EE8205: Embedded Computer Systems

Electrical, Computer & Biomedical Engineering, Ryerson University

Scheduling Real-time Applications using µVision and RTX

1. Objectives

This lab introduces students to develop RTX based multithreaded applications for ARM Cortex-M3 processors. The students will learn how to schedule multithreaded applications by employing round-robin, priority preemptive scheduling supported by µVision, RTX operating system and CMSIS libraries. Moreover, you will learn how to schedule and implement a Rate Monotonic Scheduling (RMS), which is a popular Fixed Priority Scheduling (FPS) technique.

2. Creating New Project:

- After launching µVision, select Project >> New µVision Project in the main menu bar. If a project already exists, first close the project by selecting Project >> Close Project. Then Select New µVision Project as shown in Figure 1.

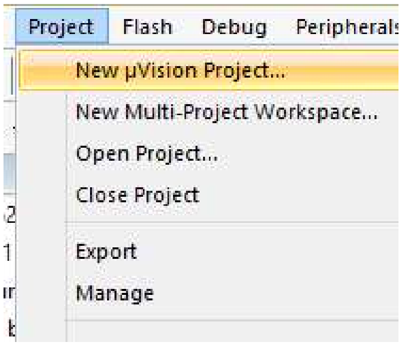


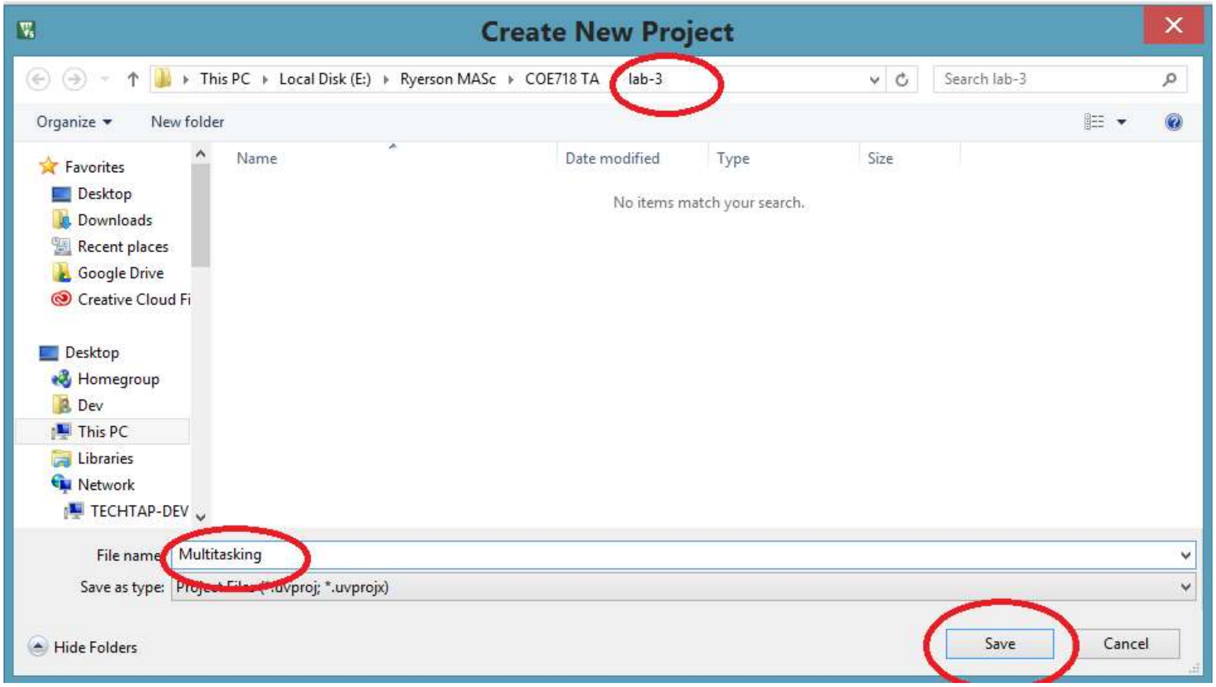
Figure 1

- You should see a window shown in Figure 2. Select the icon for "New Folder" and name your working folder “Lab3". Then name the project as “Multitasking” and Press Save.

- Type “LPC1768” as shown in the Figure 3 and select the device and press OK.

- Select the following Packages from Run Time Environment and add those to your project as depicted in Figure 4.
  a. CMSIS>CORE
  b. CMSIS>RTOS(API)>Keil RTX
  c. Device > Startup

  Press OK once selected



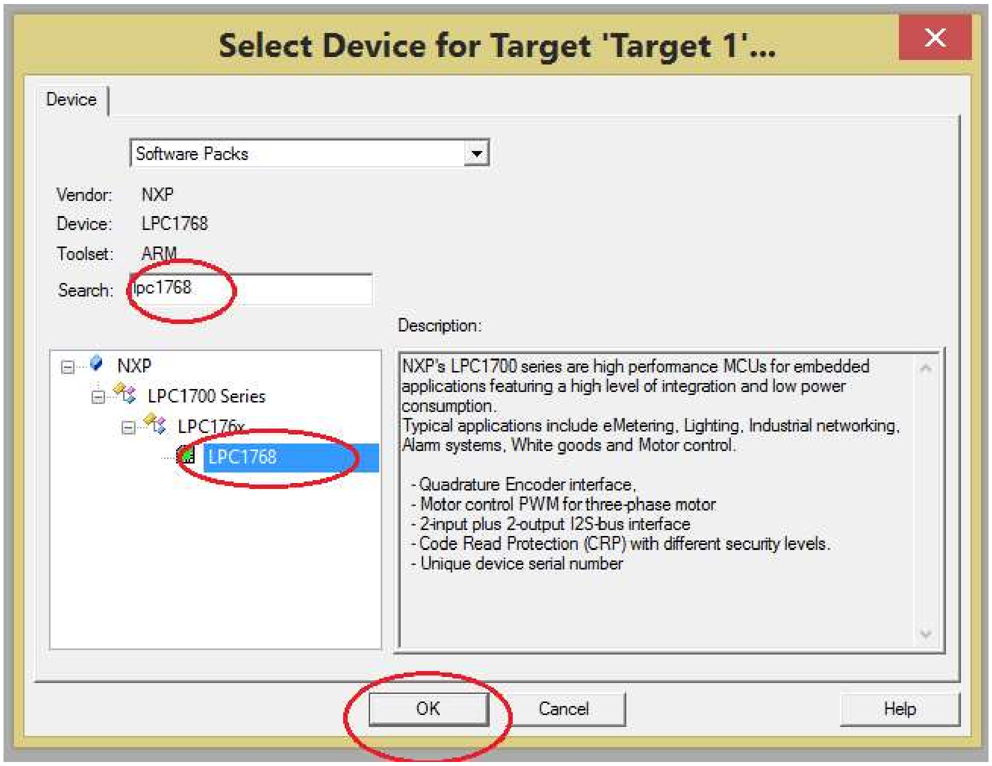
This PC › Local Disk (E:) › Ryerson MASc › COE718 TA › lab-3

No items match your search.

File name: Multitasking

Save as type: Project Files (*.uvproj; *.uvprojx)

Figure 2



Software Packs

Vendor: NXP

Device: LPC1768

Toolset: ARM

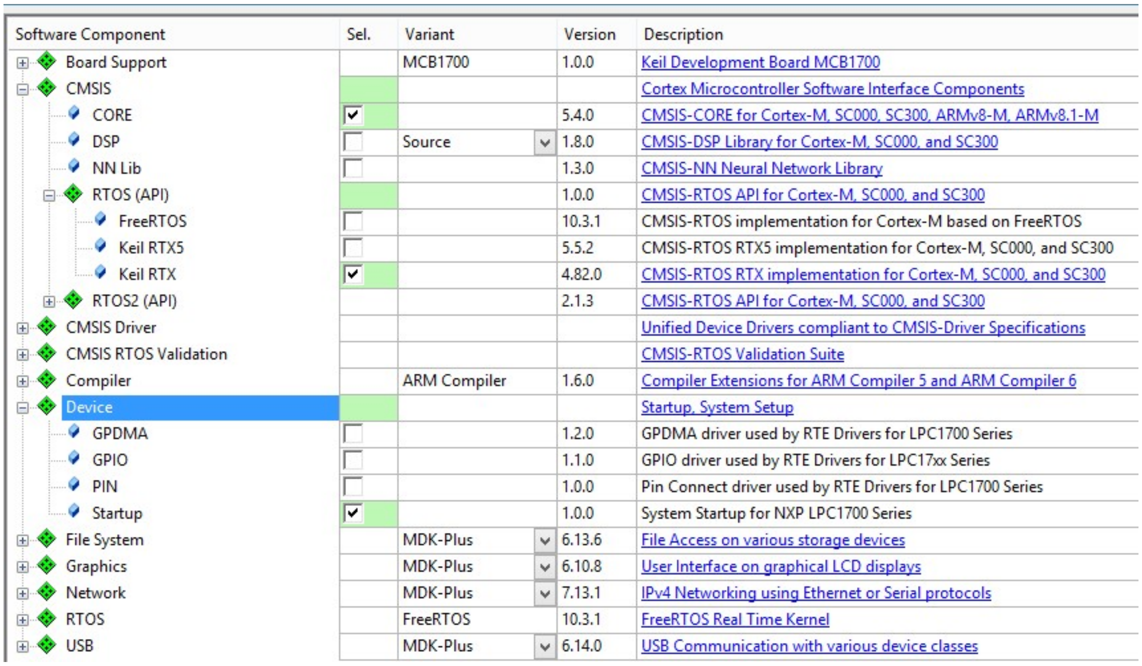
Search: lpc1768

Description:

NXP's LPC1700 series are high performance MCUs for embedded applications featuring a high level of integration and low power consumption.
Typical applications include eMetering, Lighting, Industrial networking, Alarm systems, White goods and Motor control.

- Quadrature Encoder interface,
- Motor control PWM for three-phase motor
- 2-input plus 2-output I2S-bus interface
- Code Read Protection (CRP) with different security levels.
- Unique device serial number

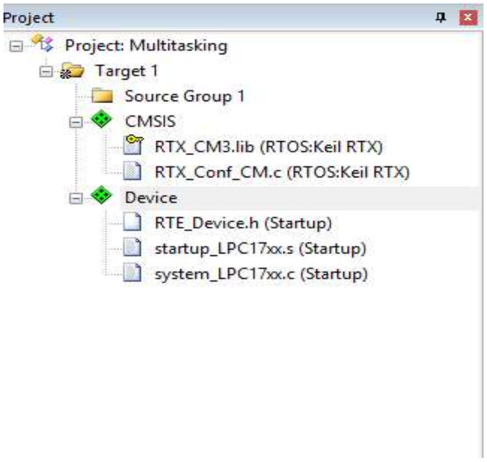
Figure 3



Software Component	Sel.	Variant	Version	Description
Board Support		MCB1700	1.0.0	Keil Development Board MCB1700
CMSIS				Cortex Microcontroller Software Interface Components
CORE	☑		5.4.0	CMSIS-CORE for Cortex-M, SC000, SC300, ARMv8-M, ARMv8.1-M
DSP	☐	Source	1.8.0	CMSIS-DSP Library for Cortex-M, SC000, and SC300
NN Lib	☐		1.3.0	CMSIS-NN Neural Network Library
RTOS (API)			1.0.0	CMSIS-RTOS API for Cortex-M, SC000, and SC300
FreeRTOS	☐		10.3.1	CMSIS-RTOS implementation for Cortex-M based on FreeRTOS
Keil RTX5	☐		5.5.2	CMSIS-RTOS RTX5 implementation for Cortex-M, SC000, and SC300
Keil RTX	☑		4.82.0	CMSIS-RTOS RTX implementation for Cortex-M, SC000, and SC300
RTOS2 (API)			2.1.3	CMSIS-RTOS API for Cortex-M, SC000, and SC300
CMSIS Driver				Unified Device Drivers compliant to CMSIS-Driver Specifications
CMSIS RTOS Validation				CMSIS-RTOS Validation Suite
Compiler		ARM Compiler	1.6.0	Compiler Extensions for ARM Compiler 5 and ARM Compiler 6
Device				Startup, System Setup
GPDMA	☐		1.2.0	GPDMA driver used by RTE Drivers for LPC1700 Series
GPIO	☐		1.1.0	GPIO driver used by RTE Drivers for LPC17xx Series
PIN	☐		1.0.0	Pin Connect driver used by RTE Drivers for LPC1700 Series
Startup	☑		1.0.0	System Startup for NXP LPC1700 Series
File System		MDK-Plus	6.13.6	File Access on various storage devices
Graphics		MDK-Plus	6.10.8	User Interface on graphical LCD displays
Network		MDK-Plus	7.13.1	IPv4 Networking using Ethernet or Serial protocols
RTOS		FreeRTOS	10.3.1	FreeRTOS Real Time Kernel
USB		MDK-Plus	6.14.0	USB Communication with various device classes

Figure 4

- Now your Project files should look like as shown in Figure 5.



Project

- Project: Multitasking
  - Target 1
    - Source Group 1
    - CMSIS
      - RTX_CM3.lib (RTOS:Keil RTX)
      - RTX_Conf_CM.c (RTOS:Keil RTX)
    - Device
      - RTE_Device.h (Startup)
      - startup_LPC17xx.s (Startup)
      - system_LPC17xx.c (Startup)

Figure 5

- Right click on Source Group1 folder and select "Add New Item to Group 'Source Group 1' as shown in Figure 6.

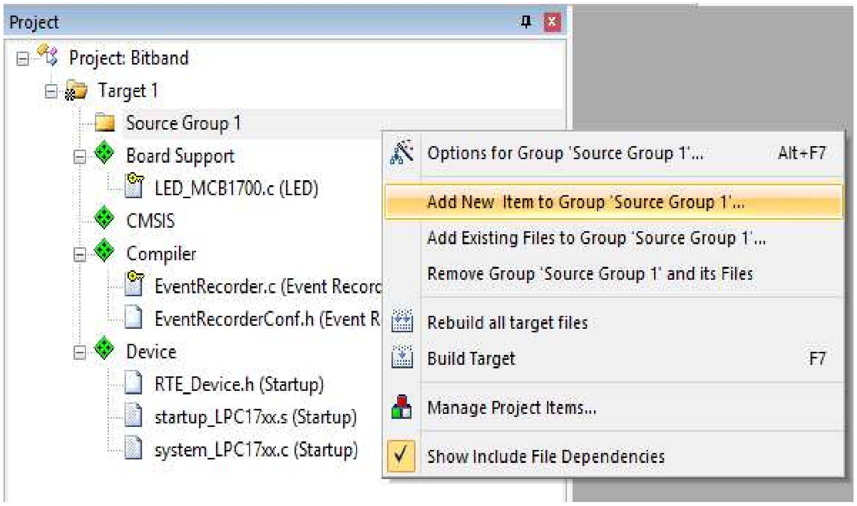
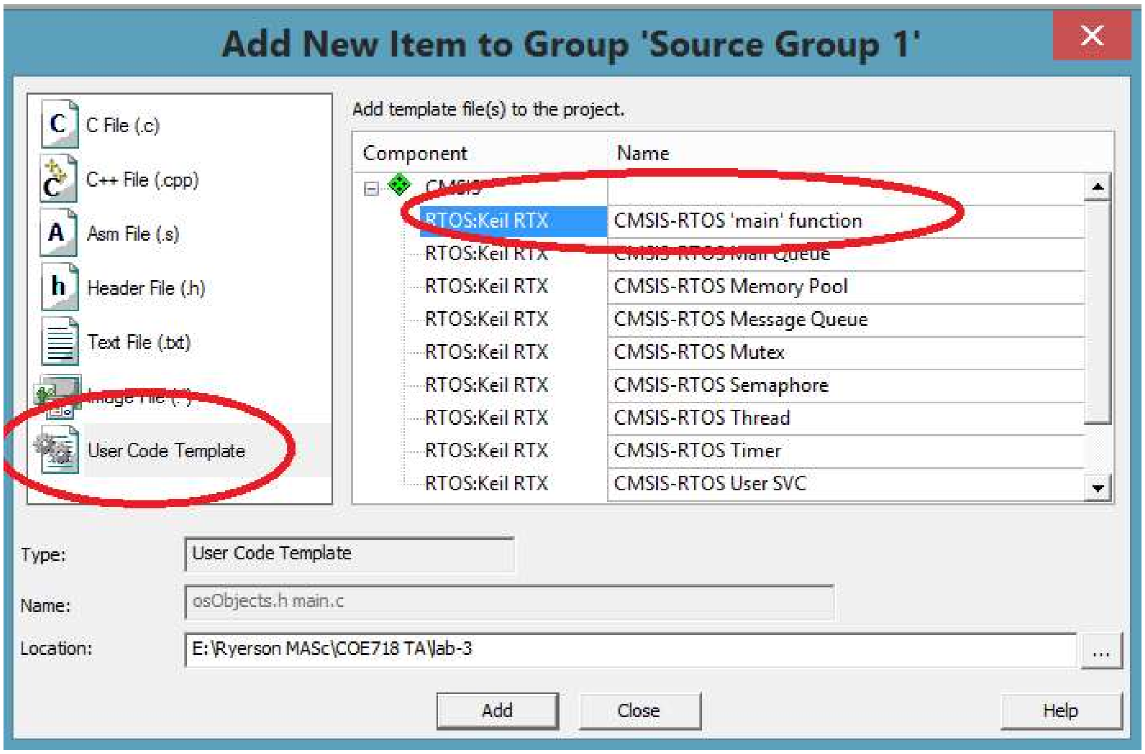

Project
Project: Bitband
Target 1
Source Group 1
Board Support
LED_MCB1700.c (LED)
CMSIS
Compiler
EventRecorder.c (Event Recorc
EventRecorderConf.h (Event R
Device
RTE_Device.h (Startup)
startup_LPC17xx.s (Startup)
system_LPC17xx.c (Startup)
Options for Group 'Source Group 1'... Alt+F7
Add New Item to Group 'Source Group 1'...
Add Existing Files to Group 'Source Group 1'...
Remove Group 'Source Group 1' and its Files
Rebuild all target files
Build Target F7
Manage Project Items...
Show Include File Dependencies

Figure 6

- Select the User Code Template >expand CMSIS >RTOS:KeilRTX> CMSIS-RTOS'main' function. Then click Add as shown in Figure 7. The file will be added to the project.



Component	Name
CMSIS	
RTOS:Keil RTX	CMSIS-RTOS 'main' function
RTOS:Keil RTX	CMSIS-RTOS Mail Queue
RTOS:Keil RTX	CMSIS-RTOS Memory Pool
RTOS:Keil RTX	CMSIS-RTOS Message Queue
RTOS:Keil RTX	CMSIS-RTOS Mutex
RTOS:Keil RTX	CMSIS-RTOS Semaphore
RTOS:Keil RTX	CMSIS-RTOS Thread
RTOS:Keil RTX	CMSIS-RTOS Timer
RTOS:Keil RTX	CMSIS-RTOS User SVC

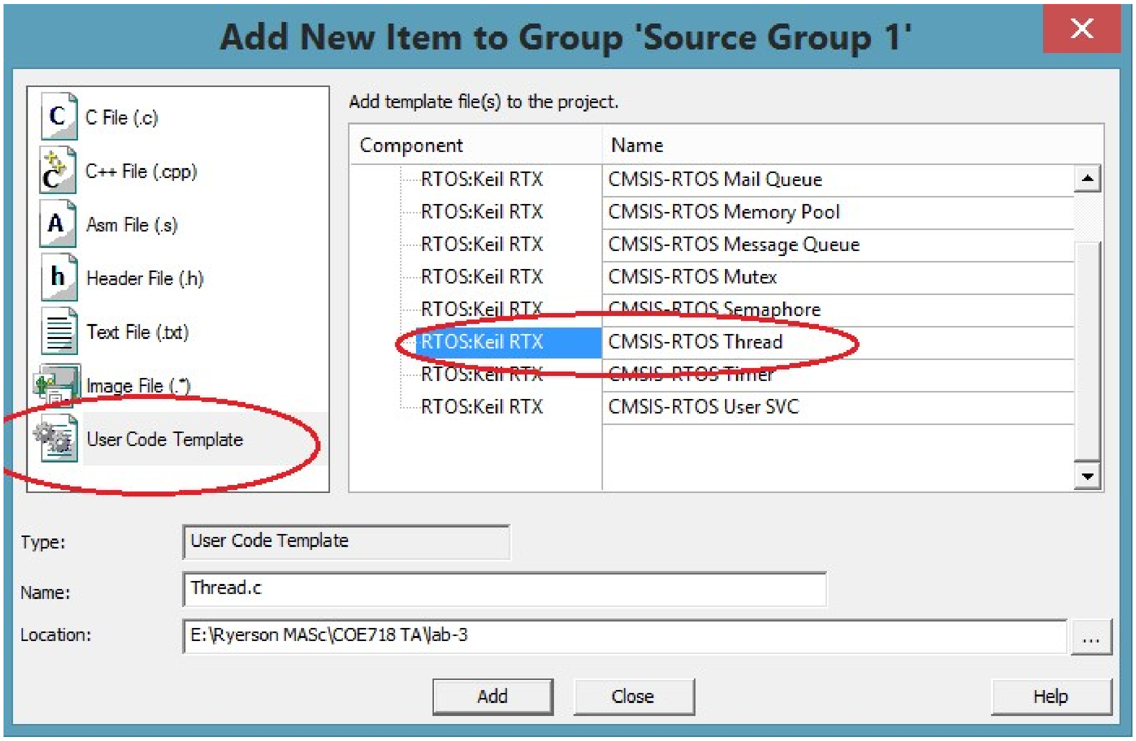
Type: User Code Template

Name: osObjects.h main.c

Location: E:\Ryerson MASc\COE718 TA\lab-3

Figure 7

- Repeat and this time select CMSIS>RTOS:Keil RTX> CMSIS-RTOS Thread. Then click Add and another file will be added to your project as shown in Figure 8.



Add template file(s) to the project.

Component	Name
RTOS:Keil RTX	CMSIS-RTOS Mail Queue
RTOS:Keil RTX	CMSIS-RTOS Memory Pool
RTOS:Keil RTX	CMSIS-RTOS Message Queue
RTOS:Keil RTX	CMSIS-RTOS Mutex
RTOS:Keil RTX	CMSIS-RTOS Semaphore
RTOS:Keil RTX	CMSIS-RTOS Thread
RTOS:Keil RTX	CMSIS-RTOS Timer
RTOS:Keil RTX	CMSIS-RTOS User SVC

Type: User Code Template

Name: Thread.c

Location: E:\Ryerson MASc\COE718 TA\lab-3

Figure 8

- Now your Project Folders should look like the Figure 9 depicted below.

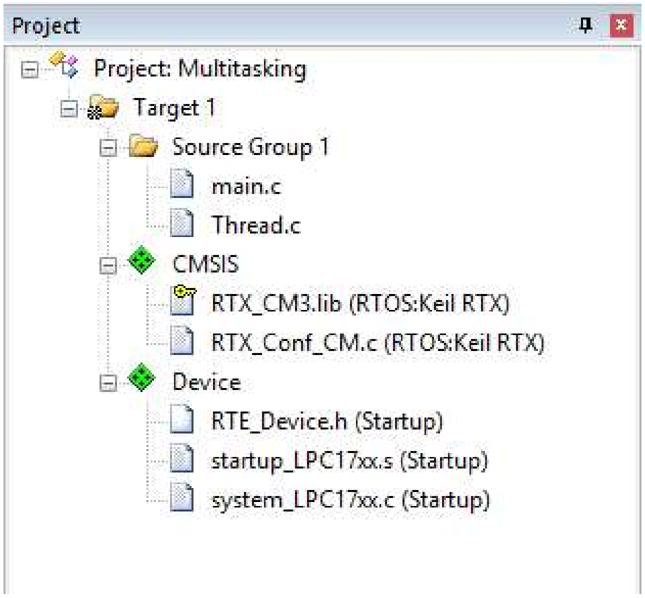

Project
Project: Multitasking
Target 1
Source Group 1
main.c
Thread.c
CMSIS
RTX_CM3.lib (RTOS:Keil RTX)
RTX_Conf_CM.c (RTOS:Keil RTX)
Device
RTE_Device.h (Startup)
startup_LPC17xx.s (Startup)
system_LPC17xx.c (Startup)

Figure 9

- Click on the icon on the top menu as shown in Figure 10

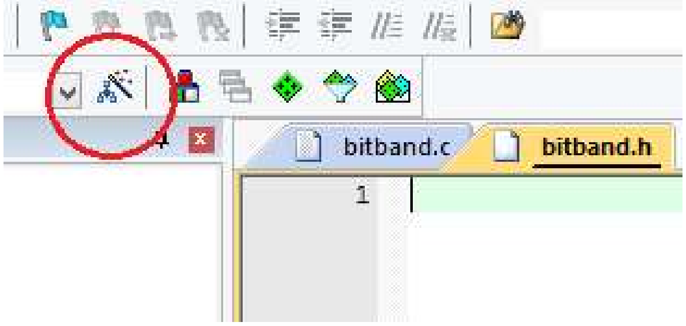
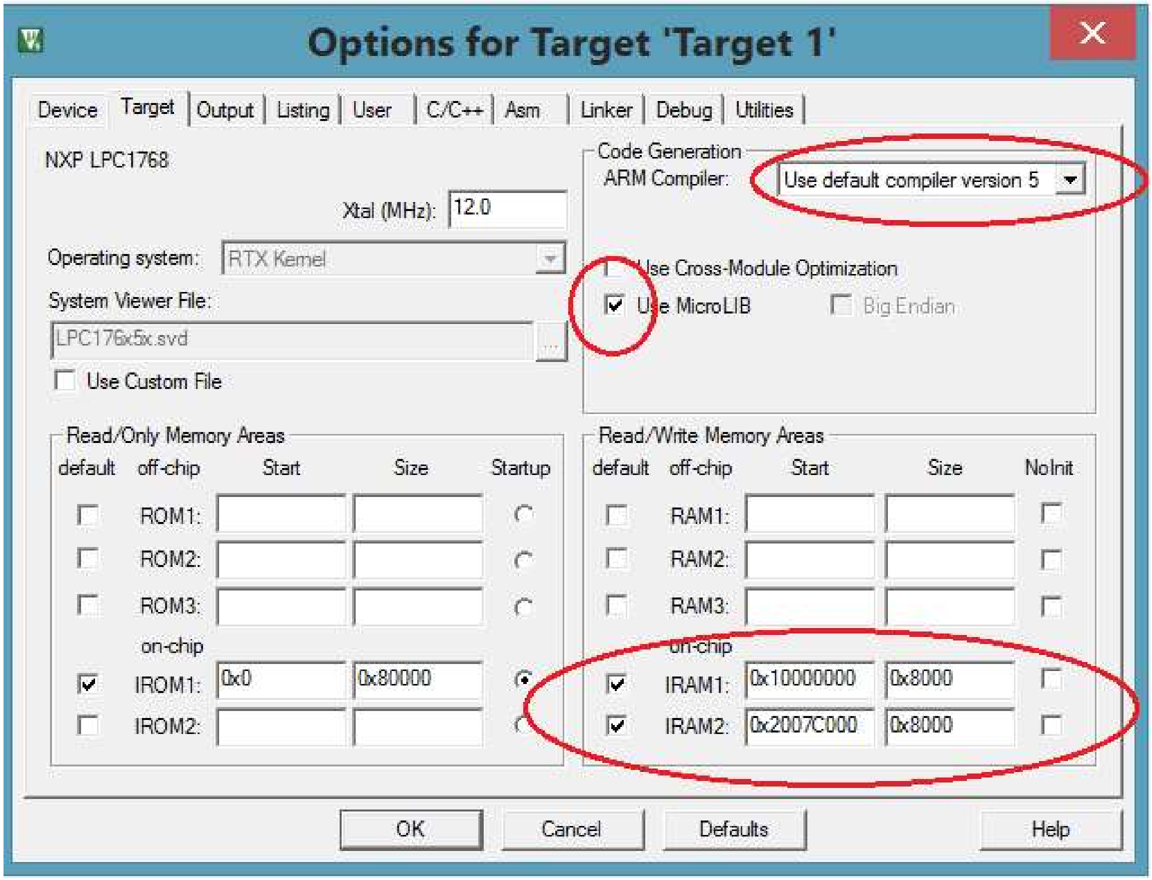


Figure 10

- Do the Following changes shown in Figure 11: Under Target. ARM Compiler> **Use default compiler version 5**. Check Use > **Micro LIB**.



Options for Target 'Target 1'

NXP LPC1768

Xtal (MHz): 12.0

Operating system: RTX Kernel

System Viewer File: LPC176x5x.svd

□ Use Custom File

Code Generation

ARM Compiler: Use default compiler version 5

□ Use Cross-Module Optimization

☑ Use MicroLIB □ Big Endian

Read/Only Memory Areas

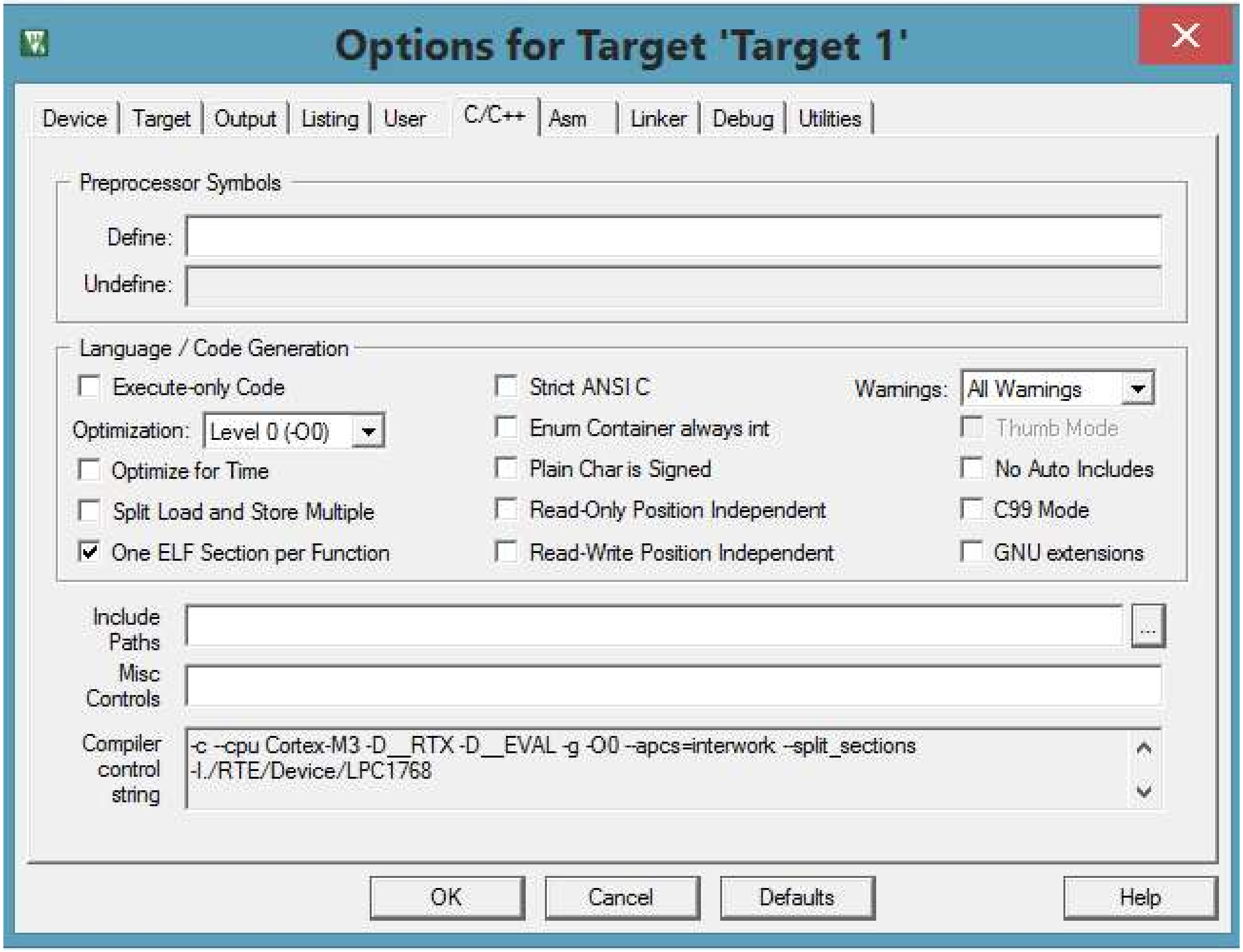
default	off-chip	Start	Size	Startup
□	ROM1:			○
□	ROM2:			○
□	ROM3:			○
	on-chip			
☑	IROM1:	0x0	0x80000	●
□	IROM2:			○

Read/Write Memory Areas

default	off-chip	Start	Size	NoInit
□	RAM1:			□
□	RAM2:			□
□	RAM3:			□
	on-chip			
☑	IRAM1:	0x10000000	0x8000	□
☑	IRAM2:	0x2007C000	0x8000	□

Figure 11

- Select C/C++ and a window similar to Figure 12 will appear.



Define:

Undefine:

□ Execute-only Code

Optimization: Level 0 (-O0)

□ Optimize for Time

□ Split Load and Store Multiple

☑ One ELF Section per Function

□ Strict ANSI C

□ Enum Container always int

□ Plain Char is Signed

□ Read-Only Position Independent

□ Read-Write Position Independent

Warnings: All Warnings

□ Thumb Mode

□ No Auto Includes

□ C99 Mode

□ GNU extensions

Include Paths

Misc Controls

Compiler control string

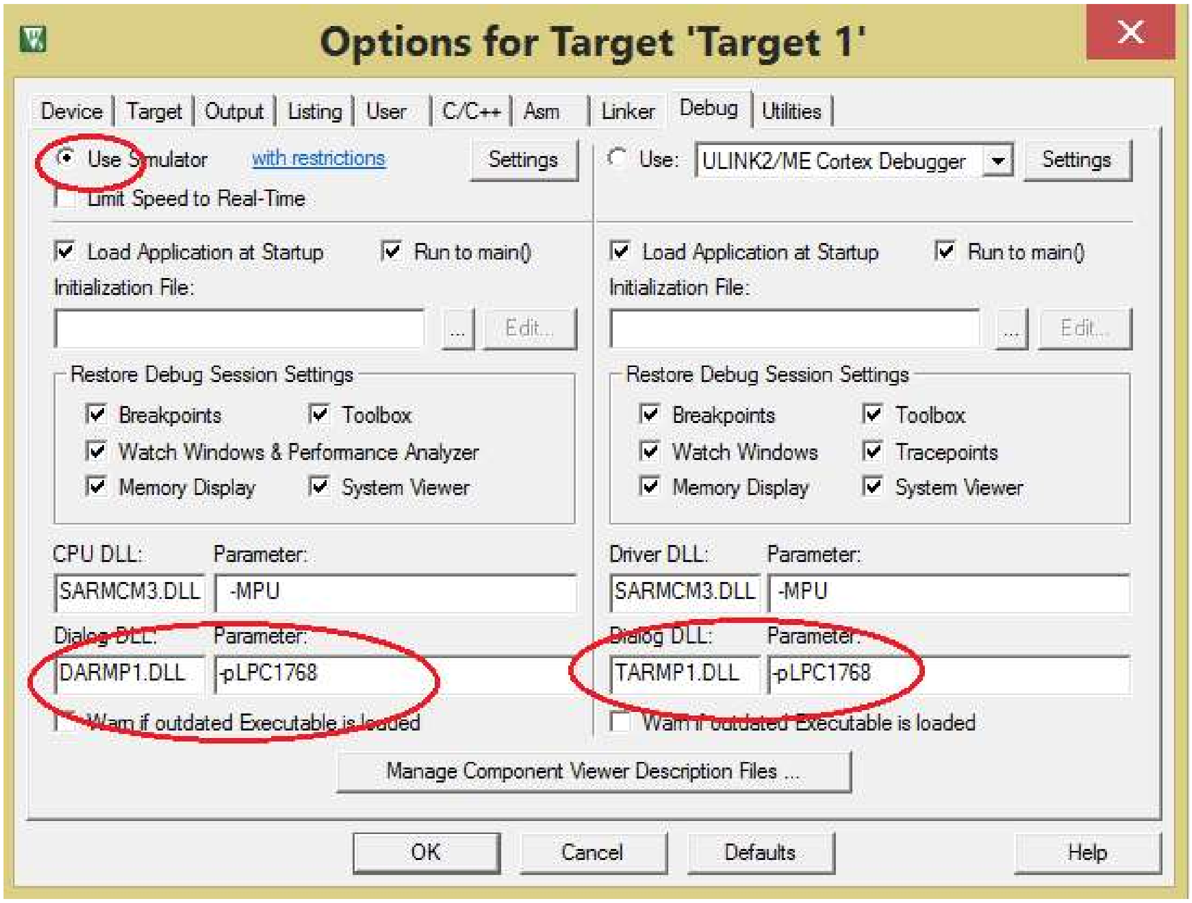
```
-c --cpu Cortex-M3 -D__RTX -D__EVAL -g -O0 --apcs=interwork --split_sections
-I./RTE/Device/LPC1768
```

Figure 12

- Then select Debug option and check Use Simulator. Perform the following changes also.
  Dialog DLL: DARMP1.DLL, **Parameter**: -pLPC1768

Debug Window should look like the one in Figure 13. Then Press **OK.**

Now the Project Directory is all set for the Lab3 and related assignment.
Debug and Run the project and make sure no errors or warnings exist.
Then Replace the contents of main.c with the main.c file provided on D2L, and Threads.c file with the contents of Threads.c from D2L.



● Use Simulator

□ Limit Speed to Real-Time

☑ Load Application at Startup ☑ Run to main()

Initialization File:

Restore Debug Session Settings

☑ Breakpoints ☑ Toolbox

☑ Watch Windows & Performance Analyzer

☑ Memory Display ☑ System Viewer

CPU DLL: SARMCM3.DLL Parameter: -MPU

Dialog DLL: DARMP1.DLL Parameter: -pLPC1768

□ Warn if outdated Executable is loaded

○ Use: ULINK2/ME Cortex Debugger

☑ Load Application at Startup ☑ Run to main()

Initialization File:

Restore Debug Session Settings

☑ Breakpoints ☑ Toolbox

☑ Watch Windows ☑ Tracepoints

☑ Memory Display ☑ System Viewer

Driver DLL: SARMCM3.DLL Parameter: -MPU

Dialog DLL: TARMP1.DLL Parameter: -pLPC1768

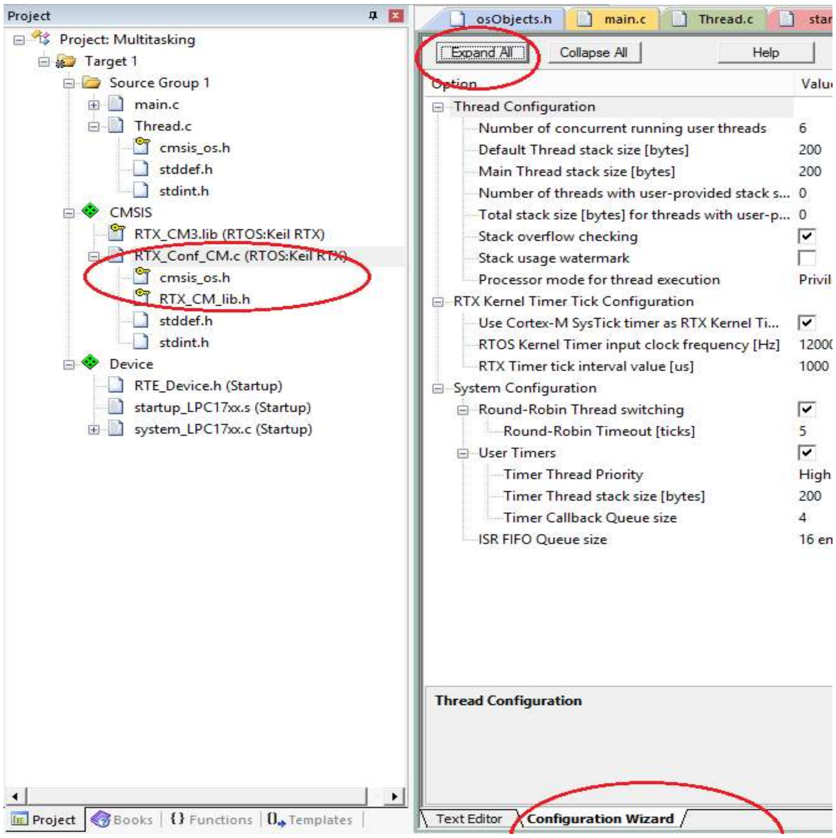
□ Warn if outdated Executable is loaded

Figure 13

RTX must be configured for specifications such as the time slice frequency of the CPU's systick timer and the arbitration techniques for the multi-threaded applications.

Open the file *RTX_Conf_CM.c*, double click from the project directory as shown in Figure 14.
Select Configuration Wizard and click Expand All.

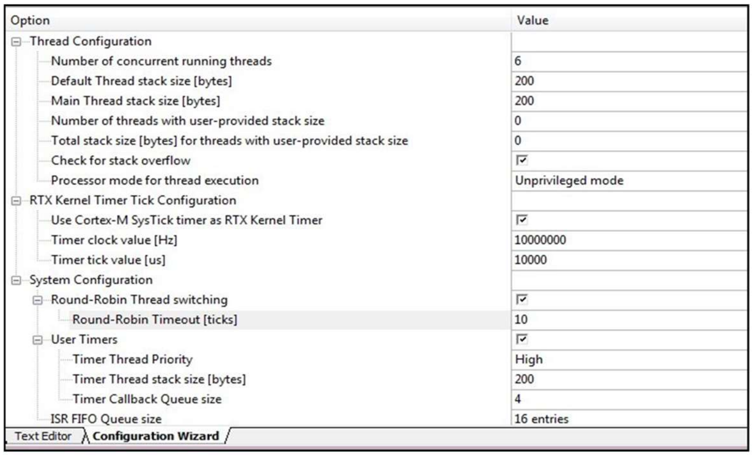
Make sure that the option *Use Cortex-M SysTick timer as RTX Kernel Timer* is selected, the RTOS Kernel *Timer input clock frequency [Hz]* option is set to **10000000** (10 MHz), and the RTX *Timer tick interval value[us]* option is set to **10000** (10ms). Make sure that the "User Timers" option is also checked for round-robin scheduling. Your configuration file should now resemble Figure 15.



Option	Valu
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 s...	0
Total stack size [bytes] for threads with user-p...	0
Stack overflow checking	☑
Stack usage watermark	☐
Processor mode for thread execution	Privil
RTX Kernel Timer Tick Configuration	
Use Cortex-M SysTick timer as RTX Kernel Ti...	☑
RTOS Kernel Timer input clock frequency [Hz]	1200(
RTX Timer tick interval value [us]	1000
System Configuration	
Round-Robin Thread switching	☑
Round-Robin Timeout [ticks]	5
User Timers	☑
Timer Thread Priority	High
Timer Thread stack size [bytes]	200
Timer Callback Queue size	4
ISR FIFO Queue size	16 en

Thread Configuration

Figure 14



Option	Value
Thread Configuration	
Number of concurrent running 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
Check for stack overflow	☑
Processor mode for thread execution	Unprivileged mode
RTX Kernel Timer Tick Configuration	
Use Cortex-M SysTick timer as RTX Kernel Timer	☑
Timer clock value [Hz]	10000000
Timer tick value [us]	10000
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

Figure 15

3. Analyzing the RTX Project

3.1 Watch Windows

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

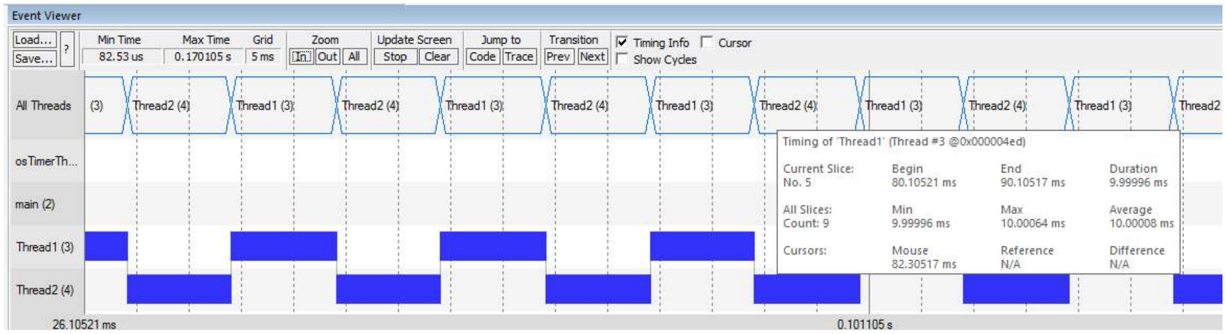
1. Open the watch window by selecting View > Watch Window > Watch 1.
2. You may hover the mouse over the variable name in the code window, right-click and select **Add ‘counta'** or **‘countb' to...** >> Watch 1.
3. When you click the RUN icon to execute the program, the values of ‘counta’ and ‘countb’ 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.

3.2 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.

3.3 RTX Event Viewer

The Event Viewer is a graphical representation of a program’s thread-based execution timeline. An example is shown in Figure 16. The Event Viewer runs on the Keil simulator but must be configured properly for CPU execution using a Serial Wire Viewer (SWV).



Min Time 82.53 us
Max Time 0.170105 s
Grid 5 ms
☑ Timing Info ☐ Cursor
☐ Show Cycles

Timing of 'Thread1' (Thread #3 @0x000004ed)

Current Slice: No. 5	Begin 80.10521 ms	End 90.10517 ms	Duration 9.99996 ms
All Slices: Count: 9	Min 9.99996 ms	Max 10.00064 ms	Average 10.00008 ms
Cursors:	Mouse 82.30517 ms	Reference N/A	Difference N/A

Figure 16

To use this feature of the Serial Wire Viewer (SWV).

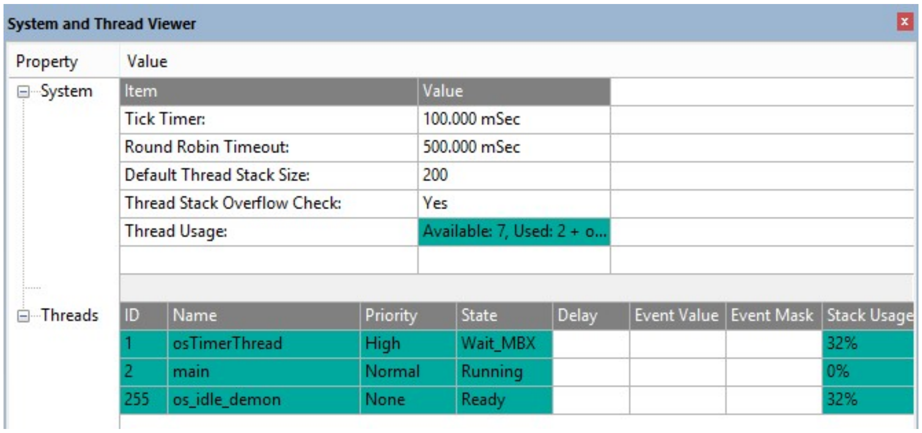
1. In the main menu select Debug >> OS Support >> Event Viewer. A window should appear.
2. Click RUN. Click the "All" button under the zoom menu in the Event Viewer window. You may also select "In" or "Out" to adjust the view of the timeline which dynamically updates as the program continues to execute.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

similar to that of Figure 16.

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.

3.4 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 17. 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.



System and Thread Viewer

Property: System

Item	Value
Tick Timer:	100.000 mSec
Round Robin Timeout:	500.000 mSec
Default Thread Stack Size:	200
Thread Stack Overflow Check:	Yes
Thread Usage:	Available: 7, Used: 2 + o...

Property: Threads

ID	Name	Priority	State	Delay	Event Value	Event Mask	Stack Usage
1	osTimerThread	High	Wait_MBX				32%
2	main	Normal	Running				0%
255	os_idle_demon	None	Ready				32%

Figure 17

To use this feature:

- Select Debug >> OS Support >> System and Thread Viewer.
- 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 Multithreaded Application with uVision and RTX

4.1 Understanding the RTX Program

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

Thread1 and Thread2 will loop infinitely using a round-robin scheduling technique. This timing specification was included in the config file (RTX_Conf_CM.c). osKernelInitialize() and osKernelStart() will setup the round-robin scheduling definition for the threads and execute the kernel respectively. 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

As in 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

1. Now that we have analyzed a simplistic multi-threaded application and its various performance features using uVision. Let's take a look at the code once more step-by-step using uVision's analysis tools.
2. Re-execute the code and take a look at the Event Viewer. Which thread executes first? osTimerThread() thread initializes and executes - this thread is responsible for executing time management functions specified by ARM's RTOS configuration.
3. The program starts executing from main(), where main() ensures that:
   a. The Cortex-M3 system and timers are initialized - *SystemInit()*
   b. An os kernel is initialized for interfacing software to hardware - *osKernelInitialize()*
   c. Creates the threads to execute thread1 and thread2 - *Init_Thread ();*
   d. Starts the kernel to begin thread switching - *osKernelStart()*

4. The Thread1 thread executes for its round-robin time slice since it is created first. After 10msec the timer thread forces control to the Thread2 thread.
5. The Thread2 thread executes during its time slice for 10msec and is forced to stop again and execute task1. This occurs infinitely.

4.4 Processor Idling Time

As an exercise, let us determine the idling time of the code we have been currently working with by using the idle demon, i.e. 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, but the countIDLE variable does not. *What does this mean?* This The CPU is currently under 100% utilization by the task 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.

Note on Pre-emptive/Non Pre-emptive Scheduling Versus Round-Robin Scheduling

To implement pre-emptive/ non pre-emptive scheduling techniques, make note that the *RTX_Conf_CM*.c file must be adjusted. Specifically in the Configuration Wizard, the option System Configuration >> Round- Robin Thread switching must be disabled. Ensure however that the systick timers are enabled.

5. Implementing Various Scheduling Algorithms

Exercise 1- Setting Priority: Exit the Debug mode to access the Thread.c file. Change the line:

```
osThreadDef(Thread1, osPriorityNormal, 1, 0); to
osThreadDef(Thread1, osPriorityAboveNormal, 1, 0);
```

Compile the program and return to Debug mode. Run the program and open the Event Viewer window.
Question 1: *What do you notice?*
By setting the priority of Thread2 to a higher priority than that of Thread1, a **pre-emptive** scheduling technique was created where the higher priority thread will execute to completion first. Since Thread1 was created first, it

was expected to run first. However, Thread1 will never be executed due to its "Normal" priority (in comparison to Thread2's "AboveNormal") and the fact that Thread2 executes infinitely. Conversely, if the code was programmed such that the Thread2 terminates after a finite time (when its workload completes), Thread1 would thereafter be able to execute. It is recommended that the CMSIS-RTOS API Thread Management and osPriority enumerations be consulted during coding.

Exercise 2 - Pre-emptive Scheduling: Exit Debug mode to access the Thread.c file again. Change the Thread1 and Thread2 function code to the following:

```
void Thread2 (void const *argument) {
	for (;;){     // Infinite loop – runs while thread2 runs.
		countb++;     // Increment global variable countb indefinitely
		osThreadYield();
	}                        // suspend thread

}

void Thread1 (void const *argument) {
	for (;;){     // Infinite loop – runs while Thread1 runs.
		counta++;     // Increment global variable counta indefinitely
		osThreadYield();
	}     // suspend thread

}
```

Also make sure to change:
`osThreadDef(Thread1, `**`osPriorityAboveNormal`**`, 1, 0);` back to
`osThreadDef(Thread1, `**`osPriorityNormal`**`, 1, 0);`

Recompile the files. Enter Debug mode. Open a Watch window to track the counta and countb variables, along with the Event Viewer. Reset the program and click RUN.

Question 2: *How does the execution of the code using* `osThreadYield()` *differ from round-robin?*
If you were successful, you will observe short execution time slices per thread in the Event Viewer, where it almost appears as if the threads were running as round-robin (after several msec). 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 2.52us 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 variable and task using the performance-based tools.
Exercise 3: Stop the previous program and exit Debug mode to gain access to the Thread.c file. Remove the `osThreadYield()` functions you implemented in the last exercise. Change the Thread1 and Thread2 function code to the following:

```
void Thread2 (void const *argument) {
  for(;;) {
	countb++;
	osDelay(1);
  }
}
void Thread1 (void const *argument) {
     for(;;) {
	counta++;
	osDelay(2);
    }
}
```

Recompile the files and enter Debug mode. Setup the Watch 1 window with the variables counta, countb, and countIDLE. RUN the program

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?*

6. Introduction to Rate Monotonic Scheduling for Real-time Applications

6.1 Virtual Timers

Virtual timers are a type of countdown timer used in the CMSIS RTX API. Each timer possesses a callback function which is triggered once the timer has counted down. This callback indicates what action the timer is to perform once triggered. Therefore, the instantiation of multiple timers can countdown various periods of time, useful for the multiple tasks executed in a real-time system.

Virtual timers are defined after the #include and #define area in the .c code as:

```
osTimerDef(timer0_handle, callback);
```

Note its callback function must be declared before defining the timer(s). The virtual timer may then be instantiated within the main() as an RTX thread.

```
osTimerId timer_0= osTimerCreate(osTimer(timer0_handle), osTimerPeriodic, (void *)0);
```

The above statement creates a timer called timer_0 that specifies information for once its countdown has triggered. The timer0_handle will call the callback function with the argument (void *)0. In terms of the frequency of the countdown timer, osTimerPeriodic defines a periodic timer whereas osTimerOnce is used to declare a single-shot timer. The timer can then be started in the main() at any time using the following statement.

```
osTimerStart(timer_0, 3000);
```

It signifies that the timer timer_0 should start, with a countdown of 3000 milliseconds. The use of multiple virtual timers can trigger the callback function at various times and/or frequencies. An example application will be provided to you in the next section after explaining the importance of inter-thread communication in RTX or any other real-time operating system.

6.2 Inter-thread Communication - Signals and Waits

Up to now we have learned how to create threads, set their priorities, and use timers provided by RTX to create and schedule applications. In many applications however, there is a need to synchronize and communicate information among various threads. There are several means to communicate between threads in an RTOS. In the first part of this lab, we will focus on the use of signal and wait flags to synchronize execution between application threads. The concept is synonymous to signal and wait flags learned in general operating systems also.

A single thread in the RTX API may contain up to 8 signal flags stored in its thread control block. Signals are used to synchronize (signal or halt) the execution of threads. To synchronize threads, a thread usually "waits" for a "signal" to continue its execution. If a thread's signal flag number matches the wait flag number that was asserted by another thread, then the waiting thread can be released from the waiting state, and it will transition to the ready state for execution. Thus, this method is used to synchronize any number of threads - the signal thread must complete a certain task before the waiting thread can continue.

Like the previous lab, a thread is created and given an ID as given below.

```
void led_Thread1 (void const *argument);
osThreadDef(led_Thread1, osPriorityNormal, 1, 0);
```

```
 osThreadId T_led_ID1;

int main(void){
...
T_led_ID1 = osThreadCreate(osThread(led_Thread1), NULL);
...}
```

A wait flag may be set in the code as follows:

```
osSignalWait (0x03,osWaitForever);
```

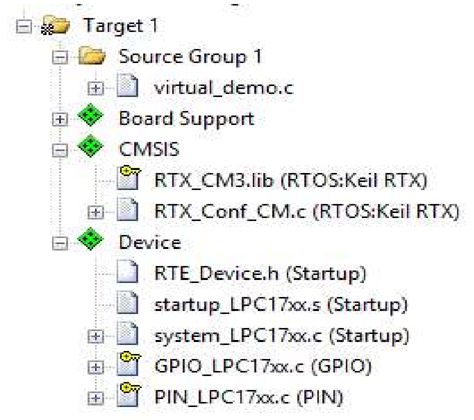
The above statement signifies that the thread is waiting for the signal flag 0x03 to be asserted. The second parameter indicates the maximum duration (in milliseconds) that the thread should wait to be signalled. In this case the wait period is osWaitForever.

A signal may be sent to a thread or cleared using:

```
osSignalSet(T_led_ID2,0x01);    or       osSignalClear(T_led_ID2, 0x01);
```

6.3 Example Application

Launch the µVision application. Create a new project "virtual_demo" in your "lab3/example" folder. Select the LPC1768 processor chip. Copy ‘virtual_demo.c’ file provided to you on D2L under lab3 folder, to your project directory. Configure your project workspace with the same settings as you did in the other exercises. Your project folder should resemble Figure 18.



```
Target 1
    Source Group 1
        virtual_demo.c
    Board Support
    CMSIS
        RTX_CM3.lib (RTOS:Keil RTX)
        RTX_Conf_CM.c (RTOS:Keil RTX)
    Device
        RTE_Device.h (Startup)
        startup_LPC17xx.s (Startup)
        system_LPC17xx.c (Startup)
        GPIO_LPC17xx.c (GPIO)
        PIN_LPC17xx.c (PIN)
```

Figure 18

Check the "Options for Target"->C/C++ -> C99 Mode. Make sure that the timers are enabled in RTX_Conf_CM.c.

Open the virtual_demo.c file and examine the code. Note the following: two virtual timers created and started with the countdown period of 3000 and 1000 respectively. The callback function is passed with the specified timer parameters once the timer has triggered. Three threads are created for the signal and wait demonstration (T_led_ID1, T_led_ID2, T_led_ID3). LEDs will flash according to the signal/wait pair that have been matched, or virtual timer currently being called back and executed.

Build the project and analyze the application execution (LEDs), its timing characteristics, and how it correlates with the sample code given. Use the debugging and analysis tools to trace variables to get an in-depth understanding. Make sure to comment out the osDelay() while debugging.

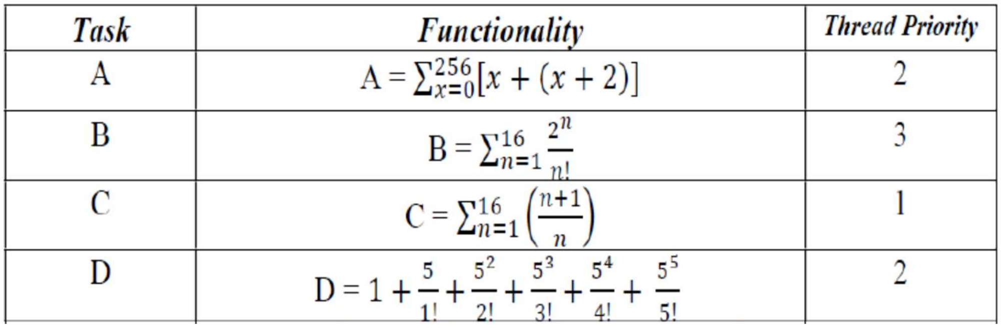
7. Optional Lab Assignments (Bonus Marks 3% of the Course Marks)
Submit Lab-3 Report through D2L

7.1 Part I

The following outlines the specifications for 3 different scheduling applications (Questions 1, 2 and 3 in section 5). You should create an analysis version for each application. The analysis will be used for debug mode to analyze performance of your applications for your report.

1. Implement a **round-robin** scheduling example using 3 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 lab. Ensure that the tasks do not run infinitely, and they have a finite workload with respect to time.

TABLE 1: LIST OF PRE-EMPTIVE TASKS



Task	*Functionality*	*Thread Priority*
A	$A = \sum_{x=0}^{256}[x + (x + 2)]$	2
B	$B = \sum_{n=1}^{16} \frac{2^n}{n!}$	3
C	$C = \sum_{n=1}^{16} \left(\frac{n+1}{n}\right)$	1
D	$D = 1 + \frac{5}{1!} + \frac{5^2}{2!} + \frac{5^3}{3!} + \frac{5^4}{4!} + \frac{5^5}{5!}$	2

2. Table 1 provides a list of pre-emptive threads, with their function and priority listed. Note: The lower the number in the Priority column, the higher the priority. Write the **pre-emptive** code for a scheduling algorithm which invokes the threads and functionalities in Table I based on their priority level (i.e. Task C should finish computing first). Each task should print their final result to stdout (using printf or the watch window).

Submit the printout of your .c code, RTX_Conf_CM.c Configuration Wizard file, and snapshots of your Event Viewer and Performance Analyzer windows for **each** application.

7.2 Part II

Implement an RMS algorithm using the following process set given in Table 2.

Table 2: 3-Process Set

Process	Period (T)	Computation Time (C)	Priority (P)
A	40000	20000	3
B	40000	10000	2
C	20000	5000	1

Schedule the above process set using inter-thread communication mechanisms and virtual timers. Since RMS is a Fixed Priority Scheduling method, ensure that the priorities are followed accordingly (i.e., lower the number, higher the priority). Note that the periods are much longer than the lecture examples due to debugging purposes. Use a custom delay() function.

Hand in the .c code for the RMS scheduling technique of this process set. Moreover, include an execution timeline for the processes (may be drawn). Submit this timeline with your code. Make sure that your program execution matches your calculated timeline for verifying the correctness.

References

1. "The Keil RTX Real Time Operating System and µVision" www.keil.com. An ARM Company.
2. "Keil µVision and Microsemi SmartFusion" *Cortex-M3 Lab* by Robert Boys www.keil.com.
3. "Keil RTX RTOS the easy way" by Robert Boys www.keil.com.