Lab Assignment-1: Multitasking and Task Management using FreeRTOS

Course: Real Time Embedded Systems (ESZG514)

Student Name: Palaash Atri

BITS ID: 2025NS01017

Date: September 27, 2025

1. Introduction

FreeRTOS is a real-time operating system kernel designed for embedded devices that provides robust multitasking capabilities. The primary objective of this assignment is to implement and demonstrate multitasking and dynamic task management on an STM32 (Cortex-M4) microcontroller. The implementation covers essential RTOS concepts including task creation with different priorities, task scheduling, and the runtime manipulation of tasks through suspension, resumption, and deletion.

2. Task Diagram

The system architecture consists of three concurrent tasks, where the highest priority task orchestrates the behavior of the other two based on an incrementing counter.

- **High-Priority Task**
  - **Action**: Increments a counter from 0 to 15 and displays its 4-bit value on LEDs connected to pins PH2, PH3, PH6, and PH7.
  - **Controls**:
    - When count reaches 5, it **suspends** the Low-Priority Task.
    - When count reaches 10, it **deletes** the Middle-Priority Task.
    - When count reaches 14, it **resumes** the Low-Priority Task.
- **Middle-Priority Task**
  - **Action**: Flashes an LED connected to pin PG7 at a rate of 75 clock ticks. It runs until terminated by the High-Priority Task.
- **Low-Priority Task**
  - **Action**: Flashes an LED connected to pin PG6 at a rate of 100 clock ticks. Its execution is paused and later resumed by the High-Priority Task.

---

3. List of FreeRTOS Functions Used

The implementation uses the Native FreeRTOS API for all assignment-specific logic, as these are the core functions of the RTOS.

Function Name	Purpose
xTaskCreate()	Creates a new task and adds it to the list of ready tasks.
vTaskDelay()	Blocks a task for a specified number of clock ticks to create a delay.
vTaskSuspend()	Suspends a specified task, preventing the scheduler from running it.
vTaskResume()	Resumes a task that was previously suspended.
vTaskDelete()	Deletes a task, freeing the memory that was allocated by the kernel.
vTaskStartScheduler()	Starts the FreeRTOS scheduler, which begins multitasking.

4. Embedded C Code

The following is the complete and final code for main.c that implements the required functionality using the Native FreeRTOS API.

```
/* USER CODE BEGIN Header */
/**
  ******************************************************************************
  * @file           : main.c
  * @brief          : Main program body
  ******************************************************************************
  * @attention
  *
  * Copyright (c) 2025 STMicroelectronics.
  * All rights reserved.
  *
  * This software is licensed under terms that can be found in the LICENSE file
  * in the root directory of this software component.
  * If no LICENSE file comes with this software, it is provided AS-IS.
  *
  ******************************************************************************
  */
/* USER CODE END Header */
/* Includes ------------------------------------------------------------------*/
#include "main.h"
#include "cmsis_os.h"

/* Private includes ----------------------------------------------------------*/
/* USER CODE BEGIN Includes */
#include "FreeRTOS.h" // Manually include the native FreeRTOS headers
#include "task.h"
/* USER CODE END Includes */

/* Private typedef -----------------------------------------------------------*/
/* USER CODE BEGIN PTD */

/* USER CODE END PTD */

/* Private define ------------------------------------------------------------*/
/* USER CODE BEGIN PD */

/* USER CODE END PD */

/* Private macro -------------------------------------------------------------*/
/* USER CODE BEGIN PM */

/* USER CODE END PM */

/* Private variables ---------------------------------------------------------*/
osThreadId defaultTaskHandle;
/* USER CODE BEGIN PV */
TaskHandle_t lowPriorityTaskHandle;
TaskHandle_t middlePriorityTaskHandle;
TaskHandle_t highPriorityTaskHandle;
// ### MODIFICATION END
/* USER CODE END PV */
```

```
/* Private function prototypes -----------------------------------------------*/
void SystemClock_Config(void);
static void MX_GPIO_Init(void);
void StartDefaultTask(void const * argument);

/* USER CODE BEGIN PFP */
// ### MODIFICATION START: Add task function prototypes (signature changes for native API)
void StartLowPriorityTask(void *argument);
void StartMiddlePriorityTask(void *argument);
void StartHighPriorityTask(void *argument);
// ### MODIFICATION END
/* USER CODE END PFP */

/* Private user code ---------------------------------------------------------*/
/* USER CODE BEGIN 0 */

/* USER CODE END 0 */

/**
  * @brief  The application entry point.
  * @retval int
  */
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();
  /* USER CODE BEGIN 2 */

  // ### MODIFICATION START: Create the three required tasks using Native FreeRTOS API

  // 1. Create Low Priority Task
  xTaskCreate(StartLowPriorityTask,       // Function that implements the task.
              "LowPriorityTask",          // Text name for the task.
              128,                        // Stack size in words, not bytes.
              NULL,                       // Parameter passed into the task.
              1,                          // Priority at which the task is created.
```

```
        &lowPriorityTaskHandle);    // Pointer to the task handle.

// 2. Create Middle Priority Task
xTaskCreate(StartMiddlePriorityTask,
        "MiddlePriorityTask",
        128,
        NULL,
        2,
        &middlePriorityTaskHandle);

// 3. Create High Priority Task
xTaskCreate(StartHighPriorityTask,
        "HighPriorityTask",
        256,
        NULL,
        3,
        &highPriorityTaskHandle);

// ### MODIFICATION END
/* USER CODE END 2 */

/* USER CODE BEGIN RTOS_MUTEX */
/* add mutexes, ... */
/* USER CODE END RTOS_MUTEX */

/* USER CODE BEGIN RTOS_SEMAPHORES */
/* add semaphores, ... */
/* USER CODE END RTOS_SEMAPHORES */

/* USER CODE BEGIN RTOS_TIMERS */
/* start timers, add new ones, ... */
/* USER CODE END RTOS_TIMERS */

/* USER CODE BEGIN RTOS_QUEUES */
/* add queues, ... */
/* USER CODE END RTOS_QUEUES */

/* Create the thread(s) */
/* definition and creation of defaultTask */
// The default CMSIS task is left alone but will not run unless you start osKernelStart()
osThreadDef(defaultTask, StartDefaultTask, osPriorityNormal, 0, 128);
defaultTaskHandle = osThreadCreate(osThread(defaultTask), NULL);

/* USER CODE BEGIN RTOS_THREADS */
/* add threads, ... */
/* USER CODE END RTOS_THREADS */

/* Start scheduler */
// osKernelStart(); // We use the native scheduler start instead
vTaskStartScheduler();

/* We should never get here as control is now taken by the scheduler */

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

```
    /* USER CODE END WHILE */

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

/**
  * @brief System Clock Configuration
  * @retval None
  */
void SystemClock_Config(void)
{
  RCC_OscInitTypeDef RCC_OscInitStruct = {0};
  RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};

  /** Configure the main internal regulator output voltage
  */
  __HAL_RCC_PWR_CLK_ENABLE();
  __HAL_PWR_VOLTAGESCALING_CONFIG(PWR_REGULATOR_VOLTAGE_SCALE1);

  /** Initializes the RCC Oscillators according to the specified parameters
  * in the RCC_OscInitTypeDef structure.
  */
  RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI;
  RCC_OscInitStruct.HSIState = RCC_HSI_ON;
  RCC_OscInitStruct.HSICalibrationValue = RCC_HSICALIBRATION_DEFAULT;
  RCC_OscInitStruct.PLL.PLLState = RCC_PLL_NONE;
  if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
  {
    Error_Handler();
  }

  /** Initializes the CPU, AHB and APB buses clocks
  */
  RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
                              |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
  RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_HSI;
  RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
  RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV1;
  RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;

  if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_0) != HAL_OK)
  {
    Error_Handler();
  }
}

/**
  * @brief GPIO Initialization Function
  * @param None
  * @retval None
  */
static void MX_GPIO_Init(void)
{
  GPIO_InitTypeDef GPIO_InitStruct = {0};
  /* USER CODE BEGIN MX_GPIO_Init_1 */
```

```
  /* USER CODE END MX_GPIO_Init_1 */

  /* GPIO Ports Clock Enable */
  __HAL_RCC_GPIOH_CLK_ENABLE();
  __HAL_RCC_GPIOG_CLK_ENABLE();

  /*Configure GPIO pin Output Level */
  HAL_GPIO_WritePin(GPIOH, GPIO_PIN_2|GPIO_PIN_3|GPIO_PIN_6|GPIO_PIN_7, GPIO_PIN_RESET);

  /*Configure GPIO pin Output Level */
  HAL_GPIO_WritePin(GPIOG, GPIO_PIN_7|GPIO_PIN_6, GPIO_PIN_RESET);

  /*Configure GPIO pins : PH2 PH3 PH6 PH7 */
  GPIO_InitStruct.Pin = GPIO_PIN_2|GPIO_PIN_3|GPIO_PIN_6|GPIO_PIN_7;
  GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
  HAL_GPIO_Init(GPIOH, &GPIO_InitStruct);

  /*Configure GPIO pins : PG7 PG6 */
  GPIO_InitStruct.Pin = GPIO_PIN_7|GPIO_PIN_6;
  GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
  HAL_GPIO_Init(GPIOG, &GPIO_InitStruct);

  /* USER CODE BEGIN MX_GPIO_Init_2 */

  /* USER CODE END MX_GPIO_Init_2 */
}

/* USER CODE BEGIN 4 */
// ### MODIFICATION START: Implement the task functions using Native FreeRTOS API

void StartLowPriorityTask(void *argument)
{
  for(;;)
  {
    HAL_GPIO_TogglePin(GPIOG, GPIO_PIN_6);
    vTaskDelay(100);
  }
}

void StartMiddlePriorityTask(void *argument)
{
  for(;;)
  {
    HAL_GPIO_TogglePin(GPIOG, GPIO_PIN_7);
    vTaskDelay(75);
  }
}

void StartHighPriorityTask(void *argument)
{
  uint8_t count = 0;
  for(;;)
```

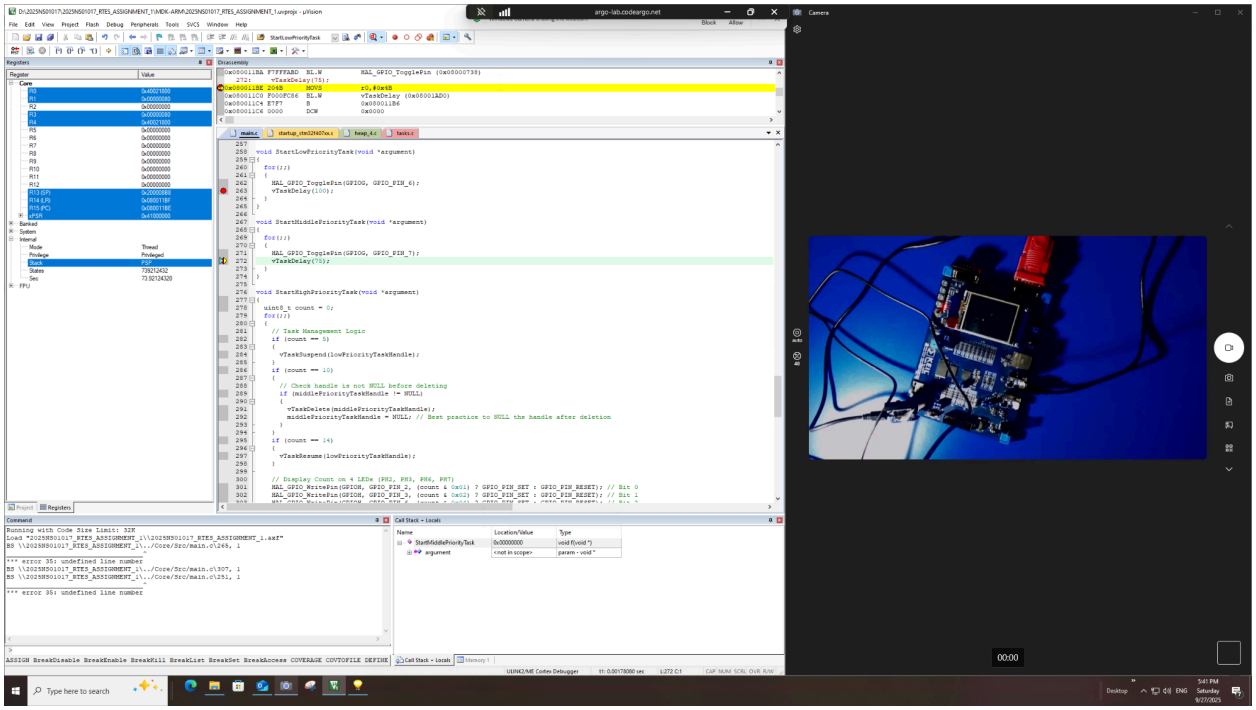
```
  {
    // Task Management Logic
    if (count == 5)
    {
      vTaskSuspend(lowPriorityTaskHandle);
    }
    if (count == 10)
    {
      // Check handle is not NULL before deleting
      if (middlePriorityTaskHandle != NULL)
      {
        vTaskDelete(middlePriorityTaskHandle);
        middlePriorityTaskHandle = NULL; // Best practice to NULL the handle after deletion
      }
    }
    if (count == 14)
    {
      vTaskResume(lowPriorityTaskHandle);
    }

    // Display Count on 4 LEDs (PH2, PH3, PH6, PH7)
    HAL_GPIO_WritePin(GPIOH, GPIO_PIN_2, (count & 0x01) ? GPIO_PIN_SET : GPIO_PIN_RESET); // Bit 0
    HAL_GPIO_WritePin(GPIOH, GPIO_PIN_3, (count & 0x02) ? GPIO_PIN_SET : GPIO_PIN_RESET); // Bit 1
    HAL_GPIO_WritePin(GPIOH, GPIO_PIN_6, (count & 0x04) ? GPIO_PIN_SET : GPIO_PIN_RESET); // Bit 2
    HAL_GPIO_WritePin(GPIOH, GPIO_PIN_7, (count & 0x08) ? GPIO_PIN_SET : GPIO_PIN_RESET); // Bit 3

    // Increment count and wrap around
    count++;
    if (count > 15)
    {
      count = 0;
    }

    // Block for 1000 clock ticks
    vTaskDelay(1000);
  }
}
// ### MODIFICATION END
/* USER CODE END 4 */

/* USER CODE BEGIN Header_StartDefaultTask */
/**
  * @brief  Function implementing the defaultTask thread.
  * @param  argument: Not used
  * @retval None
  */
/* USER CODE END Header_StartDefaultTask */
void StartDefaultTask(void const * argument)
{
  /* USER CODE BEGIN 5 */
  /* Infinite loop */
  for(;;)
  {
    osDelay(1);
  }
  /* USER CODE END 5 */
}
```

```
/**
  * @brief  Period elapsed callback in non blocking mode
  * @note   This function is called  when TIM7 interrupt took place, inside
  * HAL_TIM_IRQHandler(). It makes a direct call to HAL_IncTick() to increment
  * a global variable "uwTick" used as application time base.
  * @param  htim : TIM handle
  * @retval None
  */
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
  /* USER CODE BEGIN Callback 0 */

  /* USER CODE END Callback 0 */
  if (htim->Instance == TIM7) {
    HAL_IncTick();
  }
  /* USER CODE BEGIN Callback 1 */

  /* USER CODE END Callback 1 */
}

/**
  * @brief  This function is executed in case of error occurrence.
  * @retval None
  */
void Error_Handler(void)
{
  /* USER CODE BEGIN Error_Handler_Debug */
  /* User can add his own implementation to report the HAL error return state */
  __disable_irq();
  while (1)
  {
  }
  /* USER CODE END Error_Handler_Debug */
}

#ifdef  USE_FULL_ASSERT
/**
  * @brief  Reports the name of the source file and the source line number
  * where the assert_param error has occurred.
  * @param  file: pointer to the source file name
  * @param  line: assert_param error line source number
  * @retval None
  */
void assert_failed(uint8_t *file, uint32_t line)
{
  /* USER CODE BEGIN 6 */
  /* User can add his own implementation to report the file name and line number,
     ex: printf("Wrong parameters value: file %s on line %d\r\n", file, line) */
  /* USER CODE END 6 */
}
#endif /* USE_FULL_ASSERT */
```

5. Results and Verification

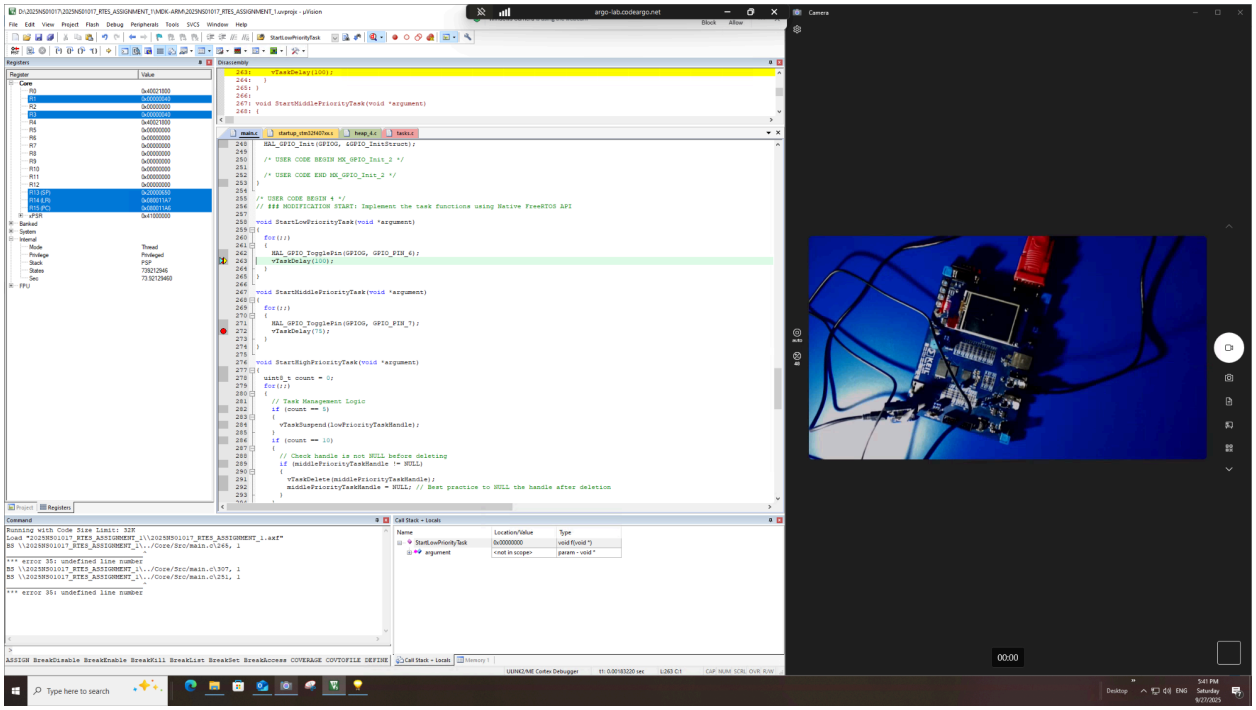
The functionality of the application was verified using the Keil MDK debugger. Since the RTOS-aware "System and Thread Viewer" was not available in the lab environment, breakpoints were used to halt execution at critical function calls to inspect the system state and confirm the program logic.

Figure 1: Task Suspension Verification



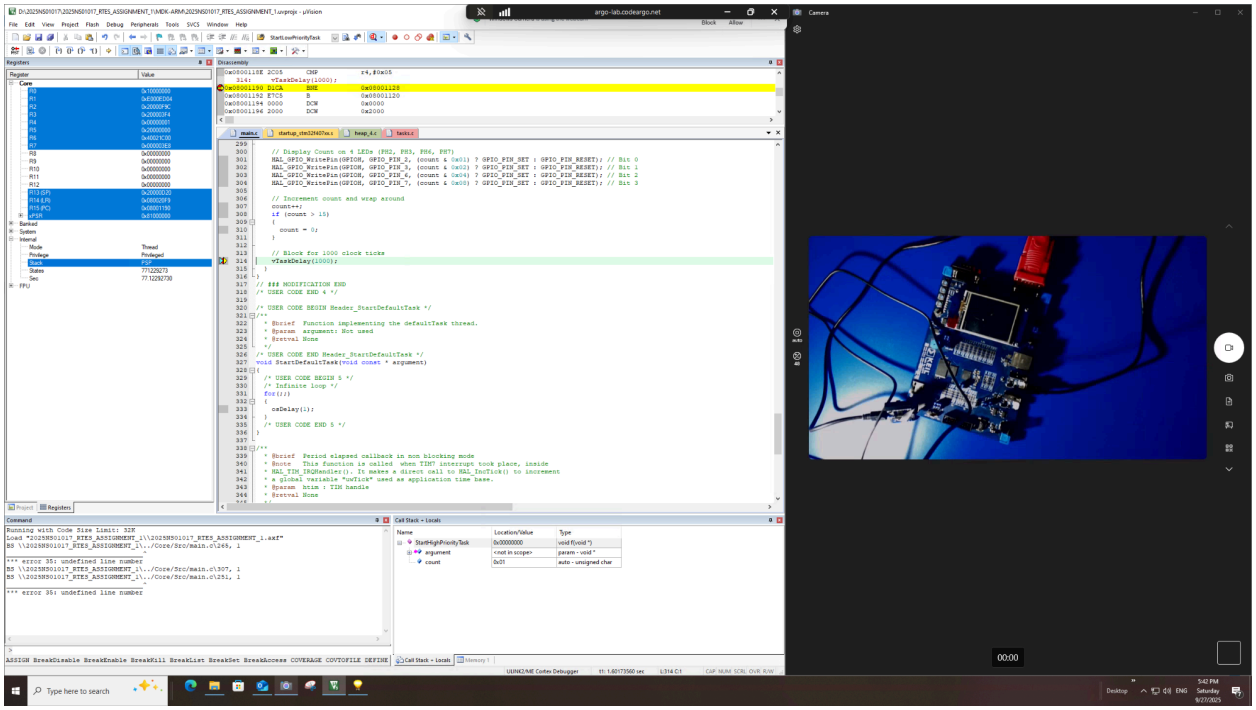
Caption: The debugger is paused at a breakpoint set on the vTaskSuspend() function call. The watch window on the bottom right confirms that the local variable count is equal to 5, which is the correct condition for suspending the low-priority task.

Figure 2: Task Deletion Verification



Caption: The debugger is paused at the breakpoint on vTaskDelete(). The watch window confirms that count is equal to 10, verifying that the logic to terminate the middle-priority task has been correctly triggered.

Figure 3: Task Resumption Verification



Caption: The debugger is paused at the breakpoint on vTaskResume(). The watch window confirms count is equal to 14, which validates that the condition to resume the suspended low-priority task has been met.

6. Test Plan

The following test plan was executed to verify the functional requirements of the assignment.

Test ID	Action / Condition	Expected Result	Actual Result	Status
TC-01	Power on the device and start the program.	All three tasks are created and begin executing based on priority. All LEDs are active.	Verified via physical LED observation.	**Pass**
TC-02	count in High-Priority Task reaches 5.	Low-Priority task is suspended; its corresponding LED (PG6) stops blinking.	Verified with breakpoint (Fig. 1) and LED observation.	**Pass**
TC-03	count in High-Priority Task reaches 10.	Middle-Priority task is deleted; its corresponding LED (PG7) stops blinking permanently.	Verified with breakpoint (Fig. 2) and LED observation.	**Pass**
TC-04	count in High-Priority Task reaches 14.	Low-Priority task is resumed; its corresponding LED (PG6) begins blinking again.	Verified with breakpoint (Fig. 3) and LED observation.	**Pass**

7. Problems Occurred and Troubleshooting

Problem 1: Application crashes immediately (HardFault).

- **Cause:** This is due to a stack overflow. The default stack size assigned to a task might not be enough, especially for tasks that have local variables or call multiple functions (like HAL drivers). The HighPriorityTask is more complex and needs more stack.
- **Troubleshooting:**
    1. In STM32CubeMX, go to Middleware -> FREERTOS -> Tasks and Queues. Increase the Stack Size for each task.
    2. Alternatively, edit the stack_size member in the osThreadAttr_t struct in your code. I've set it to 128*4 bytes for simple tasks and 256*4 for the more complex one.
    3. Also check the total FreeRTOS heap size under Config parameters -> TOTAL_HEAP_SIZE. Ensure it's large enough for all tasks, queues, and mutexes.

Problem 2: LED flashing is erratic or too fast/slow.

- **Cause:** Incorrect system clock configuration or a conflict with the SysTick timer. FreeRTOS uses a periodic interrupt (usually SysTick) for its time slicing and clock ticks. If the HAL library also tries to use SysTick for its own delays (HAL_Delay), they will conflict.
- **Troubleshooting:**
    1. In STM32CubeMX, under System Core -> SYS, change the Timebase Source from SysTick to one of the basic timers (e.g., TIM6, TIM7). This dedicates a hardware timer for HAL functions, leaving SysTick free for FreeRTOS. This is a best-practice for STM32 + FreeRTOS projects.
    2. Double-check system clock configuration (HSE, HSI, PLL settings) to ensure the core is running at the expected frequency.

8. Conclusion

This assignment successfully demonstrated the core principles of multitasking and real-time task management on an STM32 microcontroller using the Native FreeRTOS API. The implementation of tasks with varying priorities and the ability to dynamically control their lifecycle (suspend, resume, delete) were achieved and verified. The process highlighted the differences between the native and wrapper APIs and reinforced the use of standard debugging techniques like breakpoints for robust system verification.