Ryerson University

Department of Electrical, Computer, & Biomedical Engineering
Faculty of Engineering & Architectural Science

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

Instructor:	**Gul Khan**

Assignment/Lab Number:	*Lab 3a*
Assignment/Lab Title:	*RTX based Multitasking with Round-Robin Scheduling*

Submission Date:	**October 8, 2025**
Due Date:	**October 8, 2025**

Student LAST Name	**Student FIRST Name**	**Student Number**	**Section**	**Signature***
Grewal	**Arman**	**501100160**	**03**	**AG**

Table Of Contents

1.0 LAB OBJECTIVE

The objective of this lab is to design and evaluate a CMSIS-RTOS–based application on the NXP LPC1768 that demonstrates bit-banding, inter-thread synchronization (signals, semaphore, mutexCP), and simple U workload construction using conditional logic and barrel-shifter–style operations. The work includes implementing equal-priority threads, validating deterministic execution order, and comparing a headless analysis build with a demo build that visualizes active threads via the LCD and LEDs to assess timing and overhead effects.

2.0 Part I - Round Robin scheduling

main.c Code:

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

#define osObjectsPublic                     // define objects in main module
#include "osObjects.h"                      // RTOS object definitions
#include "GLCD.h"
#include "Board_LED.h"

#define __FI        1                      /* Font index 16x24               */

extern int Init_Thread (void);

/*
 * main: initialize and start the system
 */
int main (void) {

  osKernelInitialize ();                    // initialize CMSIS-RTOS
  if (Init_Thread() != 0) {
    for(;;) { }
  }
  osKernelStart ();                         // start thread execution
  osDelay(osWaitForever);
}

```

Thread.c Code:

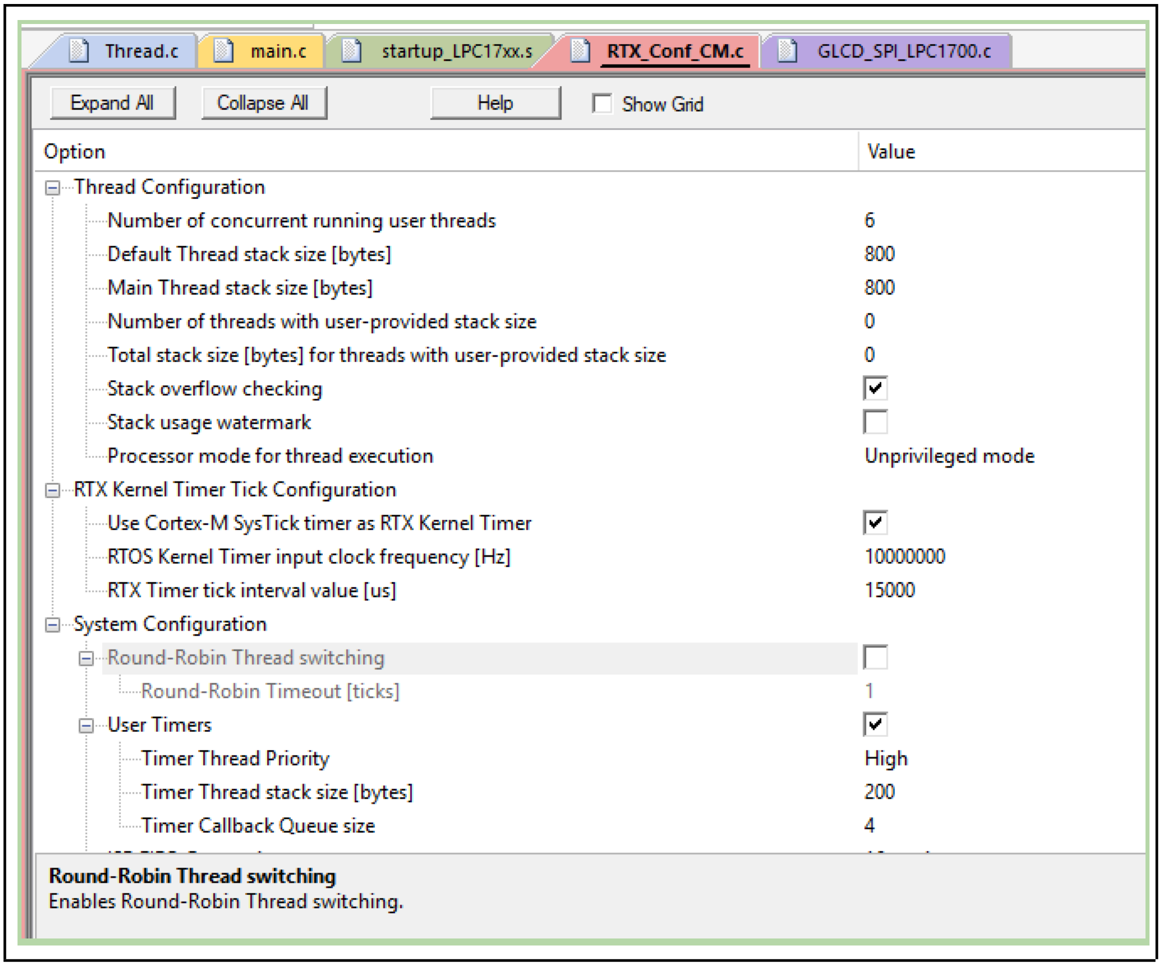
```c
#include "cmsis_os.h"         // CMSIS RTOS header file
#include "LPC17xx.h"

#define __FI        1         /* Font index 16x24 (unused in analysis) */
enum { N = 4096 };
static int g_arr[N];

static void Data_Init(void) {
  int i;
  for (i = 0; i < N; ++i) g_arr[i] = i;
}

/*----------------------------------------------------------------------------
 *      Threads
 *---------------------------------------------------------------------------*/

void Thread1 (void const *); // thread function
void Thread2 (void const *); // thread function
void Thread3 (void const *); // thread function

osThreadId tid_Thread;        // thread id
osThreadDef (Thread1, osPriorityNormal, 1, 0);

osThreadId tid2_Thread;       // thread id
osThreadDef (Thread2, osPriorityNormal, 1, 0);

osThreadId tid3_Thread;       // thread id
osThreadDef (Thread3, osPriorityNormal, 1, 0);

int Init_Thread (void) {

  Data_Init();

  tid_Thread  = osThreadCreate(osThread(Thread1), NULL);
  tid2_Thread = osThreadCreate(osThread(Thread2), NULL);
  tid3_Thread = osThreadCreate(osThread(Thread3), NULL);

  if (!tid_Thread || !tid2_Thread || !tid3_Thread) {
    return -1;
  }
  return 0;
}
```

```c

/* ================== Thread 1: Linear search (O(N)) ================== */
void Thread1 (void const *argument) {
  const int ITERATIONS = 200000;
  unsigned rng = 1u;
  int t;

  (void)argument;

  for (t = 0; t < ITERATIONS; ++t) {
    rng = rng * 1664525u + 1013904223u;
    {
      int key = (int)(rng % (unsigned)N);
      int i;
      for (i = 0; i < N; ++i) {
        if (g_arr[i] == key) break;
      }
    }
  }

  osThreadTerminate(NULL);
}

/* ================== Thread 2: Binary search (O(log N)) ================== */
void Thread2 (void const *argument) {
  const int ITERATIONS = 200000;
  unsigned rng = 2u;
  int t;

  (void)argument;

  for (t = 0; t < ITERATIONS; ++t) {
    rng = rng * 1664525u + 1013904223u;
    {
      int key = (int)(rng % (unsigned)N);
      int lo = 0, hi = N - 1;
      while (lo <= hi) {
        int mid = lo + ((hi - lo) >> 1);
        if (g_arr[mid] == key) break;
        if (g_arr[mid] < key) lo = mid + 1; else hi = mid - 1;
      }
```

```c
    }
  }

  osThreadTerminate(NULL);
}

/* ================== Thread 3: Jump search (~O(sqrt(N))) ================== */
void Thread3 (void const *argument) {
  const int ITERATIONS = 200000;
  const unsigned STEP = 64u;
  unsigned rng = 3u;
  int t;

  (void)argument;

  for (t = 0; t < ITERATIONS; ++t) {
    rng = rng * 1664525u + 1013904223u;
    {
      int key = (int)(rng % (unsigned)N);
      unsigned prev = 0u, curr = STEP;

      while (curr <= (unsigned)N) {
        if (g_arr[curr - 1] >= key) break;
        prev = curr;
        curr += STEP;
      }
      if (curr > (unsigned)N) curr = (unsigned)N;

      {
        unsigned i;
        for (i = prev; i < curr; ++i) {
          if (g_arr[i] == key) break;
        }
      }
    }
  }

  osThreadTerminate(NULL);
}

```

Inside thread.c, three different threads were implemented. Each of the threads perform a different type of search on an array. Thread one is linear search with a time complexity of O(n), thread two is binary search with time complexity O(log n) and my third thread is jump search with time complexity O(sqrt(n)). All three threads perform a search on a array for as long as the iterations are set. The faster searching algorithms will end faster compared to the more complex ones as they run faster (as seen in analysis below).

RTX_Conf_CM.c Configuration Wizard File:


Thread.c
main.c
startup_LPC17xx.s
RTX_Conf_CM.c
GLCD_SPI_LPC1700.c
Expand All
Collapse All
Help
Show Grid
Option
Value
Thread Configuration
Number of concurrent running user threads 6
Default Thread stack size [bytes] 800
Main Thread stack size [bytes] 800
Number of threads with user-provided stack size 0
Total stack size [bytes] for threads with user-provided stack size 0
Stack overflow checking
Stack usage watermark
Processor mode for thread execution Unprivileged mode
RTX Kernel Timer Tick Configuration
Use Cortex-M SysTick timer as RTX Kernel Timer
RTOS Kernel Timer input clock frequency [Hz] 10000000
RTX Timer tick interval value [us] 15000
System Configuration
Round-Robin Thread switching
Round-Robin Timeout [ticks] 1
User Timers
Timer Thread Priority High
Timer Thread stack size [bytes] 200
Timer Callback Queue size 4
Round-Robin Thread switching
Enables Round-Robin Thread switching.

Analysis:

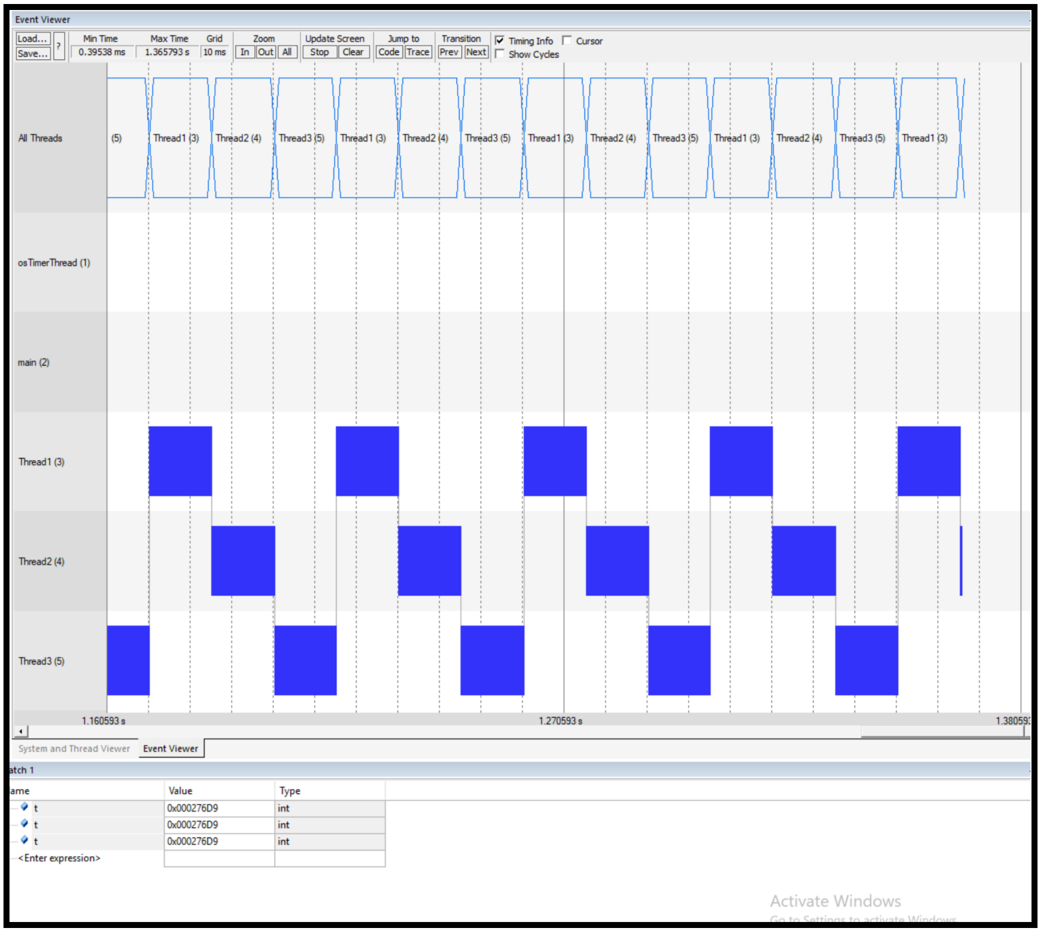


Figure 1.0: Event Viewer window for Part I (middle of process)

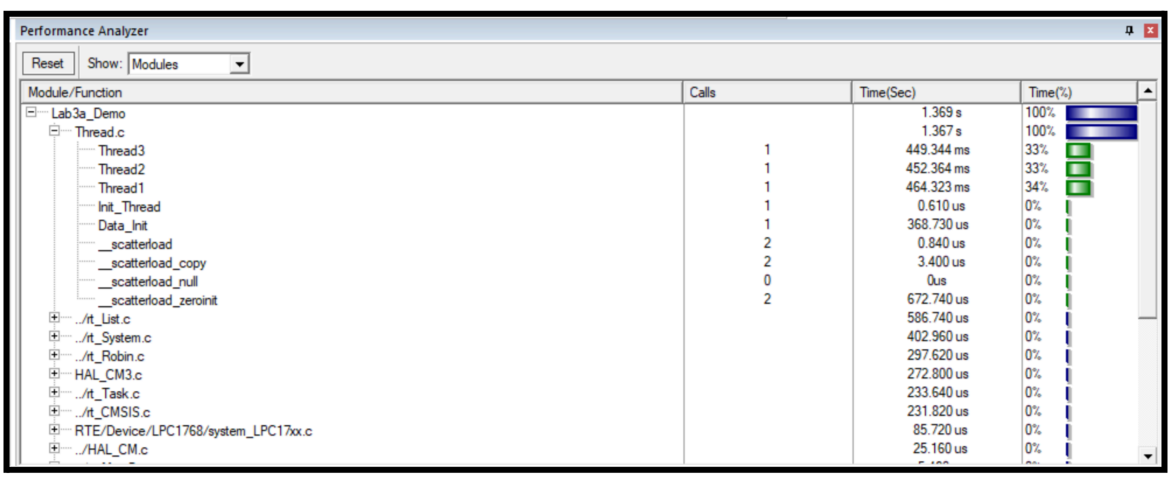


Figure 2.0: Performance Analyzer window for Part I (middle of process).

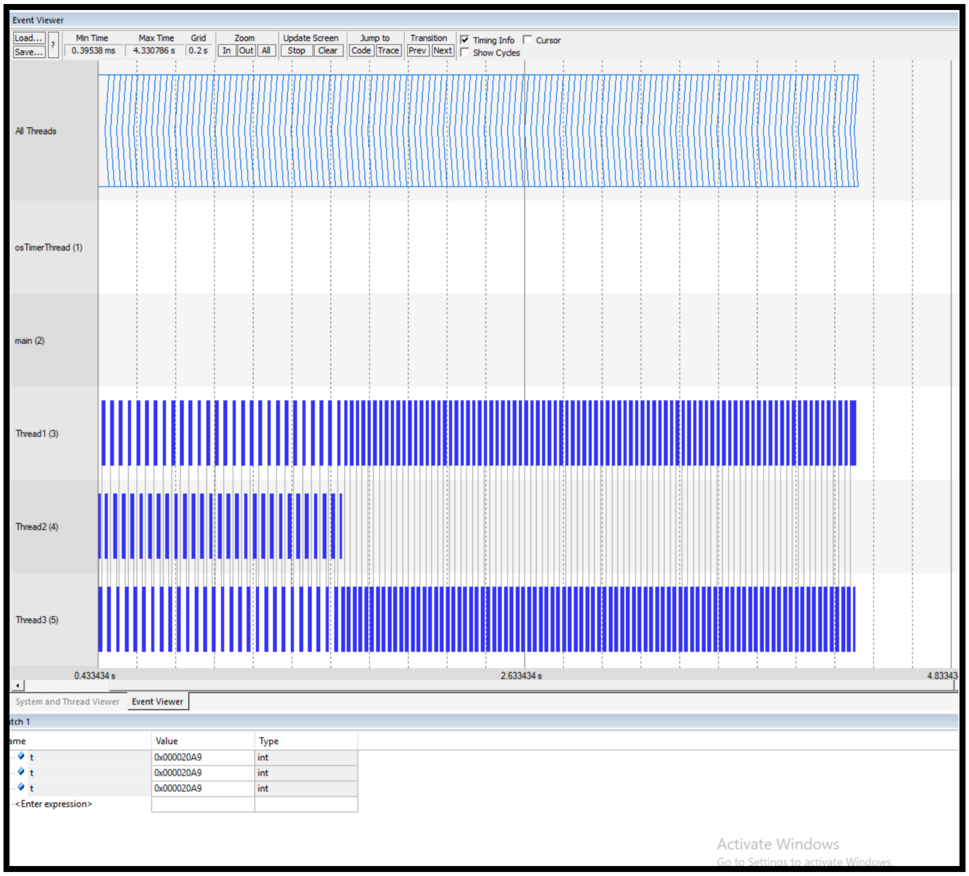


Figure 3.0: Event Viewer for Part I (End of Processes).

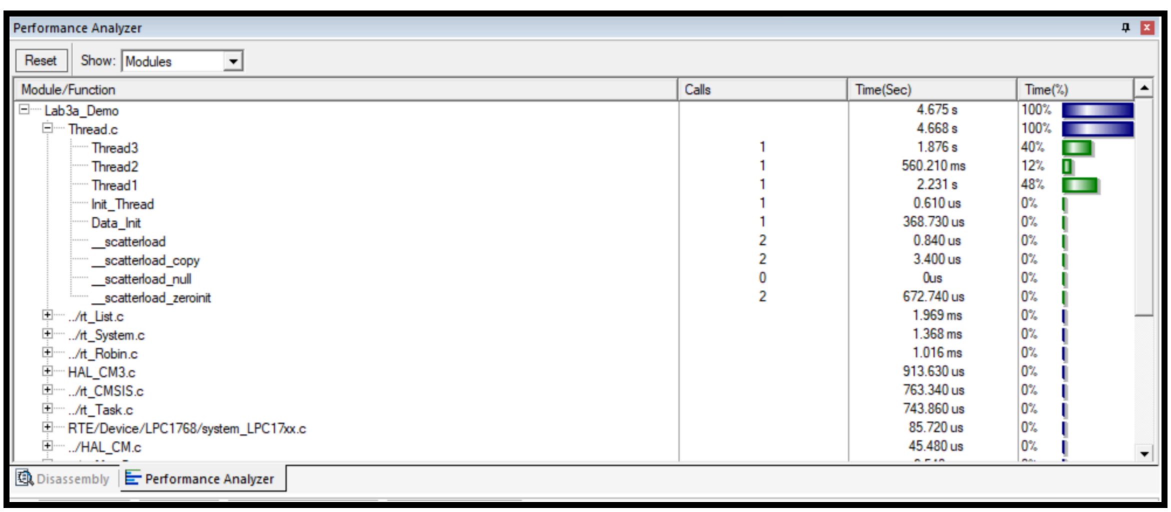


Figure 4.0: Performance Analyzer for Part I (End of Processes).

Figure one and two present the event and performance analyzer windows while in the middle of the process. In these figures it is visible that all three threads have the same priority and the round robin is doing its thing by going through all the processes with the same priority. In the performance analyzer in figure two, it presents how the time % is evenly distributed between three threads due to round robin. However, since thread two is a much faster algorithm, it finishes faster compared to thread one and three. As a result the event viewer shows that thread two finishes first and thread one and three continue on. The performance analyzer in figure four shows how the time % for dropped for thread two.

2.0 Part II - Operating System (OS) Round Robin

main.c Code:

Thread.c | main.c | startup_LPC17xx.s | RTX_Conf_CM.c

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

#define osObjectsPublic                     // define objects in main module
#include "osObjects.h"                      // RTOS object definitions
#include "GLCD.h"
#include "Board_LED.h"

#define __FI        1                      /* Font index 16x24               */

extern int Init_Thread (void);

/*
 * main: initialize and start the system
 */
int main (void) {

  osKernelInitialize ();                    // initialize CMSIS-RTOS
  if (Init_Thread() != 0) {
    for(;;) { /* creation failed */ }
  }
  osKernelStart ();                         // start thread execution
  osDelay(osWaitForever);
}

```

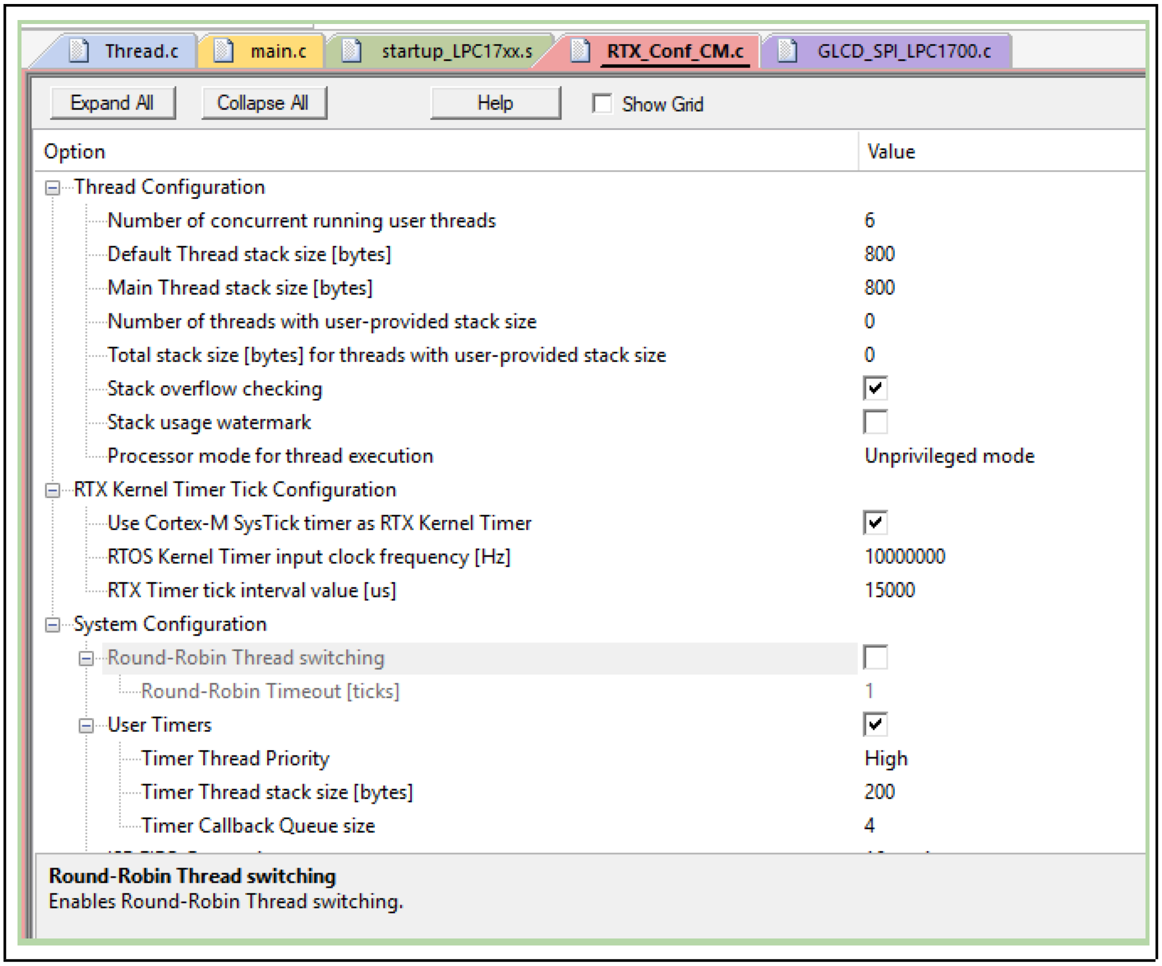
Thread.c Code:

```c
#include "cmsis_os.h"        // CMSIS RTOS header file
#include "LPC17xx.h"
#include <string.h>

#define __FI        1        /* Font index 16x24 (unused in analysis) */

/* globals */
volatile unsigned mm_access_cnt = 0;
volatile unsigned cpu_access_cnt = 0;
volatile unsigned app_cnt = 0;
volatile unsigned dev_cnt = 0;
volatile unsigned ui_users = 0;
char logger[128];


// Bit Band Macros used to calculate the alias address at run time
#define ADDRESS(x)    (*((volatile unsigned long *)(x)))
#define BitBand(x, y)   ADDRESS(((unsigned long)(x) & 0xF0000000) | 0x02000000 |(((unsigned long)(x) & 0x000FFFFF) << 5) | ((y) << 2))

#define MEM_CPU  0x01  // Memory -> CPU
#define CPU_MEM  0x02  // CPU -> Memory
#define APP_DEV  0x04  // App  -> Device
#define DEV_APP  0x08  // Device -> App



/*----------------------------------------------------------------------------
 *      Threads
 *---------------------------------------------------------------------------*/

void Memory_Management (void const *); // thread function
void CPU_Management (void const *); // thread function
void App_Interface (void const *); // thread function
void Device_Management (void const *); // thread function
void User_Interface (void const *); // thread function

osThreadId tid_Thread;        // thread id
osThreadDef (Memory_Management, osPriorityNormal, 1, 0);

osThreadId tid2_Thread;       // thread id
osThreadDef (CPU_Management, osPriorityNormal, 1, 0);

osThreadId tid3_Thread;       // thread id
osThreadDef (App_Interface, osPriorityNormal, 1, 0);

osThreadId tid4_Thread;       // thread id
osThreadDef (Device_Management, osPriorityNormal, 1, 0);

osThreadId tid5_Thread;       // thread id
osThreadDef (User_Interface, osPriorityNormal, 1, 0);
```

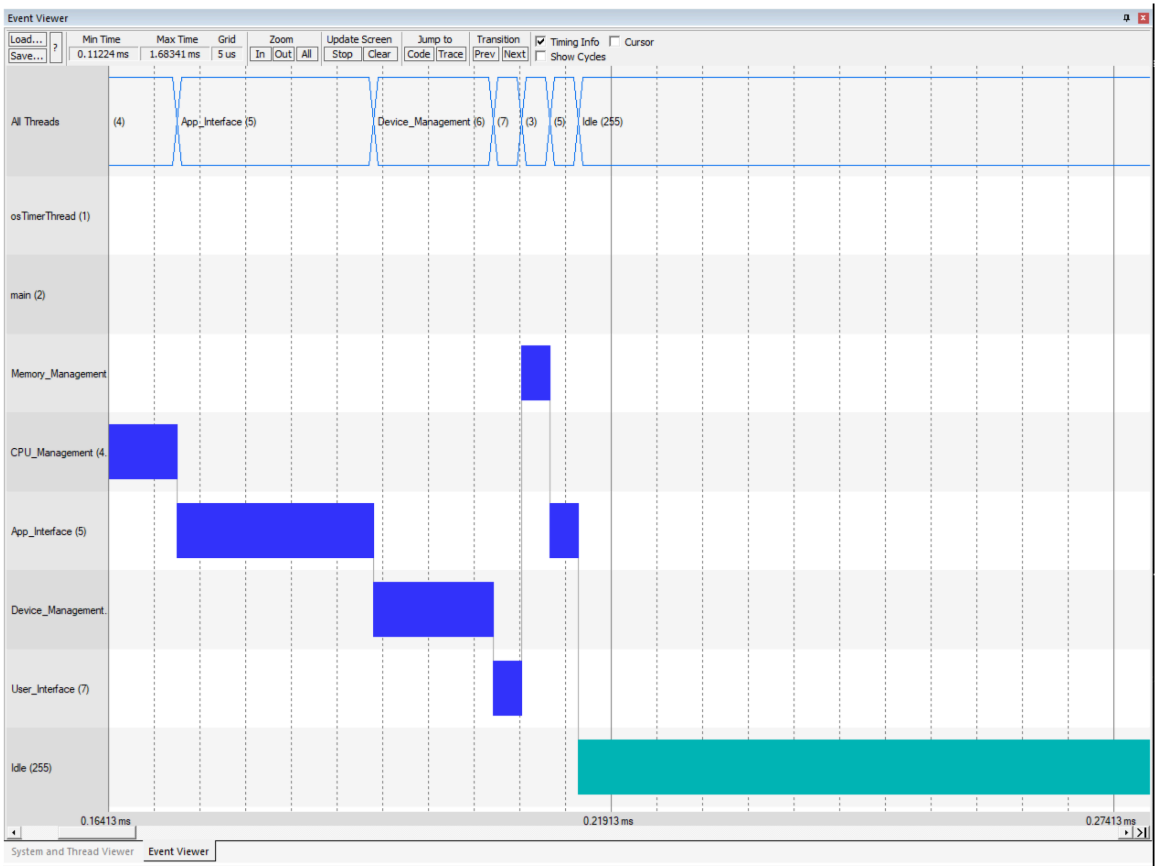
```c
osMutexId   log_lock;
osMutexDef  (log_lock);

int Init_Thread (void) {

  log_lock   = osMutexCreate(osMutex(log_lock));

  tid_Thread  = osThreadCreate(osThread(Memory_Management), NULL);
  tid2_Thread = osThreadCreate(osThread(CPU_Management), NULL);
  tid3_Thread = osThreadCreate(osThread(App_Interface), NULL);
  tid4_Thread = osThreadCreate(osThread(Device_Management), NULL);
  tid5_Thread = osThreadCreate(osThread(User_Interface), NULL);

  if (!tid_Thread || !tid2_Thread || !tid3_Thread || !tid4_Thread || !tid5_Thread) {
    return -1;
  }
  return 0;
}


void Memory_Management (void const *argument) {
  volatile unsigned long * bit;
  mm_access_cnt+=1;
  bit=  &BitBand(&LPC_ADC->ADCR, 24);
  *bit = 1;
  *bit = 0;
  osSignalSet(tid2_Thread, MEM_CPU);
  osSignalWait(CPU_MEM, osWaitForever);
  osDelay(1);
  osThreadTerminate(NULL);
}

/* ================== Thread 2: Binary search (O(log N)) ================== */
void CPU_Management (void const *argument) {
  int r1, r2, r3;
  osSignalWait(MEM_CPU, osWaitForever);
  cpu_access_cnt+=1;
  r1 = 1;
  r2 = 0;
  r3 = 5;
  while (r2 <= 0x18) {
    if ((r1 - r2) > 0) {
      r1 = r1 + 2;
      r2 = r1 + (r3 * 4);
      r3 = r3 / 2;
    } else {
      r2 = r2 + 1;
    }
```

```c
  }
  (void)r1; (void)r2; (void)r3;

  osSignalSet(tid_Thread, CPU_MEM);

  osThreadTerminate(NULL);
}

/* ================== Thread 3: Jump search (~O(sqrt(N))) ================== */
void App_Interface (void const *argument) {
  osMutexWait(log_lock, osWaitForever);
  strncpy(logger, "App_Interface:", sizeof(logger) - 1);
  osMutexRelease(log_lock);

  osSignalSet(tid4_Thread, APP_DEV);
  osSignalWait(DEV_APP, osWaitForever);

  app_cnt += 1;
  osDelay(1);
  osThreadTerminate(NULL);
}

void Device_Management (void const *argument) {
  size_t have;

  osSignalWait(APP_DEV, osWaitForever);

  osMutexWait(log_lock, osWaitForever);
  have = strlen(logger);
  strncat(logger, " DEVICE:done", (sizeof(logger) - 1) - have);
  osMutexRelease(log_lock);

  osSignalSet(tid3_Thread, DEV_APP);

  dev_cnt += 1;
  osDelay(1);
  osThreadTerminate(NULL);
}

void User_Interface (void const *argument) {
  ui_users += 1;
  osDelay(1);
  osThreadTerminate(NULL);
}
```

RTX_Conf_CM.c Configuration Wizard File:


Thread.c
main.c
startup_LPC17xx.s
RTX_Conf_CM.c
GLCD_SPI_LPC1700.c
Expand All
Collapse All
Help
Show Grid
Option
Value
Thread Configuration
Number of concurrent running user threads
6
Default Thread stack size [bytes]
800
Main Thread stack size [bytes]
800
Number of threads with user-provided stack size
0
Total stack size [bytes] for threads with user-provided stack size
0
Stack overflow checking
Stack usage watermark
Processor mode for thread execution
Unprivileged mode
RTX Kernel Timer Tick Configuration
Use Cortex-M SysTick timer as RTX Kernel Timer
RTOS Kernel Timer input clock frequency [Hz]
10000000
RTX Timer tick interval value [us]
15000
System Configuration
Round-Robin Thread switching
Round-Robin Timeout [ticks]
1
User Timers
Timer Thread Priority
High
Timer Thread stack size [bytes]
200
Timer Callback Queue size
4
Round-Robin Thread switching
Enables Round-Robin Thread switching.

Analysis:


Event Viewer
Load...
Save...
Min Time 0.11224 ms
Max Time 1.68341 ms
Grid 5 us
Zoom In Out All
Update Screen Stop Clear
Jump to Code Trace
Transition Prev Next
Timing Info
Show Cycles
Cursor
All Threads
(4)
App_Interface (5)
Device_Management (6)
(7)
(3)
(5)
Idle (255)
osTimerThread (1)
main (2)
Memory_Management
CPU_Management (4.
App_Interface (5)
Device_Management.
User_Interface (7)
Idle (255)
0.16413 ms
0.21913 ms
0.27413 ms
System and Thread Viewer
Event Viewer

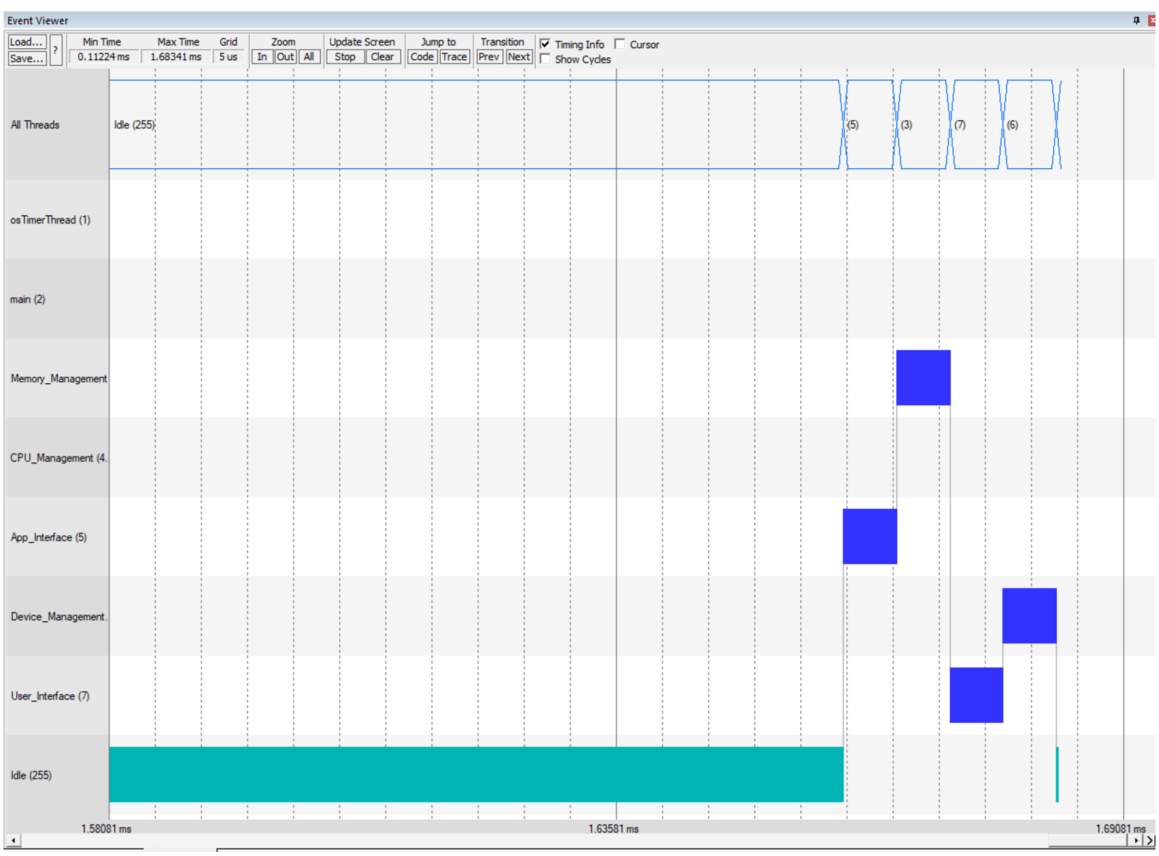

Event Viewer
Load...
Save...
Min Time 0.11224 ms
Max Time 1.68341 ms
Grid 5 us
Zoom In Out All
Update Screen Stop Clear
Jump to Code Trace
Transition Prev Next
Timing Info
Show Cycles
Cursor
All Threads
Idle (255)
(5)
(3)
(7)
(6)
osTimerThread (1)
main (2)
Memory_Management
CPU_Management (4.
App_Interface (5)
Device_Management.
User_Interface (7)
Idle (255)
1.58081 ms
1.63581 ms
1.69081 ms

Figure 5.0: Performance Analyzer for Part II .

Watch 1

Name	Value	Type
mm_access_cnt	0x00000001	uint
cpu_access_cnt	0x00000001	uint
app_cnt	0x00000001	uint
dev_cnt	0x00000001	uint
ui_users	0x00000001	uint
logger	0x10000068 logger[] "App_Interface: DEVICE:done"	uchar[128]
<Enter expression>		

Figure 6.0: Watch window for Part II .

Thread.c		76.810 us
User_Interface	1	0.260 us
Device_Manag...	1	0.790 us
App_Interface	1	0.730 us
CPU_Managem...	1	0.940 us
Memory_Manag...	1	0.590 us

Figure 7.0: Performance analyzer for Part II .

Figures five to seven present the analysis figures for part II of this lab. In this part, the OS threads were implemented. Figure 5.0 shows how the threads are traversed based on where signals are sent. The watch window in figure 6.0 shows the values for the global variables after all threads have been executed.

Pros vs. Cons of Round Robin:

With this lab, it helps to identify and outline the pros and cons of a round-robin scheme for the given operating system problem. One of the biggest advantages is that this method is simple and fair. Every thread that is ready will eventually get a turn. Similarly, there was no starvation. Each thread was bounded by wait time per slice, so eventually every thread would eventually get work done. Overall, this scheme works well for short, finite tasks such as given in this problem.

However there are also downsides to this scheme. First there is no urgency. There was no prioritization done for these threads, so a thread won't be able to jump a line no matter how important it is. Additionally, with these tiny time slices, the processor keeps saving and restoring thread state instead of making progress, ballooning overhead. Finally, blocking on signals or holding a mutex too long can stall related threads, and tuning the time slice is a trade-off: too short increases overhead, too long hurts responsiveness.