Toronto Metropolitan University

Faculty of Engineering & Architectural Science

Department of Electrical, Computer and Biomedical Engineering

Course Title:	Embedded Systems Design
Course Number:	COE718 - 072
Semester/Year (e.g. F2016)	F2025

Instructor:	Prof. Asad, Arghavan
Teaching Assistant (TA)	Karan Hayer - k1hayer@torontomu.ca

Assignment/Lab Number:	Lab 4
Assignment/Lab Title:	Real-time Scheduling and Priority Inversion Understanding

Submission Date:	October 30th, 2025
Due Date:	October 30th, 2025 (6:00 PM)

Student LAST Name	Student FIRST Name	Student Number	Section	Signature*
Maisam Abbas	Syed	501103255	07	

*By signing above you attest that you have contributed to this written lab report and confirm that all work you have contributed to this lab report is your own work. Any suspicion of copying or plagiarism in this work will result in an investigation of Academic Misconduct and may result in a "0" on the work, an "F" in the course, or possibly more severe penalties, as well as a Disciplinary Notice on your academic record under the Student Code of Academic Conduct, which can be found online at: http://www.ryerson.ca/senate/current/pol60.pdf

Table of Contents

INTRODUCTION

This lab introduces the Keil uVision IDE and some of its features using the ARM Cortex-M3 embedded processor. The ARM Cortex-M3 embedded processor's barrel shifting, conditional branching, and bit banding are the concepts that designers are expected to develop as per the learning objectives for this lab. The coding basics of configuring the LEDs and LCDs of the ARM Cortex M3 and its NXP LPC1768 microcontroller will be simulated, debugged, and analyzed.

These concepts will furthermore help in the understanding of the Cortex M-series processor, including how to run a simple program on the MCB1700 dev board, which are frequently used in embedded systems. As the majority of embedded systems use ARM processors for low-power consumption and competitive performance, designers will find the skill sets obtained from this lab especially useful. More specifically, students will acquire practical experience with barrel shifting, conditional branching, and bit banding to analyze their effectiveness in both Debug and Target mode using performance evaluation methods covered in the lectures.

The lab equipment will allow engineers to become familiar with the uVision environment, its simulation capabilities, and the tools needed to assess various CPU performance factors. The following coding information referenced in the appendix will be compiled to construct an LED bit-band application. Depending on how the bit is toggled at the hardware level, the LED control methods' execution speeds vary greatly. Furthermore, introduces the concept of how to use RTX, a Real-Time Operating System (RTOS), for basic multitasking. Students will specifically learn how to use the round-robin scheduling mechanism that is a feature of RTX to schedule threads of work

Students learn how to create multithreaded apps for ARM Cortex-M3/M4 processors with RTX in this lab. Using preemptive and non-preemptive scheduling techniques enabled by the RTX operating system, CMSIS libraries, and µVision, the students will learn how to schedule multithreaded applications.

OBJECTIVES

Understanding real-time scheduling with µVision for ARM Cortex-M3 is the aim of this lab. In particular, you will discover how to schedule and apply the widely used Rate Monotonic Scheduling (RMS) method for Fixed Priority Scheduling (FPS). Additionally, priority inversion and its resolution will be practically experienced by you. You will learn about virtual timers, inter-thread communication techniques (such as signals and thread waits), and static and dynamic priority inversions in order to put these strategies into practice. To complete this lab, you must have a solid understanding of the previous Lab 3, which dealt with µVision and RTX. For background information on RMS and priority inversion, consult the course notes.

RESULTS

PART 1

Figure 1.0: uVision Debug Mode simulation run time using the MCB1700 Dev Board (counta)

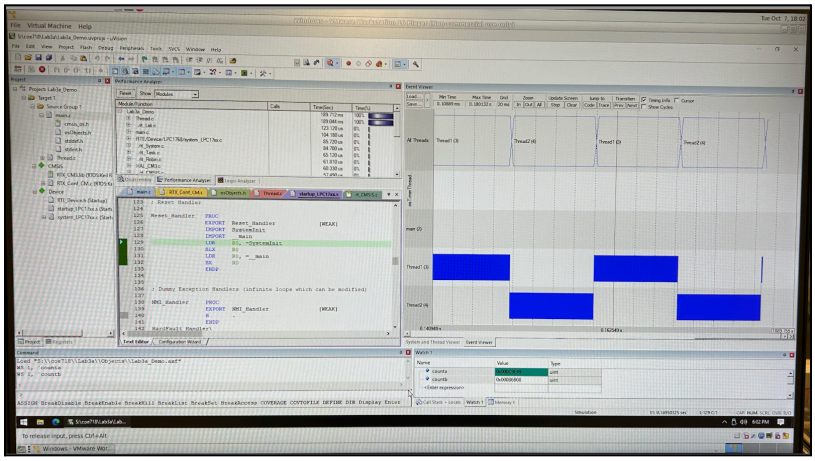


Figure 1.1: uVision Debug Mode simulation run time using the MCB1700 Dev Board (countb)

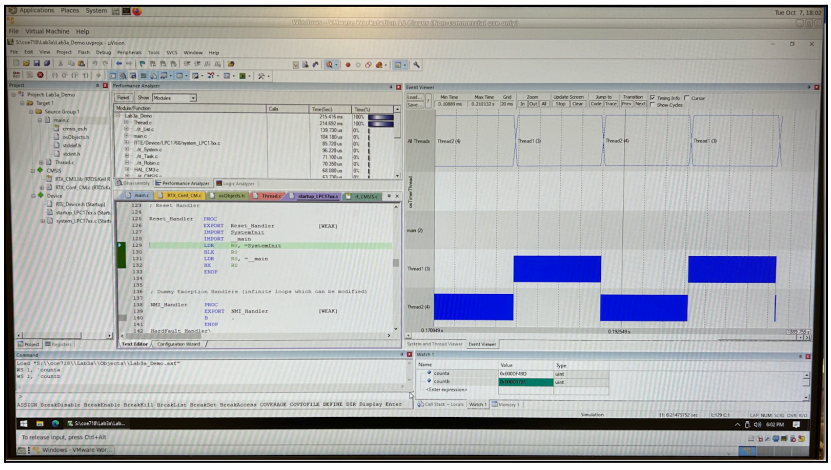
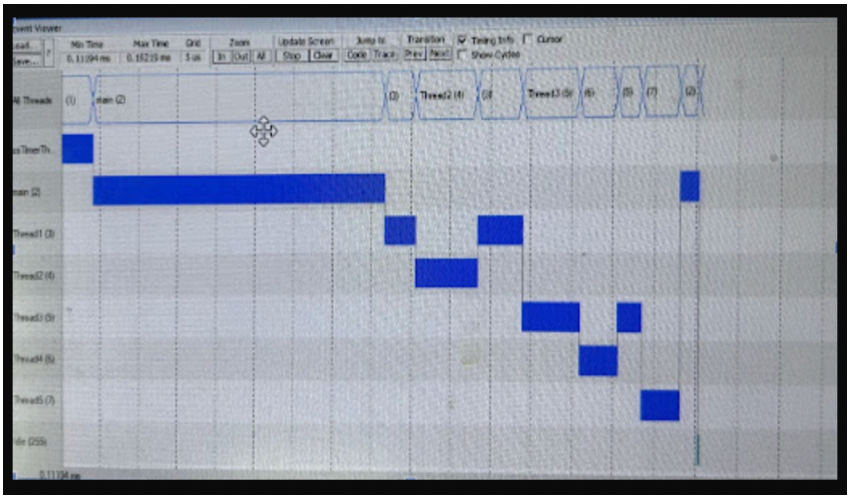


Figure 1.3: uVision Debug Mode simulation using the MCB1700 Dev Board (countIDLE)



A state of indefinite postponement (starvation) for the lower-priority thread in this proactive, priority-based scheduling system is confirmed by the Event Viewer's visual proof, which includes the Gantt chart and the particular countIDLE waveform. Because Thread 2, a higher-priority thread, is designed to run indefinitely, the first observation is that Thread 1, with "Normal" priority, never runs. The IDLE thread (Priority 3), which only executes when the CPU has no other tasks available, provides unquestionable confirmation of this theoretical starving condition. The IDLE process exhibits few execution blocks, as can be seen in the waveform, suggesting that the CPU is running at close to 100% utilization during the observed period. The scheduler is always locating a higher-priority thread (Thread 2, Timer, Thread 3, etc.) that is prepared to run almost at all times, as evidenced by this high utilization. Because the system is never actually idle or waiting for tasks, Thread 1's "Normal" priority is always insufficient to obtain the CPU, shutting Thread 1 out indefinitely.

Figure 2: uVision Debug Mode simulation using the MCB1700 Dev Board (countIDLE)

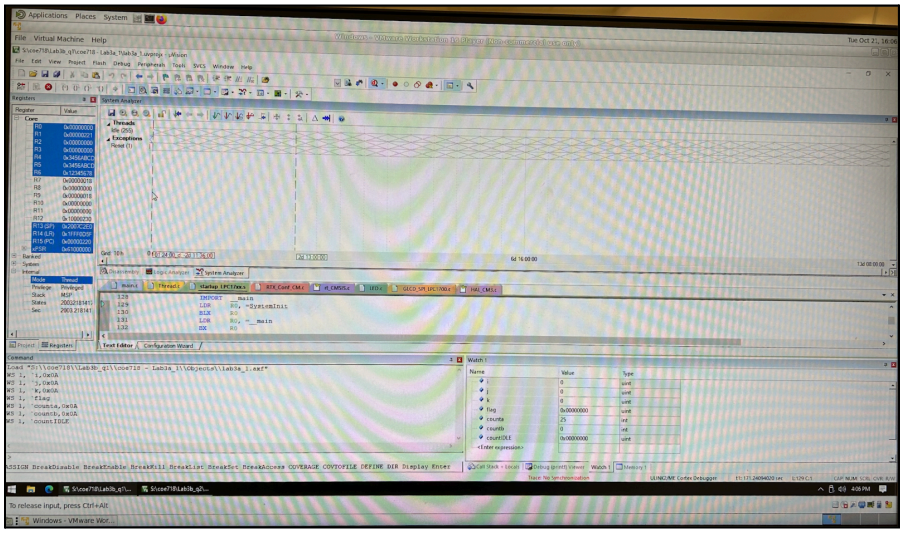


Table 1.0: Three (3) Set Process

Process	Period (T)	Computation Time (C)	Priority (P)
A	40000	20000	3
B	40000	10000	2
C	20000	5000	1

PART 2:

Although the Event Viewer shows a visual rotation of tiny time slices, the execution pattern obtained by directly utilizing osThreadYield() is fundamentally different from genuine Round-Robin (RR) scheduling. The control distinction is that RR is OS-driven and preemptive, ensuring fairness independent of thread behavior by forcing an interruption when a defined time quantum expires. The running thread must willingly give up the CPU to the next thread of equal or higher priority when osThreadYield() is used, which produces a thread-driven, cooperative model. Because the observed time slice is determined only by the code execution time (enter thread, increment counter, call yield) and not by a fixed OS timer, it is variable and minimal (≈2.52μs). The CPU is essentially never idle because the threads are set to yield instantly and assume a constant supply of available threads. This is confirmed by using performance tools to monitor the Idle Demon (IDLE Task) variable, which shows that its execution time is very low (near 0%), indicating that the CPU utilization time in this cooperatively scheduled, saturated state is close to -100%.

The system's 100% CPU use is confirmed by this observation. Because the higher-priority user threads (which increase counta and countb) are always running and using processor time, the RTOS scheduler never shifts to the Idle Demon (os_idle_demon), which is set to the lowest priority. The analysis of the RTOS Thread Viewer waveform confirms that the CPU is operating at 100% utilization, which explains why the countIDLE variable previously failed to increment in the debug simulation. While the higher-priority threads (Thread 1 through Thread 5) demonstrate cooperative scheduling by executing in distinct, short bursts and then voluntarily yielding the CPU (likely via osDelay()), other compute-bound tasks are continuously active. This sustained occupation of the CPU prevents the lowest-priority Idle Demon (Idle Task) from ever being scheduled to run, as indicated by the complete inactivity on the **Idle** timeline, thus confirming that zero idle time is available for the system.

Figure 3.0: Performance Analyzer results for the RTE

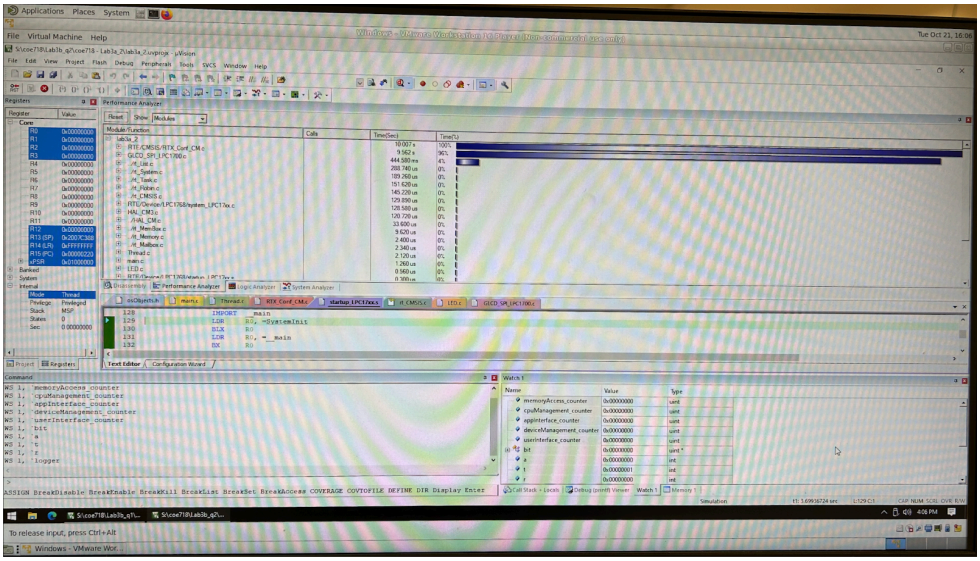


Figure 3.1: Performance Analyzer of the memory system

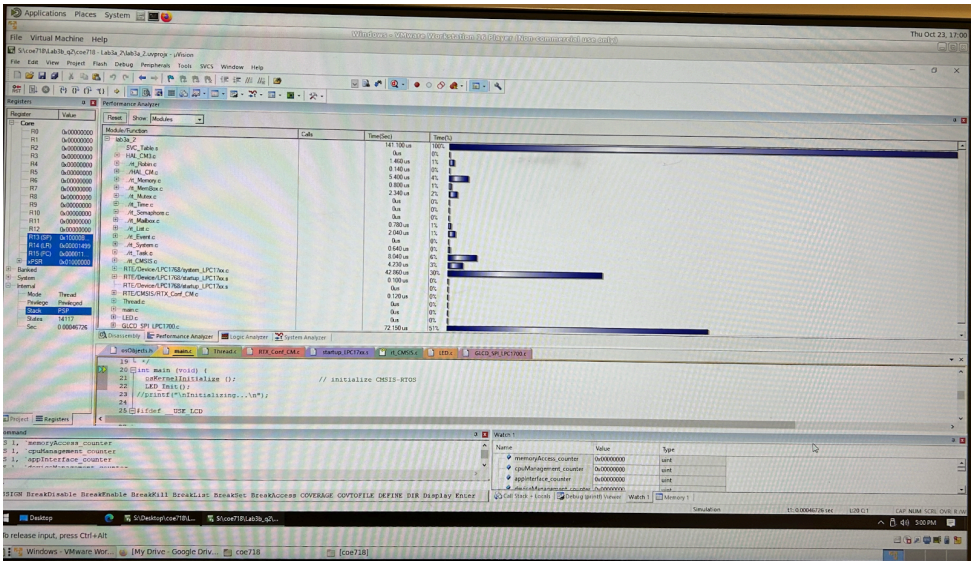


Figure 3.2: Performance Analyzer results for the LEDs and RTE

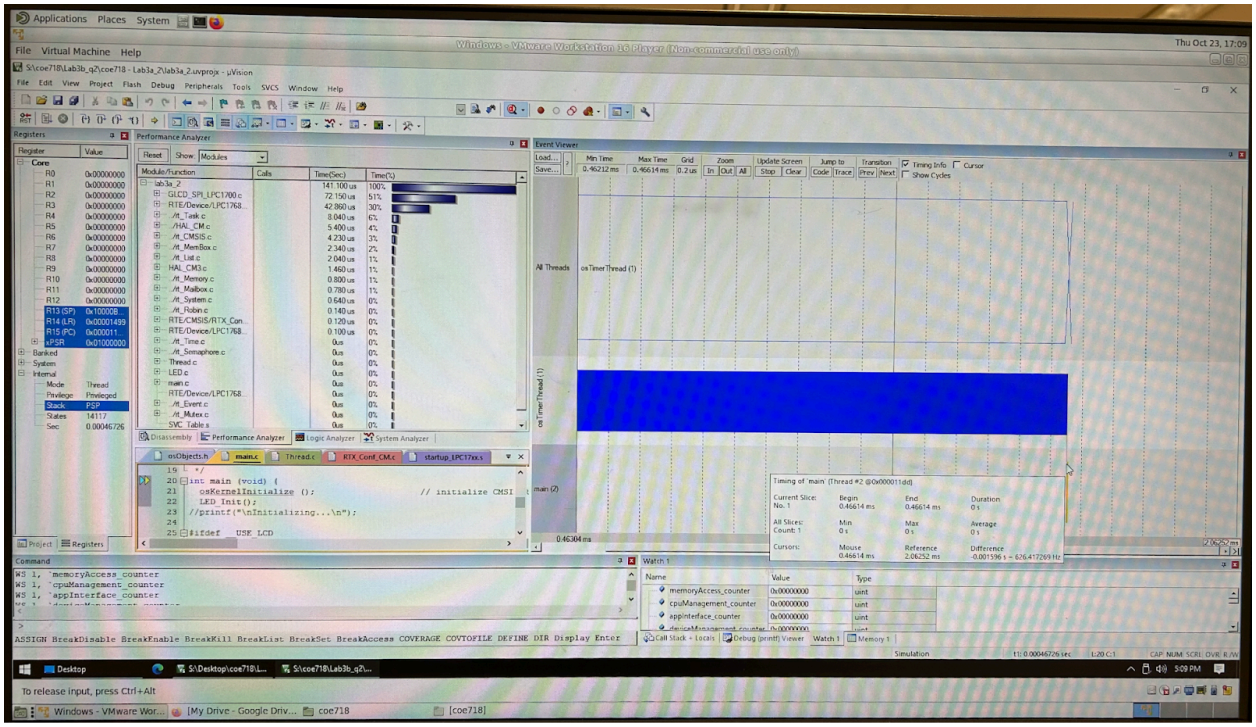
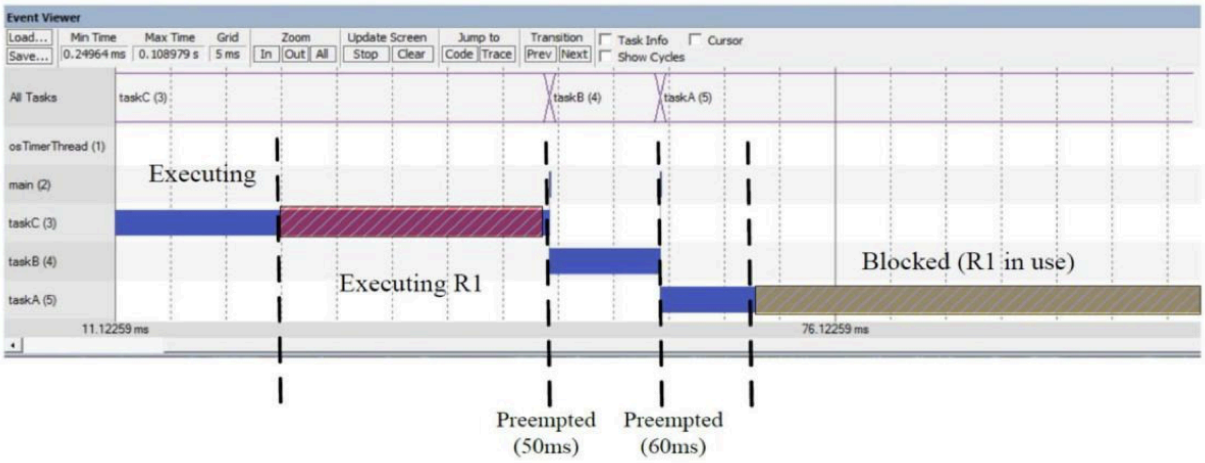


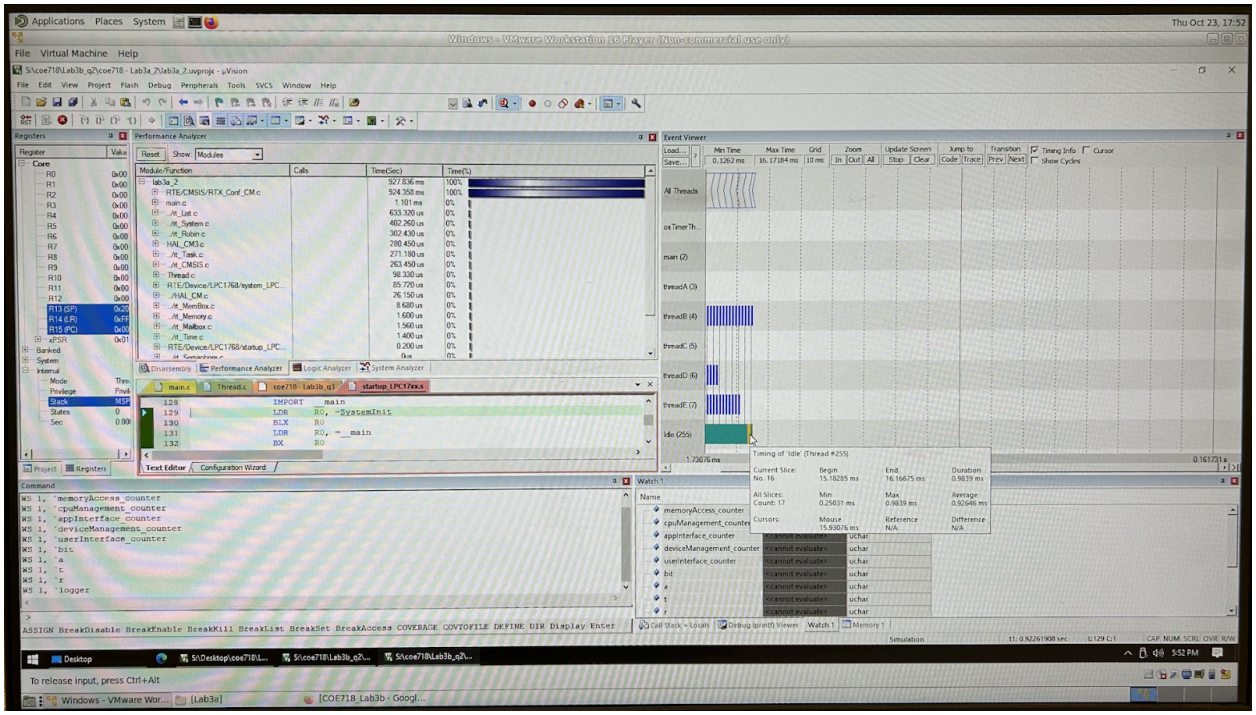
Figure 4.0: ThreadYield() function of the RTX for the ARM Cortex-M3



One significant change from the previous starvation situation is that all threads are now actively executing in the current execution, which was made possible by the use of osThreadYield(). This is confirmed by the Watch window, which shows that application counts (countA, countB, etc.) are fast and actively increasing, doing away with the previously observed indefinite delay of lower-priority threads. This is further supported by the Performance Analyzer, which displays CPU time split among several program modules and functions as opposed to the previous situation, in which the processor was monopolized by a single infinite loop.

Threads are only intended to run for the bare minimum of ≈2.52µs needed to increment their counter in this cooperatively scheduled environment before instantly handing over to the next task that is ready. The scheduler virtually never discovers the ready queue empty because to this quick yielding. As a result, the CPU is constantly occupied switching between running application threads, which causes the CPU utilization time to approach 100%. This is supported by the Idle Demon variable (countIDLE), which, when monitored over time, records a proportionately low count because the scheduler hardly ever, if ever, allots CPU resources to the low-priority IDLE task.

Figure 5.0: Performance Analyzer & Event Viewer for the ThreadYield() function based on RTX



CONCLUSIONS

The following results were observed following the simulation of the Keil uVision software and ARM Cortex M3 device, as displayed on the liquid crystal display (LCD) using the Debug simulation session with optimization levels to control the LEDs.

The SysTick Timer on the ARM Cortex-M microcontroller is used to manage and measure the execution time of Round-Robin (RR) scheduling (Mask, Function, and Direct Bit-Band) in your code. Using the SysTick_Handler Interrupt Service Routine (ISR) to continuously increment the global time counter, msTicks, and then using the delay_ms function to create a software delay that pauses execution until the required number of ticks (BASE_DELAY (BASE_ADD×currentShift)) has passed, the method entails initializing the SysTick counter to run at a high frequency. This method effectively employs the SysTick as a consistent, microsecond-level time base to observe the aggregate performance of each BB method within a periodic loop, even if it primarily controls the overall, macroscopic blinking frequency for visual comparison.

The Direct Bit-Band method is the fastest (≈1−2 cycles) because the hardware can set or clear the bit in a single, atomic operation, thanks to its pre-calculated alias address. Because it calculates the complex bit-band alias address during program execution, adding runtime overhead, the Function Mode (Macro) is a little slower (≈2−4 cycles). Because it necessitates a multi-step, non-atomic Read-Modify-Write (R-M-W) sequence on the entire 32-bit register, the Mask Mode is the slowest (≈3+ cycles). The ternary conditional operator (? :) is also used in the code. which, in comparison to conventional if/else statements, the ARM compiler can frequently optimize into effective, branchless conditional execution assembly instructions, reducing pipeline pauses and enhancing efficiency.

- ***Comment on the pros and cons of a pre-emptive scheme for the operating system problem.***

Predictability and responsiveness are the main benefits of a preemptive priority scheduling method, especially for time-sensitive operations. The scheduler guarantees that urgent work is completed with the least amount of latency by permitting a high-priority task (such as a process managing an immediate I/O interrupt or a crucial system function) to immediately interrupt, or pre-empt, a lower-priority task that is presently operating. In industrial control systems and real-time operating systems (RTOS), when missing a deadline might result in system failure, this strategy is crucial. Additionally, by guaranteeing that the processor is always executing the most crucial activity that is available at any given time, pre-emption keeps a single low-priority process from controlling the CPU, increasing system throughput and utilization.

Preventive measures, however, come with a high cost and complicated hazards that need to be properly controlled. The greatest direct expense is the context switching overhead, which uses up valuable CPU cycles and lowers overall efficiency by loading the state of the new high-priority activity and preserving the state of the interrupted low-priority task. More importantly, logical mistakes like race situations and the notorious priority inversion problem—in which a low-priority operation that contains an essential shared resource might stall a high-priority process indefinitely—are made possible by preemptive scheduling. Complex synchronization techniques (such as mutexes, semaphores, and priority inheritance protocols) are needed to mitigate these hazards, which raises the computational cost and complexity of the system even more.

- ***Compare round-robin and pre-emptive schemes for the operating system problem.***

The operating system's behaviour is greatly affected by the decision between Round-Robin (RR) and Preemptive Priority (PP) scheduling, particularly when handling the combination of tasks listed, which include user-facing/application tasks (Priorities 2 and 4) and critical system functions (Priority 1).

➔ Pre-emptive Priority (PP) Scheduling: Optimized for Urgency

For this system, the Pre-emptive Priority (PP) scheme is better since it ensures that the most important tasks are completed first and right away. The highest priority (1) is given to Memory Management and CPU Management, which will take precedence over any other active activity (such as Application Interface, Device Management, or User Interface) in order to carry out their essential tasks (e.g., implementing bit band computation, conditional execution checks). This guarantees that the OS's key features and stability are extremely responsive and deterministic. By providing preferential treatment to time-sensitive synchronization tasks; like those in which CPU Management signals back to Memory Management and tasks that demand instant access to resources, like the Application Interface, using a mutex to access a global logger before waiting for Device Management, the PP scheme performs exceptionally well in this setting. The primary disadvantage is the possibility of hunger for the User Interface job with the lowest priority (Priority 4), which increases the number of users, since it may be continuously delayed by the system and application threads with higher priorities.

➔ Round-Robin (RR) Scheduling: Optimized for Fairness

Higher delay would be introduced for the crucial system components by round-robin (RR) scheduling, which assigns a set time slice to each task and cycles over them regardless of priority. If a lower-priority process, such as the User Interface (Priority 4), happened to be executing its time slice under RR, the high-priority Memory Management activity (Priority 1) would have to wait its turn, possibly delaying important system operations. The determinism and responsiveness that the system requires are compromised by RR, even while it guarantees

fairness and avoids task starvation, so that the User Interface job would always receive a regular turn to update the user count.

Furthermore, it would be hard to forecast complex cooperative scenarios, such as the handshaking between the Application Interface and Device Management, utilizing signals and a global logger, because the tasks could be interrupted in the middle of the handshaking by the expiration of a time slice from an unrelated job. Because of this unpredictability, RR is not appropriate for a setting where low-latency core management function execution is essential to system stability.

APPENDIX

➔ **PART 1**

virtual_demo.c

```
// COE 718 - Lab4 - Part 1
// CMSIS-RTOS main function template

#define osObjectsPublic                 // define RTOS objects in this file
#include "osObjects.h"                  // RTOS object definitions
#include "cmsis_os.h"                   // main RTOS header
#include <stdio.h>
#include <math.h>
#include "Board_LED.h"                  // LED control functions

int count = 0; // keeps track of how many times the LED has flashed
int flag  = 0; // used to control when timers restart
void delay(unsigned int); // custom delay function prototype

// Declare the three LED thread functions
void led_ThreadC (void const *argument);
void led_ThreadB (void const *argument);
void led_ThreadA (void const *argument);

// Define each thread for the RTOS
osThreadDef (led_ThreadC, osPriorityNormal, 1, 0);
osThreadDef (led_ThreadB, osPriorityNormal, 1, 0);
osThreadDef (led_ThreadA, osPriorityNormal, 1, 0);


        //Virtual Timer declaration and callback method

// Thread IDs for later reference
osThreadId T_led_ID1;
osThreadId T_led_ID2;
osThreadId T_led_ID3;

// Callback function that runs when a timer expires
void callback(void const *param){
        count = count + 1;  // simply increase the count each time the timer triggers
}
```

```
// Define three software timers that use the same callback
osTimerDef(timer0_handle, callback);
osTimerDef(timer1_handle, callback);
osTimerDef(timer2_handle, callback);



  // Thread C - Flash LED3, then signal Thread B when done

void led_ThreadC (void const *argument) {

        osTimerId timer_0 = osTimerCreate(osTimer(timer0_handle), osTimerPeriodic, NULL);
// create timer
        osTimerStart(timer_0, 500); // start timer with 500ms interval

        for (;;) {
                LED_On(3);  // turn on LED3
                Delay(10);  // short delay
                LED_Off(3); // turn off LED3

                osSignalSet(T_led_ID1,0x03); // signal this thread to continue

                if (count == 4){ // after 4 timer callbacks
                        count = 0;  // reset count
                        flag = 1;   // set flag to restart timer later

                        osSignalSet(T_led_ID2,0x01); // tell next thread (Thread B) to run
                        osTimerStop(timer_0); // stop this thread's timer
                        osSignalWait(0x03, osWaitForever); // wait until signaled again
                }

                osSignalWait(0x03, osWaitForever); // wait before next cycle

                if (flag == 1){ // restart timer only once after switching
                        osTimerStart(timer_0, 500);
                        flag = 0;
                }
        }
}
```

```
  //Thread B - Flash LED4, then signal Thread A when done

void led_ThreadB (void const *argument) {
        osTimerId timer_1 = osTimerCreate(osTimer(timer1_handle), osTimerPeriodic, NULL);
        osTimerStart(timer_1, 1000); // timer with 1-second interval

        for (;;) {
                LED_On(4);  // turn on LED4
                Delay(10);  // short delay
                LED_Off(4); // turn off LED4

                osSignalSet(T_led_ID2,0x01); // continue thread

                if (count == 4){ // after 4 timer callbacks
                        count = 0;
                        flag = 1;

                        osTimerStop(timer_1); // stop timer
                        osSignalSet(T_led_ID3,0x02); // signal Thread A
                        osSignalWait(0x01, osWaitForever); // wait to be reactivated
                }

                osSignalWait(0x01, osWaitForever); // normal wait

                if (flag == 1){
                        osTimerStart(timer_1, 1000); // restart timer
                        flag = 0;
                }
        }
}


  //Thread A - Flash LED5, then signal Thread C when done


void led_ThreadA (void const *argument){
        osTimerId timer_2 = osTimerCreate(osTimer(timer2_handle), osTimerPeriodic, NULL);
        osTimerStart(timer_2, 2000); // timer with 2-second interval
```

```
        for (;;) {
                LED_On(5);  // turn on LED5
                Delay(10);  // short delay
                LED_Off(5); // turn off LED5

                osSignalSet(T_led_ID3,0x02); // continue thread

                if (count == 2){ // after 2 timer callbacks (shorter loop)
                        count = 0;
                        flag = 1;

                        osTimerStop(timer_2); // stop timer
                        osSignalSet(T_led_ID1,0x03); // signal Thread C to start
                        osSignalWait(0x02, osWaitForever); // wait for next activation
                }

                osSignalWait(0x02, osWaitForever);

                if (flag == 1){
                        osTimerStart(timer_2, 2000); // restart timer
                        flag = 0;
                }
        }
}


  //Main function - start kernel and create threads

int main (void) {
        osKernelInitialize ();  // initialize RTOS kernel
  LED_Initialize ();        // initialize LEDs on the board

        // create the three LED threads
        T_led_ID1 = osThreadCreate(osThread(led_ThreadC), NULL);
        T_led_ID2 = osThreadCreate(osThread(led_ThreadB), NULL);
  T_led_ID3 = osThreadCreate(osThread(led_ThreadA), NULL);

        osKernelStart ();       // start RTOS scheduler (threads begin running)
        osDelay(osWaitForever);  // keep main alive forever
```

```
	for (;;); // infinite loop (never reached)
}


 //Custom busy-wait delay function (simple nested loop)


void delay (unsigned int value){
	unsigned int count1 = 0;
	unsigned int count2 = 0;

	for (count1 = 0; count1 < value; count1++){
		for (count2 = 0; count2 < count1; count2++){
			// empty loop for delay
		}
	}
}
```

➔ **PART 2**

priority_inc.c

```
/*----------------------------------------------------------------------------
 * CMSIS-RTOS 'main' function template
 *----------------------------------------------------------------------------*/

#define osObjectsPublic                 // define RTOS objects in this main module
#include "osObjects.h"                  // RTOS object definitions
#include "cmsis_os.h"                   // main RTOS header
#include <stdio.h>
#include <math.h>
#include "Board_LED.h"                  // LED support functions
#include "RTE_Components.h"             // component configuration file


/*----------------------------------------------------------------------------
	CMSIS RTX Priority Inversion Example
	Priority Inversion = leave commented lines commented
	Priority Elevation = uncomment the 2 commented lines
	Author: Anita Tino
*----------------------------------------------------------------------------*/

// Declare thread functions
void P1 (void const *argument);
```

```
void P2 (void const *argument);
void P3 (void const *argument);

// Define threads and their priorities
osThreadDef(P1, osPriorityHigh, 1, 0);         // Thread P1 - high priority
osThreadDef(P2, osPriorityNormal, 1, 0);       // Thread P2 - normal priority
osThreadDef(P3, osPriorityBelowNormal, 1, 0);  // Thread P3 - low priority

// Thread IDs to reference each thread later
osThreadId t_main, t_P1, t_P2, t_P3;

// Simple delay loop (simulates work)
void delay(){
        long k, count = 0;
        for(k = 0; k < 100000; k++){
                count++; // empty loop just to waste time
        }
}

/*------------------------------------------------------
  P1: High-priority thread
  - Does some work
  - Then needs a service from P3 (low priority)
------------------------------------------------------*/
void P1 (void const *argument) {
        for (;;)
        {
                LED_On(0);              // Turn on LED0
                delay();                // Simulate doing work

                // osThreadSetPriority(t_P3, osPriorityHigh); // Uncomment to fix priority inversion

                osSignalSet(t_P3, 0x01);    // Ask P3 to perform a task
                osSignalWait(0x02, osWaitForever); // Wait for P3 to finish

                // osThreadSetPriority(t_P3, osPriorityBelowNormal); // Uncomment to restore
priority

                LED_On(6);              // Indicate completion
                LED_Off(6);
        }
}

/*------------------------------------------------------
```

```
  P2: Medium-priority thread
  - Runs continuously and can delay P3's execution
  - Causes priority inversion problem
-----------------------------------------------------*/
void P2 (void const *argument) {
        for (;;) {
                LED_On(1);   // Turn on LED1
                LED_Off(1);  // Turn off LED1
        }
}

/*-----------------------------------------------------
  P3: Low-priority thread
  - Performs a critical section that P1 needs
  - If P2 runs first, P3 gets delayed (priority inversion)
-----------------------------------------------------*/
void P3 (void const *argument) {
        for (;;) {
                delay();                        // Do some work
                osSignalWait(0x01, osWaitForever); // Wait for signal from P1
                LED_Off(0);                     // Perform "critical" action
                osSignalSet(t_P1, 0x02);        // Notify P1 that it's done
        }
}

/*-----------------------------------------------------
  main: Initializes and starts the threads
-----------------------------------------------------*/
int main(void)
{
        osKernelInitialize ();     // Initialize RTOS kernel
        LED_Initialize();          // Initialize LEDs on the board

        t_main = osThreadGetId (); // Get ID of the main thread
        osThreadSetPriority(t_main, osPriorityHigh); // Set main as high priority temporarily

        // Create threads in a staggered order
        t_P3 = osThreadCreate(osThread(P3), NULL);  // Create low-priority thread first
        osDelay(5);
        t_P2 = osThreadCreate(osThread(P2), NULL);  // Then normal-priority thread
        osDelay(5);
        t_P1 = osThreadCreate(osThread(P1), NULL);  // Finally high-priority thread

        osThreadTerminate(t_main); // Stop the main thread (no longer needed)
```

```
    osKernelStart ();         // Start the RTOS scheduler — threads start running

    for (;;) {}               // Infinite loop (should never reach here)
}
```