Brandon Ho 500727531

Lab4: Real-Time Scheduling

Introduction:

This lab introduces several key functions in the RTOS environment that allows management of thread execution. The function osTimerCreate(), osTimerStart(), osTimerStop() and OsTimerDef, allow the creations, the starting, the stopping and the execution of commands based on a "Virtual Timer". The "Inter-thread communication signal" functions osSignalWait() and osSignalSet() functions allow thread execution to be synchronized (Running/Waiting status manipulation). Together, the "Virtual Timer" and the "Inter-thread Communication Signal" commands will be used to implement an RMS algorithm that executes threads based on shortest computation time.

The semaphoreWait() and semaphoreRelease() command are used to enforce "mutual exclusion" on critical sections of code, where in this lab only one thread can execute a critical section of code at any given moment. These functions will be used to correct priority inversion error. This is a situation where a higher priority thread is indirectly pre-empted by a lower priority thread, inverting the executing order of the two processes.

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 "virtual_demo" 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.

   i) CMISI->CORE
   ii) CMSIS->RTOS(API)->Keil RTX c.
   iii) Device ->Startup
   iv) Device -> GPIO
   v) Device -> PIN
   vi) Board Support -> LED(API)->LED
   vii) Compiler -> Even recorder

5) Select "Add New Item to Group 'Source Group 1'...", via right click options under Project: virtual_demo -> Target Group 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:

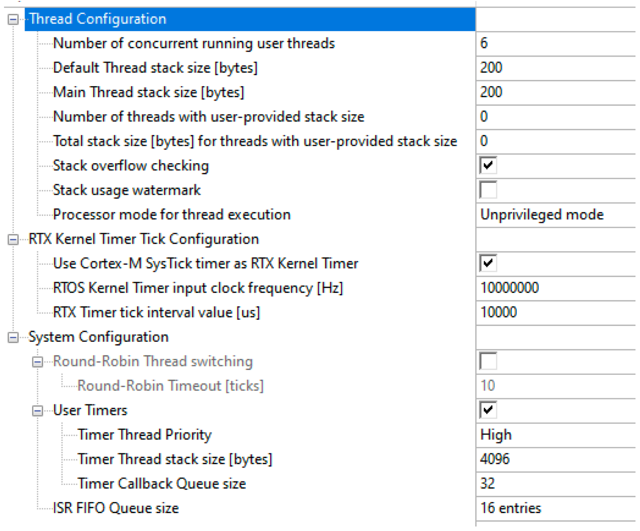
1. Under Target folder tab:
   i) Select the “Use default complier Version 5”.
   ii) Select “Use MicroLIB”.
   iii) Select “IRAM1” and “IRAM2”.
2. Under Debug tab:

   i) In the left Dialog DLL textbox insert “DARMP1.DLL” with parameter “-pLPC1768”.
   ii) In the right Dialog DLL textbox insert “TARMP1.DLL” with parameter “-pLPC1768”.
   iii) Select “Use Simulator”.
3. Under C/C++ tab:
   i) Select C99 mode.

9) Add provided “virtual_demo.c file” to project. This is done as follows:

i) Under “Project:virtual_demo ->Target 1-> Source Group 1” under the project tree hierarchy. Right click on “Source Group 1”.
ii) Select “Add Existing Files to Group…” and add “virtual_demo.c” file to project.

10) Select “Project:virtual_demo ->Target 1->CMSIS->RTX_Config_CM.c “ in the project tree hierarchy by “double left clicking” and toggle the “Configuration Wizard”. Select Expand all button. Ensure the following settings are set according to (Fig 1)



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]	10000
System Configuration	
Round-Robin Thread switching	☐
Round-Robin Timeout [ticks]	10
User Timers	✓
Timer Thread Priority	High
Timer Thread stack size [bytes]	4096
Timer Callback Queue size	32
ISR FIFO Queue size	16 entries

Fig 1: “RTX_Config_CM.c” file settings

11) Under "Project:virtual_demo ->Target 1-> Source Group 1-> virtual_demo.c , comment out all "osDelay()" function calls. Compile and enter Debug mode.

12) Enable "Event Viewer" through the menu bar "Debug->OS support->Event Viewer".

13) Enable "System and Thread Viewer" through the menu bar "Debug->OS support-> System and Thread Viewer".

14) Enable "GPIO Fast Interface 1" through the menu bar "Peripherals -> GPIO Fast Interface -> PORT 1".

15) Enable "GPIO Fast Interface 2" through the menu bar "Peripherals -> GPIO Fast Interface -> PORT 2".

16) Simulate project. Analyze the functionality of the code. See (Fig 2), (Fig 3) and (Fig 4).

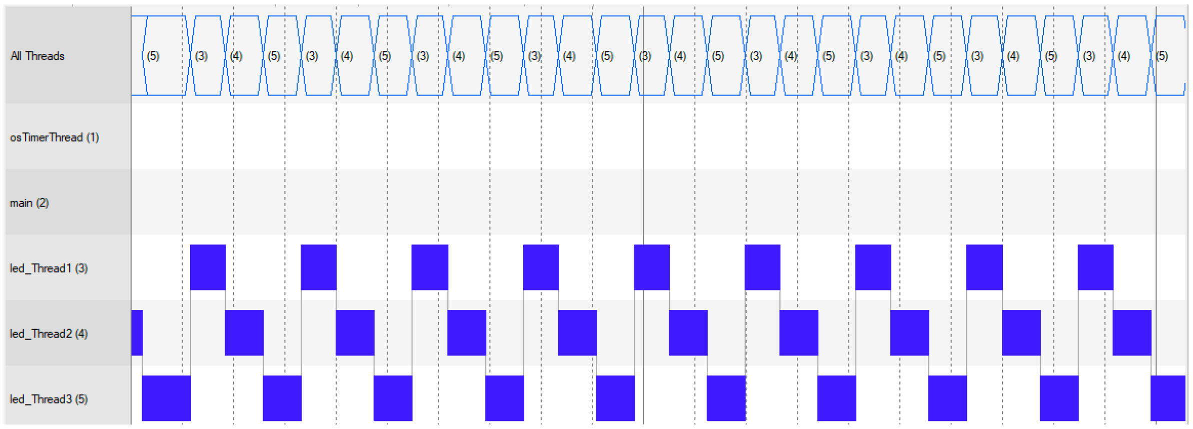
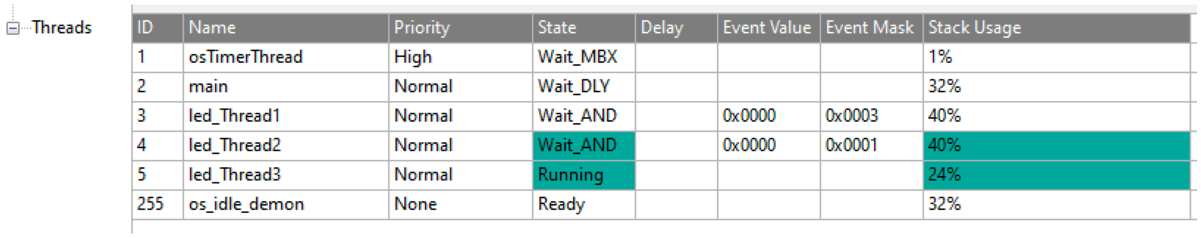
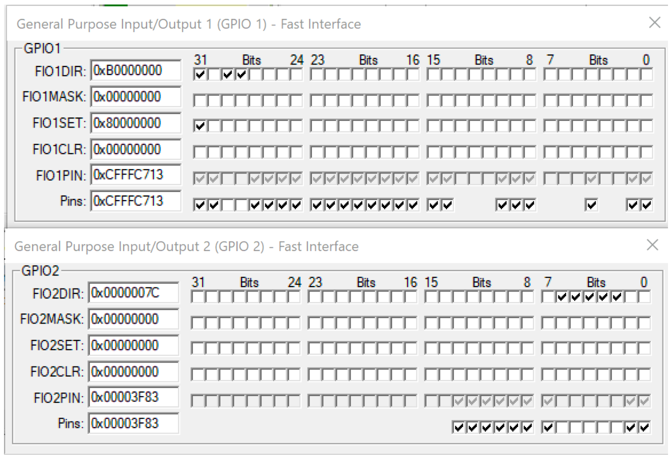


Fig2: "virtual _demo" System and Thread Viewer example


Threads

ID	Name	Priority	State	Delay	Event Value	Event Mask	Stack Usage
1	osTimerThread	High	Wait_MBX				1%
2	main	Normal	Wait_DLY				32%
3	led_Thread1	Normal	Wait_AND		0x0000	0x0003	40%
4	led_Thread2	Normal	Wait_AND		0x0000	0x0001	40%
5	led_Thread3	Normal	Running				24%
255	os_idle_demon	None	Ready				32%

Fig3: "virtual_demo" System and Thread Viewer example.


General Purpose Input/Output 1 (GPIO 1) - Fast Interface

GPIO1

FIO1DIR:	0xB0000000
FIO1MASK:	0x00000000
FIO1SET:	0x80000000
FIO1CLR:	0x00000000
FIO1PIN:	0xCFFFC713
Pins:	0xCFFFC713

General Purpose Input/Output 2 (GPIO 2) - Fast Interface

GPIO2

FIO2DIR:	0x0000007C
FIO2MASK:	0x00000000
FIO2SET:	0x00000000
FIO2CLR:	0x00000000
FIO2PIN:	0x00003F83
Pins:	0x00003F83

Fig4: “virtual_demo” GPIO output example (LED)

17) Under “Project:virtual_demo ->Target 1-> Source Group 1-> virtual_demo.c , right click on “virtual_demo.c” and select “Remove File ‘virtual_demo.c’ ”.

18) Repeat Step 9, except add “priority_inv.c” instead. Compile enter Debug mode and run simulation. Observe (Fig 5) and (Fig 6). NOTE: Inversion error.

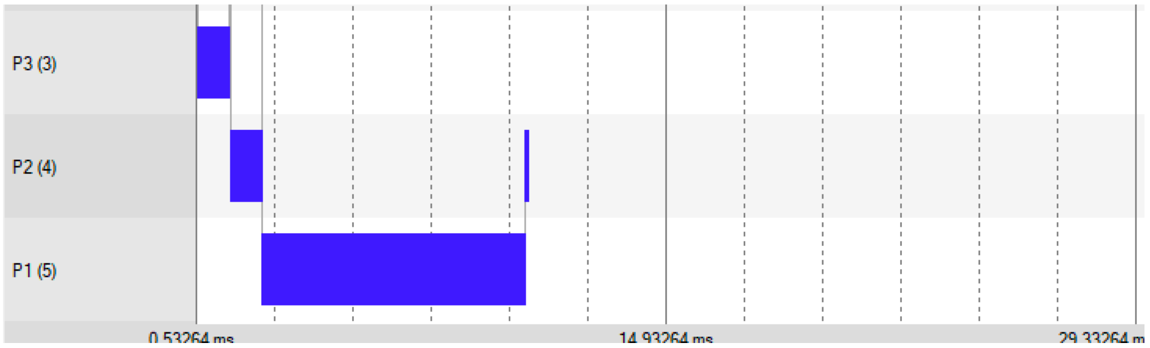
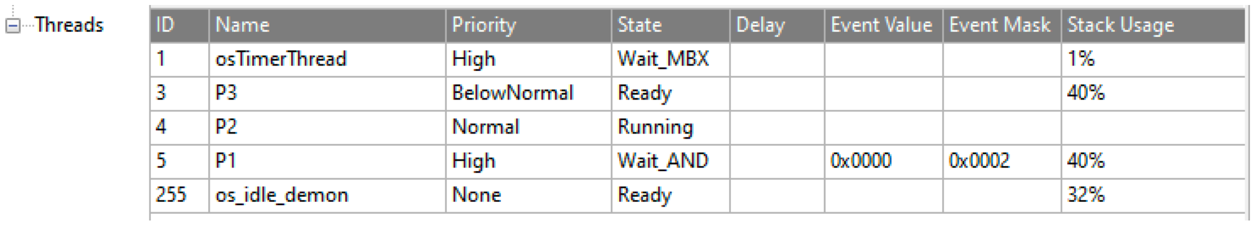


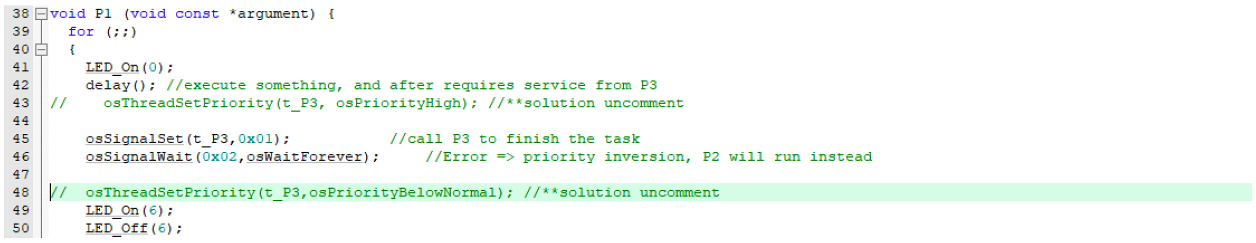
Fig 5: Inversion Example


Threads

ID	Name	Priority	State	Delay	Event Value	Event Mask	Stack Usage
1	osTimerThread	High	Wait_MBX				1%
3	P3	BelowNormal	Ready				40%
4	P2	Normal	Running				
5	P1	High	Wait_AND		0x0000	0x0002	40%
255	os_idle_demon	None	Ready				32%

Fig6: Thread ID = 5 cannot switch to Thread Id = 3. (Shared resources/Thread ID = 4 is blocking)

19) Exit Debug mode. Under “Project:virtual_demo -> Target 1-> Source Group 1-> priority_inv.c, fix the priority inversion error by dynamically changing the thread priority. This is done by uncommenting the code on lines 43 and 48. See (fig7).



```
38 void P1 (void const *argument) {
39   for (;;)
40   {
41     LED_On(0);
42     delay(); //execute something, and after requires service from P3
43 //    osThreadSetPriority(t_P3, osPriorityHigh); //**solution uncomment
44
45     osSignalSet(t_P3,0x01);           //call P3 to finish the task
46     osSignalWait(0x02,osWaitForever);     //Error => priority inversion, P2 will run instead
47
48 //  osThreadSetPriority(t_P3,osPriorityBelowNormal); //**solution uncomment
49     LED_On(6);
50     LED_Off(6);
```

Fig7: Uncomment lines 43 and 48 to dynamically assign thread priority and fix priority inversion error.

20) Compile and enter Debug mode. Simulate modified code. See (Fig 8) and (Fig 9).

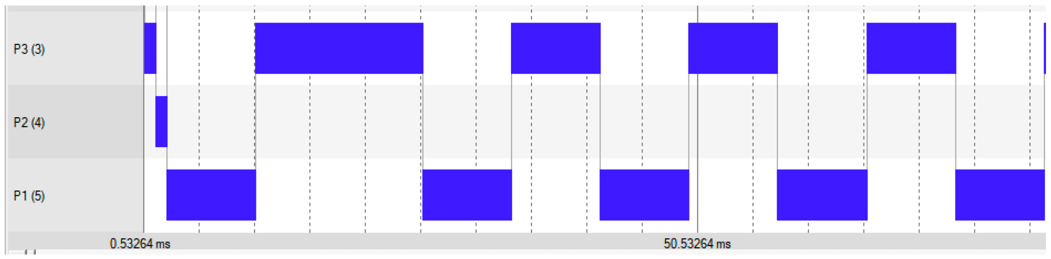
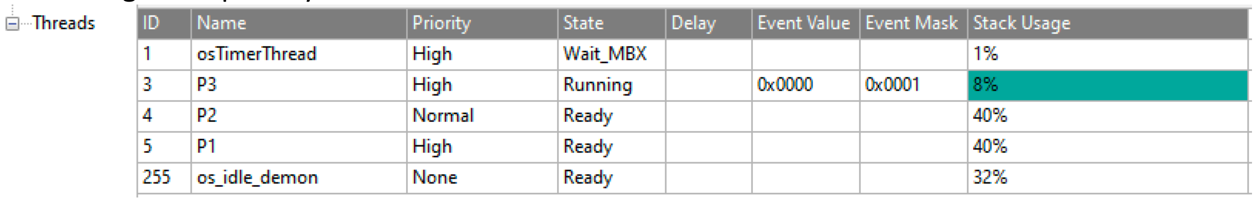


Fig 8: No priority inversion error.



Threads

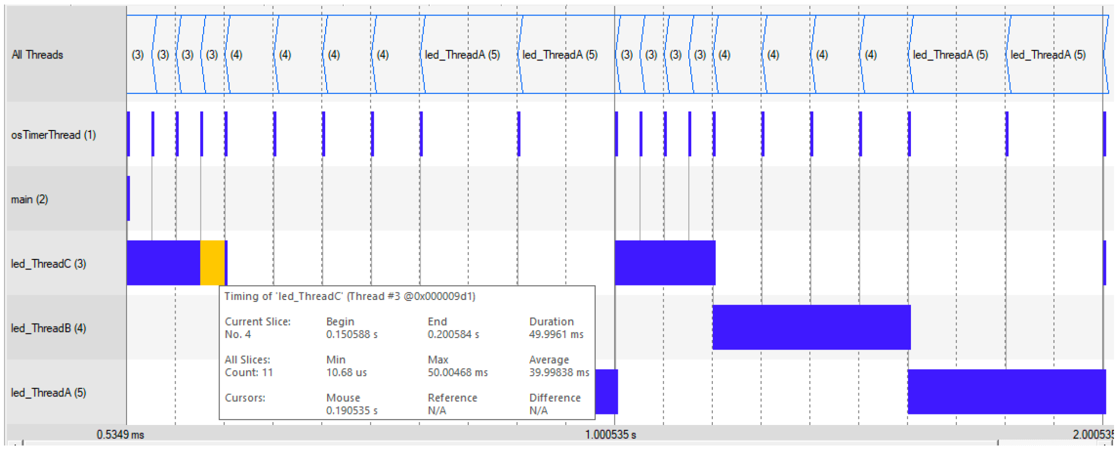
ID	Name	Priority	State	Delay	Event Value	Event Mask	Stack Usage
1	osTimerThread	High	Wait_MBX				1%
3	P3	High	Running		0x0000	0x0001	8%
4	P2	Normal	Ready				40%
5	P1	High	Ready				40%
255	os_idle_demon	None	Ready				32%

Fig 9: No blocked threads due to priority inversion. (NOTE: no threads with Wait_AND status).

Conclusion:

Part1: RMS algorithm

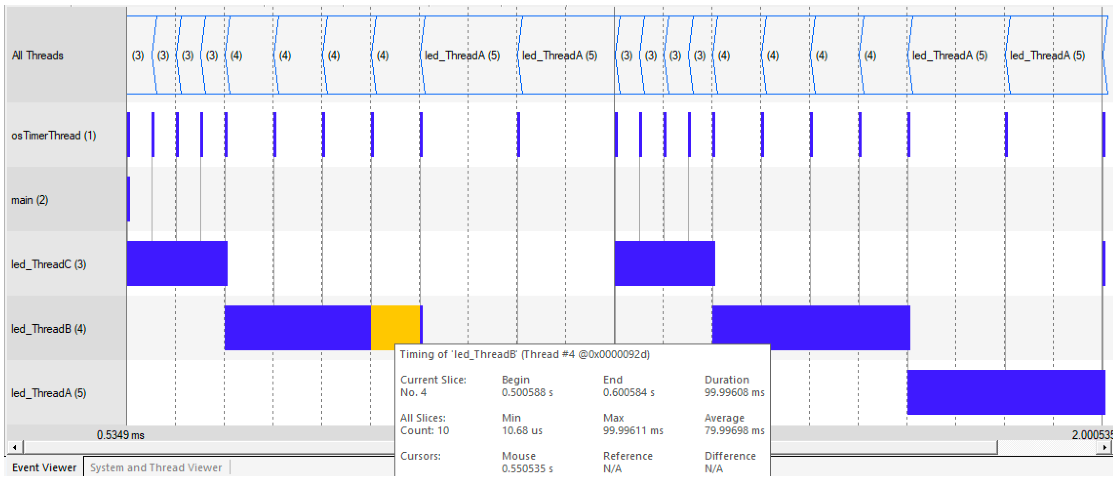
The following "Rate Monotonic Scheduler" executes in accordance with the parameters specified in the lab manual by executing the threads from shortest to longest in computation time. Additionally, each thread is executed for the duration of its computational time and is repeatedly executed for the duration of the specified period of each thread. For instance, thread C has the shortest computation time of 5000us and consequently thread C is the first thread to execute. Thread C repeatedly executes in 5000us duration thread periods for a total overall period of 20000us (See Fig 10) before switching to the next shortest computation time thread (Thread B). Thread B has a computational time of 10000us; therefore, this thread has a 10000us duration thread period and executes consecutively for a period of 40000us (See fig10). Thread A has the largest computation period of 20000us and executes last. Thread A executes in 20000us duration thread periods and repeatedly executes for a total overall period of 40000us (See fig10). The following algorithm repeats indefinitely.



Timing of 'led_ThreadC' (Thread #3 @0x000009d1)

```
Current Slice:   Begin        End          Duration
No. 4            0.150588 s   0.200584 s   49.9961 ms

All Slices:      Min          Max          Average
Count: 11        10.68 us     50.00468 ms  39.99838 ms

Cursors:         Mouse        Reference    Difference
                 0.190535 s   N/A          N/A
```

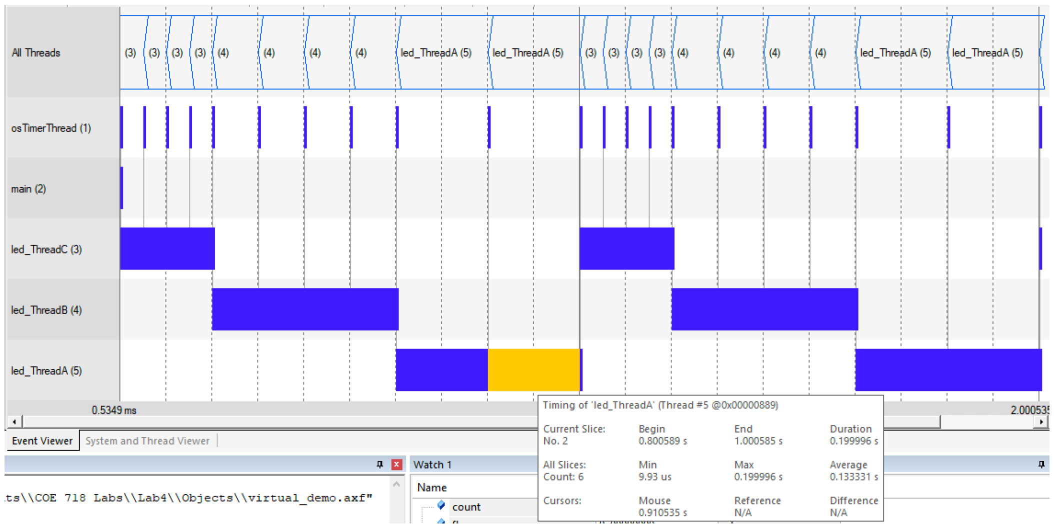
Fig 10: Thread C executes in 50ms thread duration periods and repeats consecutive execution 4 times for a total 200ms duration period of execution.



Timing of 'led_ThreadB' (Thread #4 @0x0000092d)

```
Current Slice:   Begin        End          Duration
No. 4            0.500588 s   0.600584 s   99.99608 ms

All Slices:      Min          Max          Average
Count: 10        10.68 us     99.99611 ms  79.99698 ms

Cursors:         Mouse        Reference    Difference
                 0.550535 s   N/A          N/A
```

Fig 11: Thread B executes in 100ms thread duration periods and repeats consecutive execution 4 times for a total 400ms duration period of execution.



```
ts\\COE 718 Labs\\Lab4\\Objects\\virtual_demo.axf"
```

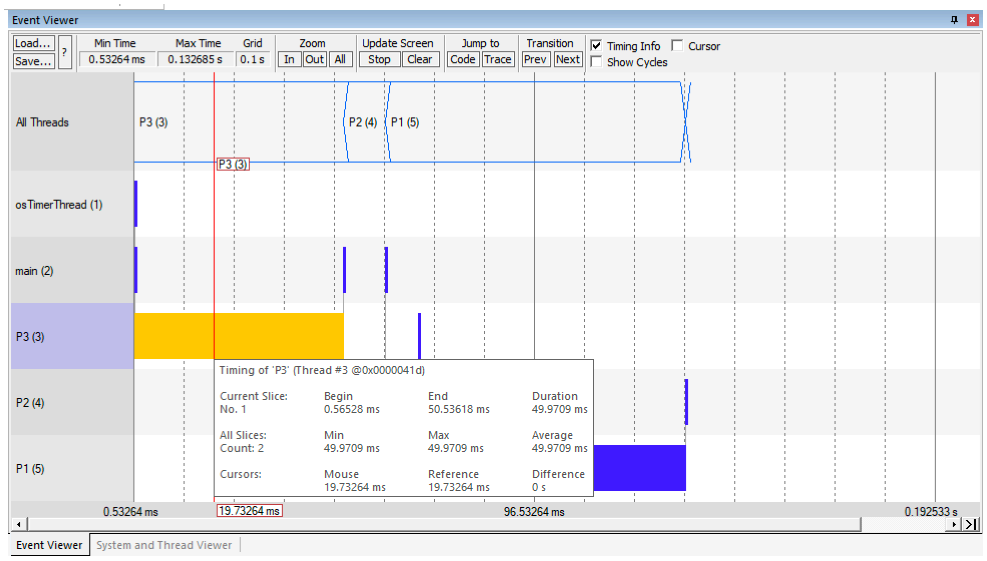
Timing of 'led_ThreadA' (Thread #5 @0x00000889)

```
Current Slice:   Begin        End          Duration
No. 2            0.800589 s   1.000585 s   0.199996 s

All Slices:      Min          Max          Average
Count: 6         9.93 us      0.199996 s   0.133331 s

Cursors:         Mouse        Reference    Difference
                 0.910535 s   N/A          N/A
```

Fig 12: Thread A executes in 200ms thread duration periods and repeats consecutive execution 2 times for a total 400ms duration period of execution

Part2: Priority Inversion. (SEE Fig 1 for RTX_Config_CM.C screenshot)

The provided code "priority_inv.c" was modified into "Part2Lab4.c" with the specified time parameters given in the lab manual. This was done by calling additional delay functions in "Thread P3" and increasing the delay between thread creation in the "main" portion of the "Part2Lab3.c" file. Fig 11, Fig 12 and Fig 13 represent the initial priority inversion error that needs to be corrected.

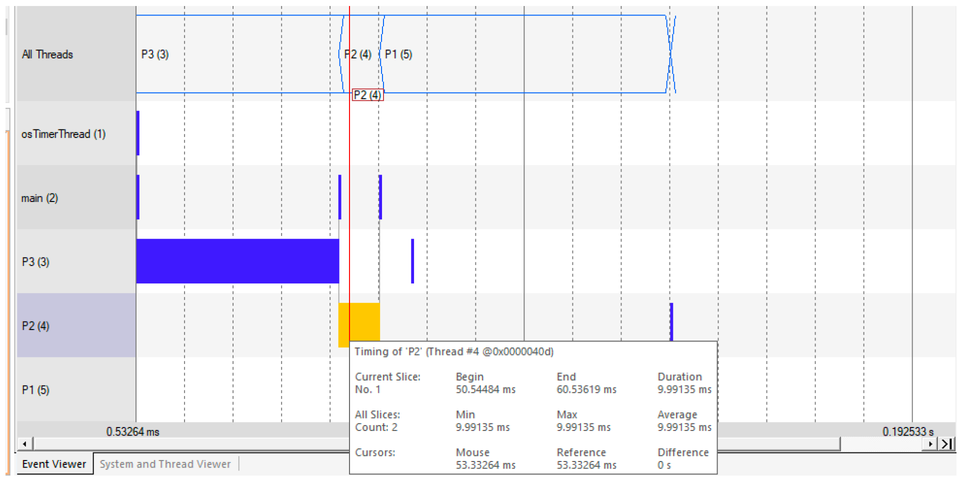


Min Time 0.53264 ms | Max Time 0.132685 s | Grid 0.1 s

Timing of 'P3' (Thread #3 @0x0000041d)

Current Slice: No. 1	Begin 0.56528 ms	End 50.53618 ms	Duration 49.9709 ms
All Slices: Count: 2	Min 49.9709 ms	Max 49.9709 ms	Average 49.9709 ms
Cursors:	Mouse 19.73264 ms	Reference 19.73264 ms	Difference 0 s

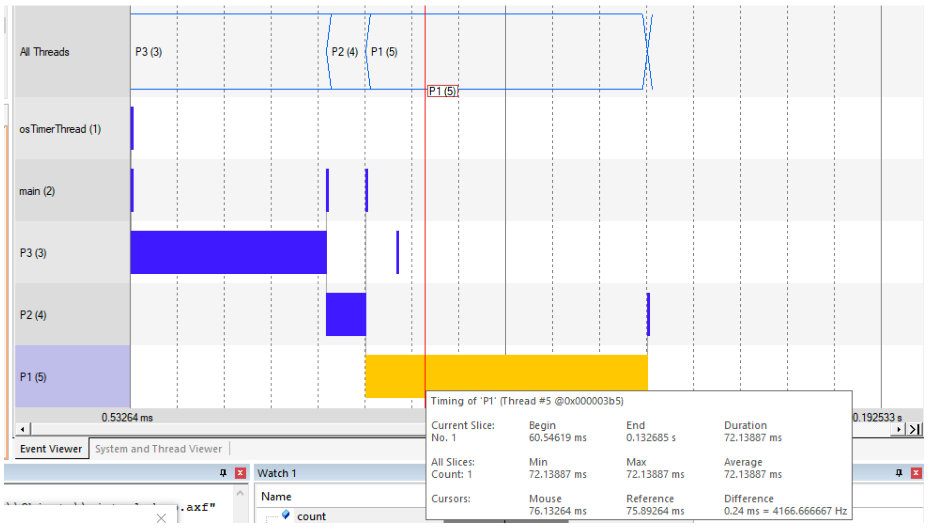
Fig13: Duration of Thread 3(50ms) up to context switch at t = 50ms.



Timing of 'P2' (Thread #4 @0x0000040d)

Current Slice: No. 1	Begin 50.54484 ms	End 60.53619 ms	Duration 9.99135 ms
All Slices: Count: 2	Min 9.99135 ms	Max 9.99135 ms	Average 9.99135 ms
Cursors:	Mouse 53.33264 ms	Reference 53.33264 ms	Difference 0 s

Fig14: Duration of Thread 2 (10ms), between context switch of Thread3(50ms) to Thread2(60ms).



```
Timing of 'P1' (Thread #5 @0x000003b5)

Current Slice:   Begin           End             Duration
No. 1            60.54619 ms     0.132685 s      72.13887 ms

All Slices:      Min             Max             Average
Count: 1         72.13887 ms     72.13887 ms     72.13887 ms

Cursors:         Mouse           Reference       Difference
                 76.13264 ms     75.89264 ms     0.24 ms = 4166.666667 Hz
```

Fig15: Duration of Thread 1(72ms) before blocking from Thread2.

It should be noted that the threads execute as follow: first thread P3, second thread P2 and third thread P1. The problem of priority inversion arises in Lab4 in the following two situations. First, when threads P3(high priority) and P1(low priority) share common resources (LED function calls) during their respective execution, thread P3 block thread P1 from executing when thread P1's higher priority stops thread P3 execution, but thread P1 still retains the resources. The second situation occurs when thread P1 requires input derived from thread P3 during execution and consequently must signal thread P1 to run intermittently. However, this thread switch operation is blocked by thread P2(medium priority), which has a higher priority than thread P3 and consequently executes instead. This forces thread P1 to wait indefinitely on the input of thread P3 since thread P3 execution is blocked by the operation of thread P2.
To fix this problem, a binary semaphore is used to protect the critical sections of thread P1 and thread P3 during operation. This is done by placing a osSemaphoreWait() and osSemaphoreRelease() in a manner that encapsulates the critical section of each thread. The osSemaphoreWait() command will only allow code to execute if a flag is not set (no other running critical sections). When a critical section is free to run, the semaphore flag is incremented for that particular thread blocking the execution of all other critical sections not currently executing. After the code in the critical section is run, the osSemaphoreRelease() command decrements the flag for that particular thread allowing other critical sections to run. Thread P2 has no signal-based thread switching in comparison to thread P3 and P1. Therefore the osSemaphoreWait() is written in conjunction with the osDelay() function to pass control to other tasks for a small period of time (700ms), and only check periodically for a free semaphore (period of time when considering execute of Thread P2 critical section). This minimizes the blocking of thread P2 on thread P1 and thread P3, while still allowing periodic execution of thread P2 (See Fig 16).

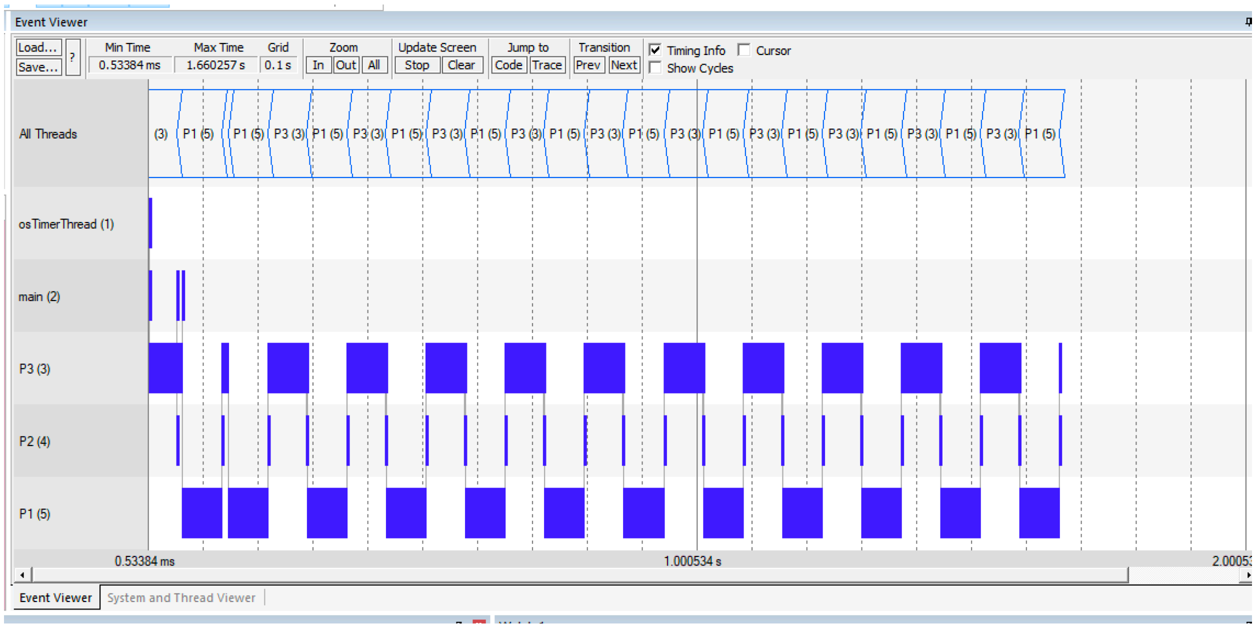


Fig16: Semaphore used to protect critical sections and prevent priority inversion.

References:

1) CMIS-RTOS API, https://www.keil.com/pack/doc/CMSIS/RTOS/html/group__CMSIS__RTOS.html, NOV 16, 2020.