RYERSON UNIVERSITY

Faculty of Engineering and Architectural Science

Department of Electrical and Computer Engineering

Course Number	COE 718
Course Title	Embedded Systems Design
Semester/Year	F2020
Lab No	3
Instructor Name	Saber Amini
Section No	03

Submission Date	10/28/2020
Due Date	10/28/2020

Name	**Student ID**	**Signature***
Vatsal Shreekant	500771363	

By signing above, you attest that you have contributed to this submission and confirm that all work you have contributed to this submission 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:

www.ryerson.ca/senate/current/pol60.pdf

Introduction

A “thread” in computer science is short for a thread of execution. Threads are a way for a program to divide (termed "split") itself into two or more simultaneously (or pseudo-simultaneously) running tasks.

The osThreadCreate() and osThreadDef() functions will create the threads and set their priorities respectively.

The osKernelInitialize() and osKernelStart() will setup the round-robin scheduling definition for the threads and execute the kernel respectively.

osTimerThread() thread initializes and executes first. This thread is responsible for executing time management functions specified by ARM's RTOS configuration.

The program starts executing from main(), where main() ensures that:

i. The Cortex-M3 system and timers are initialized -SystemInit().
ii. The os kernel is initialized for interfacing software to hardware -osKernelInitialize().
iii. Creates the threads to execute thread1 and thread2 -Init_Thread ().
iv. Starts the kernel to begin thread switching -osKernelStart().

The Thread_C thread executes for its round-robin time slice since it is the highest priority. After 15 msec the timer thread forces control to the Thread_A which has above normal priority and then over to Thread_D. After 15 msec the timer thread forces control to the Thread_B which has above below priority and then over to Thread_E.

Procedure

1) Load main.c and Thread.c example project and complete the instructions in the lab manual.
2) Select the following packages under ‘Manage Run-Time Environment’ window and select OK button:
    - CMSIS>CORE.
    - CMSIS>RTOS(API)>Keil RTX.
    - Device > Startup.
3) Modify the ‘Thread.c’ file to implement a round-robin scheduling example using 3 different tasks as listed in the table 1 of the lab manual. The code implemented in Thread.c is listed as follows:

```c
/*----------------------------------------------------------------------------
*       Lab 3: Scheduling Multithreaded Applications with RTX & uVision
*       Vatsal Shreekant, student id: 500771363
 *---------------------------------------------------------------------------*/
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#include <math.h>
#include "cmsis_os.h"
#include "LPC17xx.H"
#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 GPIO1_LED31 (*((volatile unsigned long*)0x233806FC))
#define GPIO2_LED2 (*((volatile unsigned long *) 0x23380A88))
#define __FI        1



double factor;
double var_A;
double var_B;
double var_C;
double var_D;
double var_E;
int myExponentialCalc(int, int);
void delay (unsigned int);
volatile unsigned long * GPIO2_LED4 ;



void threadA (void const *argument);                                            // thread function
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;   // thread id

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


int Init_Thread (void) {

 id_Thread_A = osThreadCreate (osThread(threadA), NULL);                          // create threads
 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);

 if(!id_Thread_A) return(-1);

  return(0);
}


void threadA (void const *arg) {
 double x =0;
 int i = 0;

 for ( i=0; i<257; i++){
  x = x + (i + (i+2));
```


Figure 1: Page 1 of Thread.c

```c
   var_A = x;
                                                        //Function that passes control to the next task of the same priority in the ready queue

  }
   delay(100);

 }

 void threadB (void const *arg) {
  double x =0;
  int i;
 int factor = 1;


  for( i = 1; i<17; i++){
   factor = factor*i;
   x = x + ((double)(myExponentialCalc(2,i))/factor);
    var_B = x;

    osDelay(1);                                                 //Function that passes control to the next task of the same priority in the ready queue

  }


 }

 void threadC (void const *arg) {
  double x =0;
  int n=0;


  for ( n=1; n<17; n++){
   x = x + (n+1)/n;
   var_C = x;
  }

 }

 void threadD (void const *arg) {

  double x=0;
  int m=0;
  factor=1;

  for ( m=0; m<6; m++){
   factor = factor*m;
   if(factor == 0){
    factor=1;
   }
   else{
    osDelay(1);                                                     //Function that passes control to the next task of the same priority in the ready queue
    x = x + ((double)(myExponentialCalc(5, m)))/(double)factor;
    var_D = x;
   }
  }

 }

 void threadE(void const *arg) {
   double x=0;
   int p=0;
   int radius=1;
```


Figure 2: Page 2 of Thread.c

```

 for (p=1; p<13; p++){
  x = x + (3.14)*((double)(myExponentialCalc(radius,2)));
   var_E = x;
   osDelay(1);                                                 //Function that passes control to the next task of the same priority in the ready queue

 }

}

int myExponentialCalc(int x, int n)
{
  int i;
  int number = 1;

  for (i = 0; i < n; i++)
    number *=x;

  return(number);
}

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++){
    }
  }
}
```


Figure 3: Page 3 of Thread.c

4) Compile project using the build button and start the simulation by selecting the debug button.
5) Select debug mode to analyze performance of the threads using Performance Analyzer and the Event Viewer.

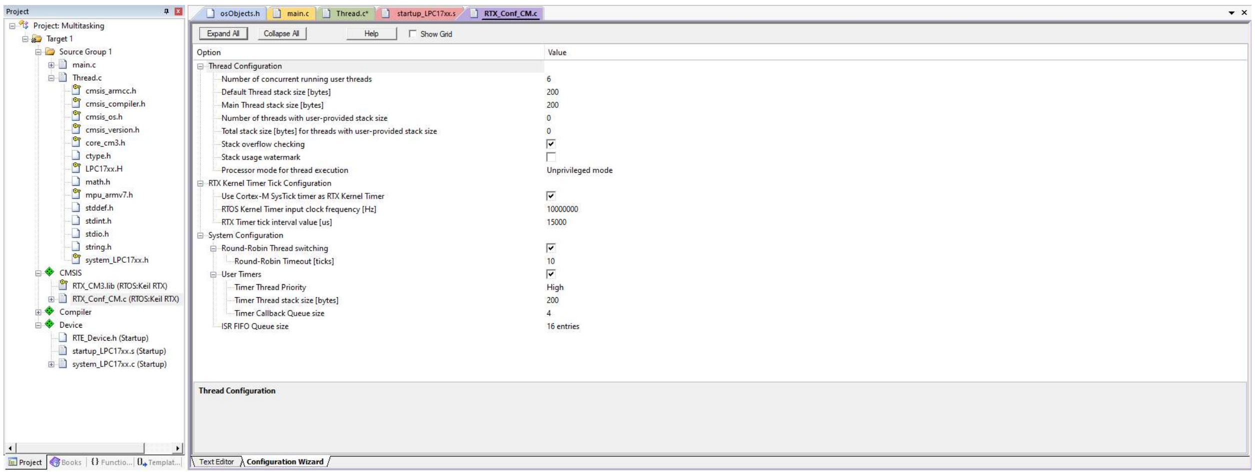

Figure 4: RTX_Conf_CM.c Configuration Wizard

Conclusion

When comparing and analyzing the results under debug mode, it is evident that the tasks are being prioritized and executed as per the priority thread and this was the initial assumption before implementing the code. Refer to figures 5 and 6 for the results. As per the instructions in the lab manual, the code for the LED was not required. The BitBand() function was used to toggle the pins at Port 2. The results for values A, B, C, D and E in the watch window were also verified against an online calculator and there were no discrepancies.

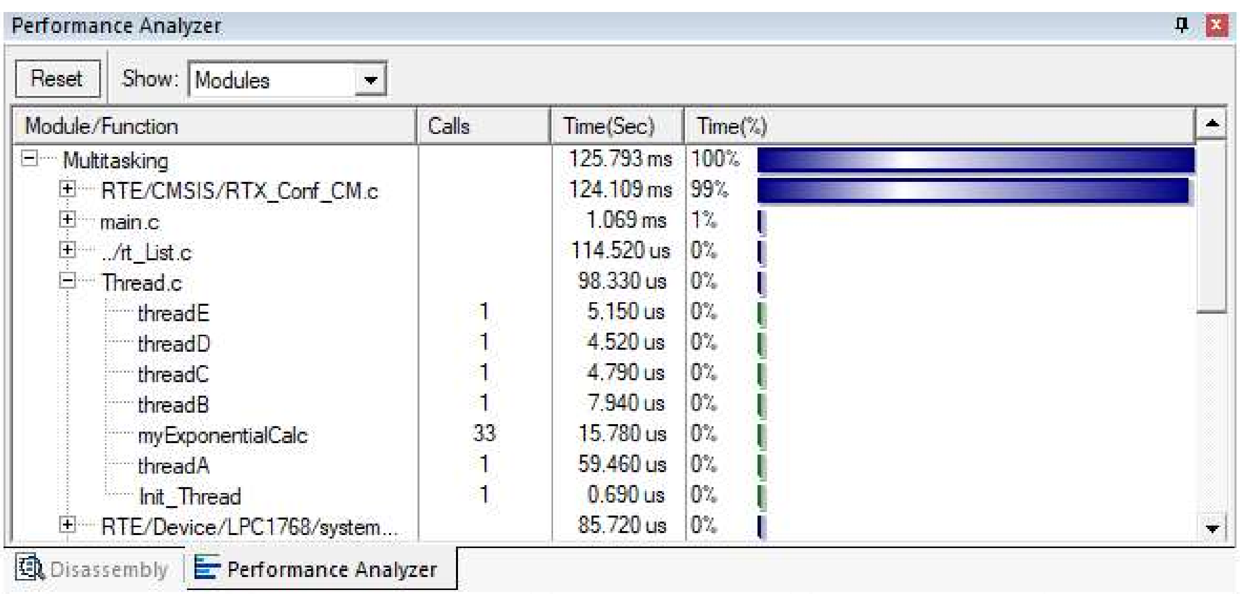

Figure 5: Performance Analyzer Window

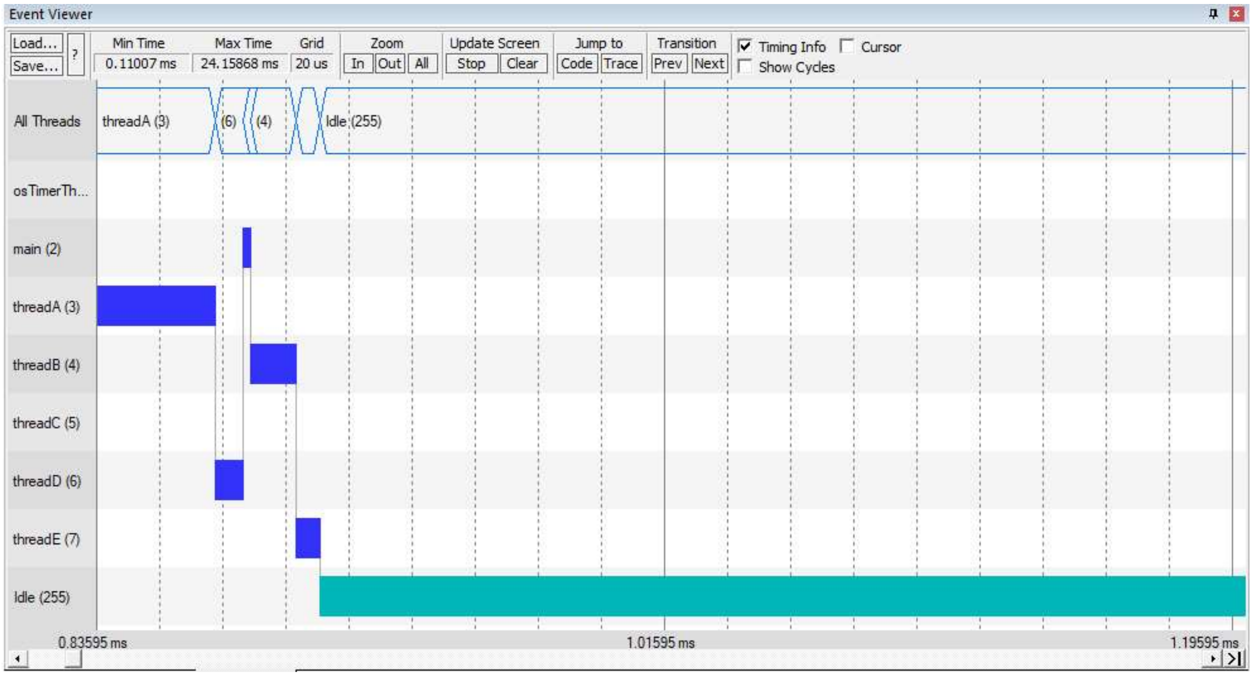

Figure 6: Event Viewer Window

References

1) NXP User Manual, https://www.nxp.com/docs/en/user-guide/UM10360.pdf, 2020
2) ARM Keil User Guide, https://www.keil.com/support/man/docs/mcb1700/mcb1700_intro.htm, 2020