Brandon Ho 500727531

Lab 3: Scheduling Multithreaded Application with RTX and uVision

Introduction:

Multithreading allows the simultaneous execution of multiple parts of a program to optimize the use of processor execution time. Multithreading can be implemented with ARM based hardware and the Kiel uVision IDE through the following process. In the startup_device file the SystemInit() configures the oscillator (PLL). For systems with variable clock, it updates the SystemCoreClock variable which is a global variable containing the clock value. The clock frequency held by the System Core Clock variable is supplied to the Systick timer. On each tick the kernel will run the scheduler to determine if it is necessary to perform a context switch and replace the running thread. “OsKernelInitialize()” is called in the main which initializes the RTOS Kernel to allow peripheral setup and creation of the “os.ThreadCreate” object. Calling “Init_Thread ()” in the main initializes threads. Calling “osKernelStart” starts thread switching. The specific function of a thread is user defined and created like a traditional function. The “osThreadcreate()” function starts a thread process by adding it to the “active threads list” and setting it to the ready state. When the priority of the thread is higher than the current running thread, the new thread starts and becomes the new running thread. Priority of each thread can be set with the “osThreadDef()” function. The priority of the thread determines the execution order. Equal priority threads are either executed sequential based on ID, by a timer mechanism that switches thread execution on a timer, or by explicit commands such as “osThreadYield()” that relinquishes current thread execution to the next thread.

Procedure:

1) Create a new uVision Project by selecting Project->New uVision Project from top menu. (NOTE: Make sure all other projects are closed)

2) Name the project “Multi-tasking” and save to destination of choice.

3) In the “Select Device for Target” window, select LPC1768 and OK.

4) Select the following Packages in the Run-time Environment window.
    a. CMISI->CORE
    b. CMSIS->RTOS(API)->Keil RTX
    c. Device ->Startup

5) Select “Add New Item to Group ‘Source Group 1’…”, via right click options under Project: Multitasking->Target 1 tree hierarchy.

6) Under the “Add new Item to Group ‘Source Group 1’” window select “User Code Template CMSIS->RTOS:KeilRTX->CMSIS-RTOS’main; function “ and add to Group.

7) Repeat 5-6, but select “User Code Template CMSIS->RTOS:KeilRTX->CMSIS-RTOS Thread” instead and add to group.

8) Under “Options for Target” select the following:
   a. Under Target folder tab:
      i) Select the “Use default complier Version 5”.
      ii) Select “Use MicroLIB”.
      iii) Select “IRAM1” and “IRAM2”.
   b. Under Debug tab:
      i) Select “Use Simulator”.
      ii) In the left Dialog DLL textbox insert “DARMP1.DLL” with parameter “-pLPC1768”.
      iii) In the right Dialog DLL textbox insert “TARMP1.DLL” with parameter “-pLPC1768”.

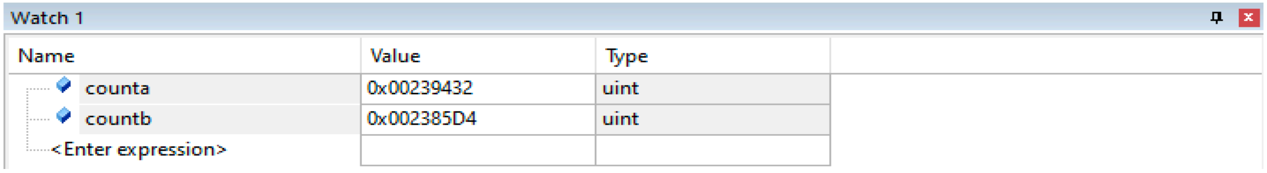
9) Replaced the contains of the main.c and threads.c with those provided. They can be accessed from “Project:Multitasking->Target 1-> Source Group 1” under the project tree hierarchy.

10) Select “Project:Multitasking->Target 1->CMSIS->RTX_Config_CM.c “ in the project tree hierarchy by “double left clicking” and toggle the “Configuration Wizard”. Ensure the following settings are selected:
   a. Under “RTX Kernel Timer-Tick Configurator tree:
      i) Select Use-Cortex-M SysTick Timer as TRX Kernel Timer
      ii) Set “Timer clock value [Hz]” to 10000000
   b. Under System Configurator tree:
      i) Round-Robin Timeout [ticks] to 10
      ii) Select User Timer

11) Compile and Enter Debug mode.

12) Enable Watch1 viewer through menu bar “View->Watch Window->Watch 1”.

13) Manually scan Thread.c file and by right-clicking over variables “counta”, add variable to Watch 1 viewer. Do the same for variable “countb”. Execute Simulation. The following should be observed (Fig1).

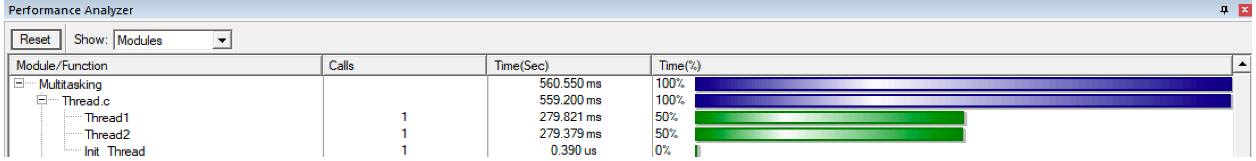


Watch 1

Name	Value	Type
counta	0x00239432	uint
countb	0x002385D4	uint
<Enter expression>		

Fig1: Watching window variables --NOTE: Observe how value has incremented.

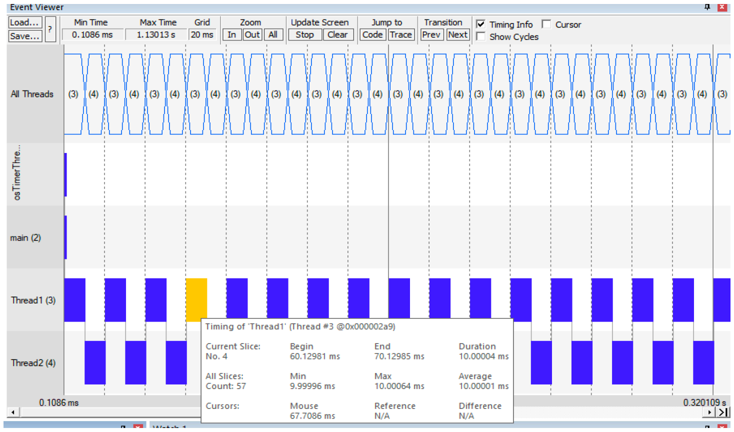
14) Enable Performance Analyzer through menu bar “View->Analysis Windows->Performance Analyzer”. Expands the tree as follows “Multitasking->Thread.c”. Reset and Re-Run the simulation. The following should be observed (Fig2).



Module/Function	Calls	Time(Sec)	Time(%)
Multitasking		560.550 ms	100%
Thread.c		559.200 ms	100%
Thread1	1	279.821 ms	50%
Thread2	1	279.379 ms	50%
Init_Thread	1	0.390 us	0%

Fig2: Performance Analyzer

15) Enable Event Viewer through the menu bar “Debug->OS support->Event Viewer”. Reset and Re-Run the simulation for a short period of time. The following should be observed (Fig3). NOTE: The threads alternate execution every 10ms.



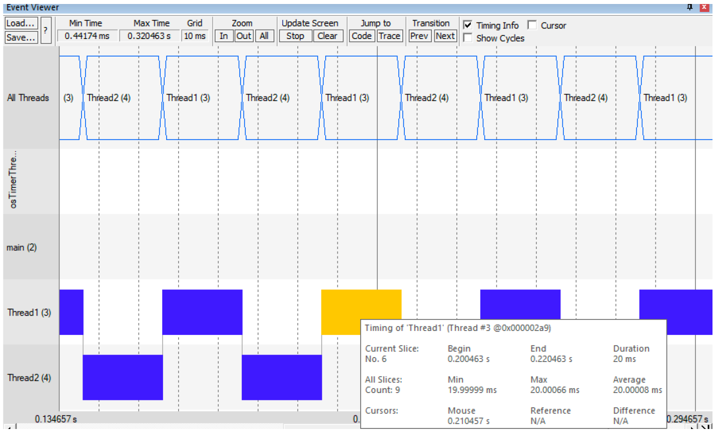
Min Time: 0.1086 ms Max Time: 1.13013 s Grid: 20 ms

```
Timing of 'Thread1' (Thread #3 @0x000002a9)

Current Slice:   Begin          End            Duration
No. 4            60.12981 ms    70.12985 ms    10.00004 ms

All Slices:      Min            Max            Average
Count: 57        9.99996 ms     10.00064 ms    10.00001 ms

Cursors:         Mouse          Reference      Difference
                 67.7086 ms     N/A            N/A
```

Fig3: Event viewer with 10ms Round-Robin cut-off-interval

16) Repeat Step 10->b.-> i) and set Round-Robin Timeout [ticks] to 20. The following changes should be observed (Fig4). NOTE: The threads alternate execution every 20ms.

17)

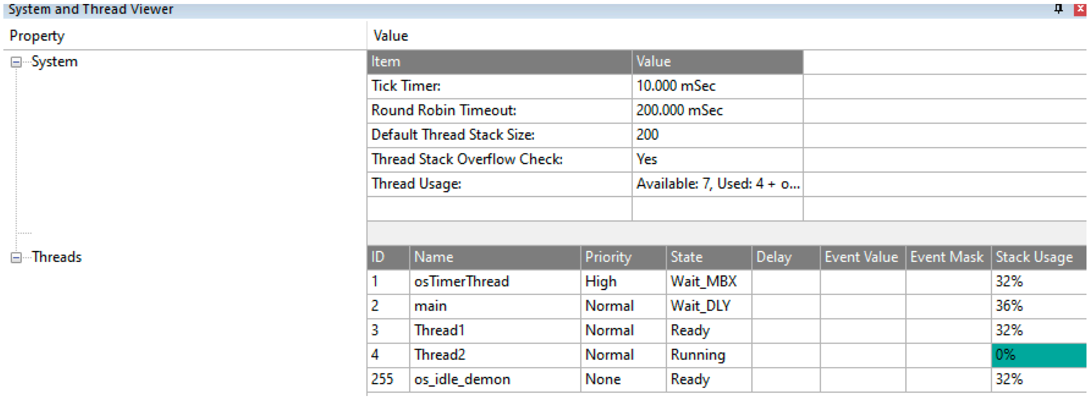


Min Time: 0.44174 ms Max Time: 0.320463 s Grid: 10 ms

```
Timing of 'Thread1' (Thread #3 @0x000002a9)

Current Slice:   Begin          End            Duration
No. 6            0.200463 s     0.220463 s     20 ms

All Slices:      Min            Max            Average
Count: 9         19.99999 ms    20.00066 ms    20.00008 ms

Cursors:         Mouse          Reference      Difference
                 0.210457 s     N/A            N/A
```

Fig4: Event viewer with 20ms Round-Robin cut-off-interval

18) Enable System and Thread Viewer through the menu bar “Debug->OS support-> System and Thread Viewer”. Reset and Re-Run simulation. The following should be observed (Fig5).



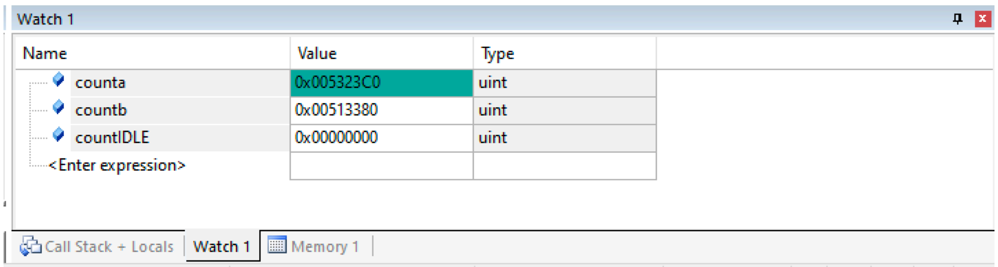
Item	Value
Tick Timer:	10.000 mSec
Round Robin Timeout:	200.000 mSec
Default Thread Stack Size:	200
Thread Stack Overflow Check:	Yes
Thread Usage:	Available: 7, Used: 4 + o...

ID	Name	Priority	State	Delay	Event Value	Event Mask	Stack Usage
1	osTimerThread	High	Wait_MBX				32%
2	main	Normal	Wait_DLY				36%
3	Thread1	Normal	Ready				32%
4	Thread2	Normal	Running				0%
255	os_idle_demon	None	Ready				32%

Fig5: System and Thread Viewer shows state of each thread being executed.

19) Exit debug mode. Under the RTX_Config_CM.c” file (Select “Project:Multitasking->Target 1->CMSIS->RTX_Config_CM.c” under project tree hierarchy) modify the text as follows:
   a. Under library declaration add global variable “unsigned int countIDLE =0”.
   b. Under the “os_idle_demon” function add “countIDLE++” in the FOR loop.

20) Re-compile and enter Debug mode. Add variable “countIDLE” to “Watch1” in accordance with Step 13. Run the simulation. The following should be observed in the Watch window (Fig6).



Name	Value	Type
counta	0x005323C0	uint
countb	0x00513380	uint
countIDLE	0x00000000	uint
<Enter expression>		

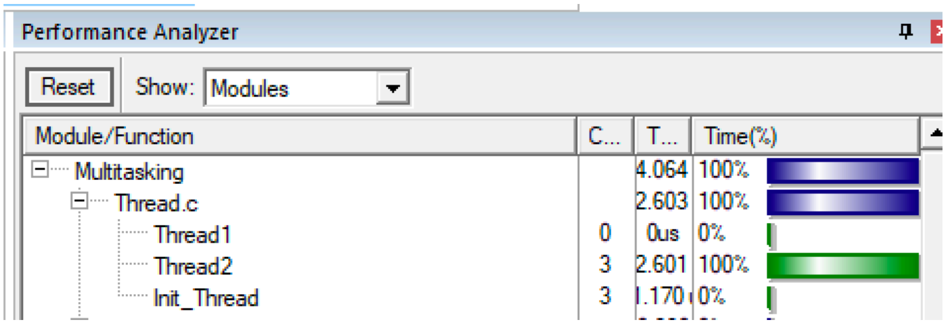
Fig 6: Watch window with “countIDLE” variable added.

NOTE: “countIDLE” does not increment. This is because Thread1 and Thread2 responsible for the incrementation of “counta” and “countb” have not finished their tasks (Never will—infinite loop) and since the “os_idle_demon” function is a lower priority it never gets executed.

21) Exit Debug mode. Change the priority of Thread2 by making the following modification to the “Thread.c” file:

Change “osTreadDef(Thread2, osPriorityNormal,1,0)” to “osTreadDef(Thread2, osPriorityAboveNormal,1,0)

Re-compile all files, enter Debug mode and re-run simulation.NOTE: Since Thread2 has higher priority than Thread1 and since both threads execute infinitely, Thread 2 executes first and hold execution until completion. This can be observed in the Performance Analyzer window with 100% CPU utilization on Thread2. (Fig7)

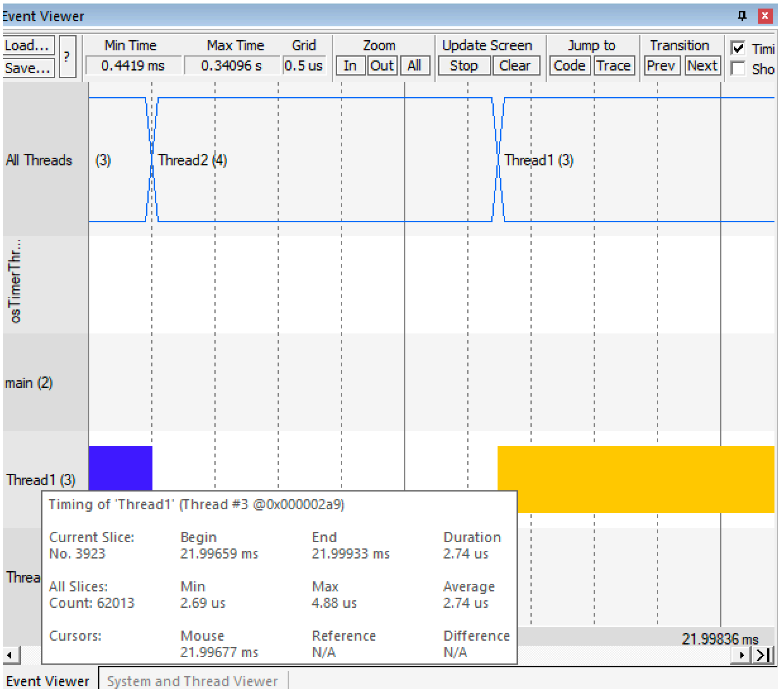


Show: Modules

Module/Function	C...	T...	Time(%)
Multitasking		4.064	100%
Thread.c		2.603	100%
Thread1	0	0us	0%
Thread2	3	2.601	100%
Init_Thread	3	1.170	0%

Fig 7: Comparison between Thread2-> Above normal priority and Thread1-> Normal Priority

22) Exit Debug mode and set all Thread priorities back to “Normal” in accordance with previous step. In “Thread.c” modify the Thread1 function and the Thread2 function after their respective “count” statements with the following “osThreadYield();” command. Re-compile all files, enter Debug mode and run simulation. NOTE: It can be observed in the “Event Viewer” window that the thread execution time has decreased significantly. The thread scheduler will now switch threads being executed as soon as the “osThreadYield()”function is called (Fig 8). Processor utilization is still shared evenly across the two threads.



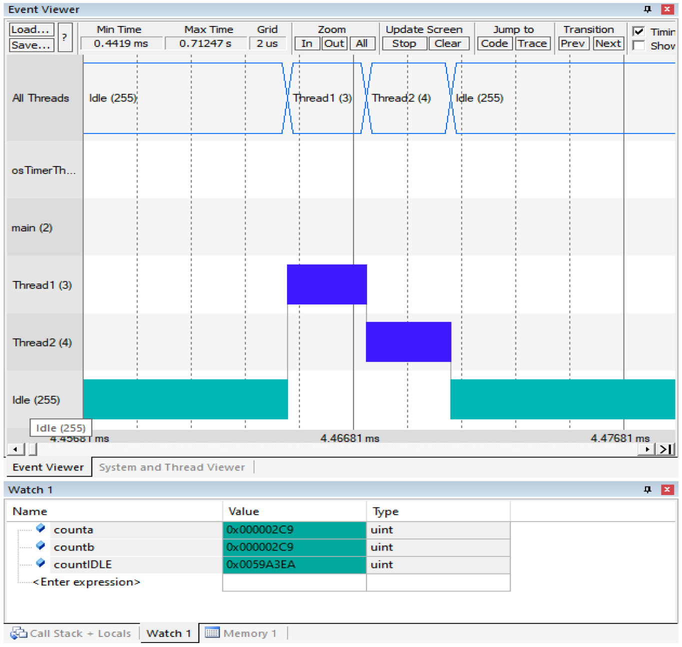
Min Time 0.4419 ms
Max Time 0.34096 s
Grid 0.5 us

Timing of 'Thread1' (Thread #3 @0x000002a9)

Current Slice: No. 3923	Begin 21.99659 ms	End 21.99933 ms	Duration 2.74 us
All Slices: Count: 62013	Min 2.69 us	Max 4.88 us	Average 2.74 us
Cursors:	Mouse 21.99677 ms	Reference N/A	Difference N/A

Fig 8: Program execution with “osThreadYield()” function in Thread1 and Thread2 FOR loops.

23) Exit Debug mode, and modify "Thread.c" by replacing the "osThreadYield()" functions from the previous step with the "osDelay(1)" and "osDelay(2)" functions for Thread1 and Thread2 respectively. Re-compile all files, enter Debug mode and run simulation. Note: Observing the "Event Viewer" and "Watch 1" windows it can be seen that each time the "osDelay()" function is executed the "os_idle_demon()" function counter is incremented. Processor unitization is no longer shared mutually between Threads but is now also shared with an "idle" period where neither Thread is executing a task (Fig9).



Min Time 0.4419 ms
Max Time 0.71247 s
Grid 2 us

Name	Value	Type
counta	0x000002C9	uint
countb	0x000002C9	uint
countIDLE	0x0059A3EA	uint
<Enter expression>		

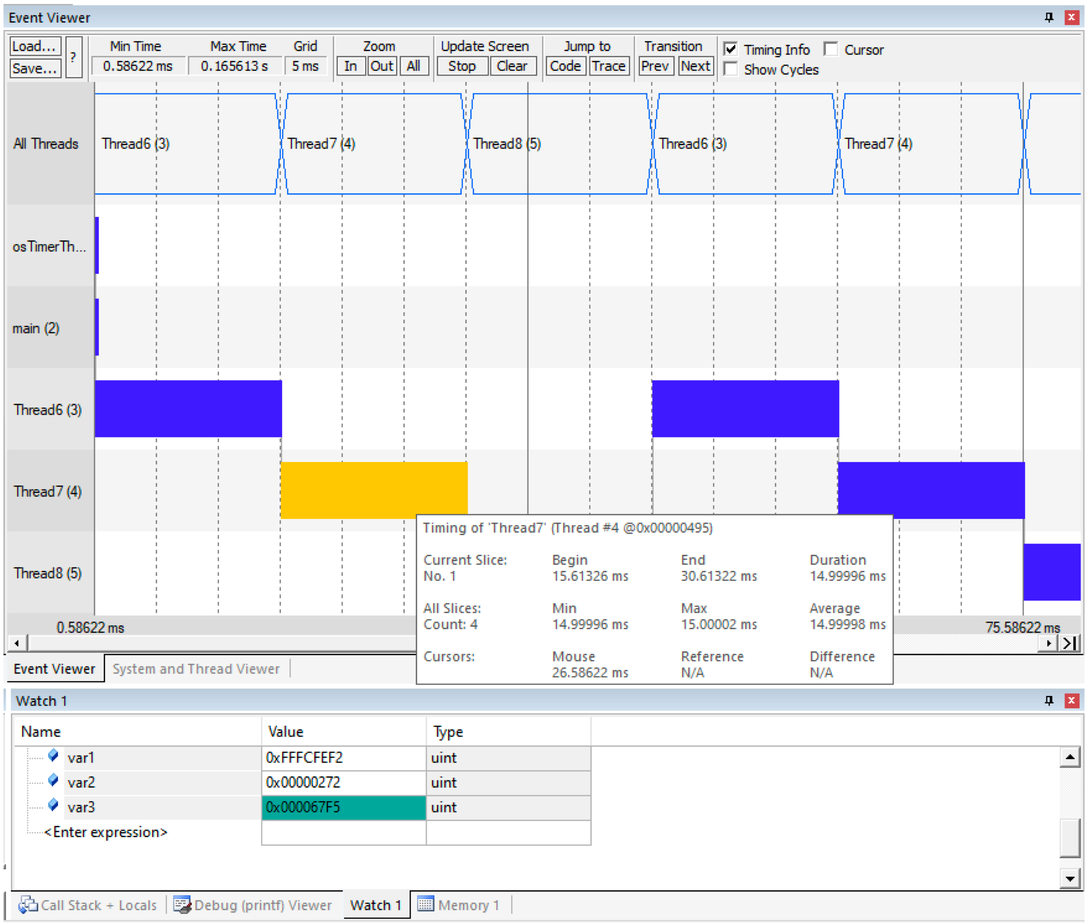
Fig 9: Program execution with osDelay() function

Conclusion:

PART1:

"Section1Lab3.txt" features code for Part1 of lab2. The three threads execute in a round-robin arrangement as per the lab manual requirements. Thread6 sums alternating positive and negative signed values counting from 1 to 75000. This is considered function one where the demo will output to LED P2.2 and the "Watch 1" window will output the value of the summation function to "var1" during execution. Thread7 sums even numbers from 1 to 75000. This is considered function two where the demo will output to LED (P2.2, P2.3) and the "Watch 1" window will output the value of the summation function to "var2" during execution. Thread8 sums random numbers between 1 and 100 for 75000 repetitions. This is considered function three where the demo will output to LED (P2.2, P2.3, P2.4) and the "Watch 1" window will output the value of the summation function to "var3" during execution. It can be observed in the "Event Viewer" the three threads switching sequentially after 15ms as per the lab manual requirements in a Round-Robin pattern (Fig12). In the "Watch 1" window the variables var1, var2 and var3 associated with each function output the summation value of each function (Fig

10). The processor utilization is shared relatively evenly across each thread until completion. However, there is deviation from equal processor utilization between threads because the functions being executed are not identical and since the tasks are finite each function is executed in slightly different time skewing the final results (Fig 11).

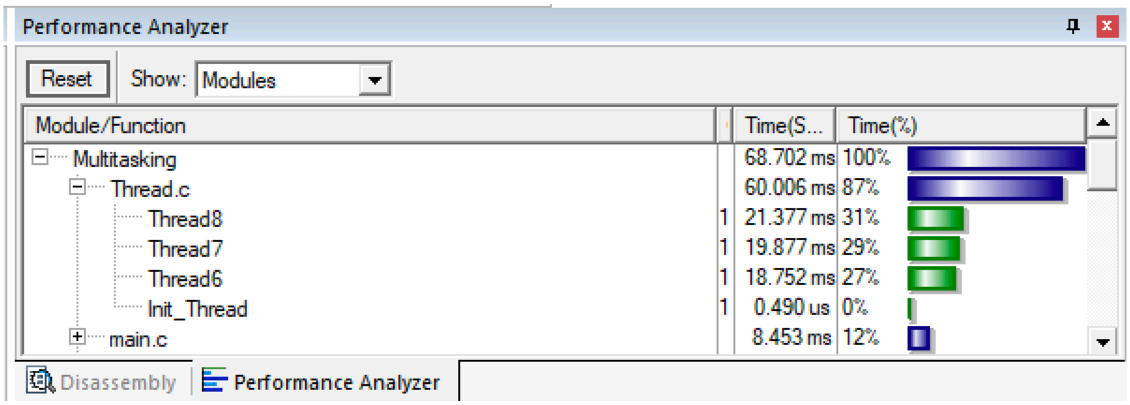


Timing of 'Thread7' (Thread #4 @0x00000495)

Current Slice: No. 1	Begin 15.61326 ms	End 30.61322 ms	Duration 14.99996 ms
All Slices: Count: 4	Min 14.99996 ms	Max 15.00002 ms	Average 14.99998 ms
Cursors:	Mouse 26.58622 ms	Reference N/A	Difference N/A

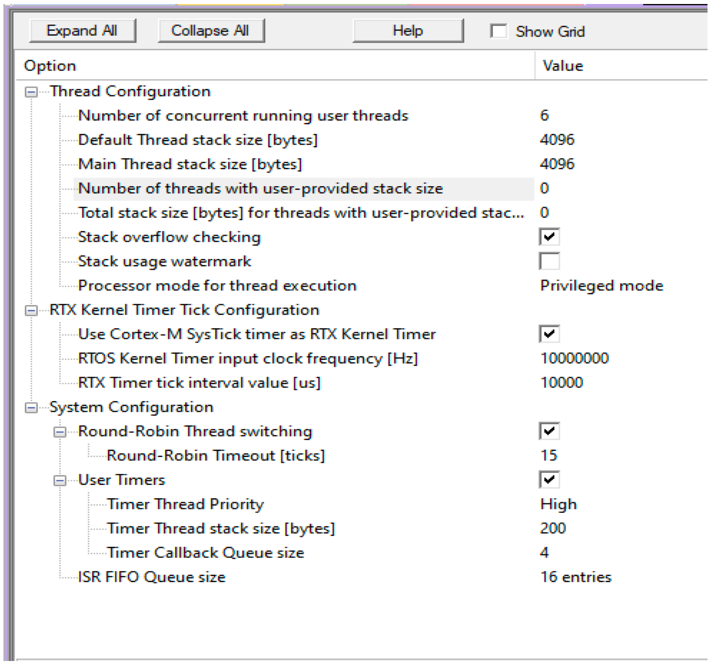
Name	Value	Type
var1	0xFFFCFEF2	uint
var2	0x00000272	uint
var3	0x000067F5	uint
<Enter expression>		

Fig 10: Three Function Round-Robin



Module/Function		Time(S...	Time(%)
Multitasking		68.702 ms	100%
Thread.c		60.006 ms	87%
Thread8	1	21.377 ms	31%
Thread7	1	19.877 ms	29%
Thread6	1	18.752 ms	27%
Init_Thread	1	0.490 us	0%
main.c		8.453 ms	12%

Fig 11: Round-Robin Processor utilization.



Option	Value
⊟ Thread Configuration	
Number of concurrent running user threads	6
Default Thread stack size [bytes]	4096
Main Thread stack size [bytes]	4096
Number of threads with user-provided stack size	0
Total stack size [bytes] for threads with user-provided stac...	0
Stack overflow checking	☑
Stack usage watermark	□
Processor mode for thread execution	Privileged 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]	10000
⊟ System Configuration	
⊟ Round-Robin Thread switching	☑
Round-Robin Timeout [ticks]	15
⊟ User Timers	☑
Timer Thread Priority	High
Timer Thread stack size [bytes]	200
Timer Callback Queue size	4
ISR FIFO Queue size	16 entries

Fig 12: Round-Robin Timeout [ticks] set to 15ms

Part2:

“Section2Lab3.txt” features code for PART2 of the lab2. The five functions, A through E are represented by five threads titled Threads1 through Thread5 as per the lab manual. There are two additional functions which are titled “factorial()” to determine the factorial of a number and “myexponentialcalc()” to determine the exponential calculation of a number. These functions supplement the execution of the general functions which comprise each thread. It can be observed from (Fig. 12) that the threads execute in accordance to their priority with function C (Thread3) executing first with above normal priority, follow by function A (Thread1) and function D (Thread4) with normal priority and last function B (Thread2) and function E (Thread5) with below normal priority (Fig13). If two threads have the same priority, their sequential ID determines their execution order by the processor. The idle time parameter can be attributed to the supplemental “myexponentialcalc()” and “factorial()” function calls, and consequently the execution time is considered separate of the thread. The execution times for each function are listed in (Table 10) with their respective outputs in (Fig. 14). In alphabetical order, function A took the most amount of time and function D took the least amount of time to execute. Please note that the processor utilization of each thread in PART2 was unable to be determined due to the rapid execute of the five threads.

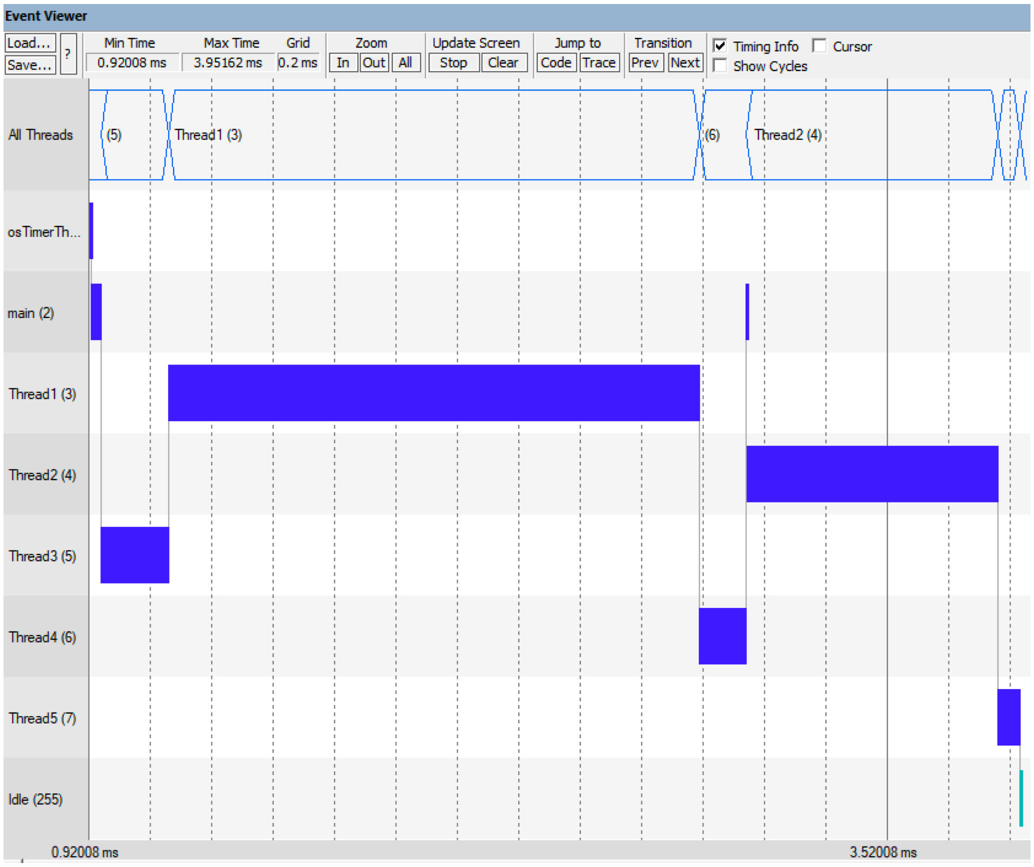
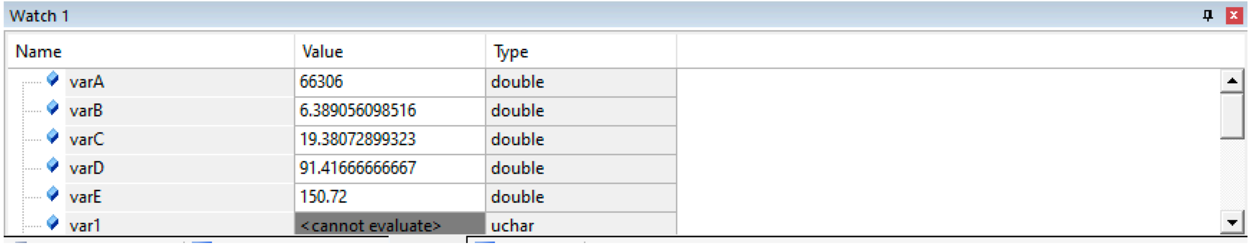


Fig 13: Five pre-emptive function execution order

Function	A	B	C	D	E
Threads	1	2	3	4	5
Time(ms)	1.723	0.815	0.222	0.154	0.074

Table 1: Execution times


Watch 1

Name	Value	Type
varA	66306	double
varB	6.389056098516	double
varC	19.38072899323	double
varD	91.41666666667	double
varE	150.72	double
var1	<cannot evaluate>	uchar

Fig 14: Function A to E output

References:

1) Systick Timer, https://www.sciencedirect.com/topics/engineering/systick-timer, 2020

2) Keil CMSIS-Core, https://www.keil.com/pack/doc/CMSIS/Core/html/index.html , 2020