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 3b
Assignment/Lab Title:	Pre-emptive scheduling with RTX

Submission Date:	October 23rd, 2025
Due Date:	October 23rd, 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

OBJECTIVES

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.

RESULTS

Question 1

- What do you notice?
    - By setting the priority of Thread2 to a higher priority than Thread1, a preemptive (interruptible) scheduling technique is created, where the higher-priority thread will execute to completion first. Since Thread1 was created first, it is also expected to run first. However, Thread1 will never be executed due to its "Normal" priority setting (in comparison to Thread2 having "AboveNormal" priority) and the fact that Thread2 executes infinitely. Conversely, if the code was programmed such that Thread2 terminates after a finite time (when its workload completes), then Thread1 would thereafter be able to execute. It is recommended that the CMSIS-RTOS API Thread Management and osPriority enumerations be consulted to familiarize yourself with the available thread-based priority options during coding.
    - **NOTE**: It is also possible to create a non-preemptive scheduling algorithm by assigning appropriate priority levels to the tasks.

Figure 1: uVision Debug Mode simulation run time using the MCB1700 Dev Board (counta)

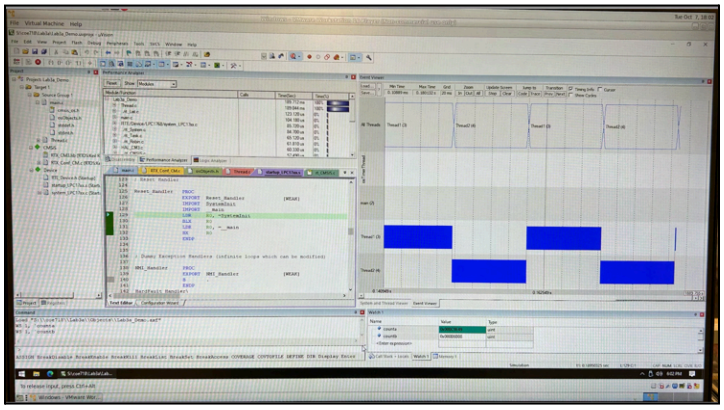


Figure 2: uVision Debug Mode simulation run time using the MCB1700 Dev Board (countb)

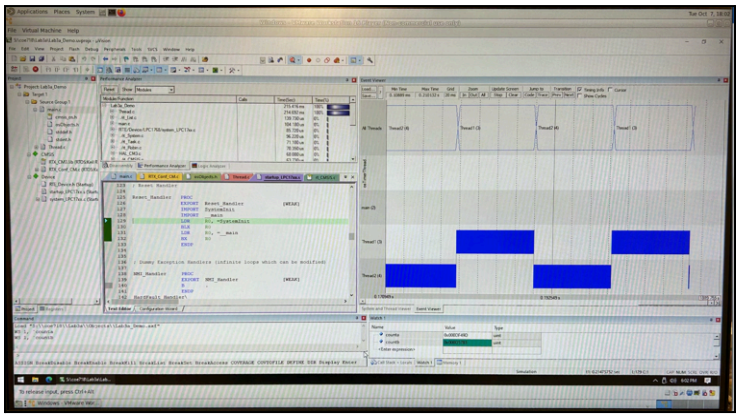
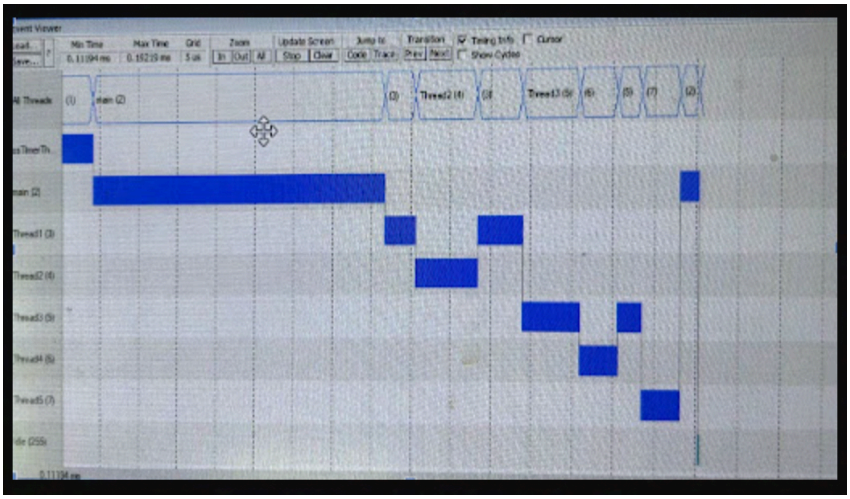
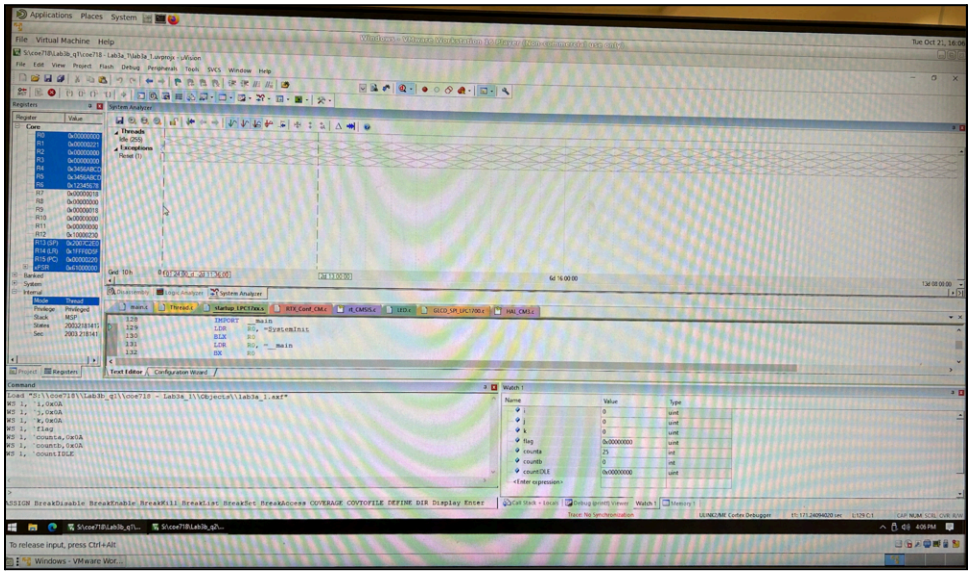


Figure 3: uVision Debug Mode simulation using the MCB1700 Dev Board (countIDLE)



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.

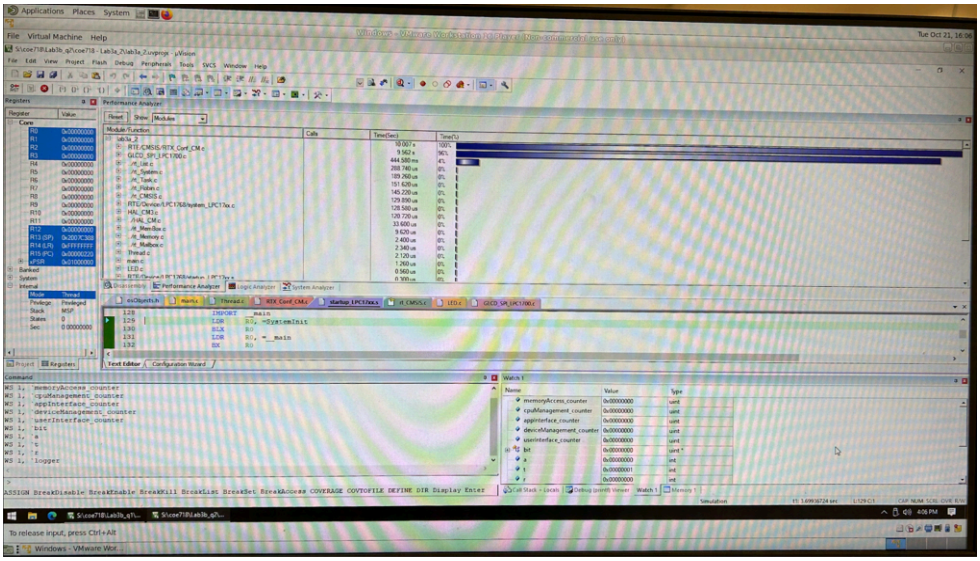


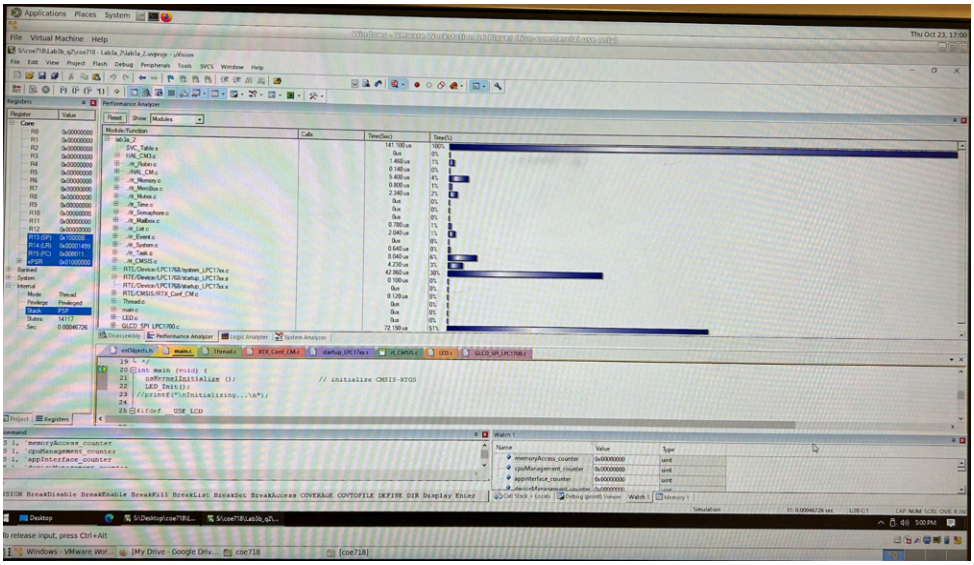
Question 2:

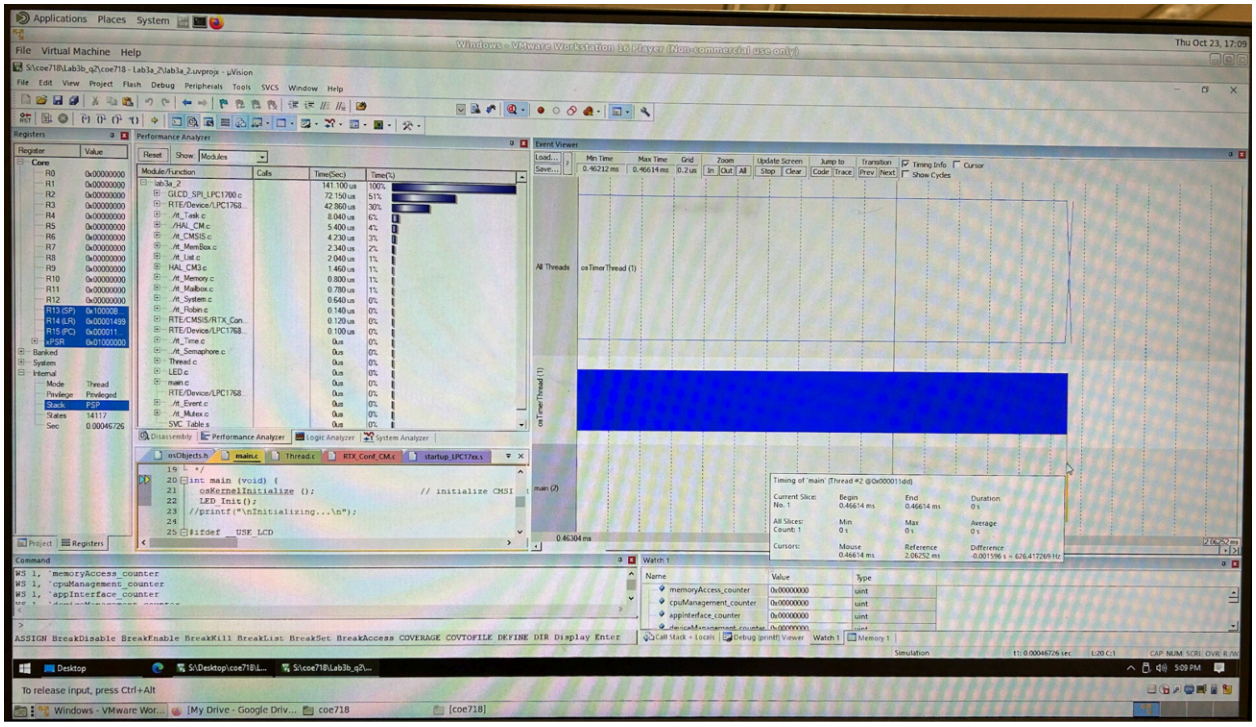
- How does the execution of the code using osThreadYield() differ from round-robin scheduling (studied in lab3a)?
    - If you were successful, you will observe short execution time slices per thread in the Event Viewer, where it appears as if the threads are running as round-robin (after several msecs). With the changes made to the program, each thread should simply increment their counter by one and pass control to the next thread of equal or greater priority using osThreadYield(). Specifically, you should observe that on average a single thread runs for 2.52µs before passing control to the next thread (which is the equivalent time spent entering the thread, incrementing the counter, and passing control).
- What is the utilization time of the processor?
    - Check the Idle Demon (refer Lab 3a manual for idle demon) variable and the task using performance-based tools.

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.





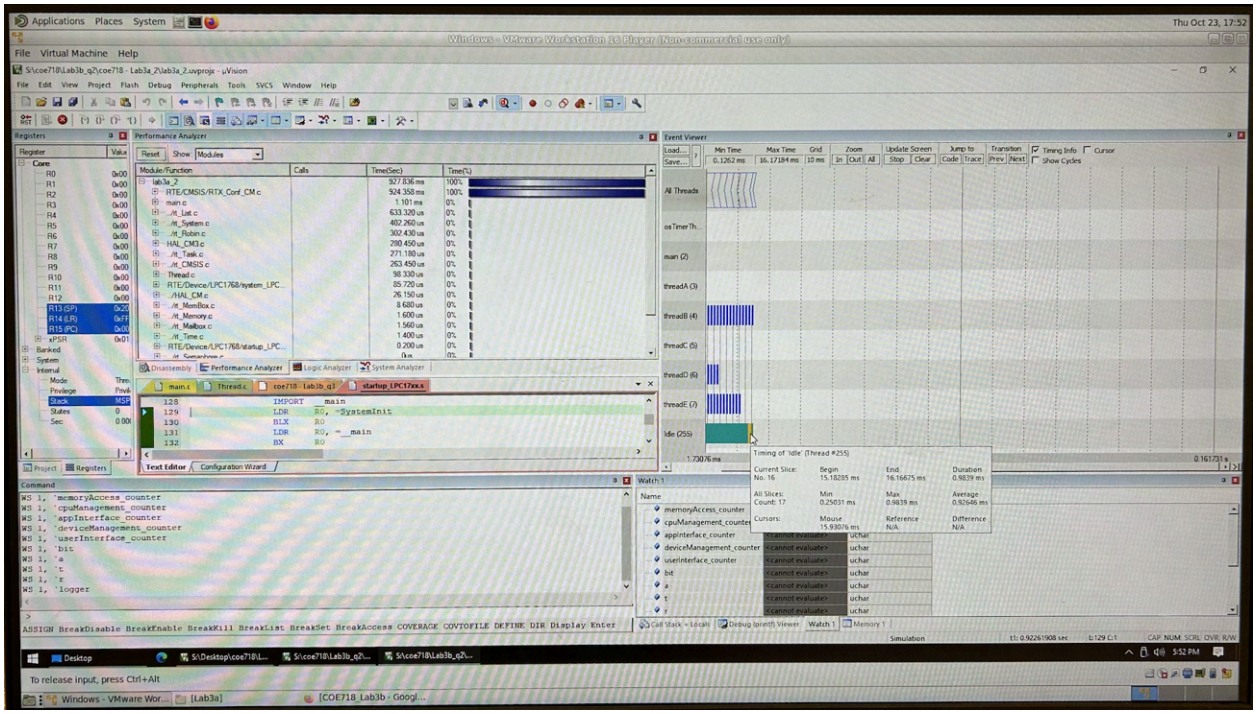


Question 3:

- Assess the Watch window and note the difference between the execution of this code and the previous code. Use the Performance Analyzer and Event Viewer to verify your findings. What is the utilization time of the CPU?

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.



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 behavior 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

➔ **Question 1**

RTX_Conf_CM.c

```
/*----------------------------------------------------------------------------
 *      CMSIS-RTOS  -  RTX
 *----------------------------------------------------------------------------
 *      Name:    RTX_Conf_CM.C
 *      Purpose: Configuration of CMSIS RTX Kernel for Cortex-M
 *      Rev.:    V4.70.1
 *----------------------------------------------------------------------------
 *
 * Copyright (c) 1999-2009 KEIL, 2009-2016 ARM Germany GmbH. All rights reserved.
 *
 * SPDX-License-Identifier: Apache-2.0
 *
 * Licensed under the Apache License, Version 2.0 (the License); you may
 * not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 * www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an AS IS BASIS, WITHOUT
 * WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 *----------------------------------------------------------------------------*/

#include "cmsis_os.h"


/*----------------------------------------------------------------------------
 *      RTX User configuration part BEGIN
 *----------------------------------------------------------------------------*/

//-------- <<< Use Configuration Wizard in Context Menu >>> ------------------
//
// <h>Thread Configuration
// =======================
//
//   <o>Number of concurrent running user threads <1-250>
//   <i> Defines max. number of user threads that will run at the same time.
//   <i> Default: 6
#ifndef OS_TASKCNT
```

```
 #define OS_TASKCNT     6
#endif

//   <o>Default Thread stack size [bytes] <64-4096:8><#/4>
//   <i> Defines default stack size for threads with osThreadDef stacksz = 0
//   <i> Default: 200
#ifndef OS_STKSIZE
 #define OS_STKSIZE     50      // this stack size value is in words
#endif

//   <o>Main Thread stack size [bytes] <64-32768:8><#/4>
//   <i> Defines stack size for main thread.
//   <i> Default: 200
#ifndef OS_MAINSTKSIZE
 #define OS_MAINSTKSIZE 50      // this stack size value is in words
#endif

//   <o>Number of threads with user-provided stack size <0-250>
//   <i> Defines the number of threads with user-provided stack size.
//   <i> Default: 0
#ifndef OS_PRIVCNT
 #define OS_PRIVCNT     0
#endif

//   <o>Total stack size [bytes] for threads with user-provided stack size <0-1048576:8><#/4>
//   <i> Defines the combined stack size for threads with user-provided stack size.
//   <i> Default: 0
#ifndef OS_PRIVSTKSIZE
 #define OS_PRIVSTKSIZE 0       // this stack size value is in words
#endif

//   <q>Stack overflow checking
//   <i> Enable stack overflow checks at thread switch.
//   <i> Enabling this option increases slightly the execution time of a thread switch.
#ifndef OS_STKCHECK
 #define OS_STKCHECK    1
#endif

//   <q>Stack usage watermark
//   <i> Initialize thread stack with watermark pattern for analyzing stack usage
(current/maximum) in System and Thread Viewer.
//   <i> Enabling this option increases significantly the execution time of osThreadCreate.
#ifndef OS_STKINIT
#define OS_STKINIT      0
```

```
#endif

//   <o>Processor mode for thread execution
//     <0=> Unprivileged mode
//     <1=> Privileged mode
//   <i> Default: Privileged mode
#ifndef OS_RUNPRIV
 #define OS_RUNPRIV     1
#endif

// </h>

// <h>RTX Kernel Timer Tick Configuration
// ======================================
//   <q> Use Cortex-M SysTick timer as RTX Kernel Timer
//   <i> Cortex-M processors provide in most cases a SysTick timer that can be used as
//   <i> as time-base for RTX.
#ifndef OS_SYSTICK
 #define OS_SYSTICK     1
#endif
//
//   <o>RTOS Kernel Timer input clock frequency [Hz] <1-1000000000>
//   <i> Defines the input frequency of the RTOS Kernel Timer.
//   <i> When the Cortex-M SysTick timer is used, the input clock
//   <i> is on most systems identical with the core clock.
#ifndef OS_CLOCK
 #define OS_CLOCK       10000000
#endif

//   <o>RTX Timer tick interval value [us] <1-1000000>
//   <i> The RTX Timer tick interval value is used to calculate timeout values.
//   <i> When the Cortex-M SysTick timer is enabled, the value also configures the SysTick timer.
//   <i> Default: 1000  (1ms)
#ifndef OS_TICK
 #define OS_TICK        10000
#endif

// </h>

// <h>System Configuration
// =======================
//
// <e>Round-Robin Thread switching
// ===============================
```

```
//
// <i> Enables Round-Robin Thread switching.
#ifndef OS_ROBIN
 #define OS_ROBIN       1
#endif

//   <o>Round-Robin Timeout [ticks] <1-1000>
//   <i> Defines how long a thread will execute before a thread switch.
//   <i> Default: 5
#ifndef OS_ROBINTOUT
 #define OS_ROBINTOUT   10
#endif

// </e>

// <e>User Timers
// ==============
//   <i> Enables user Timers
#ifndef OS_TIMERS
 #define OS_TIMERS      1
#endif

//   <o>Timer Thread Priority
//                   <1=> Low
//    <2=> Below Normal  <3=> Normal  <4=> Above Normal
//                   <5=> High
//                   <6=> Realtime (highest)
//   <i> Defines priority for Timer Thread
//   <i> Default: High
#ifndef OS_TIMERPRIO
 #define OS_TIMERPRIO   5
#endif

//   <o>Timer Thread stack size [bytes] <64-4096:8><#/4>
//   <i> Defines stack size for Timer thread.
//   <i> Default: 200
#ifndef OS_TIMERSTKSZ
 #define OS_TIMERSTKSZ  50     // this stack size value is in words
#endif

//   <o>Timer Callback Queue size <1-32>
//   <i> Number of concurrent active timer callback functions.
//   <i> Default: 4
#ifndef OS_TIMERCBQS
```

```
 #define OS_TIMERCBQS   4
#endif

// </e>

//   <o>ISR FIFO Queue size<4=>   4 entries  <8=>   8 entries
//                        <12=> 12 entries  <16=> 16 entries
//                        <24=> 24 entries  <32=> 32 entries
//                        <48=> 48 entries  <64=> 64 entries
//                        <96=> 96 entries
//   <i> ISR functions store requests to this buffer,
//   <i> when they are called from the interrupt handler.
//   <i> Default: 16 entries
#ifndef OS_FIFOSZ
 #define OS_FIFOSZ      16
#endif

// </h>

//------------- <<< end of configuration section >>> ----------------------

// Standard library system mutexes
// ===============================
//  Define max. number system mutexes that are used to protect
//  the arm standard runtime library. For microlib they are not used.
#ifndef OS_MUTEXCNT
 #define OS_MUTEXCNT    8
#endif

/*----------------------------------------------------------------------------
 *      RTX User configuration part END
 *----------------------------------------------------------------------------*/

#define OS_TRV          ((uint32_t)(((double)OS_CLOCK*(double)OS_TICK)/1E6)-1)


/*----------------------------------------------------------------------------
 *      Global Functions
 *----------------------------------------------------------------------------*/

/*--------------------------- os_idle_demon ---------------------------------*/

/// \brief The idle demon is running when no other thread is ready to run
unsigned int countIDLE = 0;
```

```
void os_idle_demon (void) {

  for (;;) {
       countIDLE++;
    /* HERE: include optional user code to be executed when no thread runs.*/
  }
}

#if (OS_SYSTICK == 0)   // Functions for alternative timer as RTX kernel timer

/*---------------------------- os_tick_init ----------------------------------*/

/// \brief Initializes an alternative hardware timer as RTX kernel timer
/// \return                         IRQ number of the alternative hardware timer
int os_tick_init (void) {
  return (-1);  /* Return IRQ number of timer (0..239) */
}

/*---------------------------- os_tick_val -----------------------------------*/

/// \brief Get alternative hardware timer's current value (0 .. OS_TRV)
/// \return                         Current value of the alternative hardware timer
uint32_t os_tick_val (void) {
  return (0);
}

/*---------------------------- os_tick_ovf -----------------------------------*/

/// \brief Get alternative hardware timer's  overflow flag
/// \return                         Overflow flag\n
///                                 - 1 : overflow
///                                 - 0 : no overflow
uint32_t os_tick_ovf (void) {
  return (0);
}

/*---------------------------- os_tick_irqack --------------------------------*/

/// \brief Acknowledge alternative hardware timer interrupt
void os_tick_irqack (void) {
  /* ... */
}

#endif   // (OS_SYSTICK == 0)
```

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

/* OS Error Codes */
#define OS_ERROR_STACK_OVF      1
#define OS_ERROR_FIFO_OVF       2
#define OS_ERROR_MBX_OVF        3
#define OS_ERROR_TIMER_OVF      4

extern osThreadId svcThreadGetId (void);

/// \brief Called when a runtime error is detected
/// \param[in]   error_code   actual error code that has been detected
void os_error (uint32_t error_code) {

  /* HERE: include optional code to be executed on runtime error. */
  switch (error_code) {
    case OS_ERROR_STACK_OVF:
      /* Stack overflow detected for the currently running task. */
      /* Thread can be identified by calling svcThreadGetId().   */
      break;
    case OS_ERROR_FIFO_OVF:
      /* ISR FIFO Queue buffer overflow detected. */
      break;
    case OS_ERROR_MBX_OVF:
      /* Mailbox overflow detected. */
      break;
    case OS_ERROR_TIMER_OVF:
      /* User Timer Callback Queue overflow detected. */
      break;
    default:
      break;
  }
  for (;;);
}


/*----------------------------------------------------------------------------
 *      RTX Configuration Functions
 *---------------------------------------------------------------------------*/

#include "RTX_CM_lib.h"

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

```
 * end of file
 *----------------------------------------------------------------------------*/


}
```

➔ **Question 2**

RTX_Conf_CM.c

```
/*----------------------------------------------------------------------------
 *      CMSIS-RTOS  -  RTX
 *----------------------------------------------------------------------------
 *      Name:    RTX_Conf_CM.C
 *      Purpose: Configuration of CMSIS RTX Kernel for Cortex-M
 *      Rev.:    V4.70.1
 *----------------------------------------------------------------------------
 *
 * Copyright (c) 1999-2009 KEIL, 2009-2016 ARM Germany GmbH. All rights reserved.
 *
 * SPDX-License-Identifier: Apache-2.0
 *
 * Licensed under the Apache License, Version 2.0 (the License); you may
 * not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 * www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an AS IS BASIS, WITHOUT
 * WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 *----------------------------------------------------------------------------*/

#include "cmsis_os.h"


/*----------------------------------------------------------------------------
 *      RTX User configuration part BEGIN
 *----------------------------------------------------------------------------*/

//-------- <<< Use Configuration Wizard in Context Menu >>> ------------------
//
```

```
// <h>Thread Configuration
// ======================
//
//   <o>Number of concurrent running user threads <1-250>
//   <i> Defines max. number of user threads that will run at the same time.
//   <i> Default: 6
#ifndef OS_TASKCNT
 #define OS_TASKCNT     6
#endif

//   <o>Default Thread stack size [bytes] <64-4096:8><#/4>
//   <i> Defines default stack size for threads with osThreadDef stacksz = 0
//   <i> Default: 200
#ifndef OS_STKSIZE
 #define OS_STKSIZE     50      // this stack size value is in words
#endif

//   <o>Main Thread stack size [bytes] <64-32768:8><#/4>
//   <i> Defines stack size for main thread.
//   <i> Default: 200
#ifndef OS_MAINSTKSIZE
 #define OS_MAINSTKSIZE 50      // this stack size value is in words
#endif

//   <o>Number of threads with user-provided stack size <0-250>
//   <i> Defines the number of threads with user-provided stack size.
//   <i> Default: 0
#ifndef OS_PRIVCNT
 #define OS_PRIVCNT     0
#endif

//   <o>Total stack size [bytes] for threads with user-provided stack size <0-1048576:8><#/4>
//   <i> Defines the combined stack size for threads with user-provided stack size.
//   <i> Default: 0
#ifndef OS_PRIVSTKSIZE
 #define OS_PRIVSTKSIZE 0       // this stack size value is in words
#endif

//   <q>Stack overflow checking
//   <i> Enable stack overflow checks at thread switch.
```

```
//   <i> Enabling this option increases slightly the execution time of a thread switch.
#ifndef OS_STKCHECK
 #define OS_STKCHECK    1
#endif

//   <q>Stack usage watermark
//   <i> Initialize thread stack with watermark pattern for analyzing stack usage
(current/maximum) in System and Thread Viewer.
//   <i> Enabling this option increases significantly the execution time of osThreadCreate.
#ifndef OS_STKINIT
#define OS_STKINIT      0
#endif

//   <o>Processor mode for thread execution
//     <0=> Unprivileged mode
//     <1=> Privileged mode
//   <i> Default: Privileged mode
#ifndef OS_RUNPRIV
 #define OS_RUNPRIV     1
#endif

// </h>

// <h>RTX Kernel Timer Tick Configuration
// ======================================
//   <q> Use Cortex-M SysTick timer as RTX Kernel Timer
//   <i> Cortex-M processors provide in most cases a SysTick timer that can be used as
//   <i> as time-base for RTX.
#ifndef OS_SYSTICK
 #define OS_SYSTICK     1
#endif
//
//   <o>RTOS Kernel Timer input clock frequency [Hz] <1-1000000000>
//   <i> Defines the input frequency of the RTOS Kernel Timer.
//   <i> When the Cortex-M SysTick timer is used, the input clock
//   <i> is on most systems identical with the core clock.
#ifndef OS_CLOCK
 #define OS_CLOCK       10000000
#endif
```

```
//   <o>RTX Timer tick interval value [us] <1-1000000>
//   <i> The RTX Timer tick interval value is used to calculate timeout values.
//   <i> When the Cortex-M SysTick timer is enabled, the value also configures the SysTick
timer.
//   <i> Default: 1000  (1ms)
#ifndef OS_TICK
 #define OS_TICK        10000
#endif

// </h>

// <h>System Configuration
// ========================
//
// <e>Round-Robin Thread switching
// ================================
//
// <i> Enables Round-Robin Thread switching.
#ifndef OS_ROBIN
 #define OS_ROBIN       1
#endif

//   <o>Round-Robin Timeout [ticks] <1-1000>
//   <i> Defines how long a thread will execute before a thread switch.
//   <i> Default: 5
#ifndef OS_ROBINTOUT
 #define OS_ROBINTOUT    10
#endif

// </e>

// <e>User Timers
// ===============
//   <i> Enables user Timers
#ifndef OS_TIMERS
 #define OS_TIMERS      1
#endif

//   <o>Timer Thread Priority
//                      <1=> Low
```

```
//     <2=> Below Normal  <3=> Normal  <4=> Above Normal
//                   <5=> High
//                   <6=> Realtime (highest)
//   <i> Defines priority for Timer Thread
//   <i> Default: High
#ifndef OS_TIMERPRIO
 #define OS_TIMERPRIO   5
#endif

//   <o>Timer Thread stack size [bytes] <64-4096:8><#/4>
//   <i> Defines stack size for Timer thread.
//   <i> Default: 200
#ifndef OS_TIMERSTKSZ
 #define OS_TIMERSTKSZ  280     // this stack size value is in words
#endif

//   <o>Timer Callback Queue size <1-32>
//   <i> Number of concurrent active timer callback functions.
//   <i> Default: 4
#ifndef OS_TIMERCBQS
 #define OS_TIMERCBQS   4
#endif

// </e>

//   <o>ISR FIFO Queue size<4=>   4 entries  <8=>   8 entries
//                   <12=> 12 entries  <16=> 16 entries
//                   <24=> 24 entries  <32=> 32 entries
//                   <48=> 48 entries  <64=> 64 entries
//                   <96=> 96 entries
//   <i> ISR functions store requests to this buffer,
//   <i> when they are called from the interrupt handler.
//   <i> Default: 16 entries
#ifndef OS_FIFOSZ
 #define OS_FIFOSZ      16
#endif

// </h>

//------------- <<< end of configuration section >>> ----------------------
```

```
// Standard library system mutexes
// ================================
//  Define max. number system mutexes that are used to protect
//  the arm standard runtime library. For microlib they are not used.
#ifndef OS_MUTEXCNT
 #define OS_MUTEXCNT    8
#endif

/*----------------------------------------------------------------------------
 *      RTX User configuration part END
 *----------------------------------------------------------------------------*/

#define OS_TRV          ((uint32_t)(((double)OS_CLOCK*(double)OS_TICK)/1E6)-1)


/*----------------------------------------------------------------------------
 *      Global Functions
 *----------------------------------------------------------------------------*/

/*--------------------------- os_idle_demon ---------------------------------*/

/// \brief The idle demon is running when no other thread is ready to run
void os_idle_demon (void) {

  for (;;) {
    /* HERE: include optional user code to be executed when no thread runs.*/
  }
}

#if (OS_SYSTICK == 0)   // Functions for alternative timer as RTX kernel timer

/*--------------------------- os_tick_init ----------------------------------*/

/// \brief Initializes an alternative hardware timer as RTX kernel timer
/// \return                         IRQ number of the alternative hardware timer
int os_tick_init (void) {
  return (-1);  /* Return IRQ number of timer (0..239) */
}
```

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

/// \brief Get alternative hardware timer's current value (0 .. OS_TRV)
/// \return                      Current value of the alternative hardware timer
uint32_t os_tick_val (void) {
  return (0);
}

/*---------------------------- os_tick_ovf ----------------------------------*/

/// \brief Get alternative hardware timer's  overflow flag
/// \return                      Overflow flag\n
///                            - 1 : overflow
///                            - 0 : no overflow
uint32_t os_tick_ovf (void) {
  return (0);
}

/*---------------------------- os_tick_irqack -------------------------------*/

/// \brief Acknowledge alternative hardware timer interrupt
void os_tick_irqack (void) {
  /* ... */
}

#endif   // (OS_SYSTICK == 0)

/*---------------------------- os_error -------------------------------------*/

/* OS Error Codes */
#define OS_ERROR_STACK_OVF      1
#define OS_ERROR_FIFO_OVF       2
#define OS_ERROR_MBX_OVF        3
#define OS_ERROR_TIMER_OVF      4

extern osThreadId svcThreadGetId (void);

/// \brief Called when a runtime error is detected
/// \param[in]   error_code   actual error code that has been detected
void os_error (uint32_t error_code) {
```

```
  /* HERE: include optional code to be executed on runtime error. */
  switch (error_code) {
    case OS_ERROR_STACK_OVF:
      /* Stack overflow detected for the currently running task. */
      /* Thread can be identified by calling svcThreadGetId().   */
      break;
    case OS_ERROR_FIFO_OVF:
      /* ISR FIFO Queue buffer overflow detected. */
      break;
    case OS_ERROR_MBX_OVF:
      /* Mailbox overflow detected. */
      break;
    case OS_ERROR_TIMER_OVF:
      /* User Timer Callback Queue overflow detected. */
      break;
    default:
      break;
  }
  for (;;);
}


/*----------------------------------------------------------------------------
 *      RTX Configuration Functions
 *----------------------------------------------------------------------------*/

#include "RTX_CM_lib.h"

/*----------------------------------------------------------------------------
 * end of file
 *----------------------------------------------------------------------------*/
```

➔ **Question 3**

RTX_Conf_CM.c

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#include <math.h>
#include "cmsis_os.h"
```

```
#include "RTL.H"
#include "LPC17xx.H"
#include "GLCD.h"

#define __FI        1                   // Font index for LCD display
#define __USE_LCD   0                   // Uncomment for DEMO mode

#define PI 3.142

double factor;

// Thread function declarations
void threadA (void const *argument);
void threadB (void const *argument);
void threadC (void const *argument);
void threadD (void const *argument);
void threadE (void const *argument);

osThreadId id_Thread_A, id_Thread_B, id_Thread_C, id_Thread_D, id_Thread_E;

osThreadDef (threadA, osPriorityAboveNormal, 1, 0);   // Priority level #2
osThreadDef (threadB, osPriorityBelowNormal, 1, 0);   // Priority level #3
osThreadDef (threadC, osPriorityHigh, 1, 0);          // Priority level #1 (highest)
osThreadDef (threadD, osPriorityAboveNormal, 1, 0);   // Priority level #2
osThreadDef (threadE, osPriorityBelowNormal, 1, 0);   // Priority level #3

//  initialize and create all threads
int Init_Thread (void) {
  id_Thread_A = osThreadCreate (osThread(threadA), NULL);
  id_Thread_B = osThreadCreate (osThread(threadB), NULL);
  id_Thread_C = osThreadCreate (osThread(threadC), NULL);
  id_Thread_D = osThreadCreate (osThread(threadD), NULL);
  id_Thread_E = osThreadCreate (osThread(threadE), NULL);

  return(0);
}


// Thread A: Performs addition operation in a loop
__task void threadA (void const *arg) {
```

```
  char out[20];
  int A = 0;
  int x = 0;
  for (x = 0; x < 257; x++) {
    A = A + (x + (x + 2));
    os_tsk_pass();                          // Pass control to next ready task
  }
    //printf("Thread A done. A = %d\n", A);

}

// Thread B: Calculates exponential series using factorial
__task void threadB (void const *arg) {
  float B = 0;
  char out[20];
  int i = 0;
  factor = 1;
  for (i = 1; i < 17; i++) {
    factor = factor * i;                    // Calculate factorial
    B = B + pow(2, i) / factor;             // Add term to series
    os_tsk_pass();                          // Pass control to next ready task
  }
   // printf("Thread B done. B = %.3f\n", B);

}

// Thread C: Performs division and addition operations
__task void threadC (void const *arg) {
  float C = 0;
  char out[20];
  int n = 0;
  for (n = 1; n < 17; n++) {
    C = C + (n + 1) / n;                    // Simple arithmetic operation
  }
   // printf("Thread C done. C = %.3f\n", C);

}

// Thread D: Calculates factorial-based power series
__task void threadD (void const *arg) {
```

```
  float D = 0;
  char out[20];
  int m = 0;
  factor = 1;
  for (m = 0; m < 6; m++) {
    factor = factor * m;                    // Calculate factorial
    if (factor == 0) {
      factor = 1;                           // Avoid division by zero
    } else {
      os_tsk_pass();                        // Pass control to next ready task
      D = D + pow(5, m) / (double)factor;      // Add term to power series
    }

  }
   //printf("Thread D done. D = %.3f\n", D);
}

// Thread E: Calculates area-like values using p and radius
__task void threadE (void const *arg) {
  int E = 0;
  char out[20];
  int p = 0;
  int radius = 1;
  for (p = 1; p < 13; p++) {
    E = E + p * PI * (pow(radius, 2));        // Calculate area * iteration
    os_tsk_pass();                          // Pass control to next ready task
  }
    //printf("Thread E done. E = %.3f\n", E);

}
```