

LAB-5

Timers

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-5

Timers

Introduction:

The STM32 microcontroller's timers are configured and used in this lab for activities like PWM signal synthesis, phase-shifted PWM generating, and timer-triggered ADC acquisitions. For accurate control of features like waveform generation and data gathering, timers are essential in embedded systems. Phase-shifted PWM, which enables the synchronization of numerous PWM signals—essential for multiphase converters and inverters—is explored after we begin with basic PWM signal production using STM32CubeIDE for applications like LED blinking.

In order to allow the microcontroller to function without continuous CPU involvement, we also set up a timer to produce interrupts at predetermined intervals, namely every 1.5 seconds. In order to ensure precise and consistent data sampling—which is crucial for applications like sensor monitoring—this interrupt is then used to initiate ADC acquisitions. We hope to learn more about the many timer configurations and uses in real-time embedded systems through these investigations.

Materials:

- Software tools: STM32CubeIDE
- Hardware tools: STM32 Nucleo Board

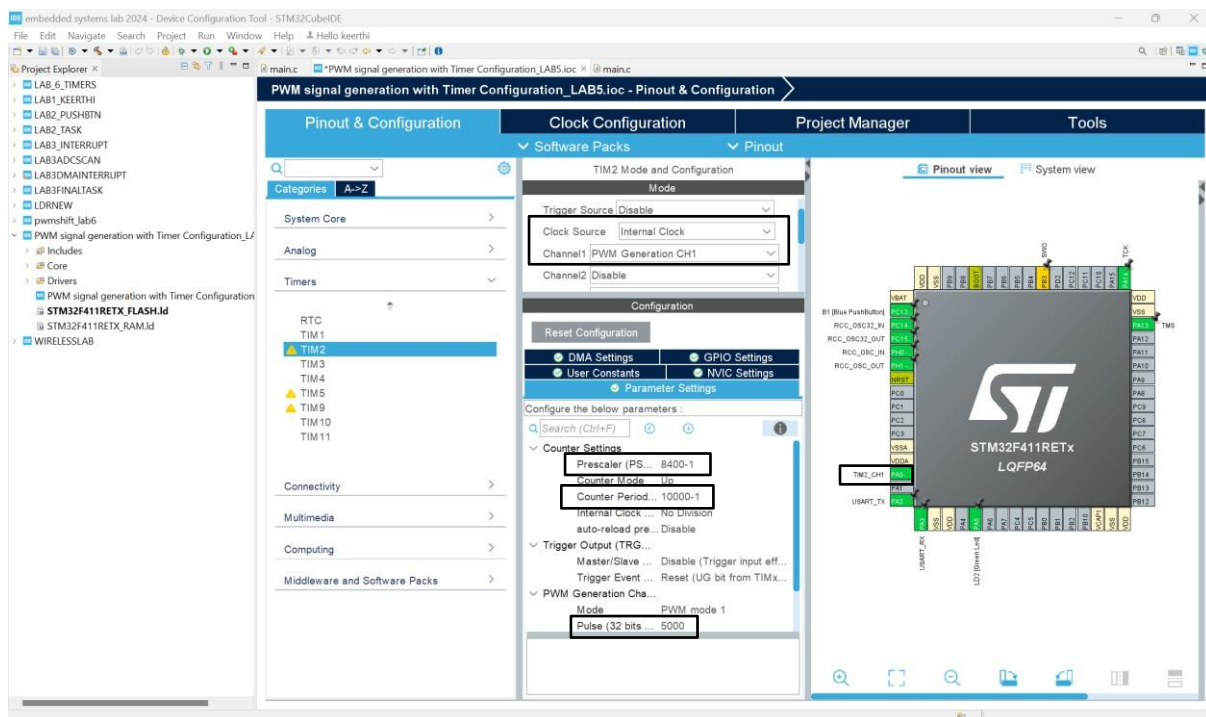
Experiment 1 – PWM signal generation with Timer Configuration

Procedure:

- **Configure Clock Source:** The **Clock Source** is set to be **Internal Source**.
- **Channel 1 as PWM Generation CH1:** we are enabling PWM on Channel 1.
- **Set Prescaler and Counter Period:**

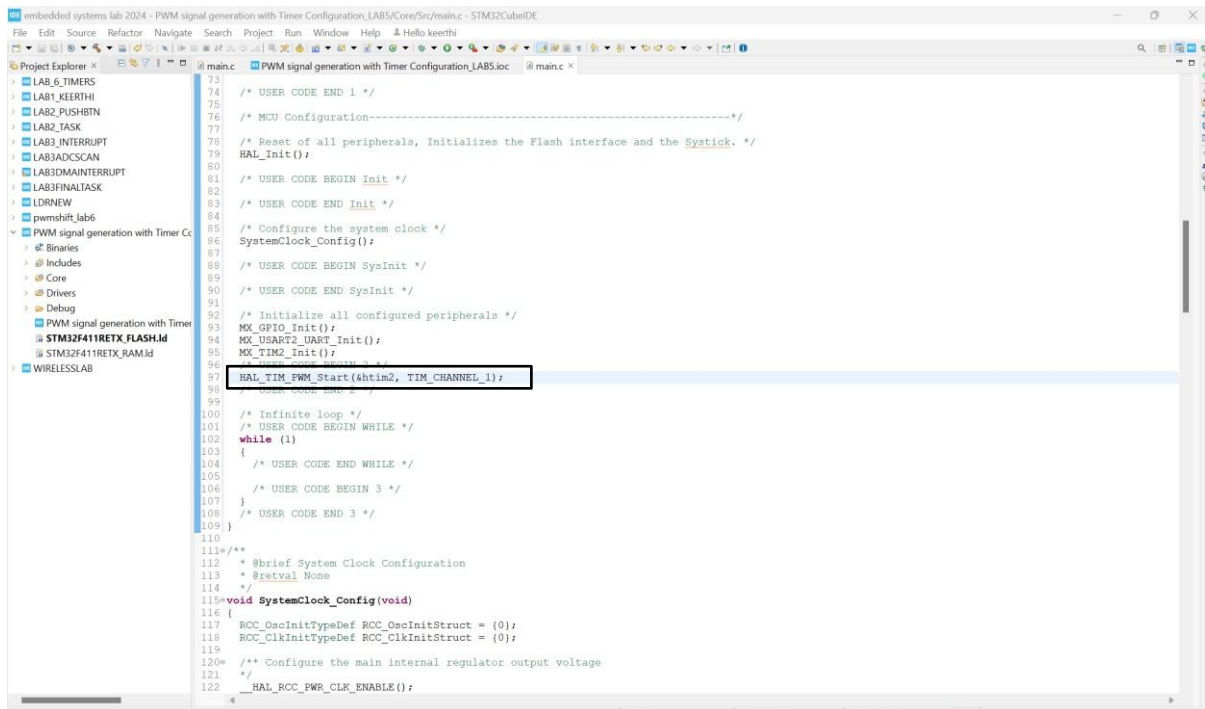
In the **Configuration** tab (under Timer settings TM2), set the **Prescaler** to 8400 - 1 (which is 8399) and the **Counter Period** to 10000 - 1 (which is 9999). This will set up the timer to count from 0 to 9999 with a clock frequency divided by 8400.

- We have Set the **Pulse Width** (for Channel 1) to 5000. This sets the duty cycle to 50%.



- So, automatically initializes and starts PWM on PA0 after configuring the timer settings (TIM2_CH1)

➤ Code Review:



```
73
74 /* USER CODE END 1 */
75
76 /* MCU Configuration-----*/
77
78 /* Reset of all peripherals, Initializes the Flash interface and the Systick. */
79 HAL_Init();
80
81 /* USER CODE BEGIN Init */
82
83 /* USER CODE END Init */
84
85 /* Configure the system clock */
86 SystemClock_Config();
87
88 /* USER CODE BEGIN SysInit */
89
90 /* USER CODE END SysInit */
91
92 /* Initialize all configured peripherals */
93 MX_GPIO_Init();
94 MX_USART2_UART_Init();
95 MX_TIM2_Init();
96
97 HAL_TIM_PWM_Start(&htim2, TIM_CHANNEL_1);
98
99
100 /* Infinite loop */
101 /* USER CODE BEGIN WHILE */
102 while (1)
103 {
104     /* USER CODE END WHILE */
105
106     /* USER CODE BEGIN 3 */
107 }
108 /* USER CODE END 3 */
109
110
111 /**
112  * @brief System Clock Configuration
113  * @retval None
114  */
115 void SystemClock_Config(void)
116 {
117     RCC_OscInitTypeDef RCC_OscInitStruct = {0};
118     RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};
119
120     /** Configure the main internal regulator output voltage
121     */
122     HAL_RCC_PWR_CLK_ENABLE();
```

- The code “**HAL_TIM_PWM_Start(&htim2, TIM_CHANNEL_1);**” starts PWM generation on Channel 1 of Timer 2. This function uses the timer settings as configured:

Clock Source: Internal

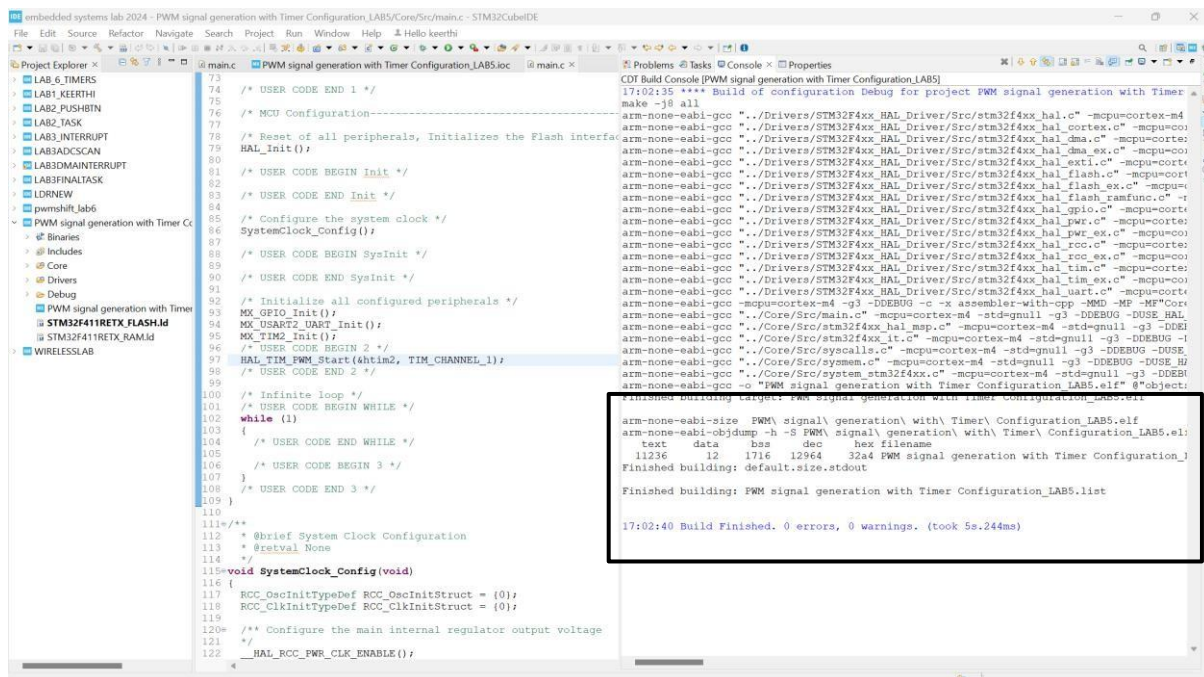
Prescaler: Divides the timer clock by 8400

Counter Period: Counts from 0 to 9999, defining the PWM frequency

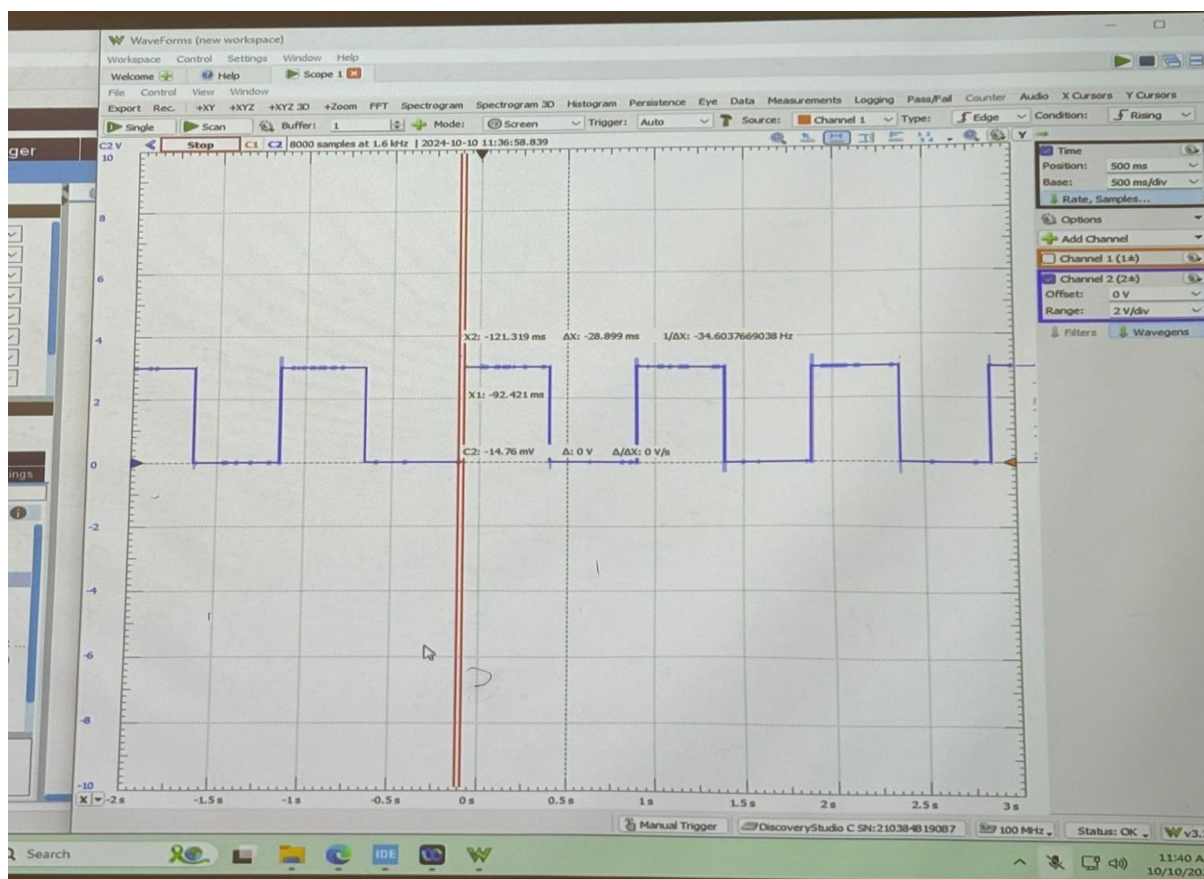
Pulse Width: High for half of the period (50% duty cycle)

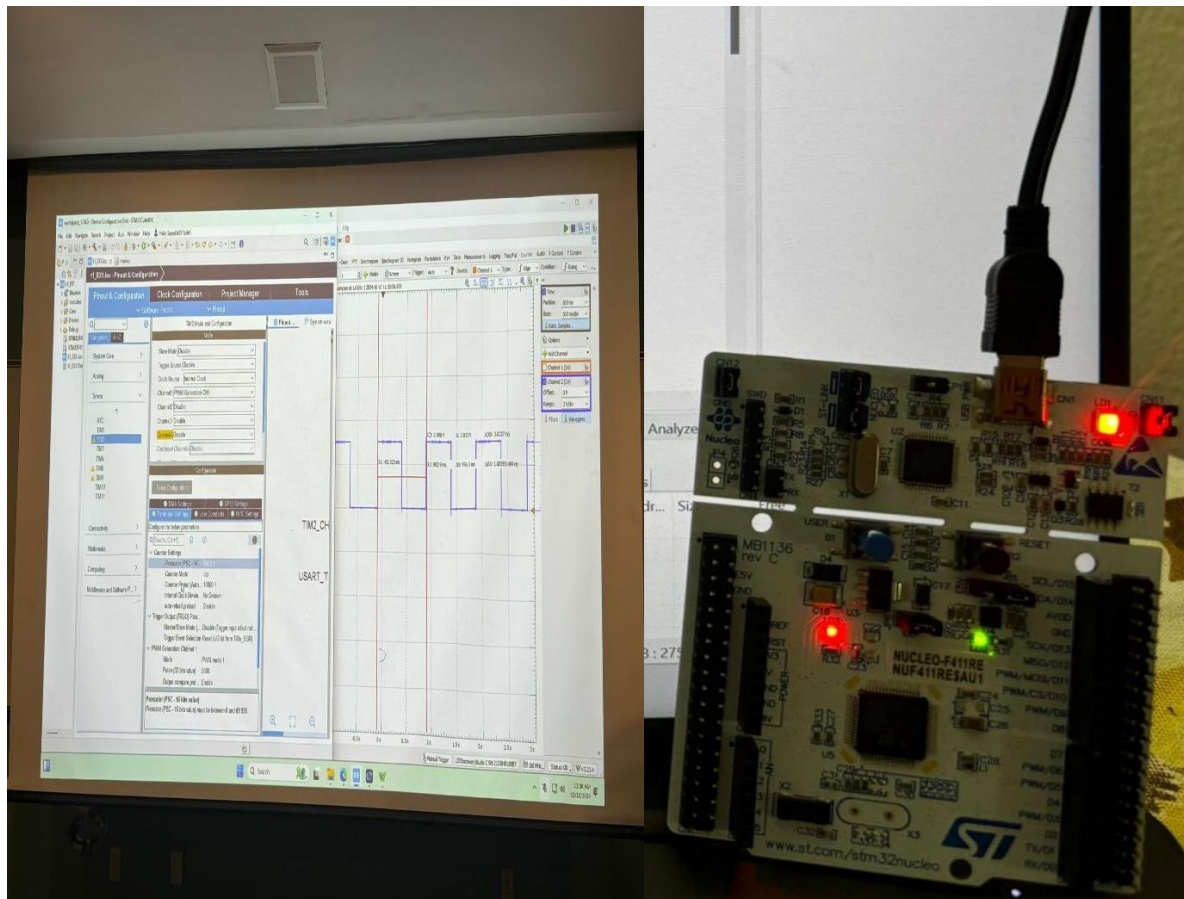
When this line is executed in your main loop, it begins outputting a PWM signal on the specified pin with the defined characteristics.

Results:



- Once after successfully compiling the code, PWM wave is generated. The output will be a square wave that toggles between high and low states at the calculated frequency, with a 50% duty cycle.

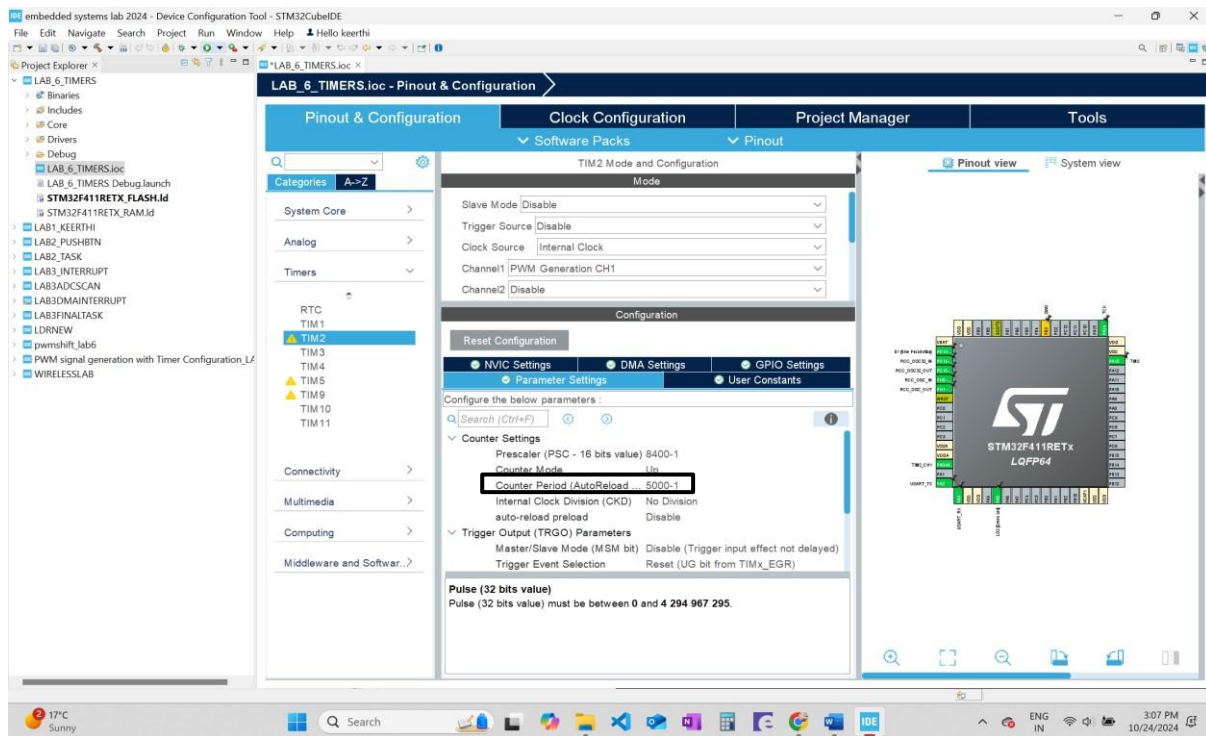




Observation of User LED:

I observed the user LED after I had compiled and uploaded the code to Nucleo Board. The LED ought to be continuously illuminated at half brightness with a 50% duty cycle.

New Configuration of Counter Period:



when I set the Counter Period to 5000-1, it produced a higher frequency PWM signal. As a result, the LED seemed to be lighted at the same brightness level since it blinked more quickly while keeping its duty cycle at 50%. The flashing effect was more noticeable because the LED's blinking was substantially faster than it was at the previous level (10000 - 1). By experimenting with various Counter Period numbers, I was able to comprehend how they affect the brightness and blinking frequency of the LED. Changing the counter period shows good PWM control by influencing the frequency and hence the apparent LED brightness.

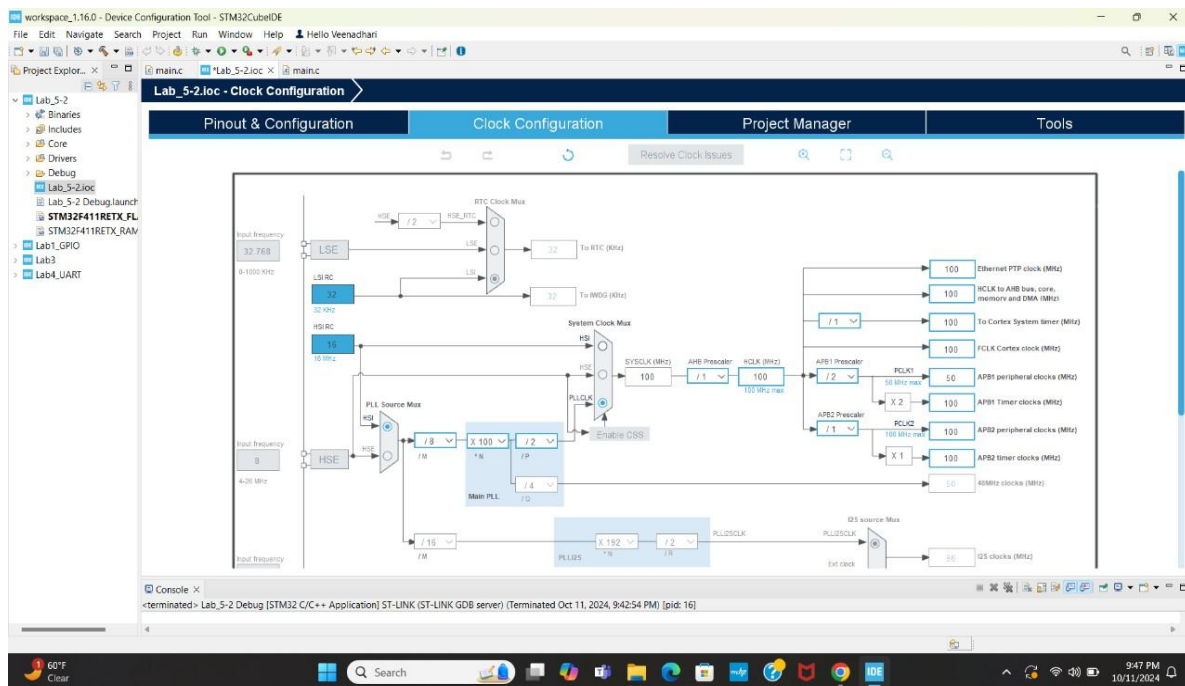
Experiment 2 – Phase shifted PWM Signal Generation

Procedure:

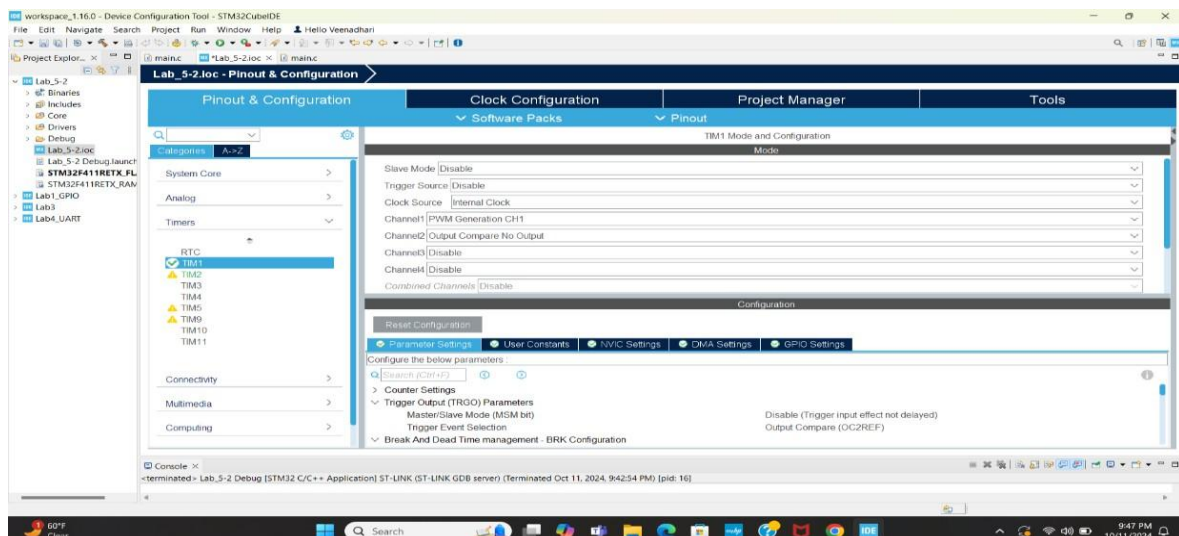
- **Set System Clock:** Configure the system clock to **100 MHz** to ensure proper timer operation.

Timer 1 Configuration:

- **Configure Clock Source for Timer 1:** Set the Clock Source to **Internal Clock**.



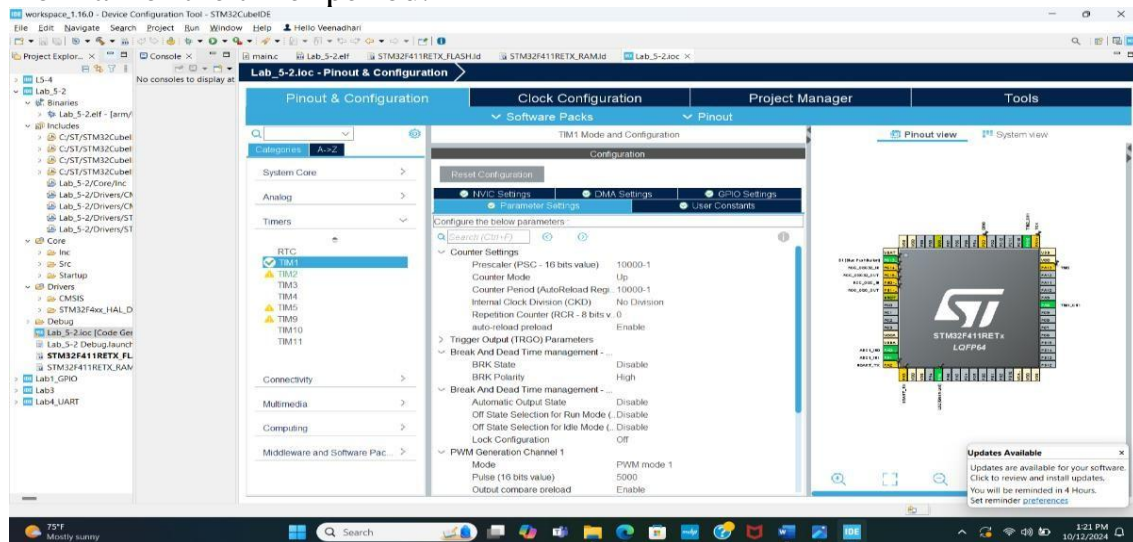
- **Channel Configuration for Timer 1:**
Enable **PWM Generation on Channel 1 (TIM1_CH1)**.
Set **Channel 2 (TIM1_CH2) to Output Compare No Output**.



➤ Set Prescaler and Counter Period for Timer 1:

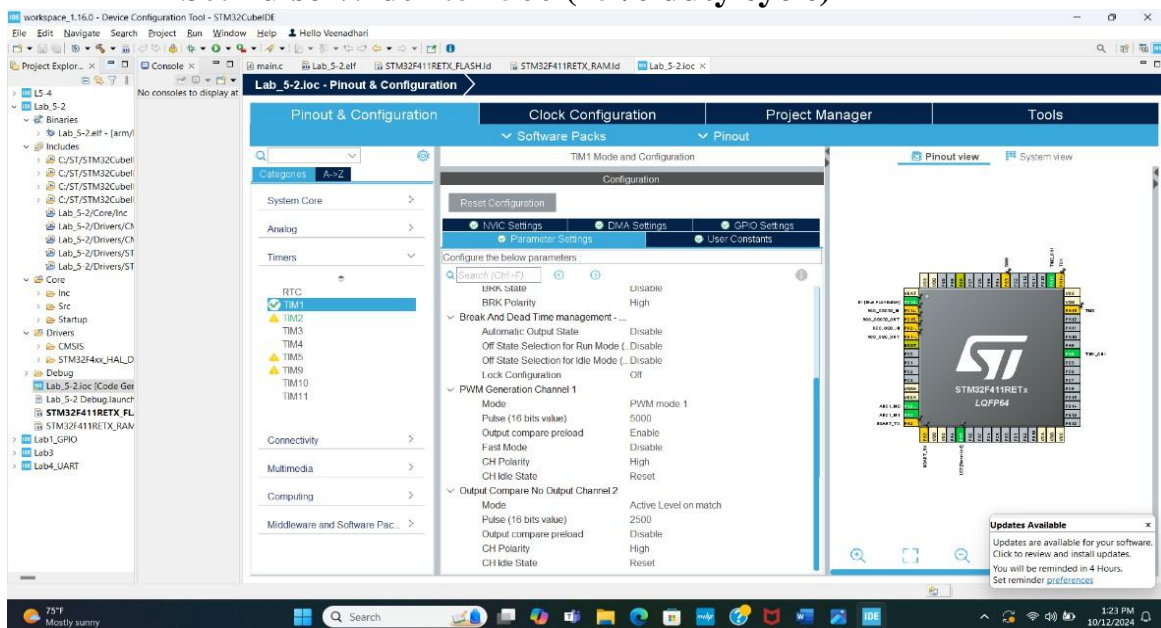
- Set **Prescaler (PSC)** to **10000 - 1 (9999)**.
- Set **Counter Period (AutoReload Register)** to **10000 - 1 (9999)**.
This configures the timer to count from 0 to 9999.

- We should **Enable Auto-reload Preload for Timer 1**, This option is enabled to allow automatic reloading of the counter value.
- **Trigger Output Parameters for Timer 1:** We should Set **Trigger Event Selection** to **Output Compare (OC2REF)**.
- Set **Pulse Width** to **5000 (50% duty cycle)**, meaning the signal will be high for half of the timer period.



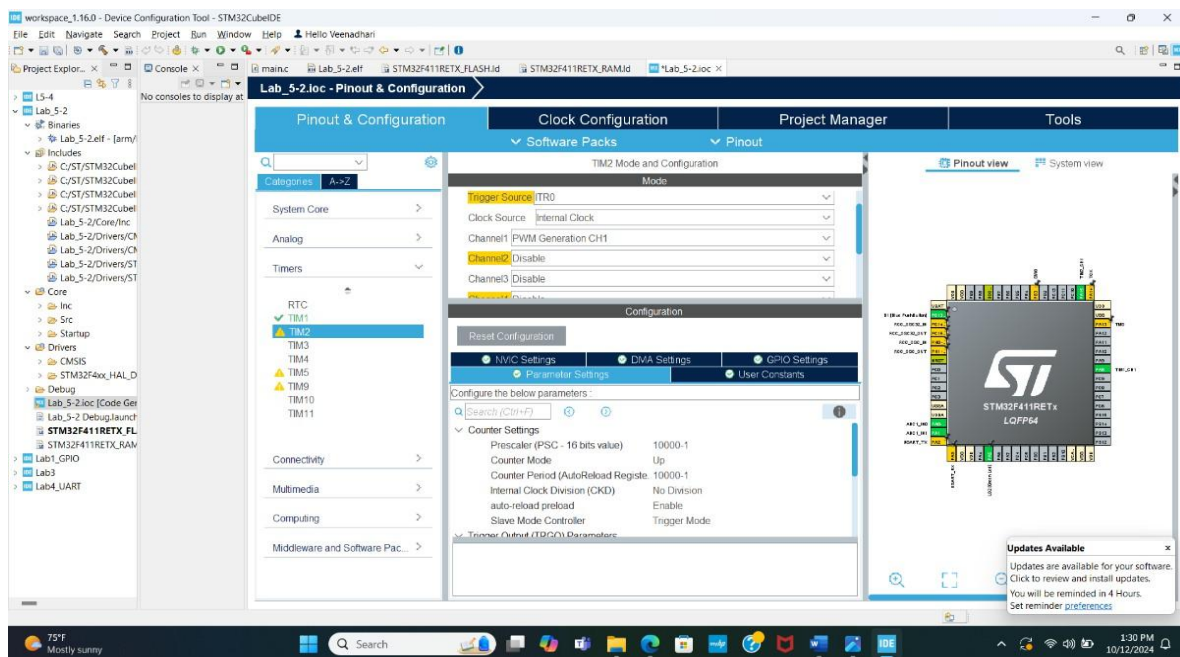
➤ Channel 2 Configuration:

- Set **Mode** to **Active Level on Match**.
- Set **Pulse Width** to **2500 (25% duty cycle)**

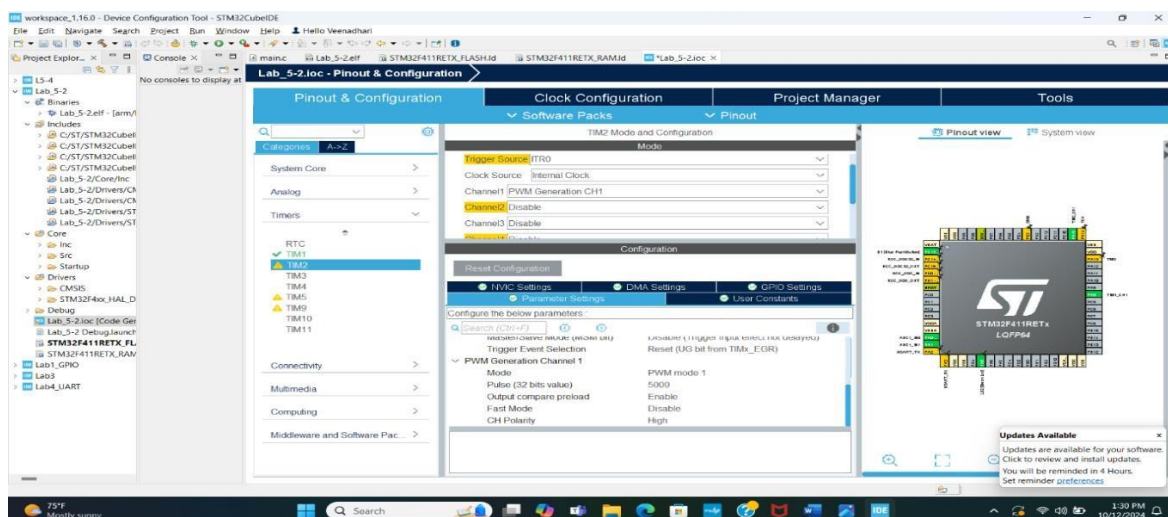


Timer 2 Configuration:

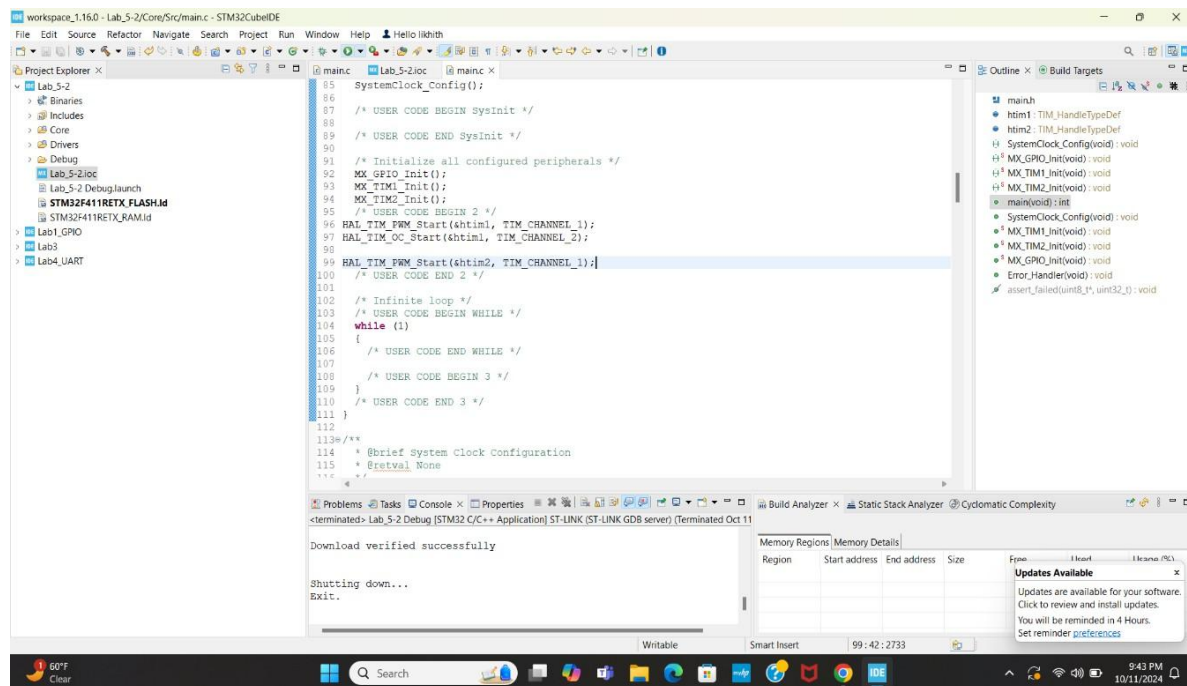
- **Slave Mode:** Set to **Trigger Mode**, allowing Timer 2 to respond to events from Timer 1.
- **Trigger Source:** Set to **ITRO** to utilize Timer 1's output as a trigger for Timer 2.
- **Clock Source:** Set to **Internal Clock**
- **Channel Configuration:** Enable **PWM Generation** on **Channel 1 (TIM2_CH1)** to generate a synchronized PWM signal.
- Set **Prescaler (PSC)** to **10000 - 1 (9999)**.
- Set **Counter Period** to **10000 - 1 (9999)**. This configures the timer to count from 0 to 9999.
- We should **Enable Auto-reload Preload**, this allows Timer 2 to continuously operate without interruption.



- We should set **Pulse Width** to **5000 (50% duty cycle)** to maintain synchronization Under PWM Generation Channel 1.



Code Review:



HAL_TIM_PWM_Start(&htim1, TIM_CHANNEL_1);

Function: Timer 1's Channel 1 PWM production is started by this function.

&htim1: This is a pointer to the timer handle structure of Timer 1, which contains all of the status information and configurable parameters.

TIM_CHANNEL_1: Signals that the PWM will be generated via Channel 1.

HAL_TIM_OC_Start(TIM_CHANNEL_2, &htim1);

Function: This function initiates the output comparison on Timer 1's Channel 2.

&htim1: This is the reference to Timer 1's timer handle once more.

TIM_CHANNEL_2: Indicates that Channel 2 will have the output compare function enabled.

HAL_TIM_PWM_Start(&htim2, TIM_CHANNEL_1);

Function: This function initiates the PWM generation on Timer 2's Channel 1.

&htim2: This is a reference to Timer 2's timer handle structure.

TIM_CHANNEL_1: Indicates that Timer 2's Channel 1 will provide the PWM signal.

The image shows a screenshot of the Visual Studio IDE interface. The main editor window displays a C++ source file named 'SystemClock_Config.cpp' with the following code:

```
85 SystemClock_Config():
86
87 /* USER CODE BEGIN SysInit */
88
89 /* USER CODE END SysInit */
90
91 /* Initialize all configured peripherals */
92 MX_GPIO_Init();
93 MX_TIM1_Init();
94 MX_TIM2_Init();
95 /* USER CODE BEGIN 2 */
96 HAL_TIM_PWM_Start(&htim1, TIM_CHANNEL_1);
97 HAL_TIM_OC_Start(&htim1, TIM_CHANNEL_2);
98
99 HAL_TIM_PWM_Start(&htim2, TIM_CHANNEL_1);
100 /* USER CODE END 2 */
101
102 /* Infinite loop */
103 /* USER CODE BEGIN WHILE */
104 while (1)
105 {
106 /* USER CODE END WHILE */
107
108 /* USER CODE BEGIN 3 */
109 }
110 /* USER CODE END 3 */
111 }
112
113 /**
114 * @brief System Clock Configuration
115 * @retval None
116 */
```

The Project Explorer on the left shows the file structure of the project, including 'Lab_5-2', 'Binaries', 'Includes', 'Core', 'Drivers', 'Debug', 'Lab_5-2\ioc', 'Lab_5-2 Debug\launch', 'STM32F411RETx_FLASH.ld', 'STM32F411RETx_RAM.ld', 'Lab1_GPIO', 'Lab3', and 'Lab4_UART'. The Output window at the bottom shows the following messages:

```
<terminated> Lab_5-2 Debug (STM32 C++ Application) ST-LINK (ST-LINK GDB server) (Terminated Oct 11)
Download verified successfully
Shutting down...
Exit.
```

The Memory Regions window is also visible, showing a table with columns: Region, Start address, End address, Size, Free, Used, and Usage (%). The table is currently empty. The status bar at the bottom shows the temperature as 60°F, the time as 9:49 PM, and the date as 10/11/2024.

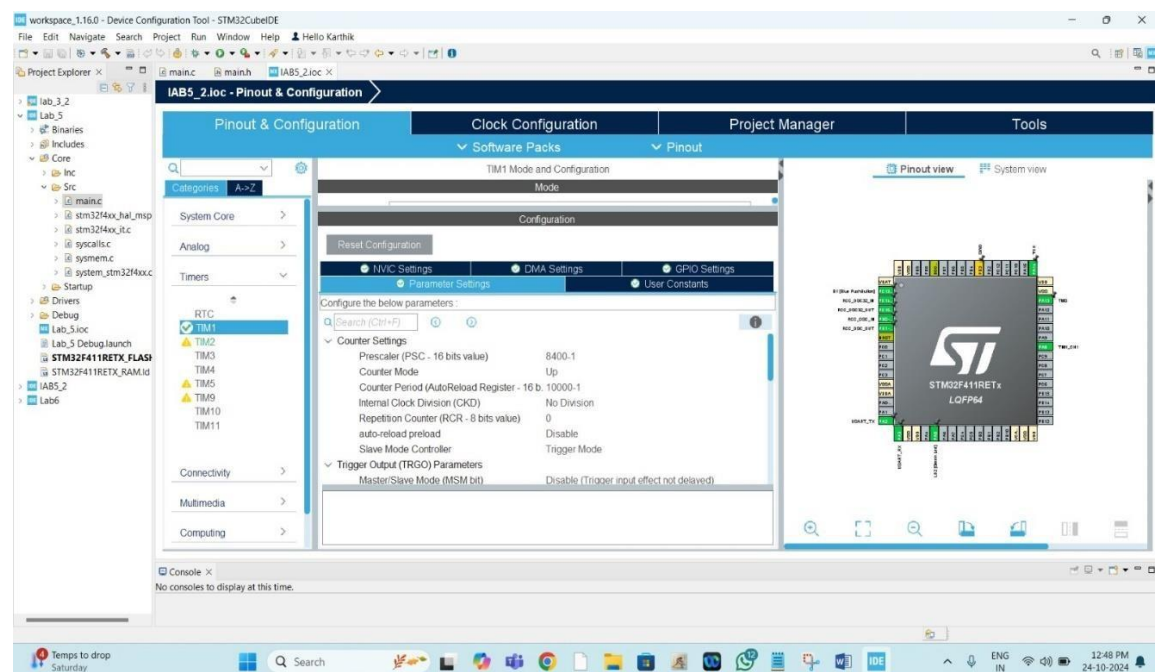
- The produced signals can be observed by connecting oscilloscope probes to the PA5 and PA8 pins. The two PWM signals with a 180 degree phase difference are shown on the oscilloscope.

Experiment 3 – Timer Interrupt

Procedure:

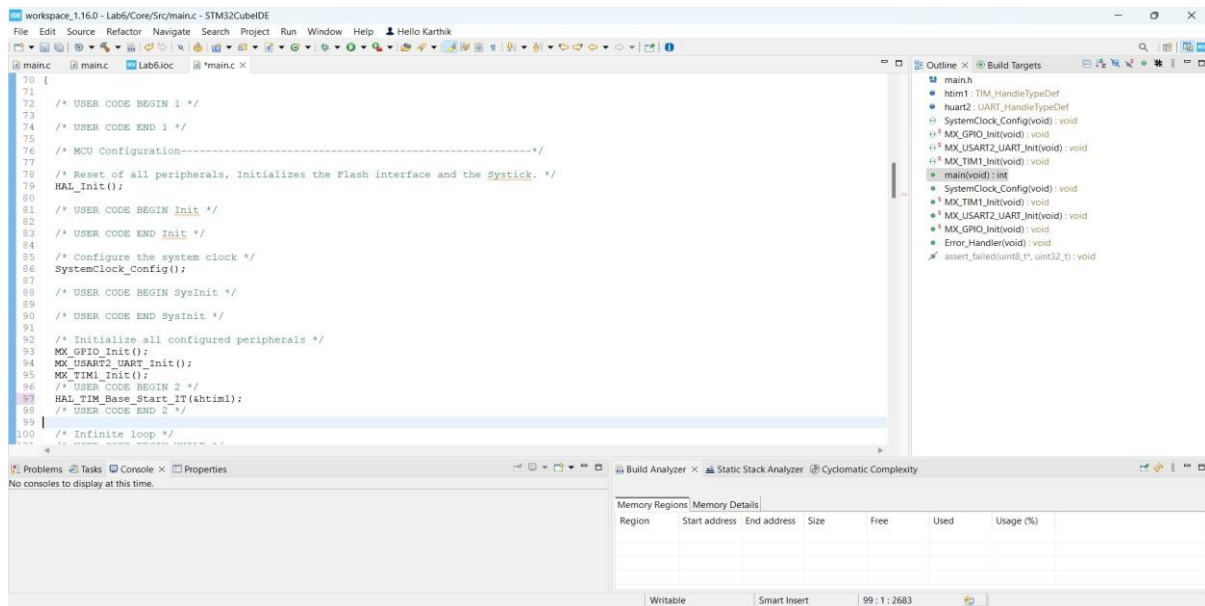
- **Configure Clock Source:** The **Clock Source** is set to be **Internal Source**.
- **Set Prescaler and Counter Period:**

In the **Configuration** tab (under Timer settings TM1), set the **Prescaler** to 8400 - 1 (which is 8399) and the **Counter Period** to 10000 - 1 (which is 9999). This will set up the timer to count from 0 to 9999.



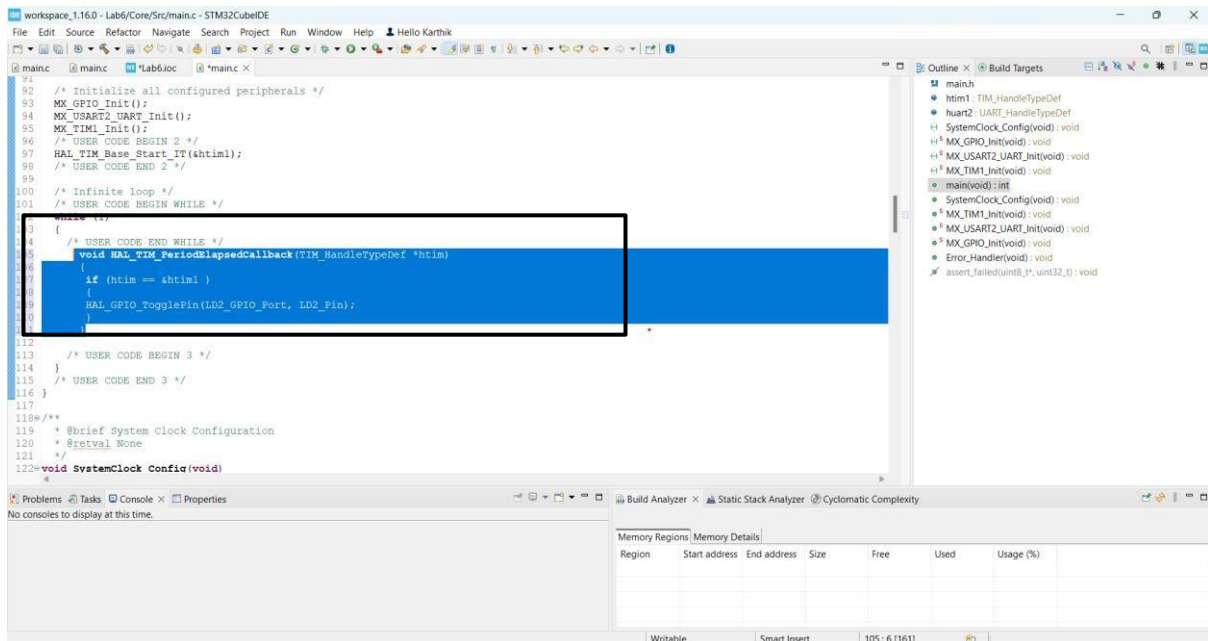
Code Review:

- The code **HAL_TIM_Base_Start_IT(&htim1);** starts Timer 1 (**htim1**) in interrupt mode using the STM32 HAL library.
- It enables periodic interrupts when the timer reaches its set period. The **&htim1** parameter is a pointer to the timer handle for Timer 1.
- An ISR will be triggered on each interrupt, allowing you to execute code at precise intervals.



- TIM1 update interrupt and TIM10 global interrupt should be **Enabled** and Preemption Priority changed to **1**.

Code for LED Blinking

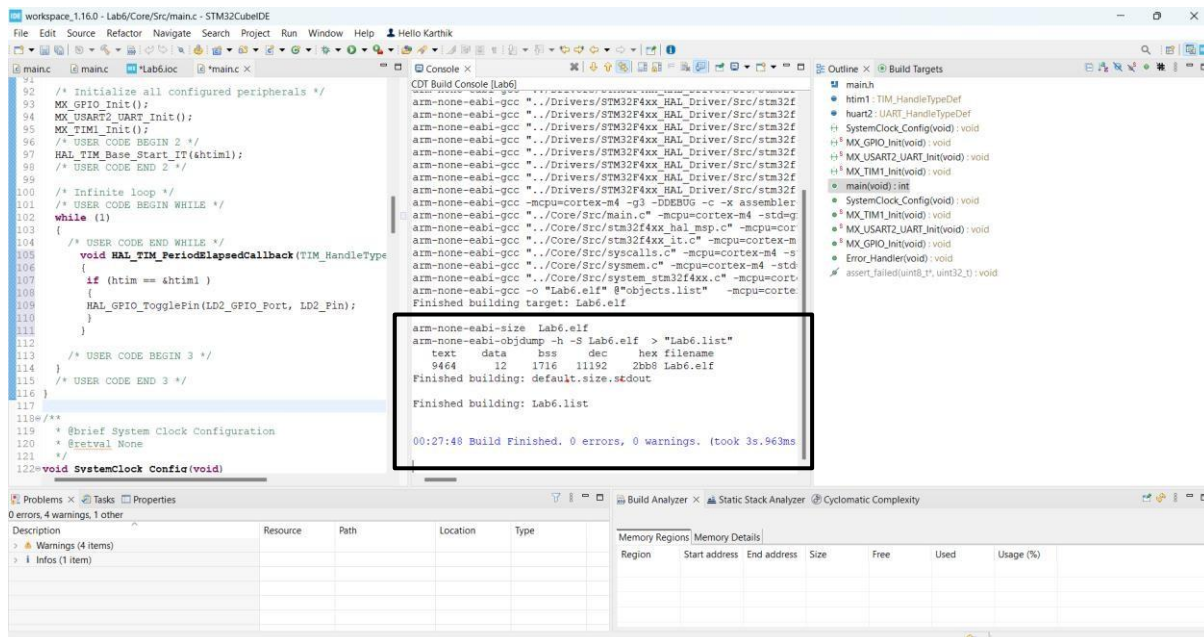


```
92 /* Initialize all configured peripherals */
93 MX_GPIO_Init();
94 MX_USART2_UART_Init();
95 MX_TIM1_Init();
96 /* USER CODE BEGIN 2 */
97 HAL_TIM_Base_Start_IT(&htim1);
98 /* USER CODE END 2 */
99
100 /* Infinite loop */
101 /* USER CODE BEGIN WHILE */
102 while (1)
103 {
104     /* USER CODE END WHILE */
105     void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
106     {
107         if (htim == &htim1)
108         {
109             HAL_GPIO_TogglePin(LD2_GPIO_Port, LD2_Pin);
110         }
111     }
112     /* USER CODE BEGIN 3 */
113 }
114 /* USER CODE END 3 */
115
116 /**
117  * @brief System Clock Configuration
118  * @retval None
119  */
120 void SystemClock_Config(void)
121 {
122     /* ... */
123 }
```

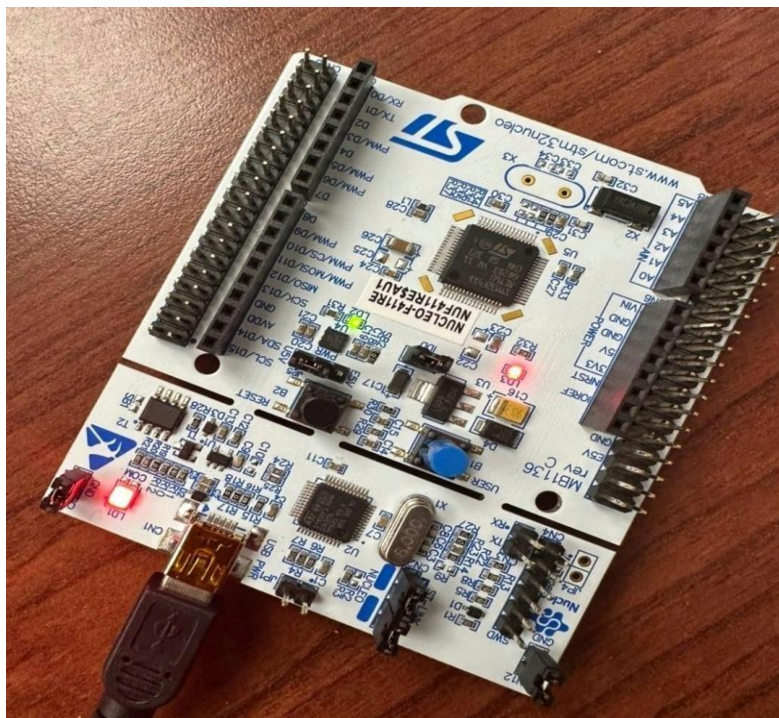
Region	Start address	End address	Size	Free	Used	Usage (%)
Writeable						
Smart Insert						
105 : 6 (161)						

- The **HAL_TIM_PeriodElapsedCallback()** function is triggered by a timer interrupt each time the timer period elapses.
 1. **Function Call:** The **TIM_HandleTypeDef *htim** parameter identifies the timer that caused the interrupt.
 2. **Timer Check:** It checks if **htim** corresponds to Timer 1 (**htim1**).
 3. **GPIO Toggle:** If true, **HAL_GPIO_TogglePin(LD2_GPIO_Port, LD2_Pin);** toggles the **LED (LD2)**, turning it on or off.
- This callback effectively blinks the **LD2** LED whenever Timer 1's period ends.

Result



- As a result of this code, the LED on the `LD2` pin will blink at intervals set by Timer 1. Each time the timer period elapses, the LED will toggle between on and off, creating a steady blinking effect. The blink rate is determined by the timer configuration.

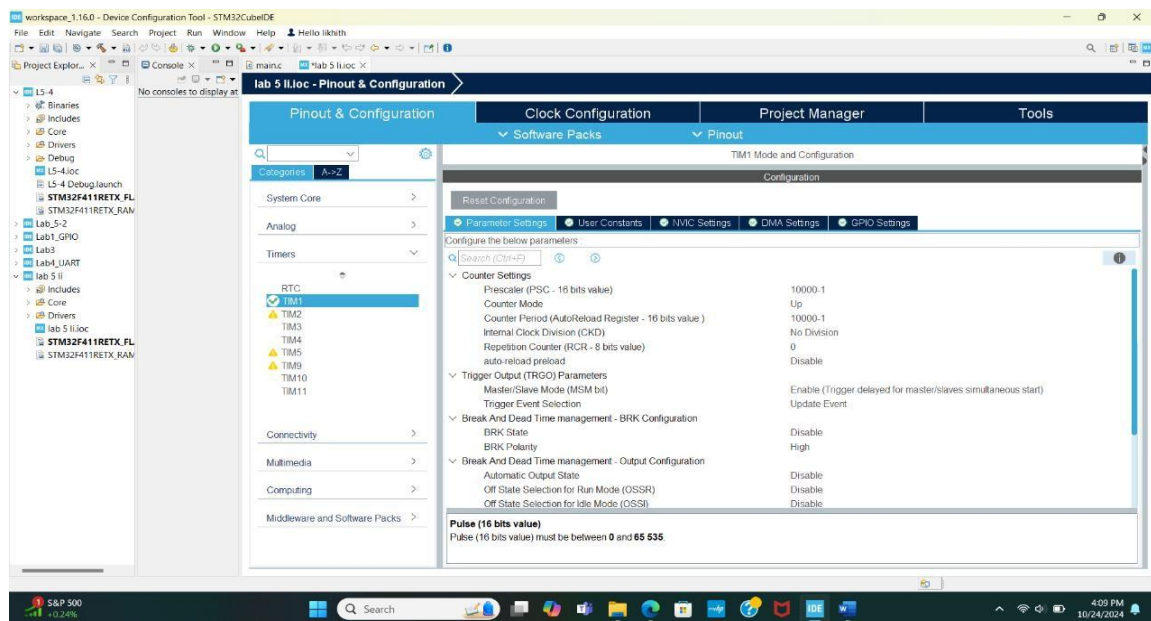
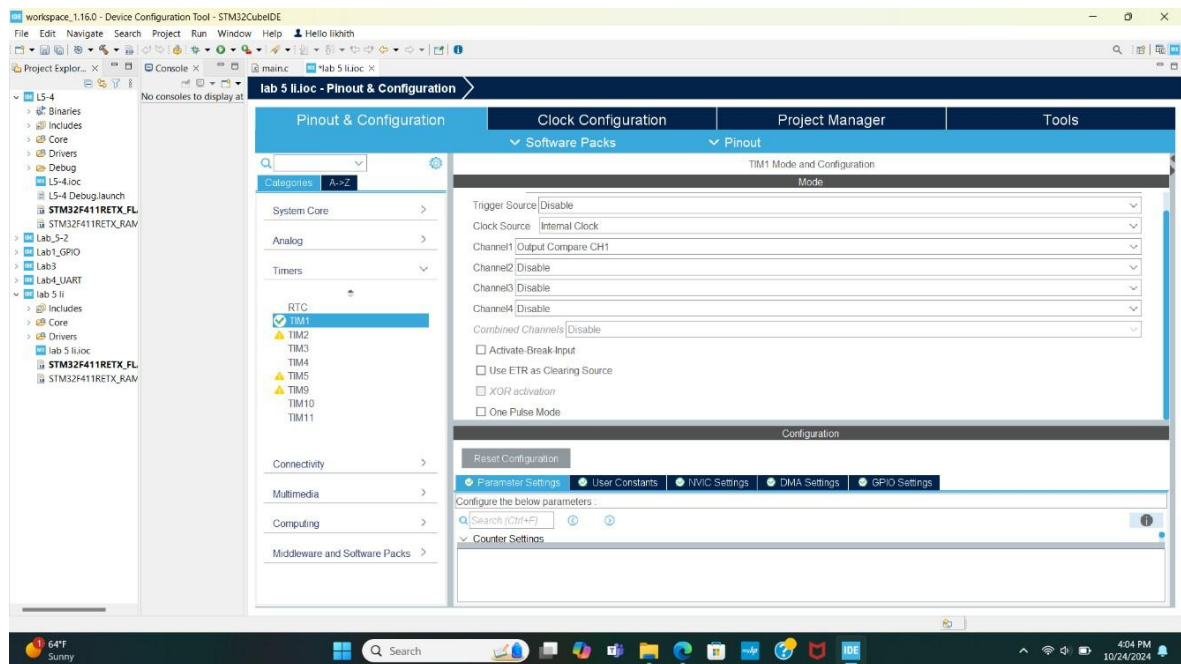


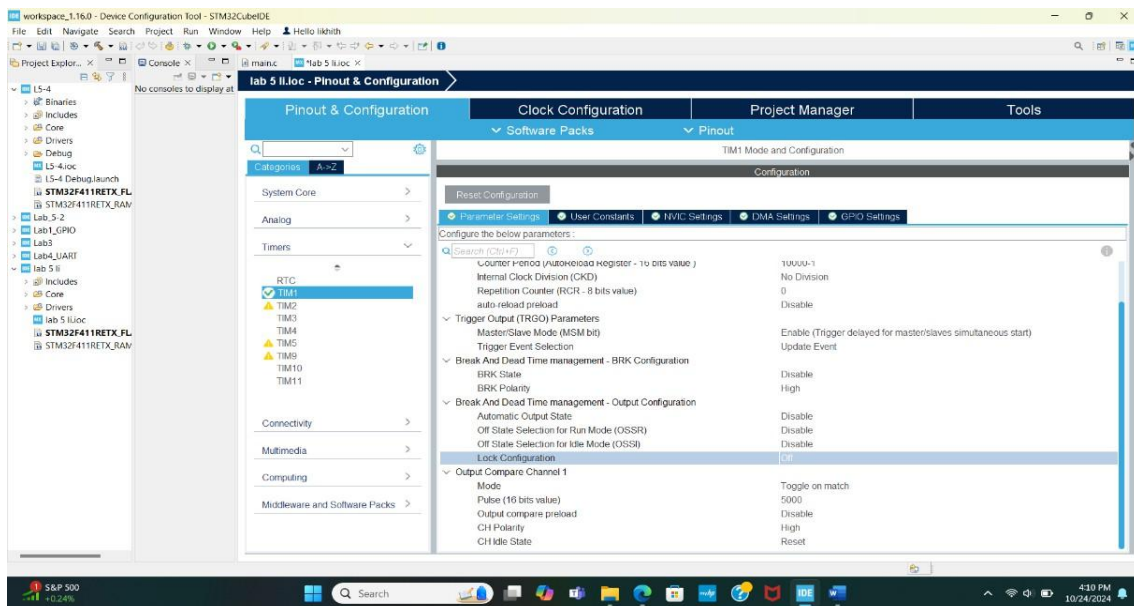
Experiment 4 – Timer Triggered Peripherals (ADC Injected Mode)

Procedure:

1. Project Setup:

- Open **STM32CubeIDE** and create a new STM32 project.
- Configure the project settings, ensuring the correct **clock settings** for your application.





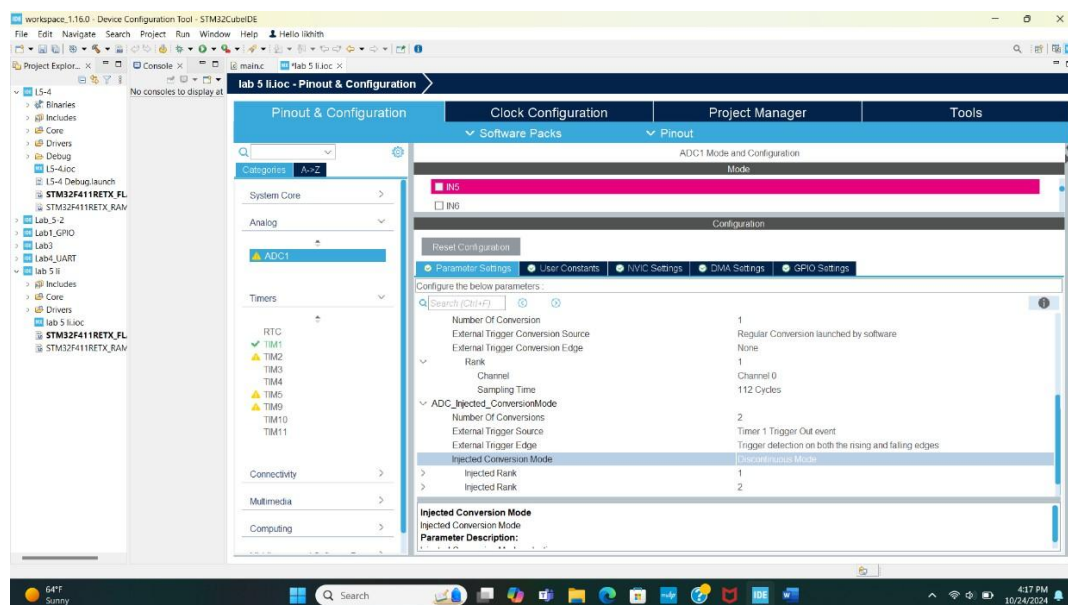
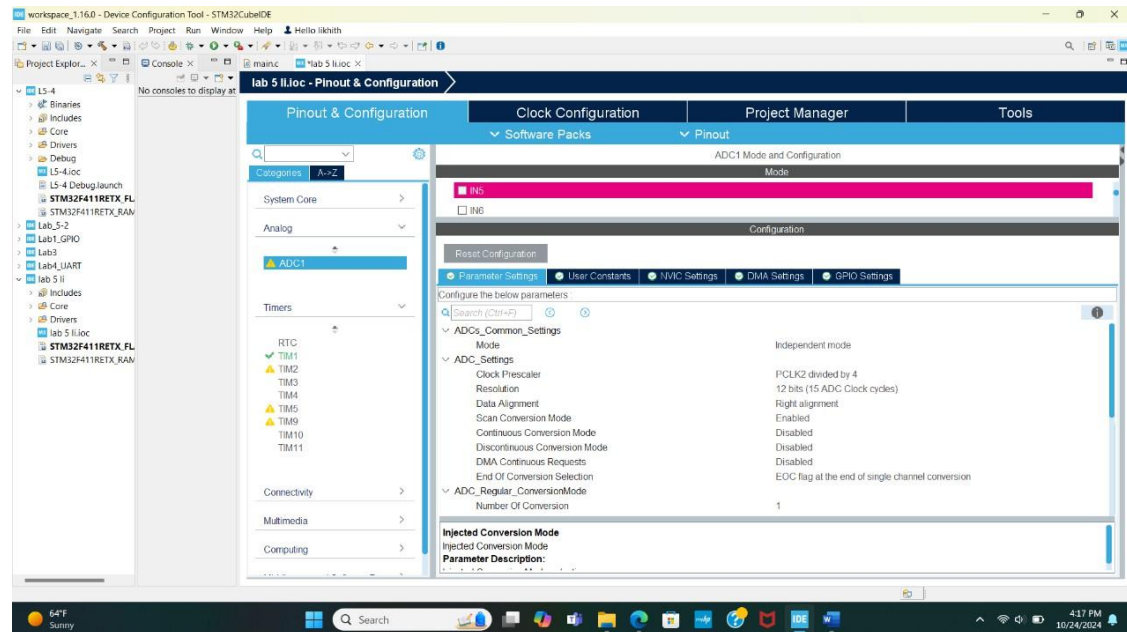
2. Configure TIM1:

- In the **Pinout & Configuration** tab, select **TIM1** and enable **Output Compare mode for Channel 1**.
- **Disable Slave Mode** and **Trigger Source** to prevent unwanted triggering during initial setup.
- Set the **Clock Source** to **Internal**, which uses the internal timer clock for operation.
- Configure the **Prescaler** to **9999** to scale down the timer clock, effectively slowing down the counting speed.
- Set the **Counter Period** to **9999**, allowing the timer to count from 0 to 9999, defining the frequency of the timer's operation.
- For the **Trigger Output (TRGO)** setting, choose **Toggle on match** to create a **toggle signal** when the timer reaches the defined count.

3. Configure ADC1:

- Navigate to the **ADC configuration** settings in STM32CubeIDE.
- Set the **Mode** to **Independent**, which allows ADC1 to operate independently of any other ADC.
- We should enable the **scan conversion mode** and **DMA continuous requests mode**.
- **ADC_Regular_conversion mode** : Set the **sampling time** to **112 cycles**.
- **Disable Scan Conversion Mode** to limit conversions to the selected channels only and **disable Continuous Conversion Mode** for manual control of conversion starts.

- For external trigger settings, we should specify that the **External Trigger Source is Timer 1 Trigger out event**, linking ADC conversions directly to TIM1 outputs and External trigger edge to trigger detection on both the rising and falling edges.
- Configure injected conversions for **Channel 0 and Channel 1**, with a Sampling Time of **56 cycles** each, to ensure accurate sampling.

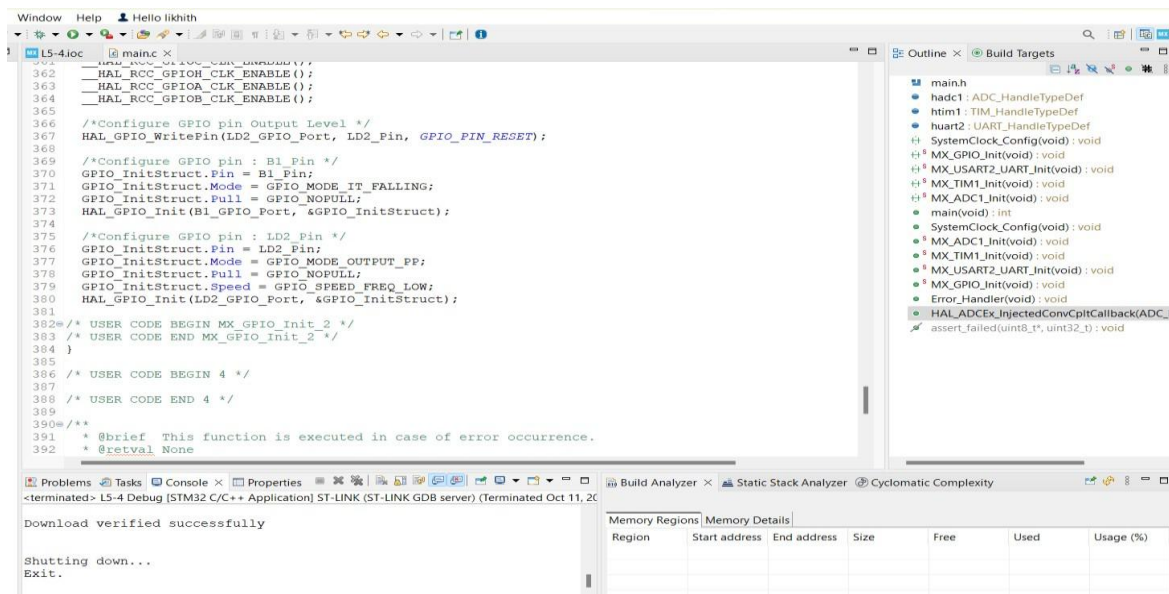



```

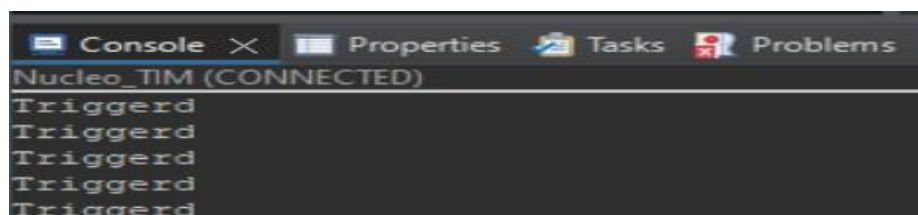
15-4ioc | @mainc x
379 GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
380 HAL_GPIO_Init(LD2_GPIO_Port, &GPIO_InitStruct);
381
382/* USER CODE BEGIN MX_GPIO_Init_2 */
383/* USER CODE END MX_GPIO_Init_2 */
384}
385
386/* USER CODE BEGIN 4 */
387
388/* USER CODE END 4 */
389
390/**
391 * @brief This function is executed in case of error occurrence.
392 * @retval None
393 */
394void Error_Handler(void)
395{
396 /* USER CODE BEGIN Error_Handler_Debug */
397 /* User can add his own implementation to report the HAL error return state */
398 __disable_irq();
399 while (1)
400 {
401 }
402 /* USER CODE END Error_Handler_Debug */
403}
404void HAL_ADCEx_InjectedConvCpltCallback(ADC_HandleTypeDef *hadc)
405{
406 HAL_UART_Transmit(&huart2, (uint8_t *)"Triggerd\n\r",
407 strlen("Triggerd\n\r"), HAL_MAX_DELAY);
408}
409
410#ifdef USE_FULL_ASSERT
411/**
412 * @brief Reports the name of the source file and the source line number
413 * where the assert param error has occurred.
414 * @param file: pointer to the source file name
415 * @param line: assert_param error line source number
416 * @retval None
417 */
418void assert_failed(uint8_t *file, uint32_t line)
419{
420 /* USER CODE BEGIN 6 */
421 /* User can add his own implementation to report the file name and line number,

```

Once the code compiles successfully, we have to upload it to the STM32 microcontroller using the debugger connected to the board.



By observing the frequency of the triggered messages in the monitor, we need to verify the ADC functionality.

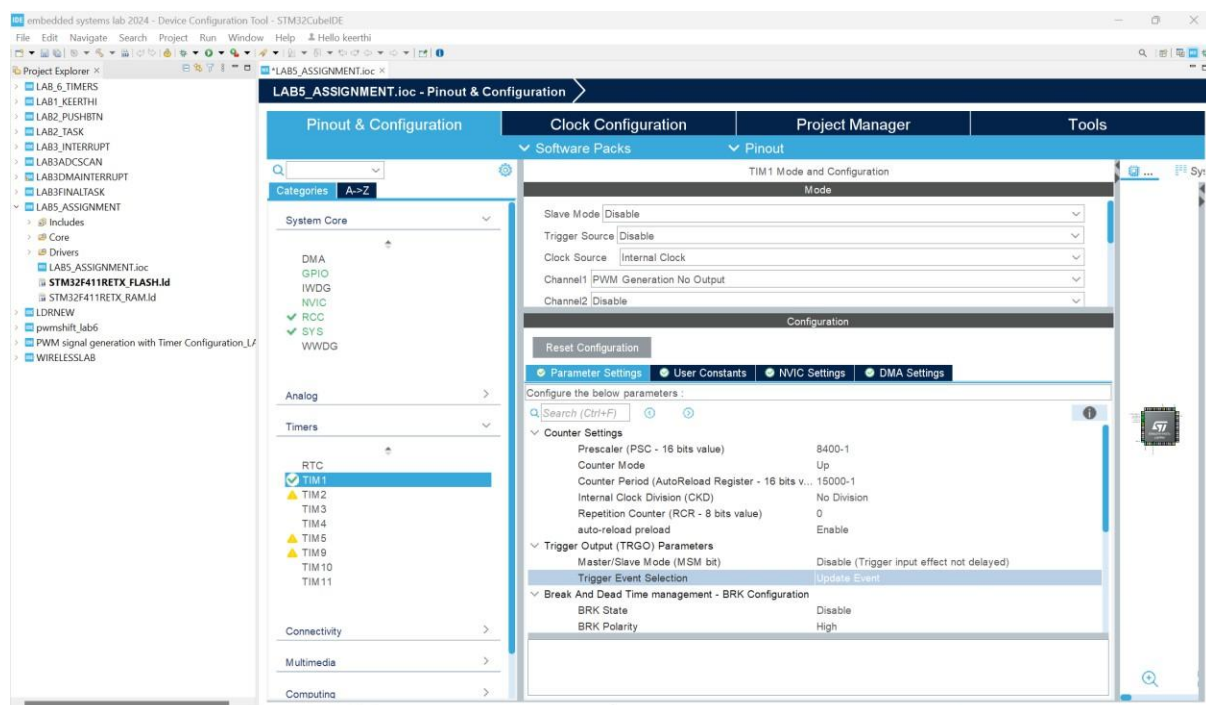
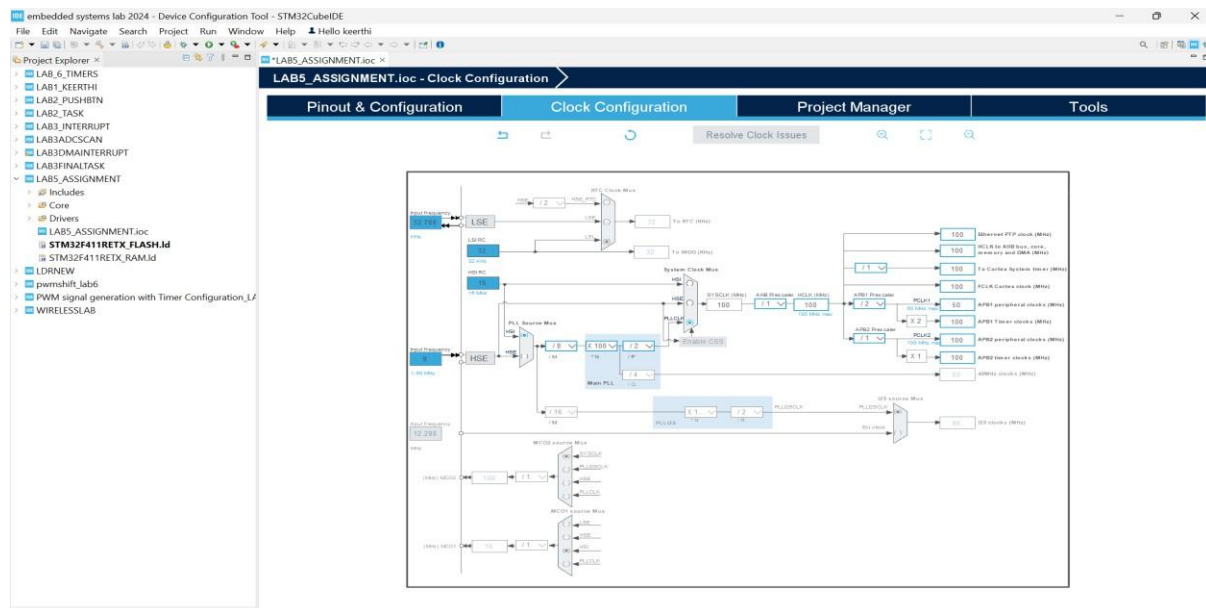


Assignment:

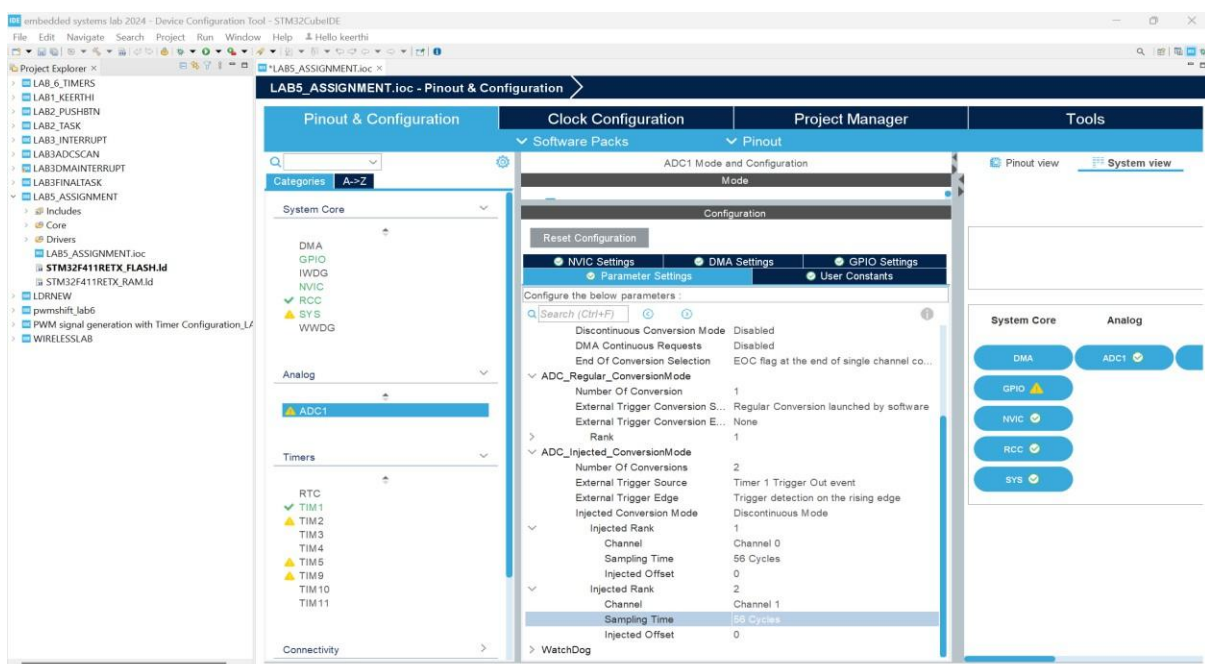
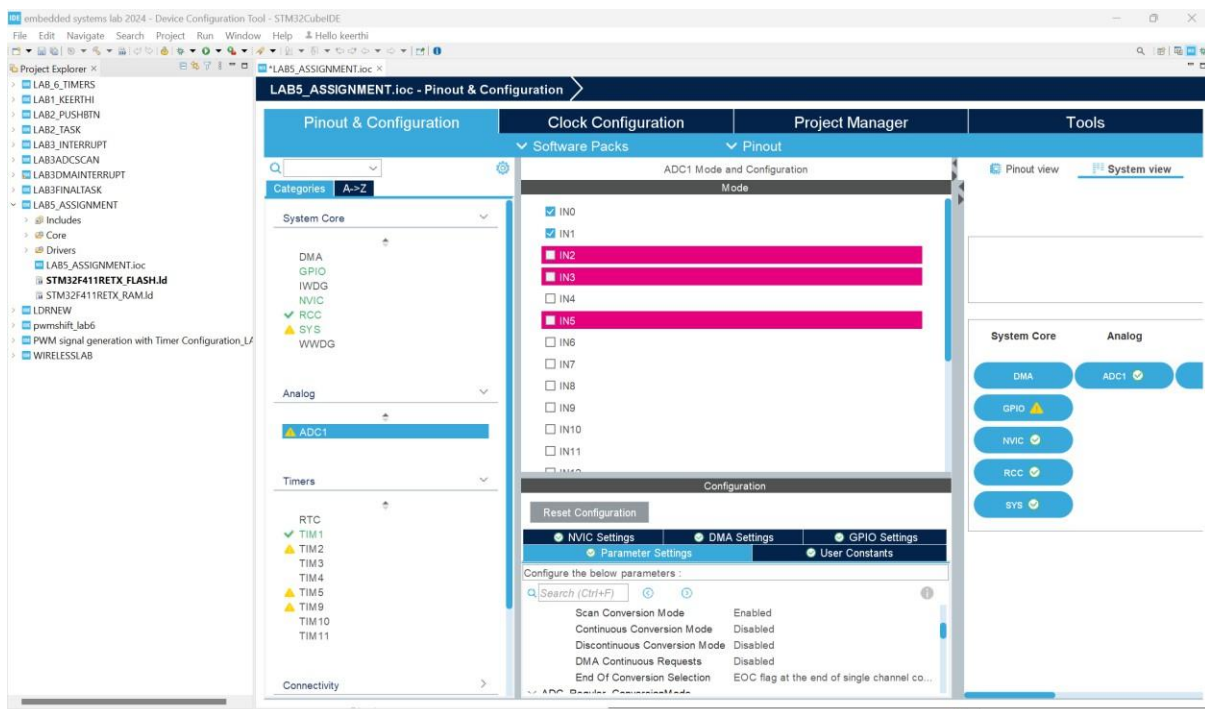
- First step is to Launch STM32CubeIDE.
- We have Set the System Clock to 100MHz
- Then enabled the Timer-1 and set the Prescaler to 8400-1 and counter period to 17868-1 (which generates an interrupt every 1.5 seconds).

Prescaler= 8399 (which gives a timer clock of $100\text{ MHz}/8400 \approx 11,905\text{ Hz}$).

Counter Period= $\text{Timer Clock} \times \text{Interval} - 1$
 $= 11,905\text{ Hz} \times 1.5\text{ seconds} - 1 \approx 17,867$



- I have enabled the ADC1 and added channel 0 and 1 to the configuration and set the scan mode to allow readings from multiple channels.
- To set the Trigger source we have **External Trigger for Regular Conversion** and selected **Timer 1 TRGO**. This basically initiates ADC conversions when Timer 1 triggers.
- Then updating main.c and Compile the code and run the code on STM32 board.
- We see that the ADC values are obtained every 1.5 seconds.



Code:

```
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
    if (htim->Instance == TIM1)
    {
        HAL_ADC_Start_IT(&hadc1); // This Starts ADC conversion in interrupt mode
    }
}
```

```
void HAL_ADC_ConvCpltCallback(ADC_HandleTypeDef *hadc)
{
    if (hadc->Instance == ADC1)
    {
        uint32_t adcValue0 = HAL_ADC_GetValue(&hadc1);
        uint32_t adcValue1 = HAL_ADC_GetValue(&hadc1);
    }
}
```

- The HAL_TIM_PeriodElapsedCallback function triggers an ADC conversion in interrupt mode every time Timer 1 overflows, while the HAL_ADC_ConvCpltCallback function retrieves and processes the ADC values once the conversion is complete, allowing for periodic data sampling.
- In this configuration, Timer 1 is programmed to provide an interrupt every 1.5 seconds, facilitating continuous ADC conversions. By setting the prescaler to 8399 and the counter period to 17867, we achieve the desired timer frequency, which allows for accurate timing of ADC readings. This integration between Timer 1 and ADC1 allows the microcontroller to effectively sample data from multiple channels.

Conclusion:

The lab successfully demonstrated the STM32 timers' adaptability for timer-triggered ADC acquisitions, phase-shifted PWM control, and PWM production. We investigated how altering parameters like frequency and duty cycle can control external devices like motors or LEDs by setting up a timer to produce a PWM signal. This practical experience prepared the ground for more complex applications that call for accurate signal modulation.

The lab also demonstrated effective management of time-sensitive processes by highlighting the significance of timers in initiating interrupts for periodic chores like ADC sampling. Setting up an interrupt every 1.5 seconds demonstrated how well the STM32 can handle precise data sampling in real-time applications, such as sensor interfaces. All things considered, this lab offered insightful information about how timer designs might be modified for particular requirements in embedded systems.