

RTX based Multitasking with Round-Robin Scheduling

COE718: Embedded Systems Design Lab 3a

1. Objectives

The purpose of this lab is to introduce students to simple multitasking by using RTX a Real-Time Operating System (RTOS). Specifically, students will learn how to schedule threads (tasks) using round-robin scheduling scheme available as part of RTX.

2. Working with uVision and RTX

2.1. Setting up an RTX (RTOS) Project

1. Launch the uVision application. Create a new project "Lab3a_Demo" in your "S:\\\\" folder. Select the **LPC1768** chip. Copy the files provided in the course directory "U:\\coe718\\labs\\lab3a\\" to your project directory.
2. Select the following Packages from the Run Time Environment and add them to your project as depicted in Figure 1.
 - i. CMSIS>CORE
 - ii. CMSIS>RTOS(API)>Keil RTX
 - iii. Device > Startup

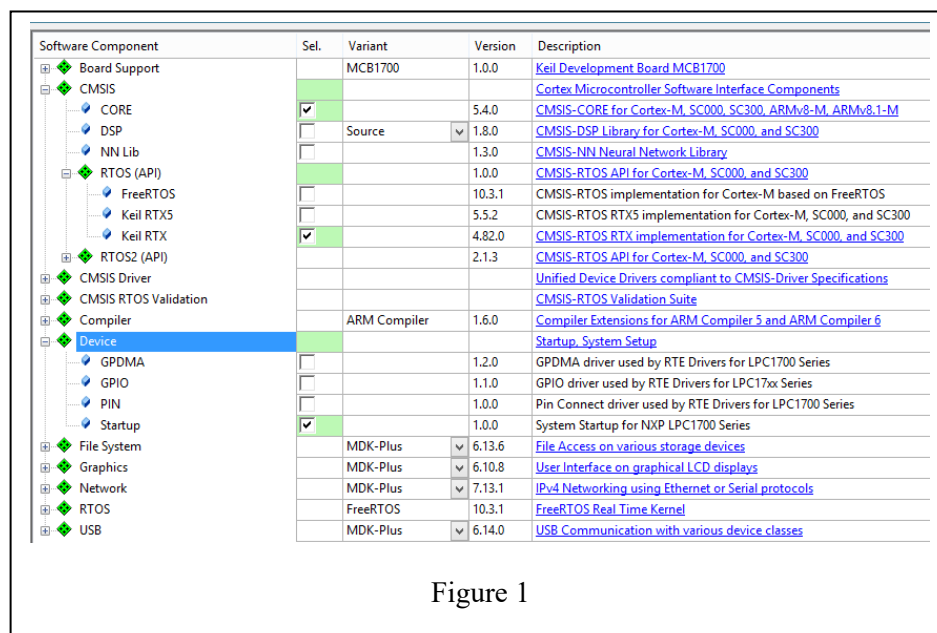
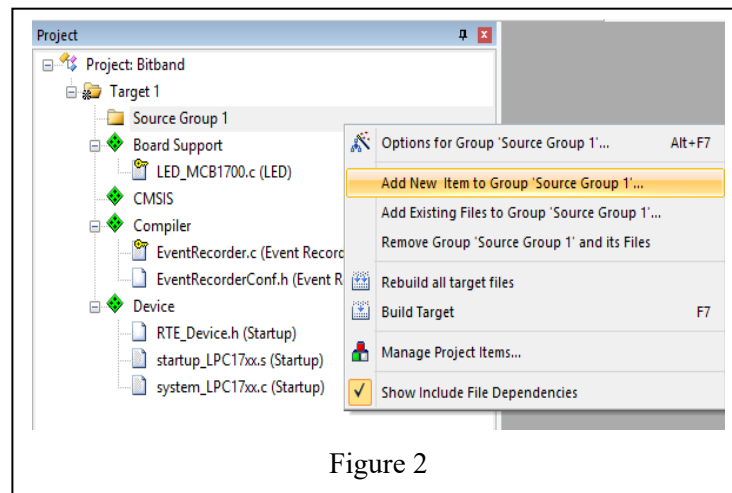
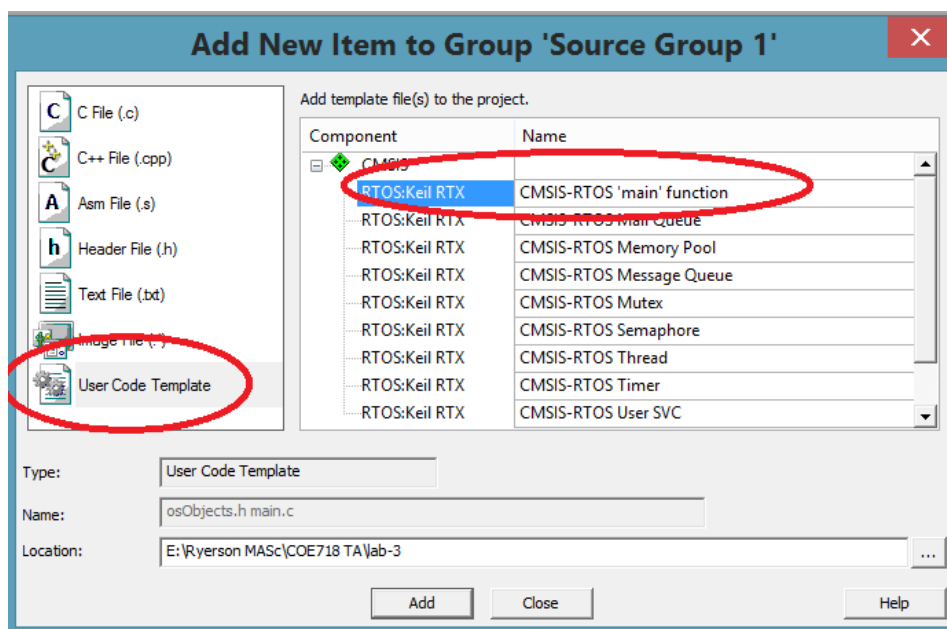


Figure 1

3. Right click on Source Group1 folder and select “Add New Item to Group ‘Source Group 1’ as shown in Figure 2.



4. Select the User Code Template >expand CMSIS >RTOS:Keil RTX> CMSIS-RTOS'main' function. Then click Add as shown in Figure 3. The file will be added to the project.



5. Repeat Step 4 and this time select CMSIS>RTOS:Keil RTX> CMSIS-RTOS Thread. The click Add and another file will be added to your project. See Figure 4.

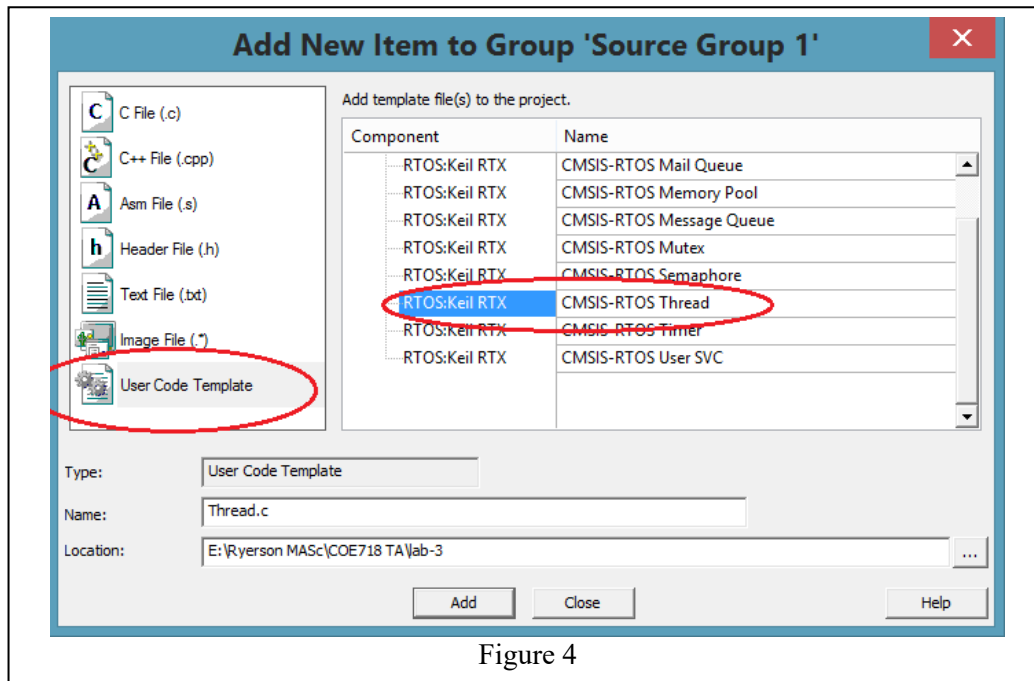


Figure 4

6. Do the Following changes by opening the options for project:
 - i. Under Target. ARM Compiler> **Use default compiler version 5**. Check Use > **Micro LIB**.
 - ii. Then select Debug option and check Use Simulator. Perform the following changes also.
Dialog DLL: DARMP1.DLL, Parameter: -pLPC1768

We have learned to add the template files from library. See 'main.c' and 'Threads.c' code files and try to understand. However, we don't use the 'main.c' and 'Threads.c' added from template instead we will replace them with the starter code provided. The Project Directory is also set for Lab3a and its related assignment. Now, you may delete 'main.c' and 'Threads.c' and add the 'main.c' and 'Threads.c' codes provided. Compile the code and check for any errors, etc.

2.2 Configuring the RTX

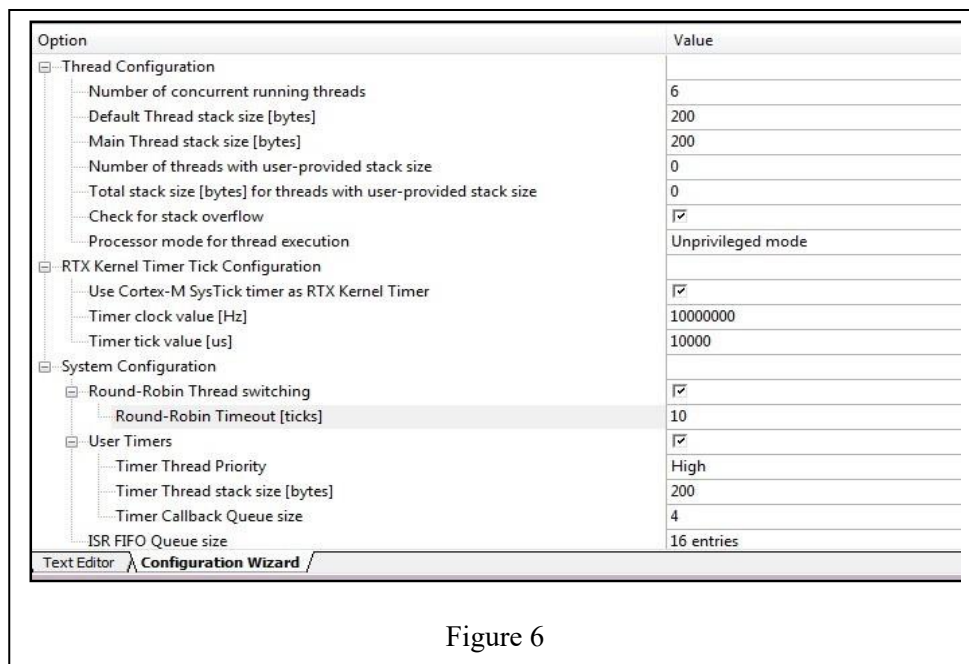
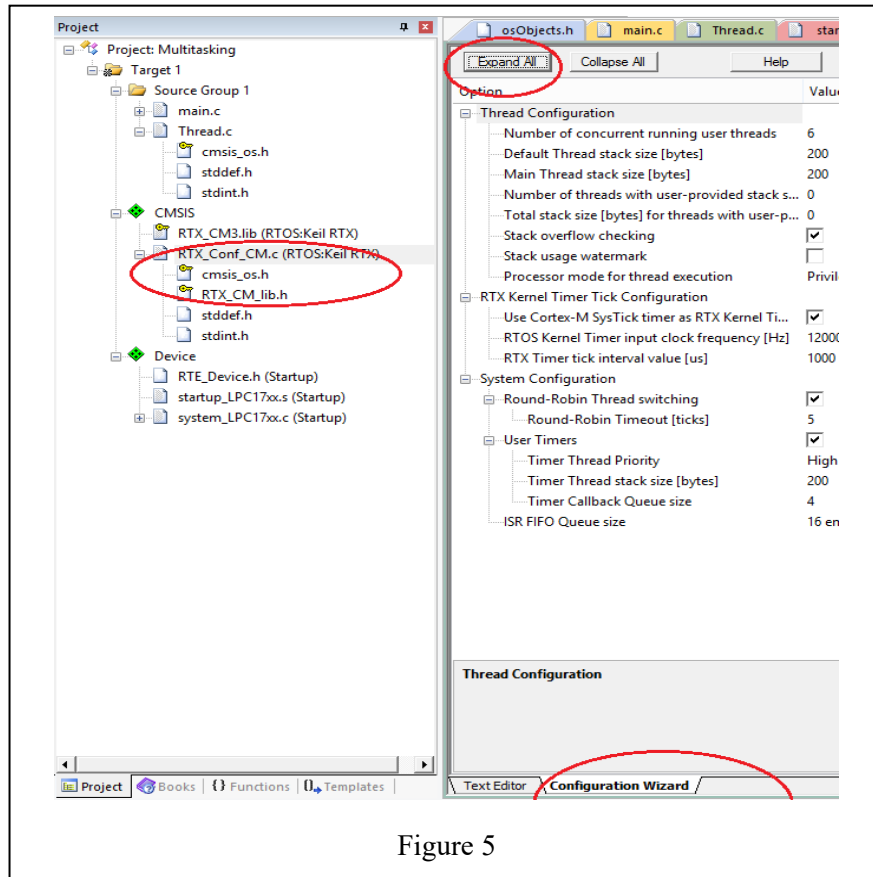
RTOS must be configured for round-robin scheduling related specifications such as the time-slice, frequency of the CPU's systick timer and the arbitration techniques for round-robin scheduling.

Open the file *RTOS_Conf_CM.c*, double click from the project directory as shown in Figure 5. Then Select Configuration Wizard and click Expand All.

Make sure that the following options are selected

- *Use Cortex-M SysTick timer as RTX Kernel Timer*
- the RTOS Kernel Timer input clock frequency [Hz] option is set to **10000000** (10 MHz)
- RTOS Timer tick interval value[us] option is set to **10000** (10ms)

Moreover, the "User Timers" option is also checked for round-robin scheduling. Your configuration should now resemble as depicted in Figure 6



3. Analyzing the RTX Project

a) Watch Windows

Watch windows allow the programmer to keep track of the variables 'counta' and 'countb'.

1. Open the watch window by selecting View > Watch Window > Watch 1.
2. You may hover the mouse over the variable name in the code window, right-click and select **Add 'counta' or 'countb' to... >> Watch 1.**
3. When you click the RUN icon to execute the program, the values of 'counta' and 'countb' should alternatively increment depending on the thread, which is currently executing.
4. It is also possible to change 'counta' and 'countb' values as its incrementing during execution. If you enter a '0' in the value field, you may modify the variable's value without affecting the CPU cycles during executing. This technique can work both in simulation and while executing on the CPU.

b) Performance Analyzer

1. Select View >> Analysis Windows >> Performance Analyzer (PA).
2. Expand the "Multitasking" in the PA window by pressing the "+" sign located next to the heading. There should be a list of functions (like a tree) present under this heading. There should also be another subheading titled "Thread.c". Press the "+" sign again to collapse the tree further. There you can observe the execution of thread1 and thread2.
3. Reset the program (ensure that the program has been stopped first). Click RUN.
4. Watch the program execute and how the functions are called.

c) RTX Event Viewer

The Event Viewer is a graphical representation of a program's thread-based execution timeline. An example is shown in Figure 7. The Event Viewer runs on the Keil simulator but must be configured properly for CPU execution using a Serial Wire Viewer (SWV).

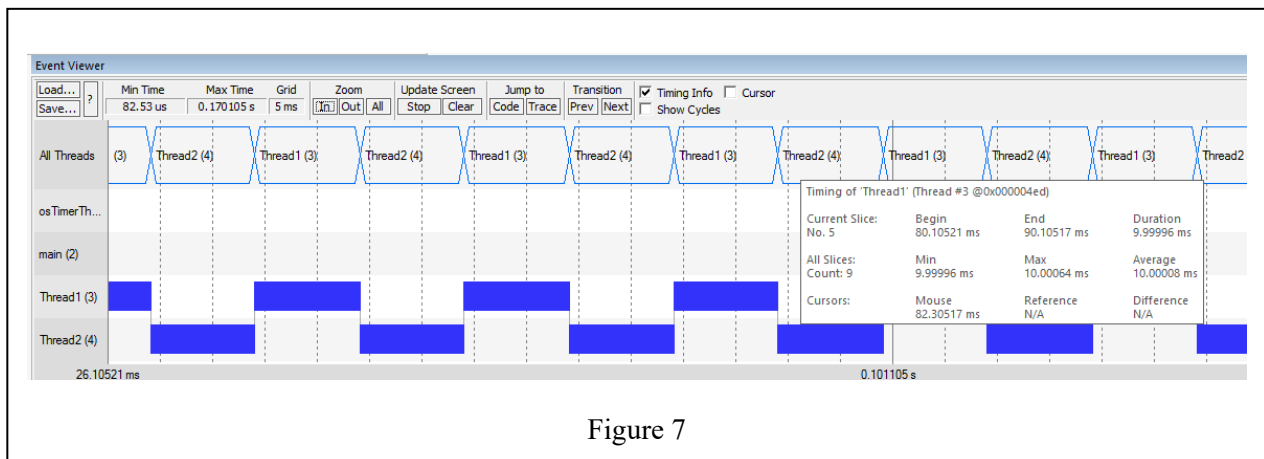


Figure 7

To use this feature of the Serial Wire Viewer (SWV), you need to perform the following.

1. In the main menu select Debug >> OS Support >> Event Viewer. A window should appear.
2. Click RUN. Click the "All" button under the zoom menu in the Event Viewer window. You may also select "In" or "Out" to adjust the view of the timeline which dynamically updates as the program continues to execute. Note the other threads other than thread1 and thread2 that are also

present in the execution timeline.

3. Let the program execute for approximately 50 msec. Click STOP. Your window should now look like that of Figure 7.
4. Hover the mouse over one of the thread time slices (blue blocks indicating execution of the task). You will see stats of the thread appear. The stats should concur with the round-robin scheduling we set up in *RTX_Conf_CM.c* (i.e., 10ms time slices).
5. Try going back to the *RTX_Conf_CM.c* file and changing the time stats of the round-robin scheduler. Rebuild the project and run it again in Debug mode. See if the Event Viewer reflects the changes you made to the file.

d) RTX Tasks and System Window

This window provides an RTX kernel summary with detailed specifications of *RTX_Conf_CM.c*, along with the execution profiling information of executing tasks. An example window is provided in Figure 8. The information obtained in this window comes from the Cortex-M3 DAP (Debug Access Port). The DAP acquires such information by reading and writing to memory locations continuously using the JTAG port.

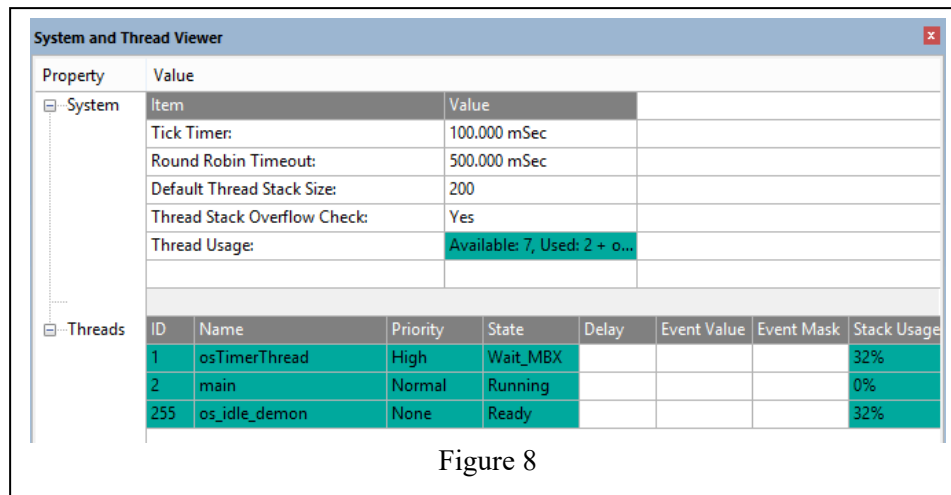


Figure 8

To use this feature, perform the following steps.

1. Select Debug >> OS Support >> System and Thread Viewer.
2. As you run the program (or Reset and RUN), the state of the "Thread" heading will change dynamically. However, the "System" information will remain the same as these information values are specified prior to runtime in *RTX_Conf_CM.c*.

4. Programming with uVision and RTX

4.1 Understanding the RTX OS

Open the *Thread.c* file and examine the code. This program presents an example of a multithreaded RTX application consisting of two simple threads, each executing its own code. The *osThreadCreate()* and *osThreadDef()* functions will create the threads (tasks) and set their priorities respectively.

Thread1 and Thread2 will loop infinitely following the round-robin scheduling technique. The timing specification was included in the config file (*RTX_Conf_CM.c*). *osKernelInitialize()* and *osKernelStart()* will setup the round-robin scheduling parameters for the threads and execute the kernel and threads. Compile the application and enter Debug mode. We will now use the uVision tools to analyze the RTX program.

4.2 Analyzing the RTX Project

Similar to the previous section, use the Watch Window, Watchpoints, Performance Analyzer, Event Viewer and RTX System, and Thread Window to analyze the application.

4.3 Reviewing Thread.c and Main.c Code

We have analyzed a simplistic multi-threaded application and its various performance features using the uVision. Let's look at the code once more step-by-step by using the uVision analysis tools.

1. Re-execute the code and look at the Event Viewer. Which thread executes first? *osTimerThread()* thread initializes and executes - this thread is responsible for the executing time management functions specified by the ARM's RTX (RTOS) configuration.
2. The program starts executing from *main()*, where it ensures that:
 - a. The Cortex-M3 system and timers are initialized - *SystemInit()*
 - b. the OS kernel is initialized for interfacing software to the hardware - *osKernelInitialize()*
 - c. Create the threads to execute Thread1 and Thread2 - *Init_Thread()*
 - d. Starts the kernel to begin thread switching - *osKernelStart()*
3. The Thread1 executes for its round-robin time slice since it is created first. After 10msec the timer thread forces control to Thread2.
4. The Thread2 executes during its time slice for 10msec and is forced to stop again and execute Thread1. This occurs infinitely.

4.4 Processor Idling Time

As an exercise, let us determine the idling time of the code. We are currently working by using the idle demon. Open the RTX_Conf_CM.c file. Under the line `#include <cmsis_os.h>` insert the definition for the global variable `unsigned int countIDLE = 0;` and setup.

```
void os_idle_demon (void) (  
    for (;;) {  
        countIDLE++;  
    }  
}
```

1. Save the file and compile the project. Re-enter Debug mode. Open the Watch window. Add `counta`, `countb`, and `countIDLE` to the expression list of variables to watch during execution. Click reset, and RUN.
2. Observe the Watch 1 window, and as 'counta' and 'countb' increment, however `countIDLE` variable does not. *What does this mean?*

Basically, CPU is currently under 100% utilization by the threads. Note that Idle Demon is set with the lowest priority in the task list. You can verify this by using the System and thread viewer tool.

5. Lab Assignment

This lab is due in **week 5** at the beginning of your lab session. The following outlines the specification of different scheduling applications. You must create TWO versions (analysis and demo) for each application in questions 1, and 2.

The analysis code will be used for debug mode and to analyze the performance of your applications. Therefore, it **must not** include any LED or LCD code. The demo version will include the LCD and LED functions and the supporting files. Further, you can develop one code using macros to disable LCD and LED part during compilation (Remember the way we used LCD in Lab 1 by using macros).

Note, you may need to increase the stack size to accommodate the LED/LCD code in RTX_Conf_CM.h. You will be marked on both versions of the code but are only required to submit the analysis version for grading. The demo version will be presented during the lab.

1. Write a round-robin scheduling example using three different tasks. Each task should be allotted a time slice of 15msec. Note: Your code must perform a different functionality than the one provided in this demo. Marks will be awarded for creativity. Ensure that the tasks do not run infinitely, and they have a finite workload with respect to time. For the demo version only, use the LEDs and the LCD to indicate the threads that are currently executing in your program.
2. Create a round-robin scheduling algorithm for the following Operating System (OS) based problem given below:

OS Task Function	Functionality
Memory Management	<ul style="list-style-type: none"> -Increments its memory access counter. -Implements a bit band computation to a volatile memory location. -Passes control to CPU management. -After obtaining a signal back from CPU management, delays the OS for one tick and deletes itself.
CPU Management	<ul style="list-style-type: none"> -Increments its CPU management access counter. -Creates a conditional execution and barrel shifting scenario (to check its internal hardware). -Signals back to the memory management task and deletes itself.
Application Interface	<ul style="list-style-type: none"> -Ensure that Application executes before Device Management. -Accesses the global variable logger using a mutex. -Writes a partial message to logger. Waits for Device Management to finish writing to logger. -Increases its global counter. Delays the OS for one tick and deletes itself.
Device Management	<ul style="list-style-type: none"> -Writes the ending to logger variable started by the Application Interface. -Signals back to the App Interface. -Increases its global counter. Delays for one tick to ensure the file is closed and deletes itself.
User Interface	<ul style="list-style-type: none"> -Increments the number of users (its variable), delays the OS for one tick then deletes itself.

Commence thread execution in main() with the memory management task. Implement each thread's functionality as specified in the table, noting that each function has its own global variable and may be dependent on another thread. The threads presented here are of a finite workload. For the demo version only, use the LEDs and the LCD to indicate the threads that are currently executing in your program. (Note you may need to add delays in your demo version). Finally, comment the pros and cons of round-robin scheme for the given operating system problem.

For all solutions:

Hand in the printout of the **analysis version** of your .c code, RTX_Conf_CM.c Configuration Wizard file, and snapshots of your Event Viewer and Performance Analyzer windows for **each** application. Question 2 may require several snapshots of your Event Viewer, and a well thought out choice regarding where you should halt your program for the Performance Analyzer window. Moreover, comment the pros and cons of round-robin scheme for the given operating system problem. Your TA/Supervisor will ask you to demonstrate the **demo version** for each application during your lab session. You may also be required to answer questions regarding the implementation and simulations of your applications.