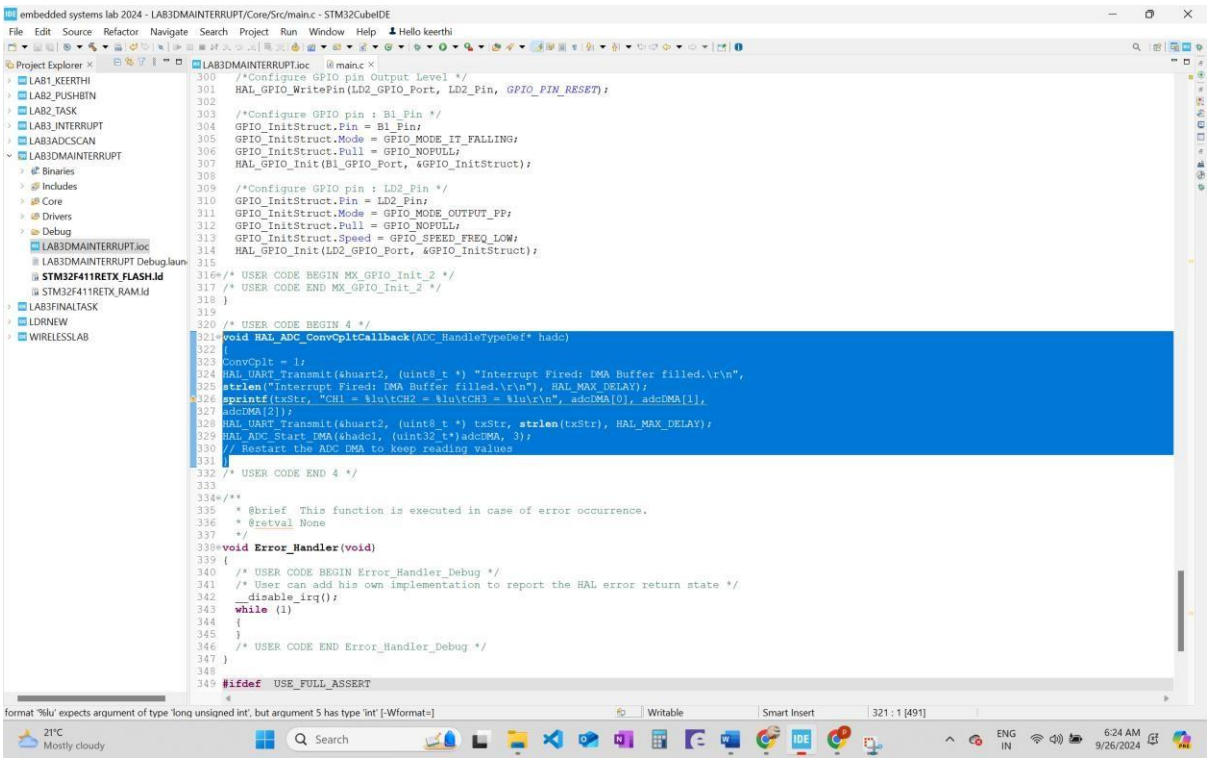
These headers provide functions for standard input/output operations and string manipulation.



- ConvCplt: A flag variable used to indicate whether an ADC conversion is complete.
- adcDMA[3]: An array of 3 uint16_t elements to store the ADC conversion results.
- txStr[100]: A character array to store the formatted string for UART transmission.








```
/* USER CODE BEGIN 4 */
void HAL_ADC_ConvCpltCallback(ADC_HandleTypeDef* hadc)
{
ConvCplt = 1;
HAL_UART_Transmit(&huart2, (uint8_t *) "Interrupt Fired: DMA Buffer filled.\r\n",
strlen("Interrupt Fired: DMA Buffer filled.\r\n"), HAL_MAX_DELAY);
sprintf(txStr, "CH1 = %lu\tCH2 = %lu\tCH3 = %lu\r\n", adcDMA[0], adcDMA[1],
adcDMA[2]);
HAL_UART_Transmit(&huart2, (uint8_t *) txStr, strlen(txStr), HAL_MAX_DELAY);
HAL_ADC_Start_DMA(&hadc1, (uint32_t*)adcDMA, 3);
// Restart the ADC DMA to keep reading values
}
/* USER CODE END 4 */
```

This is the callback function that is automatically called by the HAL when an ADC conversion is complete.

Sets the flag indicating that the ADC conversion has completed.

Sends a message to the UART indicating that the ADC conversion is complete and the DMA buffer is filled.

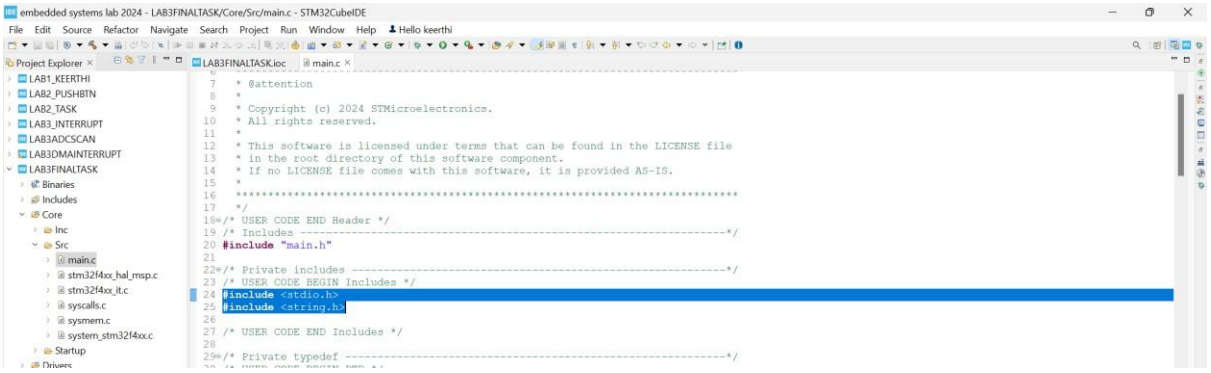
Formats the ADC conversion results into a string for transmission.

Sends the formatted string containing ADC values over UART.

Restarts the ADC DMA to continue reading values.

Part 4. ADC Continuous Mode with DMA (& Interrupt)

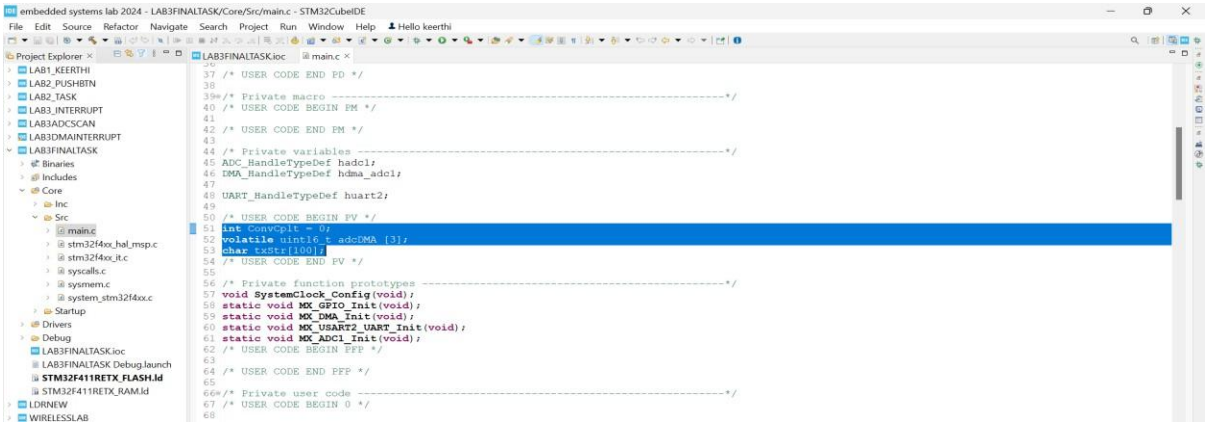
- The code initializes the ADC in DMA mode, which allows the ADC to continuously convert analog inputs and store the results in memory without requiring constant CPU intervention.
- The main loop periodically (every 500 milliseconds) checks the ADC conversion results and sends them to the UART for display.



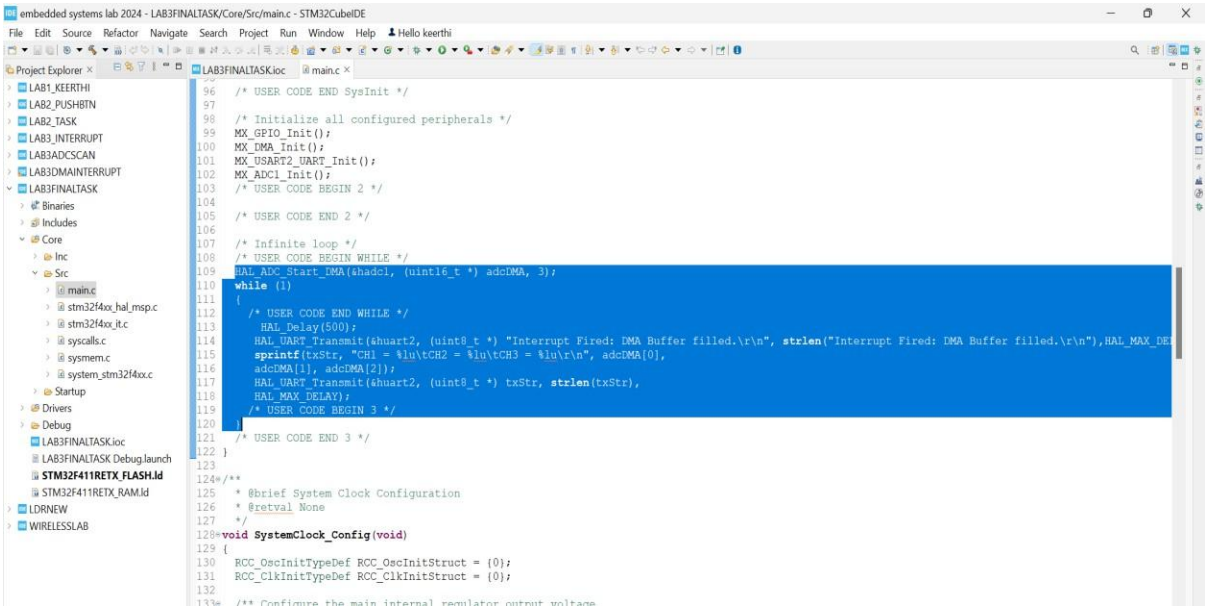
Code:

```
/* USER CODE BEGIN Includes */
#include <stdio.h>
#include <string.h>
/* USER CODE END Includes */
```

These headers provide functions for standard input/output operations and string manipulation.



- ConvCplt: A flag variable used to indicate whether an ADC conversion is complete.
- adcDMA[3]: An array of 3 uint16_t elements to store the ADC conversion results.
- txStr[100]: A character array to store the formatted string for UART transmission.




```
/* USER CODE BEGIN WHILE */
HAL_ADC_Start_DMA(&hadc1, (uint16_t *) adcDMA, 3);
while (1)
{
/* USER CODE END WHILE */
HAL_Delay(500);
HAL_UART_Transmit(&huart2, (uint8_t *) "Interrupt Fired: DMA Buffer
filled.\r\n", strlen("Interrupt Fired: DMA Buffer filled.\r\n"),
HAL_MAX_DELAY);
sprintf(txStr, "CH1 = %lu\tCH2 = %lu\tCH3 = %lu\r\n", adcDMA[0],
adcDMA[1], adcDMA[2]);
HAL_UART_Transmit(&huart2, (uint8_t *) txStr, strlen(txStr),
HAL_MAX_DELAY);
/* USER CODE BEGIN 3 */
}
/* USER CODE END 3 */
```

Starts the ADC in DMA mode, using hadc1 as the ADC handle, adcDMA as the destination buffer for conversion results, and 3 as the number of channels to be converted.

An infinite loop that keeps the main program running indefinitely.

Introduces a 500 millisecond delay, allowing the system to pause briefly between UART transmissions.

Sends a message to the UART indicating that the ADC conversion is complete and the DMA buffer is filled.

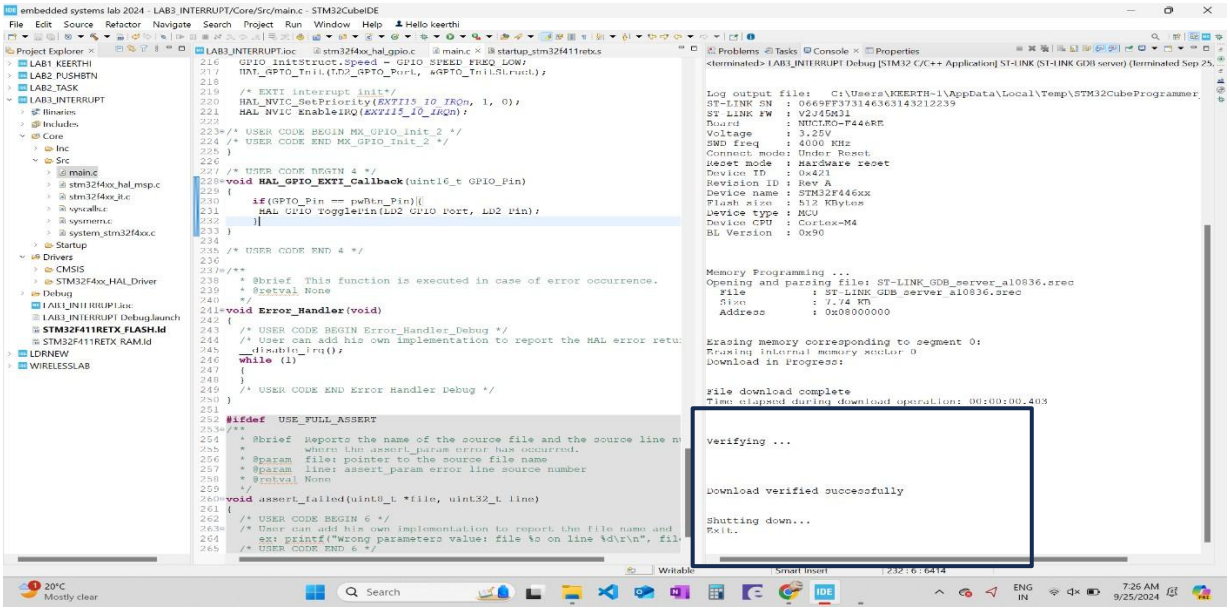
Formats the ADC conversion results into a string for transmission.

Sends the formatted string to the UART.

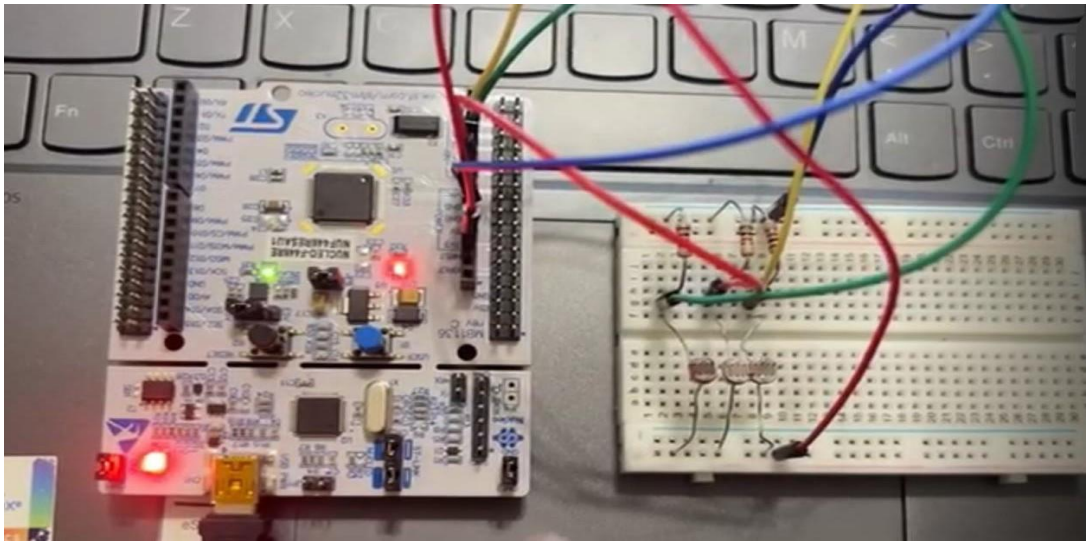
Results:

Part 1. GPIO External Interrupt (EXTI) Configuration

- After saving the file and writing the code in main.c and connecting the hardware setup, I have successfully debugged the file and the output is shown in console window.



- After running the code, I have observed that when I push the button externally the LED glows and when I again push the button the LED turns off indicating the external interrupt.
- When external interrupt is given through push button LED glows on and off.

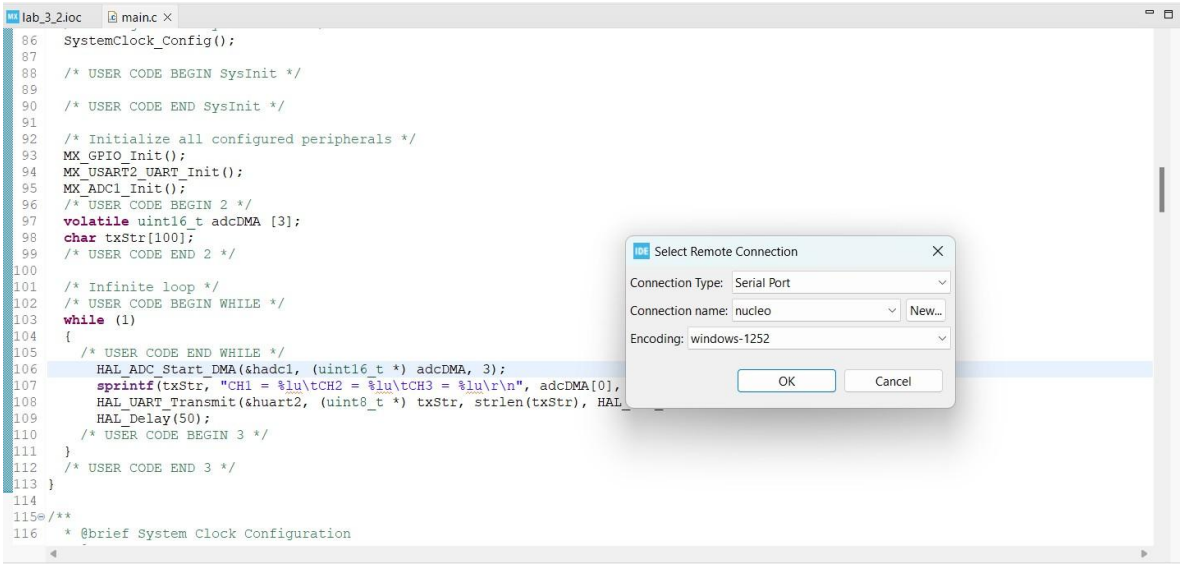


- A video demonstrating the configuration of the GPIO External Interrupt (EXTI) is provided below.

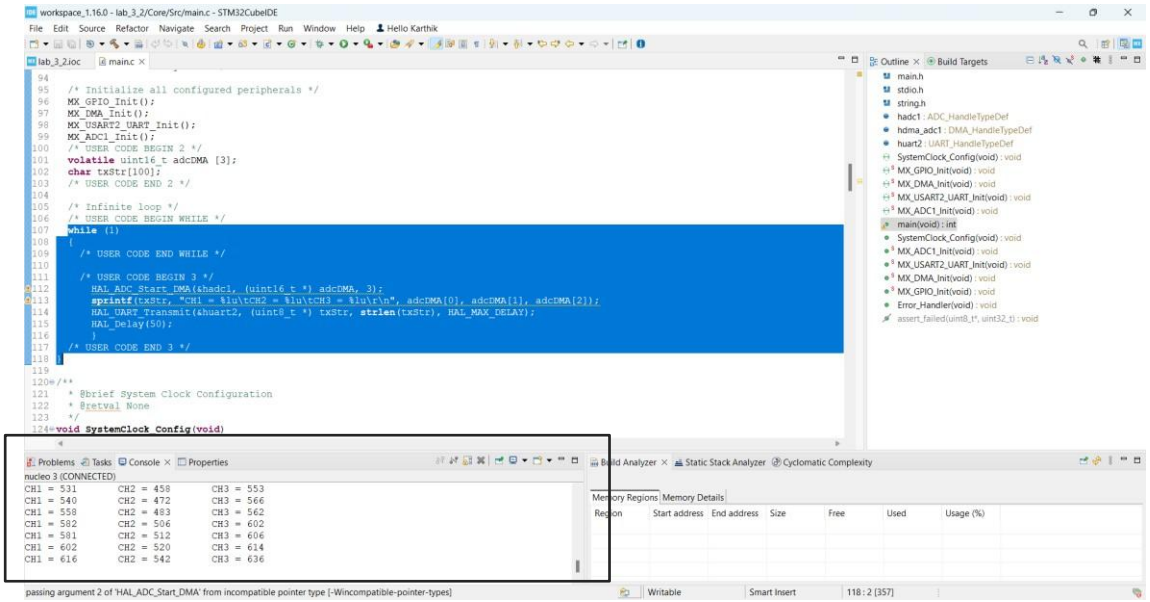
https://drive.google.com/file/d/1FlYNUDNJH3u3hqsnTS2RaTpOfxiPfY98/view?usp=drive_link

Part 2. ADC Scan Mode Configuration

- After successfully compiling, in the console window to display the values on the serial monitor. I have selected the command shell console and then selected the serial port and added new connection name (This connection name is named as **nucleo**).



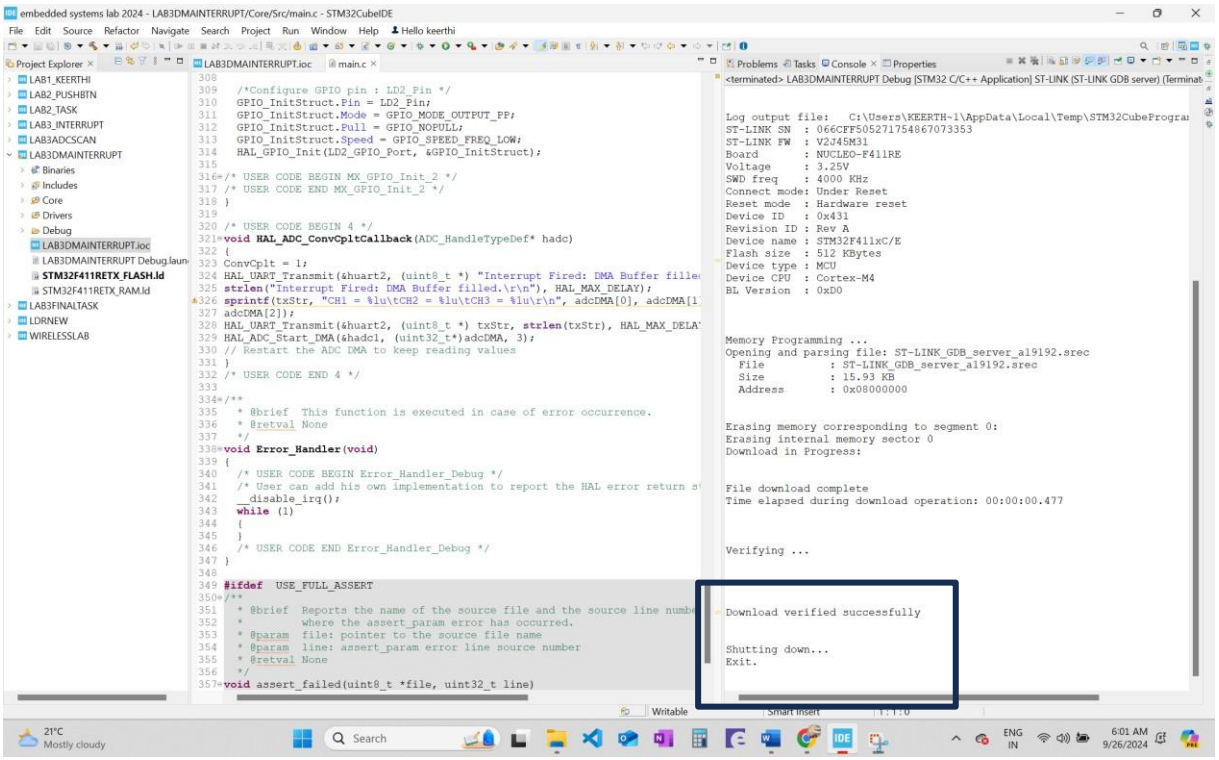
- The values appear on the serial monitor as soon as the remote connection is established successfully.

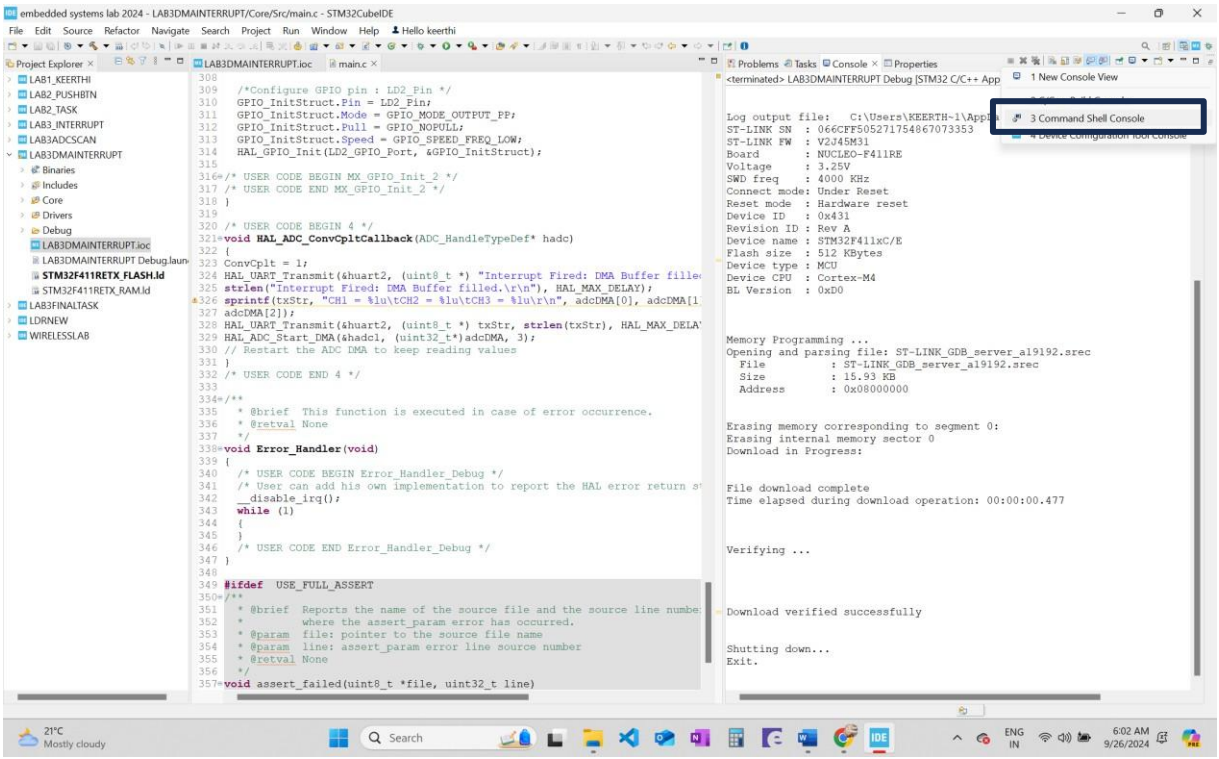


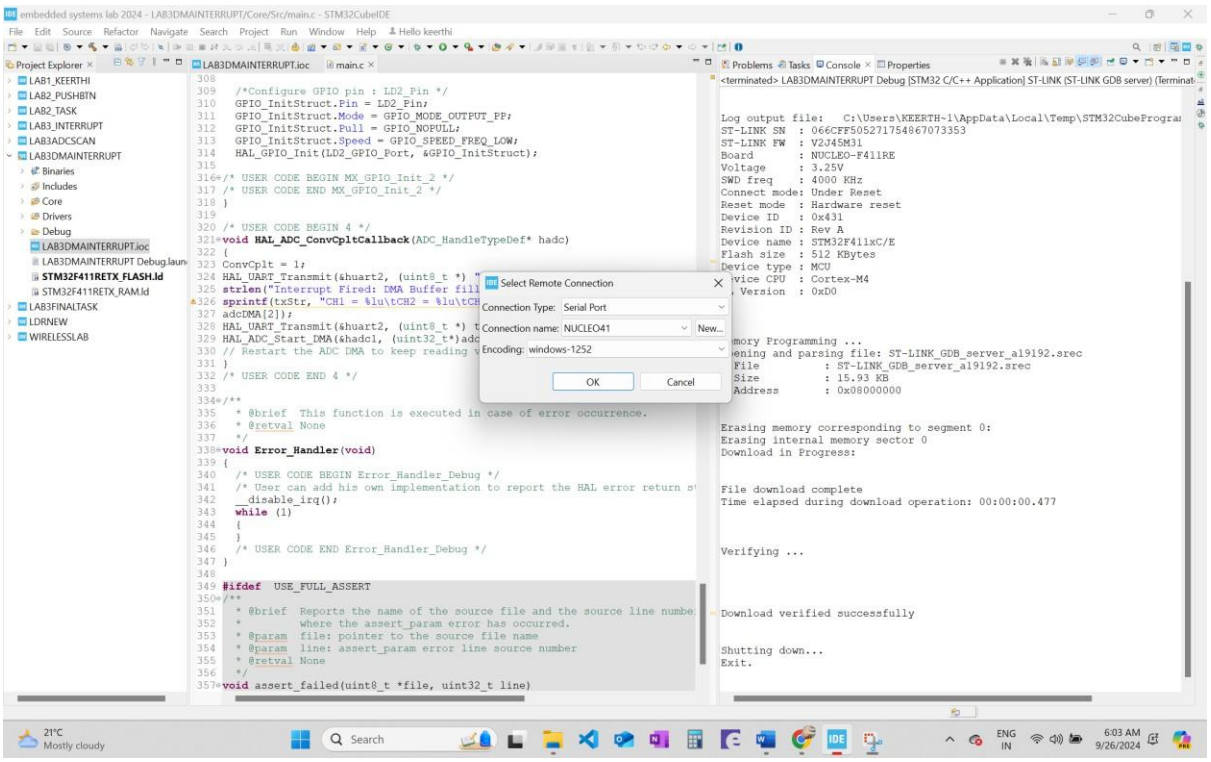
- The values that are displayed on serial monitor are the LDR values. I have made channel 0, channel 1 and channel 4. These channel values resemble the LDR values. As the LDR is connected to ADC pin on board, ADC converts the analog value to digital values and display those digital values on the serial monitor. The values are constantly shown, and they are adjusted and mirrored on the serial monitor whenever the LDR is covered. Data from the ADC channels is displayed in the serial terminal, showing values in real-time.

Part 3. ADC Scan Mode with DMA (& Interrupt)

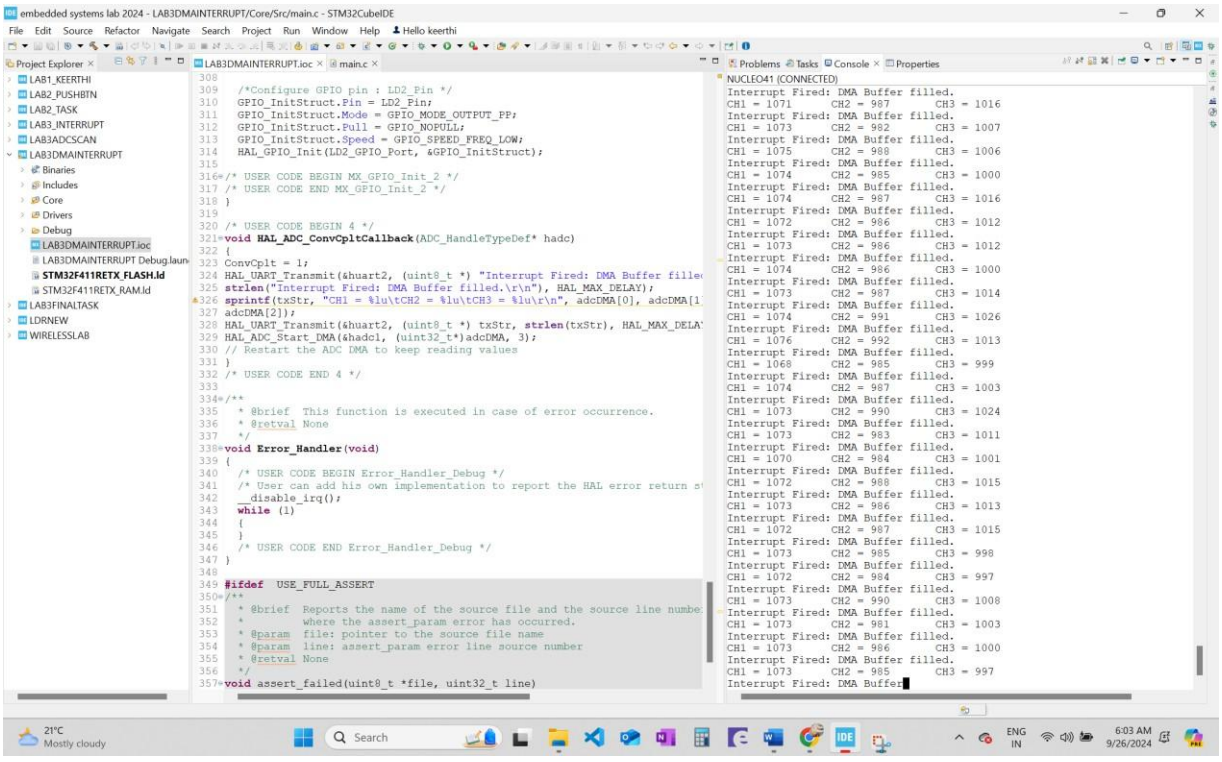
- After successfully compiling, in the console window to display the values on the serial monitor. I have selected the command shell console and then selected the serial port and added new connection name (This connection name is named as NUCLEO41).



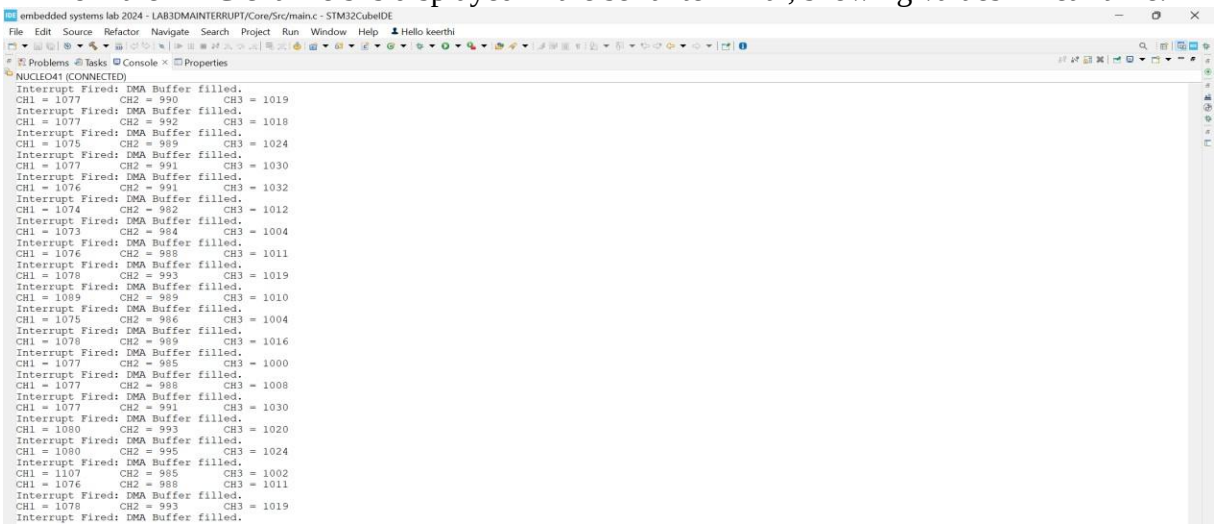


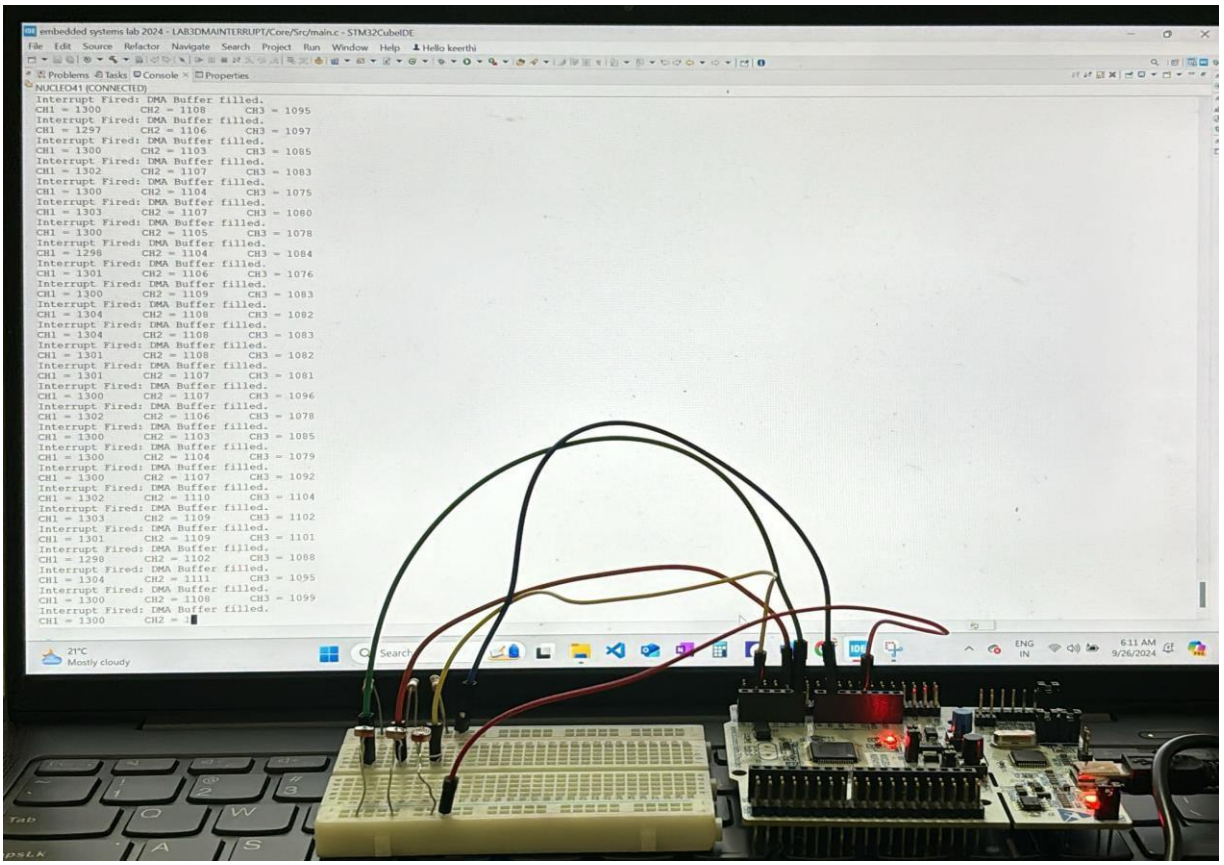


- The values appear on the serial monitor as soon as the remote connection is established successfully.



- The values that are displayed on serial monitor are the LDR values. I have **Enabled** the **scan conversion mode** and increased the **Number of conversions** to **3** and I have made channel 0, channel 1 and channel 4. These channel values resemble the LDR values. As the LDR is connected to ADC pin on board, ADC converts the analog value to digital values and display those digital values on the serial monitor. The values are constantly shown, and they are adjusted and mirrored on the serial monitor whenever the LDR is covered.Data from the ADC channels is displayed in the serial terminal, showing values in real-time.

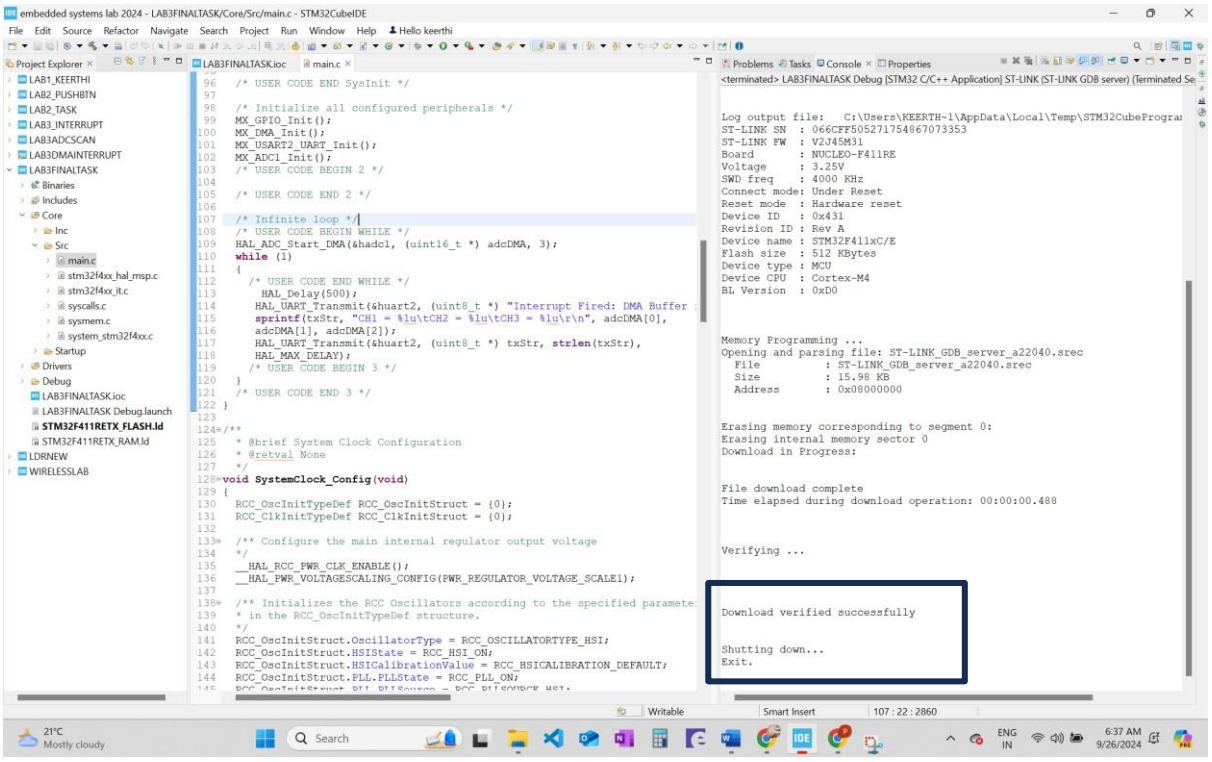


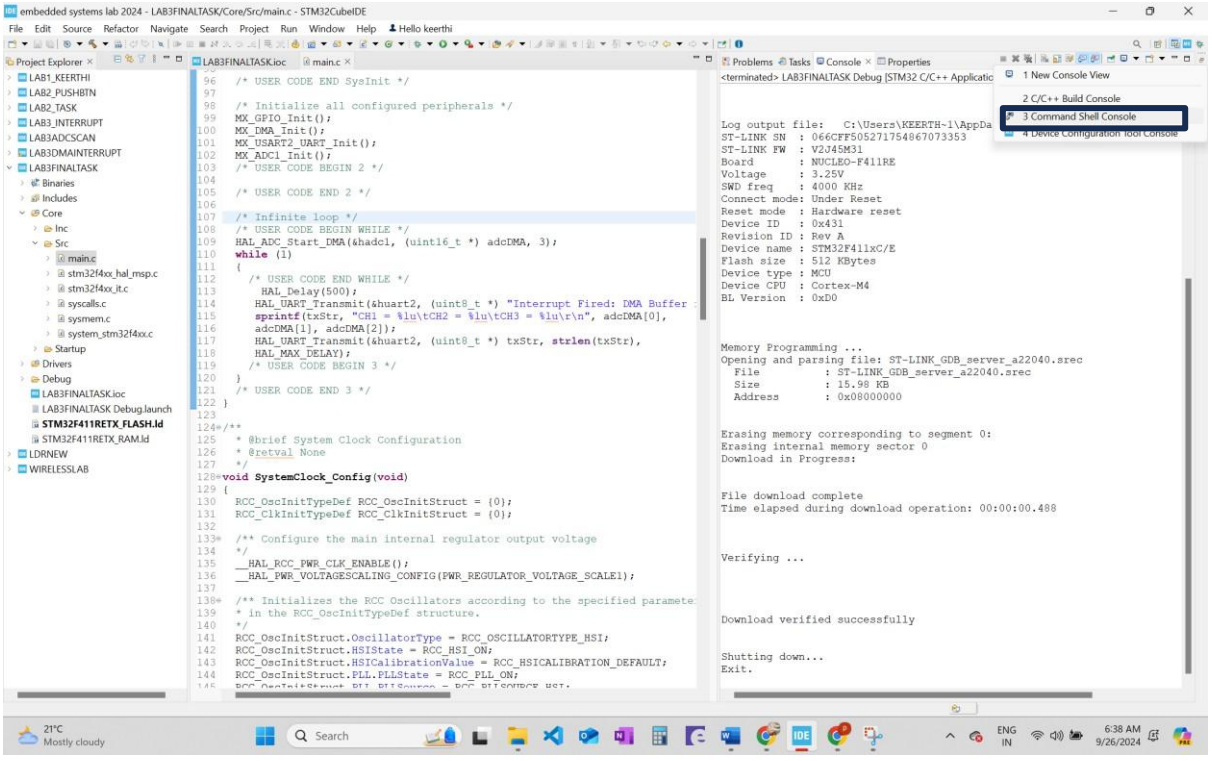


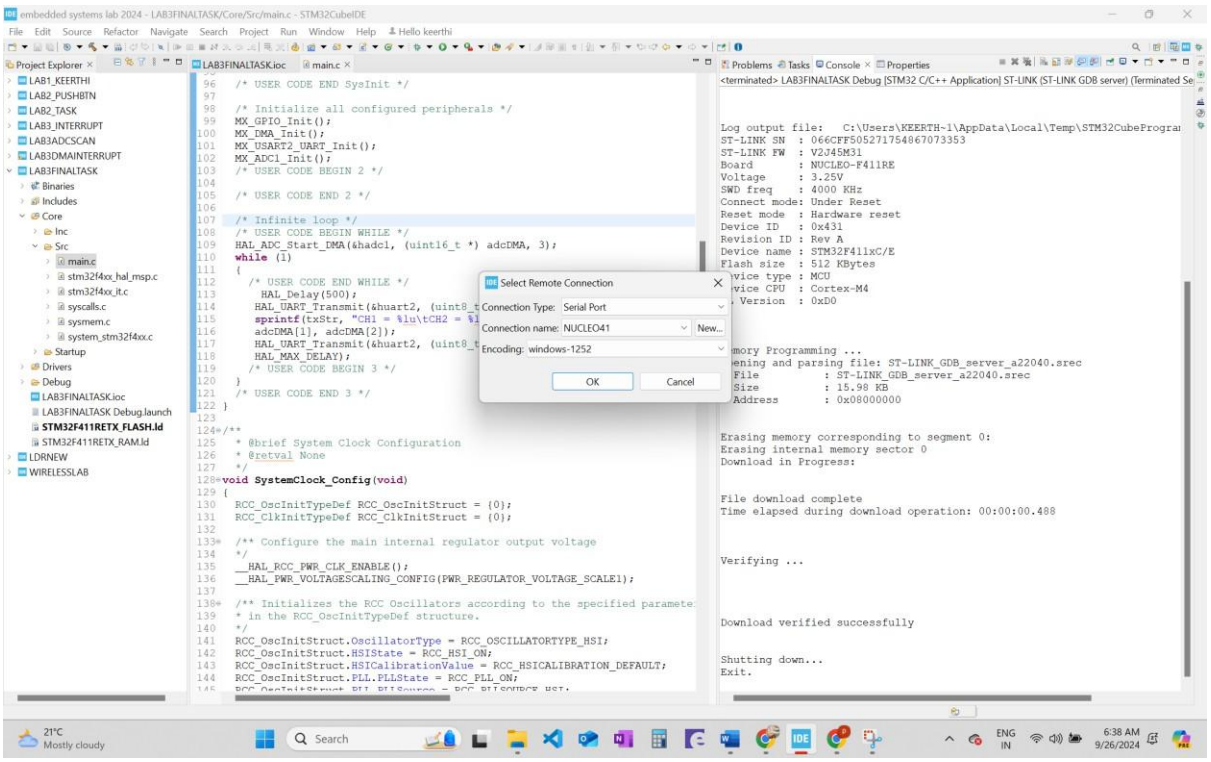
- A video demonstrating the configuration of the GPIO External Interrupt (EXTI) is provided below.
  https://drive.google.com/file/d/1x3q4iTDf7nkQmfFzUDWT_1YAHEf7H9zl/view?usp=drive_link

Part 4. ADC Continuous Mode with DMA (& Interrupt)

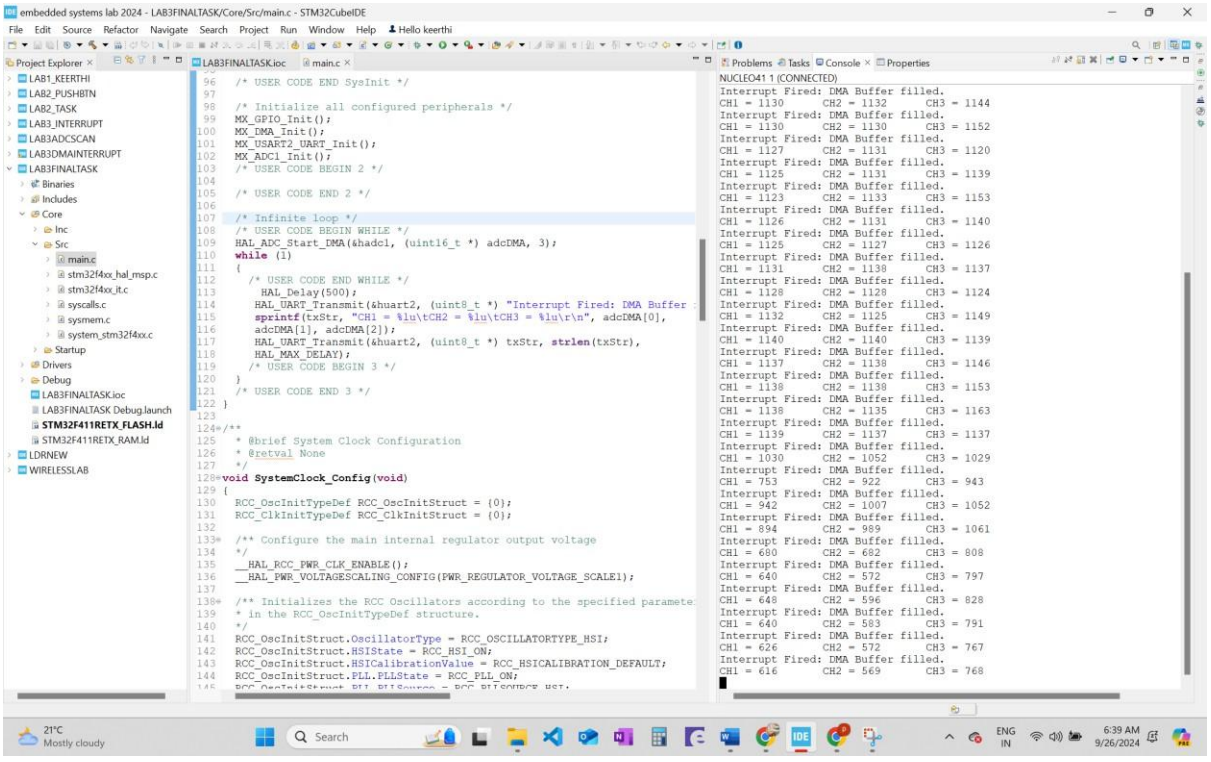
- After successfully compiling, in the console window to display the values on the serial monitor. I have selected the command shell console and then selected the serial port and added new connection name (This connection name is named as NUCLEO41).



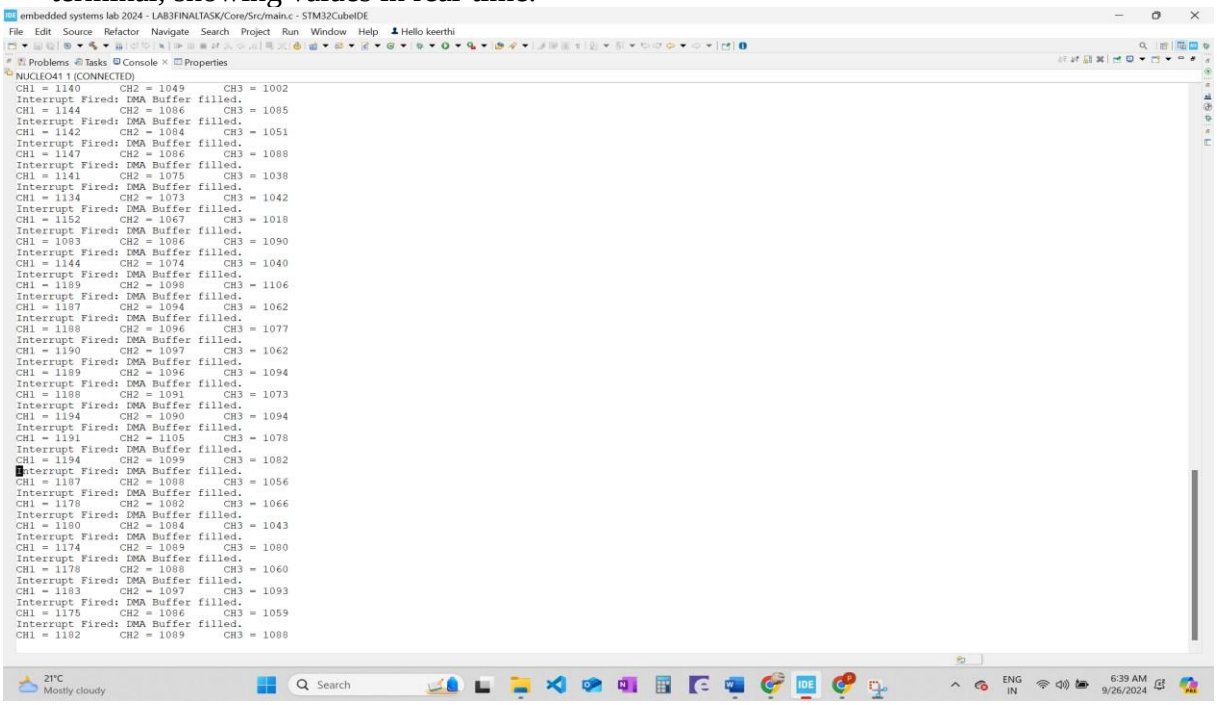


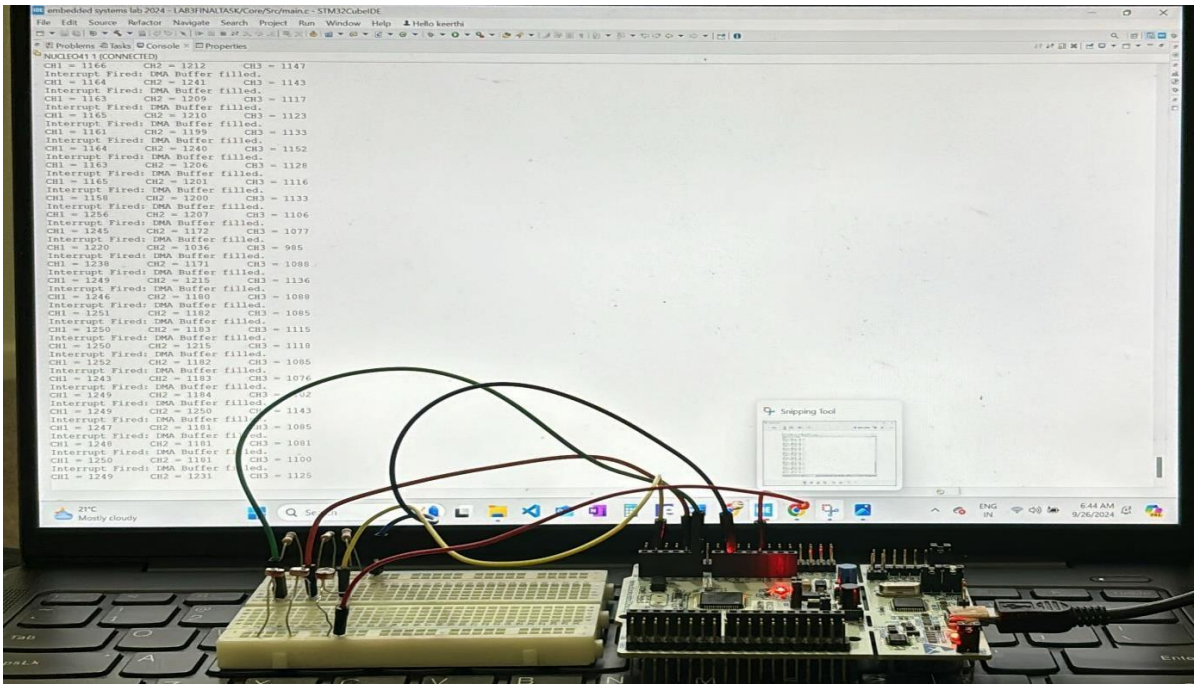


- The values appear on the serial monitor as soon as the remote connection is established successfully.



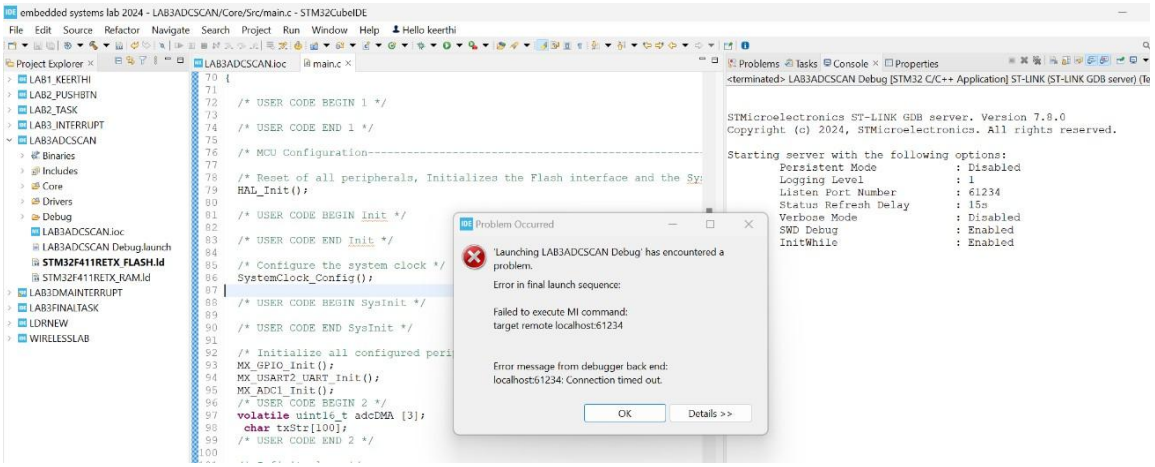
- The values that are displayed on serial monitor are the LDR values. I have **Enabled** the **continuous mode and DMA Continuous request Mode** and increased the **Number of conversions** to **3** and I have made channel 0, channel 1 and channel 4. These channel values resemble the LDR values. As the LDR is connected to ADC pin on board, ADC converts the analog value to digital values and display those digital values on the serial monitor. The values are constantly shown, and they are adjusted and mirrored on the serial monitor whenever the LDR is covered. Data from the ADC channels is displayed in the serial terminal, showing values in real-time.



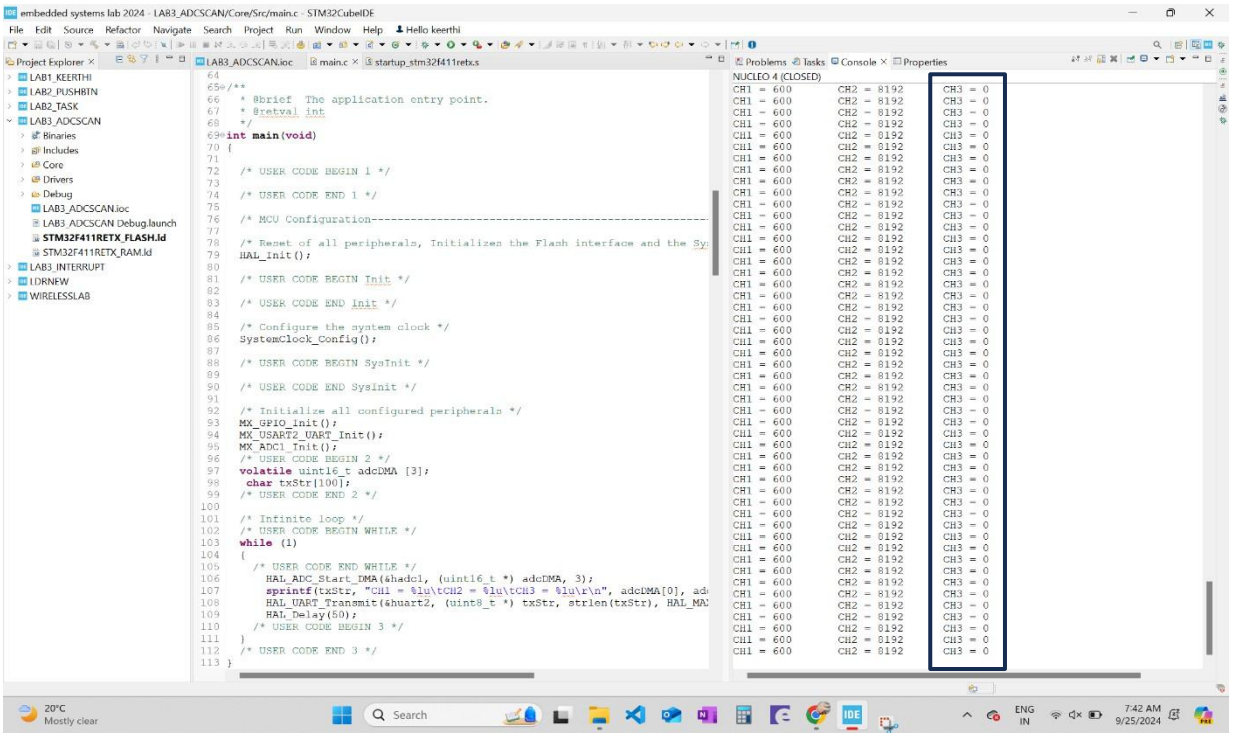


- A video demonstrating ADC Scan Mode with DMA (& Interrupt) can be found below.
https://drive.google.com/file/d/1sOf1NgYR3-ucHL-2H4emA4luLDapTFdR/view?usp=drive_link

Challenges and Solutions:

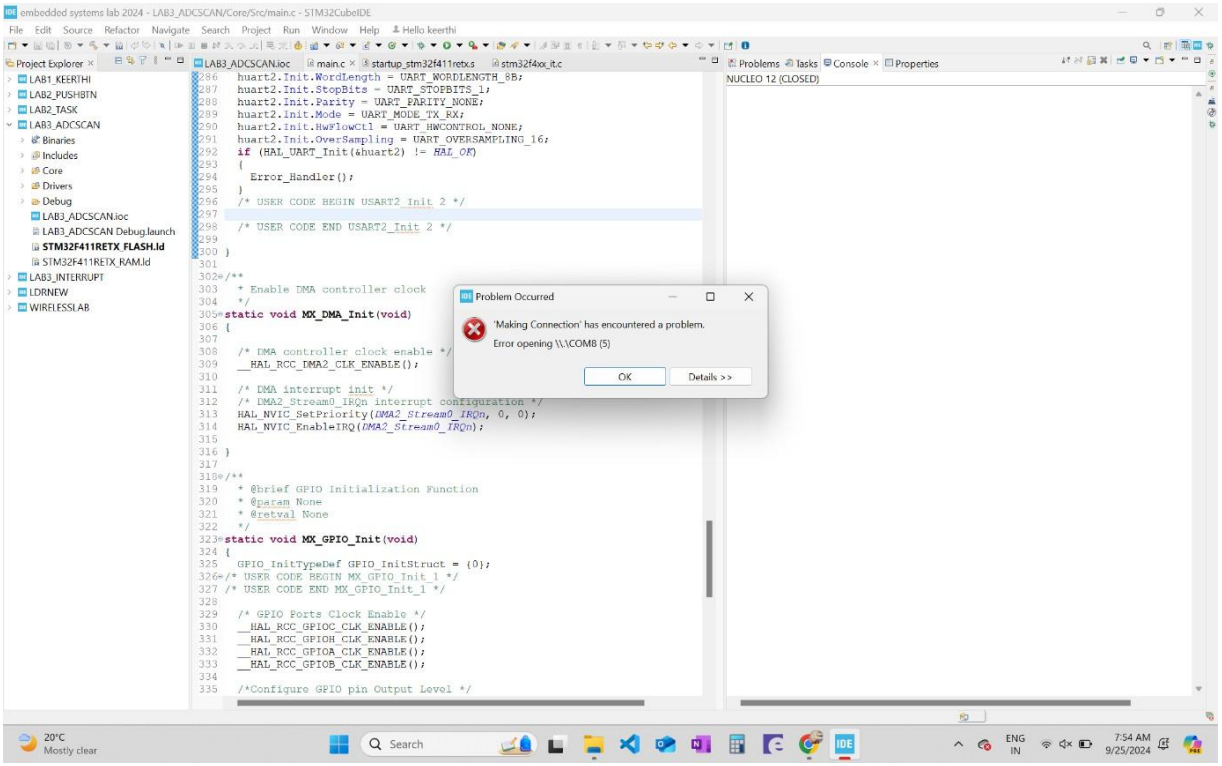


Challenge 1: The error "Failed to execute MI command: target remote localhost:61234" in STM32CubeIDE usually indicates a failure in connecting the debugger/ ST-LINK GDB.
Resolution 1: After changing the properties and fixing the STM- LINK GDB connection we got the proper result in the console window.



Challenge 2: After building the project 2 we have found that CH3 is always displaying 0.

Resolution 2: After doing the proper connections on board with LDR and rebuilding the project the result got displayed.



Challenge 3: We got this COMB error so many times because of the port connection with the windows.

Resolution 3: After changing the port connection settings in device manager of my laptop, the port connection error is solved.

Conclusion:

The experiments successfully illustrated the practical applications of interrupts and ADC configurations in the STM32 microcontroller environment. Working together as a group, we effectively set up an external interrupt on pin PC13, demonstrating how user inputs could initiate actions like toggling an LED without requiring constant monitoring in the main loop. This approach enhanced responsiveness and highlighted the benefits of using interrupt-driven programming in embedded systems.

Moreover, our exploration of ADC functionalities showcased the STM32's capability to handle multiple input channels efficiently. The incorporation of DMA in the later experiments underscored its significance in streamlining data transfer and processing, particularly for real-time applications. Throughout the process, we encountered challenges, such as configuring DMA settings and debugging UART communication, which enriched our hands-on experience. Overall, the insights gained reinforced the vital role of interrupts and DMA in improving system performance and set the stage for future projects that involve more complex sensor integrations and data management tasks.

Critical thinking

1. What is the difference between Lab Experiment 2 last week and Lab Experient 3 (this one), how are they different?

- In Lab 2, you set up a push button and one LDR (Light Dependent Resistor) as an interrupt to control a light. The interrupt method switched off the light exposing LDR senses an alteration of light intensity. It used interrupt-driven control, where the system responded to light changes sensed by the LDR, immediately. The push button was probably making manual control or reset this lab would be a primer to function bases on how interrupts are integrated in sensors with the STM32 Nucleo board.

  The complexity of the setup started with Lab 3, where a parallel circuit was used, and three channels in total using LDRs. The LDRs were likely made for interrupting but here they controlled the blinking of lights, possibly dependent on ambient light. Each was likely a single channel LDR, doing its thing on separate channels — potentially the same way using timers or PWM to blink away. By learning multi-channel control and parallel processing.

2. How is the LED being controlled without any input monitoring condition within the main function loop?

- The LED in the STM32F411RE can be controlled without any input monitoring in the main function loop by utilizing interrupts triggered by external devices like a push button and LDRs (Light Dependent Resistors). When the push button or LDR detects a change, an interrupt service routine (ISR) is triggered, which controls the LED based on the input event, without needing to constantly monitor the inputs in the main loop. Additionally, hardware timers can be used to manage LED behaviour, such as blinking, at regular intervals. This allows the push button and LDRs to function as independent triggers for the LED, bypassing the need for direct input monitoring in the main function loop.

3. What is the difference between External Interrupt Mode and External Event Mode?

- The External Interrupt Mode creates an interrupt request (IRQ) when it detects a specific change (like a rise or fall) in a signal on a chosen pin. On the other hand, the External Event Mode triggers an event without causing an interrupt, which means it can help synchronize tasks or events without stopping the current process to deal with a high-priority event.

4. What'd happen if you had chosen Pull-up in the GPIO settings?

- When the pull-up resistor is enabled, the GPIO pin will stay high (logic level 1) when the button is not pressed. Pressing the button connects the pin to ground (logic level 0). This configuration helps prevent floating pin states and ensures a clear logic level when the button is not active.

5. What are the use cases of ADC Scan Mode?

- ADC Scan Mode is useful for applications that need to monitor several analog signals at the same time. It's great for systems with multiple sensors, like environmental monitoring, where different factors such as temperature, humidity, and light must be tracked together. In industrial control, it allows efficient sampling of various signals like pressure and flow rates, improving process management. It’s also important in medical devices that track multiple physiological signals for a complete health overview. Other applications include data acquisition systems that require quick sampling from several inputs and consumer electronics that need smooth interaction with various analog controls. Overall, ADC Scan Mode makes it easier to collect and process data from multiple sources efficiently.

6. Unlike previous experiments, why was HAL_ADC_Start_DMA **function never called after starting ADC in the final experiment?**

- HAL_ADC_Start_DMA is called just once because the ADC is in continuous mode, which lets it keep reading signals without needing a restart. This first call also sets up the DMA (Direct Memory Access) to automatically transfer data from the ADC to memory. Once set up, the DMA handles the data transfer on its own. This makes the system more efficient, allowing the CPU to work on other tasks or enter low-power mode without needing to keep starting the ADC. The system also uses interrupts to process the data only when it’s ready, ensuring everything runs smoothly without extra calls to start the ADC again.