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

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

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

<u>Table Of Contents</u>

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 preemptive based scheduling for different tasks. This lab looks into assigning priorities to different tasks and running primitive scheduling on said tasks.

2.0 Part I - Mathematical Preemptive Tasks

main.c Code:

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

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

extern int Init_Thread (void);


/*
 * main: initialize and start the system
 */
int main (void) {
  osKernelInitialize ();                    // initialize CMSIS-RTOS
	Init_Thread ();
  osKernelStart ();                         // start thread execution
	osDelay(osWaitForever);
}
```

Main.c code above is where the code starts from. It starts off by initializing the Kernel and creating the threads using osKernelInitialize and Init_Thread. Once the kernel has been started, the scheduler starts cycling through the threads while the main function keeps on hold forever.

Thread.c Code:

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

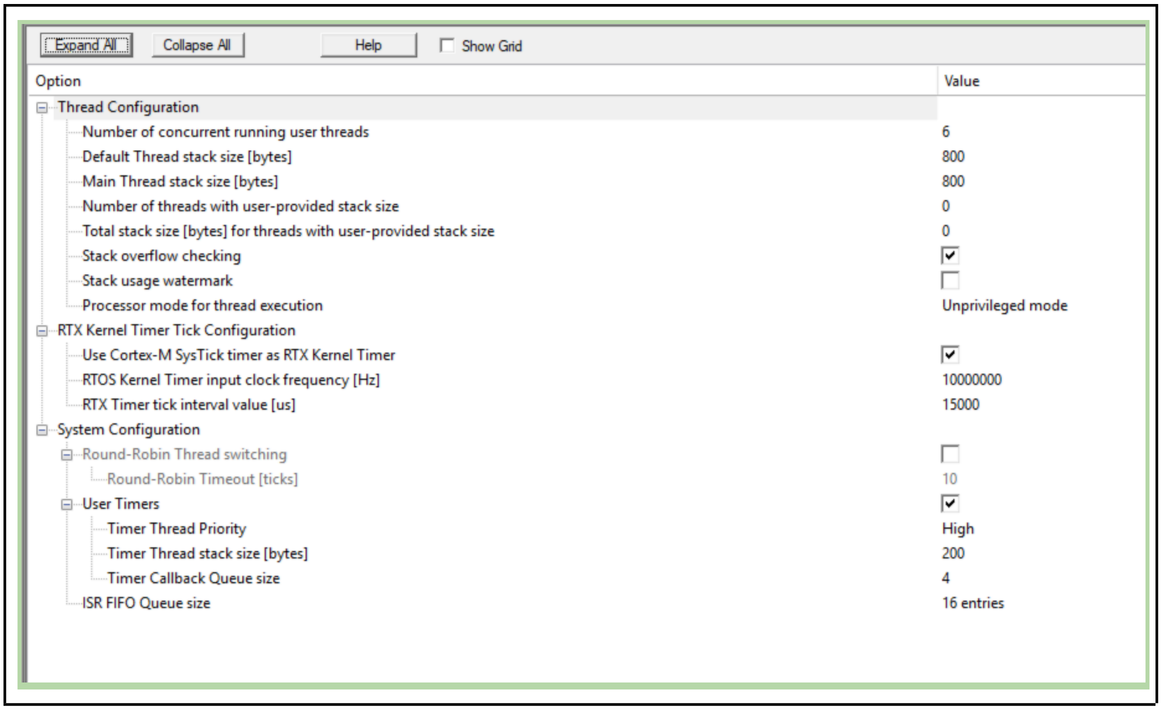
```
#define PI 3.14159265359

/*----------------------------------------------------------------------------
 *      Sample threads
 *---------------------------------------------------------------------------*/
 unsigned int counta=0;
unsigned int countb=0;



void Thread1 (void const *argument); // thread function
void Thread2 (void const *argument); // thread function
void Thread3 (void const *argument); // thread function
void Thread4 (void const *argument); // thread function
void Thread5 (void const *argument); // thread function

volatile uint32_t resultA;
volatile double resultB;
volatile double resultC;
volatile double resultD;
volatile double resultE;


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

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

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

osThreadId tid4_Thread; // thread id
osThreadDef (Thread4, osPriorityAboveNormal, 1, 0);                 // thread object


osThreadId tid5_Thread; // thread id
osThreadDef (Thread5, osPriorityNormal, 1, 0);                 // thread object

int Init_Thread (void) {

  tid_Thread = osThreadCreate (osThread(Thread1), NULL);
        tid2_Thread = osThreadCreate (osThread(Thread2), NULL);
        tid3_Thread = osThreadCreate (osThread(Thread3), NULL);
        tid4_Thread = osThreadCreate (osThread(Thread4), NULL);
        tid5_Thread = osThreadCreate (osThread(Thread5), NULL);
```

```
  if(!tid_Thread || !tid2_Thread || !tid3_Thread || !tid4_Thread || !tid5_Thread )
return(-1);

  return(0);
}

void Thread5 (void const *argument) {
	double r = 1.0;
	int k;
	double ans;
	double coeff_sum = 0.0;
   for (k = 1; k <= 12; ++k) {
      coeff_sum += k;
   }

   ans = PI * r * r * coeff_sum;
   resultE = ans;
	osThreadTerminate(NULL);

}

void Thread4 (void const *argument) {
	int i;
	double ans=1.0;
	double factorial = 1.0;
	for(i=1; i<=5; i++){
		factorial *= i;
		ans  += pow(5.0, (double)i) / factorial;
	}
	 resultD = ans;			// ˜ 91.4166666667
	osThreadTerminate(NULL);

}

void Thread3 (void const *argument) {
	int i;
	double ans=0.0;

	for (i=1; i<=16; i++){
		ans += (double)(i + 1) / (double)i;
	}
	resultC = ans;
	osThreadTerminate(NULL);
}
```

```
void Thread2 (void const *argument) {
	int i;
	double ans=0.0;
	double factorial=1.0;

	for (i=1; i<=16; i++){
		factorial*=i;
		ans+=pow(2.0,i)/factorial;

	}
	resultB = ans;
	osThreadTerminate(NULL);
}
void Thread1 (void const *argument) {
	int ans=0;
	int i;
	for (i=0; i<=256; i++){
		ans+=i+(i+2);
	}
	resultA = ans;

	osThreadTerminate(NULL);
}
```

Inside thread.c, five threads are created, each handling one of the mathematical tasks. Thread5 corresponds to TaskA, Thread4 corresponds to TaskB and so forth. Each of these threads does its calculations, and stores the answer in a global variable (to be observed in analysis). The most important part here is that all the threads are assigned different priority levels, which will allow for preemptive scheduling. All these threads are then run based on the thread's priority and terminate themselves once calculations are done.

RTX_Conf_CM.c Configuration Wizard File:



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]	10
User Timers	☑
Timer Thread Priority	High
Timer Thread stack size [bytes]	200
Timer Callback Queue size	4
ISR FIFO Queue size	16 entries

Analysis:

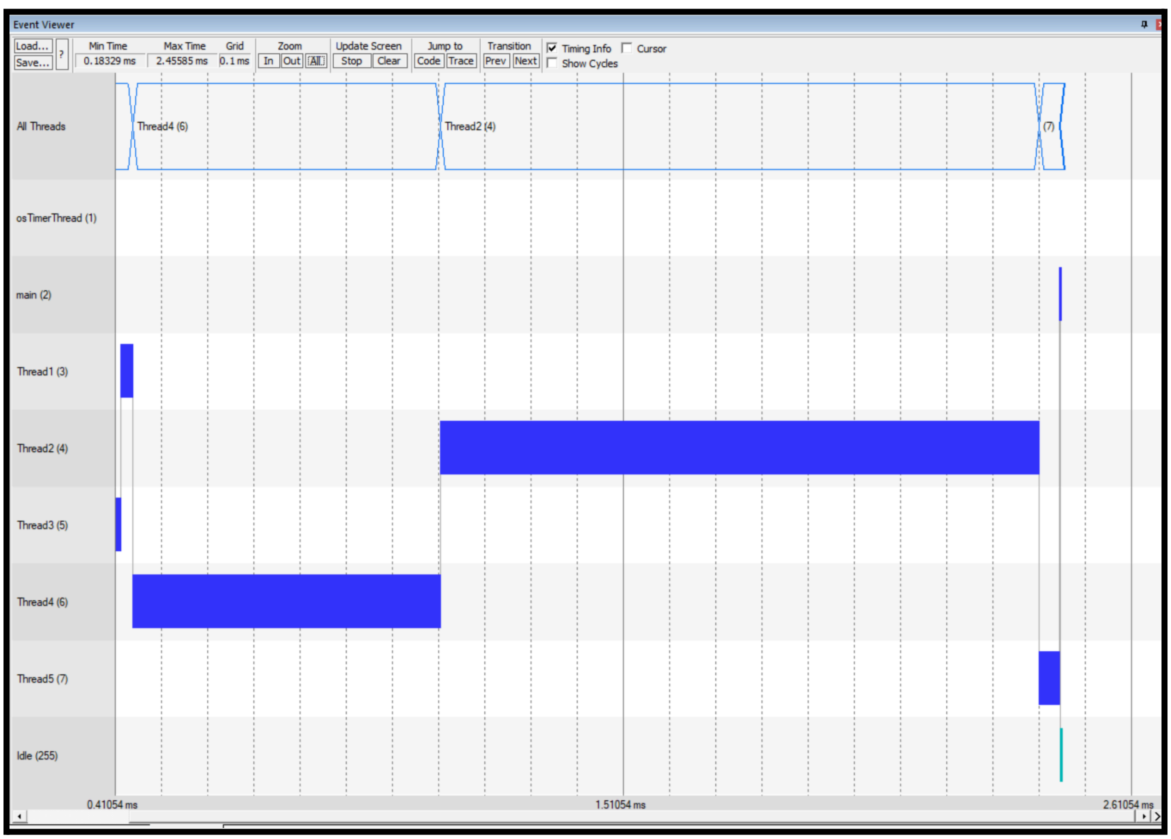
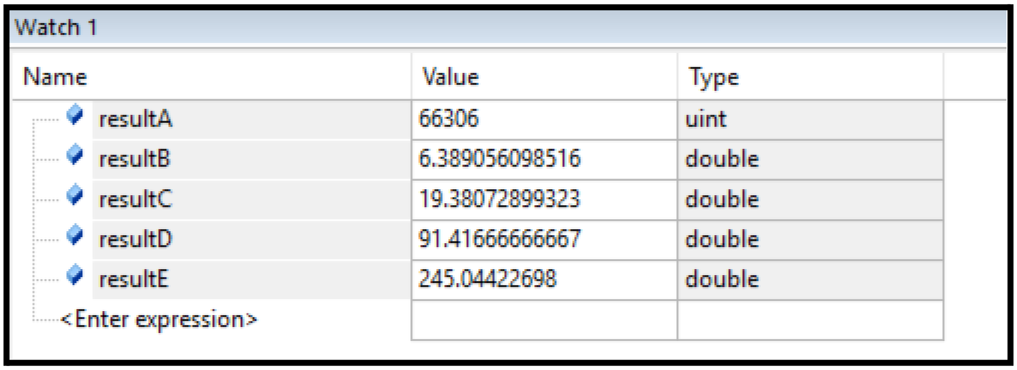


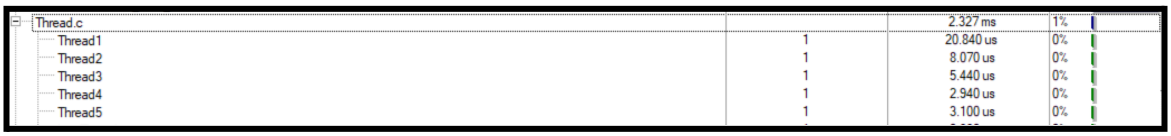
Figure 1.0: Event Viewer window for Part I



Watch 1

Name	Value	Type
resultA	66306	uint
resultB	6.389056098516	double
resultC	19.38072899323	double
resultD	91.41666666667	double
resultE	245.04422698	double
<Enter expression>		

Figure 2.0: Watch window with all calculated values from threads



Thread.c		2.327 ms	1%
Thread1	1	20.840 us	0%
Thread2	1	8.070 us	0%
Thread3	1	5.440 us	0%
Thread4	1	2.940 us	0%
Thread5	1	3.100 us	0%

Figure 3.0:Performance Analyzer window for Part I

After running in demo mode, the preemptive based scheduling can be seen to be working. Figure 1.0 makes it clear in what order the tasks ran in and for how long. Since all the threads are assigned priorities at the start, the scheduler is able to figure out which tasks to run first, and which threads should be put on hold (preempt). Since TaskC (thread 3) was assigned the highest priority, it was run first. Task A and D had the same priority but scheduler went with A then D. Then the scheduler was met with the lowest priority tasks B and E, with E completing last. Round robin on the other hand gives each task an equal time slice, the preemptive approach ensures that higher-priority thread completes first. In the watch window, the global variables (answers) are being watched for all the tasks. A quick double check shows that these calculated values are correct and match the expected results. Overall, these results confirm the scheduler is handling threads based on priority, and preempts lower priority ones.

2.0 Part II - Operating System (OS) Round Robin

main.c Code:

```
/*----------------------------------------------------------------------------
 * 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);
}
```

Main.c code above is where the code starts from. It starts off by initializing the Kernel and creating the threads using osKernelInitialize and Init_Thread. Once the kernel has been started, the scheduler starts cycling through the threads while the main function keeps on hold forever.

Thread.c Code:

```
#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
```

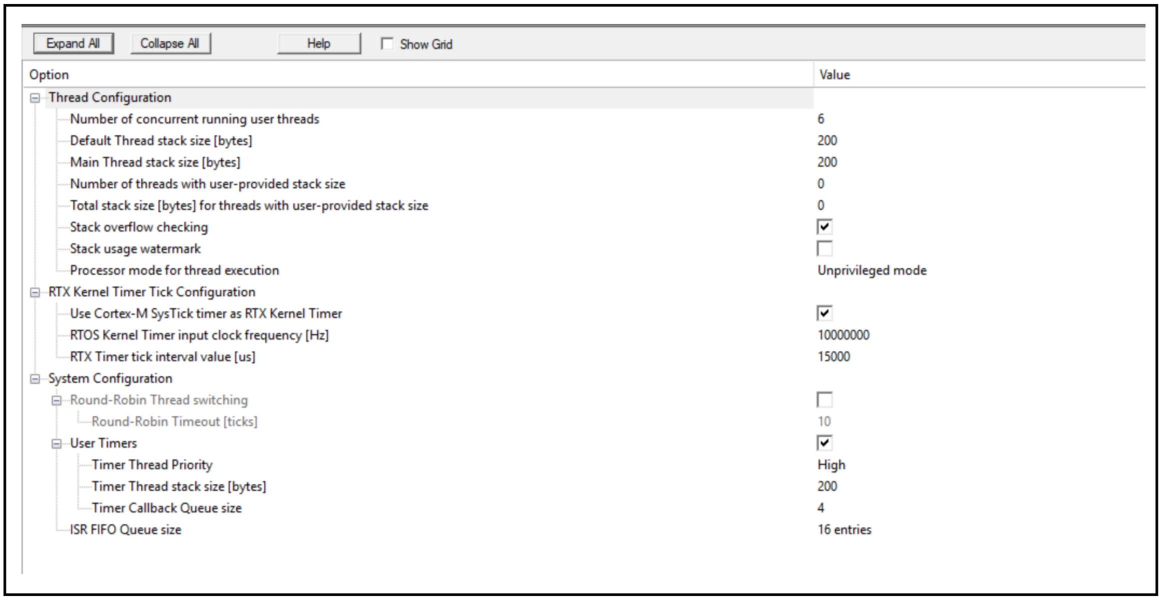
```
|(((unsigned long)(x) & 0x000FFFFF) << 5) | ((y) << 2))

#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, osPriorityHigh, 1, 0);

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

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

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

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

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);
}

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

	osSignalSet(tid_Thread, CPU_MEM);

 osThreadTerminate(NULL);
}

void App_Interface (void const *argument) {
	osMutexWait(log_lock, osWaitForever);
 strncpy(logger, "App_Interface:", sizeof(logger) - 1);
```

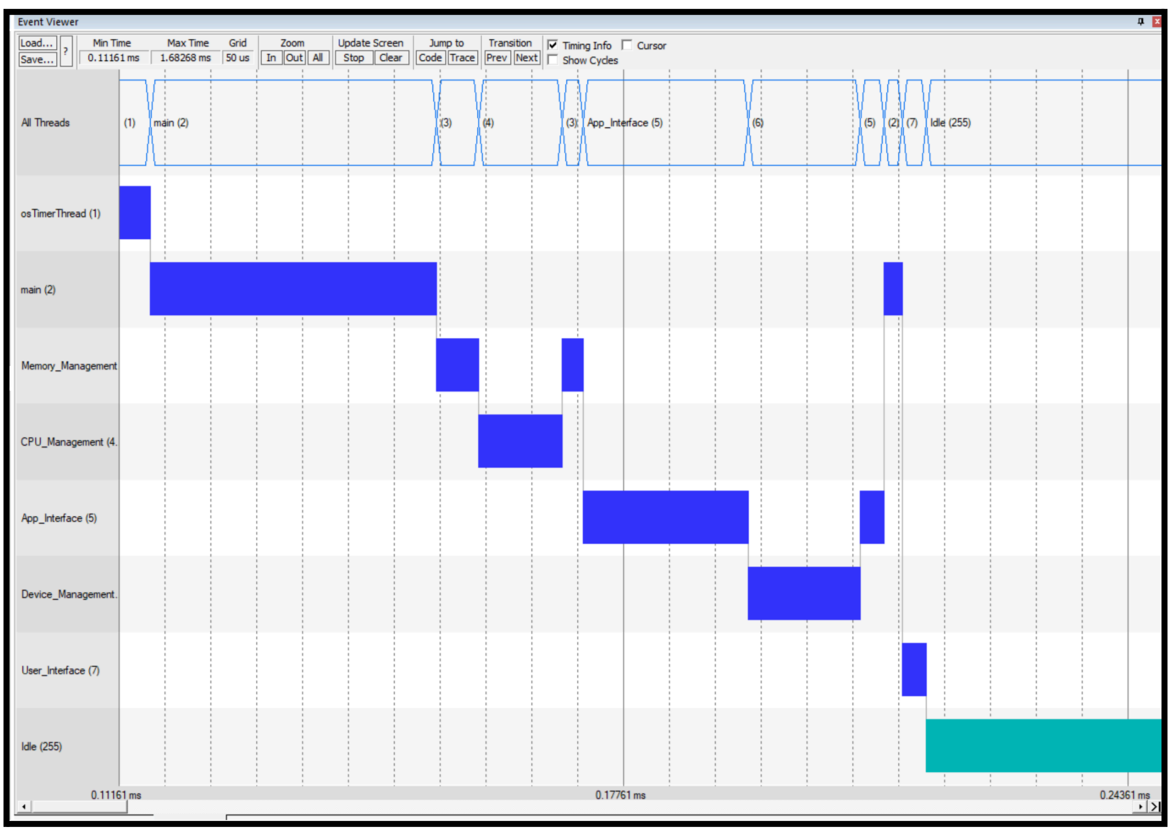
```
 osMutexRelease(log_lock);

	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);
}
```

In this thread.c file, the tasks for OS based threads are created. Each thread does its required operation as instructed. The important part here is that all the threads are assigned priorities. This way, the scheduler knows what to run first.

RTX_Conf_CM.c Configuration Wizard File:


Expand All
Collapse All
Help
Show Grid
Option
Value
Thread Configuration
Number of concurrent running user threads 6
Default Thread stack size [bytes] 200
Main Thread stack size [bytes] 200
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] 10
User Timers
Timer Thread Priority High
Timer Thread stack size [bytes] 200
Timer Callback Queue size 4
ISR FIFO Queue size 16 entries

Analysis:


Event Viewer
Load...
Save...
Min Time 0.11161 ms
Max Time 1.68268 ms
Grid 50 us
Zoom In Out All
Update Screen Stop Clear
Jump to Code Trace
Transition Prev Next
Timing Info
Cursor
Show Cycles
All Threads
(1)
main (2)
(3)
(4)
App_Interface (5)
(6)
(7)
Idle (255)
osTimerThread (1)
main (2)
Memory_Management
CPU_Management (4.
App_Interface (5)
Device_Management
User_Interface (7)
Idle (255)
0.11161 ms
0.17761 ms
0.24361 ms

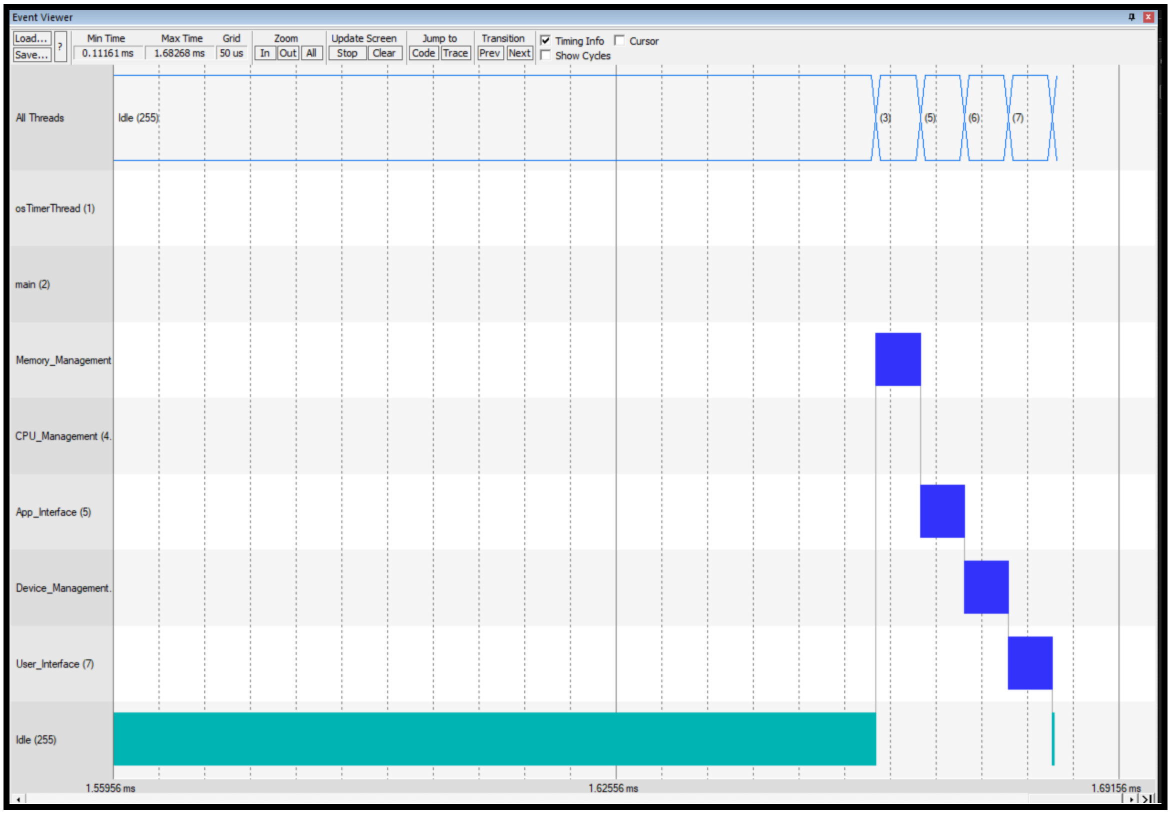


Figure 5.0: Event Views for Part II .

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		76.630 us	0%
User_Interface	1	0.260 us	0%
Device_Management	1	0.790 us	0%
App_Interface	1	0.730 us	0%
CPU_Management	1	0.940 us	0%
Memory_Management	1	0.590 us	0%
Init_Thread	1	0.970 us	0%
strncpy	1	13.980 us	0%
strlen	1	0.930 us	0%
strncat	1	2.610 us	0%
__scatterload	1	0.420 us	0%

Figure 7.0: Performance analyzer for Part II .

The event viewer in figure 5.0 visually shows the execution order of the different threads. The threads for Memory/CPU management have the highest priority. For this reason, these two threads are seen to be running first in the event viewer. Next, Application interface and Device management have the next highest priorities, and are seen to be running next in the even viewer. Finally, the User interface has the lowest priority out of all the other threads, which is why it was run last. After this, all the threads are in OSDelay which is why there is the idle time. After the threads become available again, and the order they finish and terminate are in the order of priority (scheduler heads to highest priority first). The reason there is a cut and switch between some of the threads (for example, Memory to CPU back to Memory) is because there is a signal set and wait between some threads. Regardless, this preemptive approach ensured predictable and efficient task ordering with proper inter-task communication. Unlike round robin, this method ensured unnecessary delays. Finally, the watch window was also useful in confirming all the threads were run correctly. The access counts were all observed to be incremented, and the logger had the correct char values stored.

3.0 Part II - Conclusion

In conclusion, this lab demonstrated how a preemptive scheduler manages multiple threads based on priority. The higher priority tasks are run first while the rest are preempted, and how the use of signals ensures proper communication. The correct execution order and final results confirmed proper coordination. Overall, preemptive scheduling provided efficient, predictable task control compared to round-robin execution.