Real Time Operating Systems Design and Programming

Lab 6

OS Debugging Lab

Tuning Performance

Contents

1 Overview

Tuning a program to a good, if not optimal, state is a time-consuming task. In this lab, you are expected to improve the performance of the provided project. You will first be introduced to how to evaluate the performance of the provided project with various debugging tools and techniques. By doing so, you will notice performance or, in some cases, even functional bugs.

2 The project Setup

The project is based on RTX and you should not modify the configuration, but make sure the timer clock value is correct for your board. It applies prioritised Round-Robin task switching scheduling with timeout of 50µs.

There will be three types of tasks:

The init task creates all other tasks, the dispatcher task and all other LED tasks, and then deletes itself.

The dispatcher task will dispatch all other LEDs based on the priorities assigned (LED_priority), by raising LED's RTX priority to 10.

The LED task will turn on an LED, do some computation (delay), turn off the LED, do some computation (delay), set its own priority to 1 and then pass the control to other tasks (dispatcher).

For a dispatch cycle, one LED task will run several times based on the priorities assigned if its LED_priority is 4, then it will be called 4 times in a cycle). All LED tasks will be called at least once during the cycle. However, different LED tasks take different time to finish, this reflects the preference of the program and is decided by LED_preference.

The program will stop when it finishes MAX_COUNT cycles (by default 100). So the target is to finish all cycles using least time by carefully choosing the priority strategy that meets the preference of the program.

The precise timing behaviour depends on the random number generator, which will be seeded by the ADC input. So please do not feel surprised by the slightly different results when you run the exact same program many times.

This setup is reasonably artificial and for demonstration purposes only.

3 Lab Procedure

1. Open the template project. Scan through the code. In particular the LED task and the dispatcher task.
2. Run the code and measure the time of LED activity with an oscilloscope. How long does it roughly take to finish everything? (From the first LED pulse to the last LED pulse.)

   Around 0.3 seconds.

3. The *end_time=os_time_get();* in the dispatcher task records the ending time in the unit of 10µs. You were introduced to the watch window and memory monitoring function in µVision debugger in previous labs. Recall that this is supported by the ARM CoreSight debugging technology (part of Cortex-M processors) and only global variables can be monitored in real time.

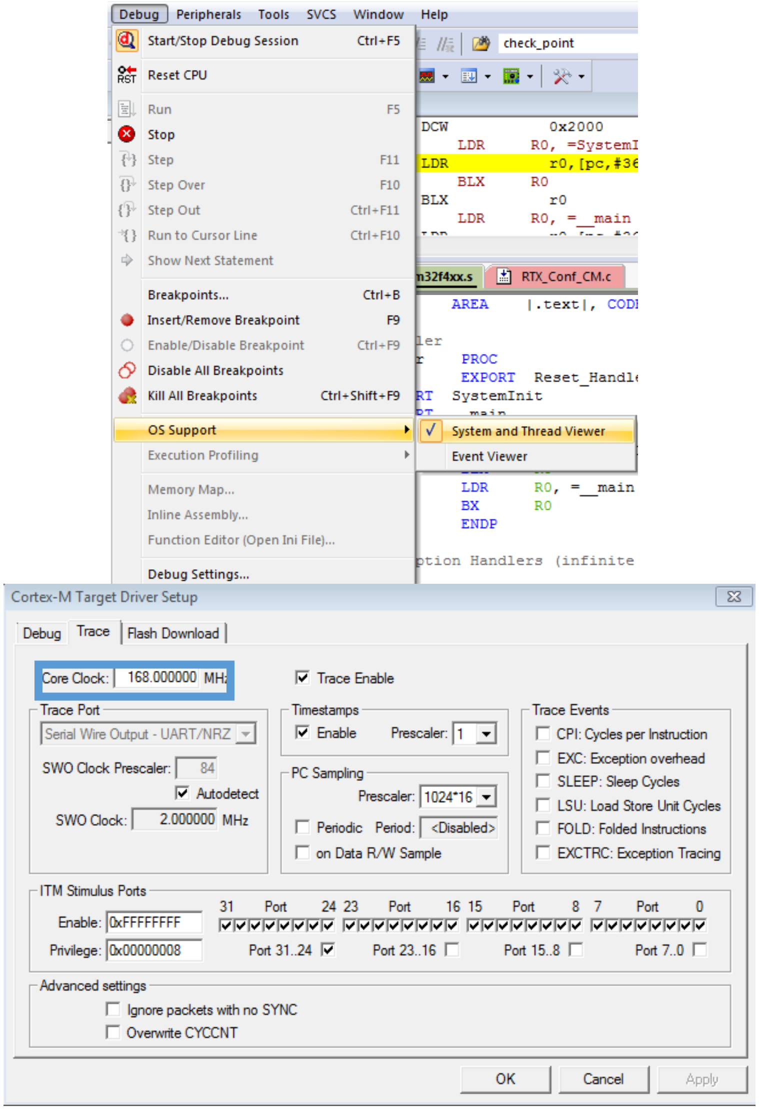
   Now run the program in debugger mode and watch the *end_time*. Does that correspond to the result of Q2?

   Yes, the value of end_time is around 30600, which means 0.3 seconds.

4. To what extent is this technique of measuring time intrusive?

   Not very much in terms of timing, but the program has to explicitly record the time:
   end_time=os_time_get();

5. µVision also supports RTX and provides OS-aware debugging tools. In debug mode, click on the debug menu and select the **System and Thread Viewer** in OS Support. However the window will not response during debugging as you needs to activate and configure the Serial Wire Viewer (SWV). This is actually similar to the logic analyser and we have introduced how to setup and configure the SWV in previous lab. It enables the debugger to trace the CPU activity in real-time through the serial wire.
   Configure the SWV if the project has not done this for you already. In particular, make sure that the Core clock is matched with your board.



```
DCW          0x2000
    LDR      R0, =SystemI
LDR          r0,[pc,#36
    BLX      R0
BLX          r0
    LDR      R0, =__main
    AREA     |.text|, CODE
     PROC
     EXPORT  Reset_Handle
     SystemInit
     LDR     R0, =__main
     BX      R0
     ENDP
ption Handlers (infinite
```

Cortex-M Target Driver Setup

Debug | Trace | Flash Download

Core Clock: 168.000000 MH

☑ Trace Enable

Trace Port: Serial Wire Output - UART/NRZ
SWO Clock Prescaler: 84
☑ Autodetect
SWO Clock: 2.000000 MHz

Timestamps: ☑ Enable Prescaler: 1

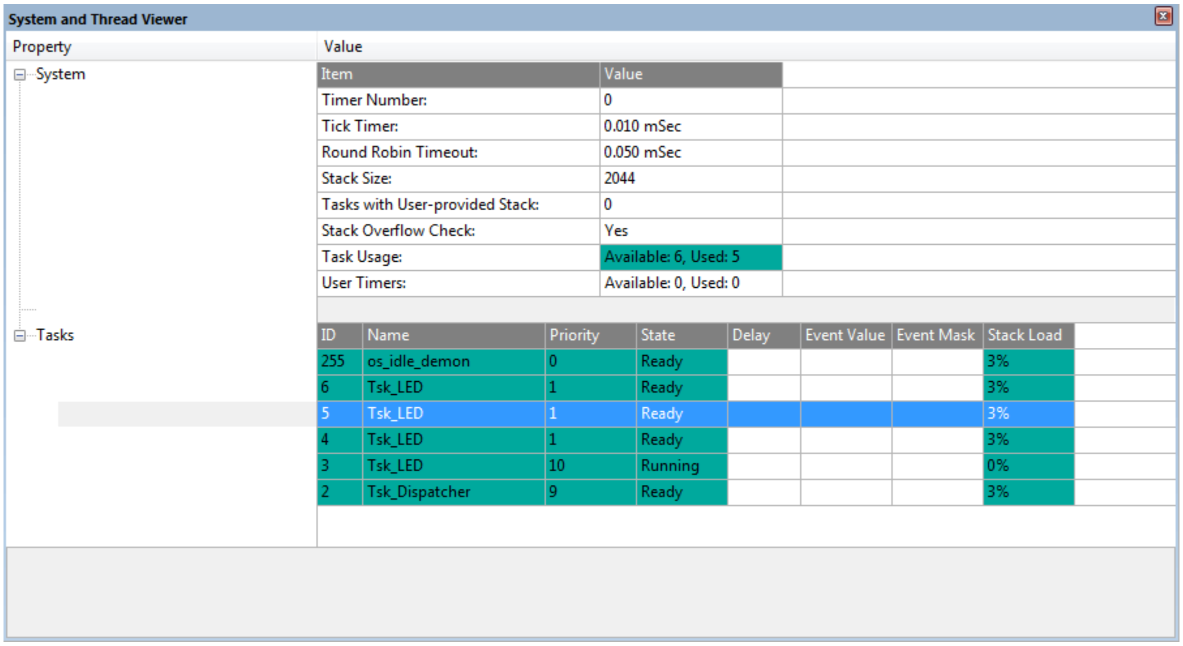
PC Sampling: Prescaler: 1024*16
□ Periodic Period: <Disabled>
□ on Data R/W Sample

Trace Events:
□ CPI: Cycles per Instruction
□ EXC: Exception overhead
□ SLEEP: Sleep Cycles
□ LSU: Load Store Unit Cycles
□ FOLD: Folded Instructions
□ EXCTRC: Exception Tracing

ITM Stimulus Ports
31 Port 24 23 Port 16 15 Port 8 7 Port 0
Enable: 0xFFFFFFFF ☑☑☑☑☑☑☑☑ ☑☑☑☑☑☑☑☑ ☑☑☑☑☑☑☑☑ ☑☑☑☑☑☑☑☑
Privilege: 0x00000008 Port 31..24 ☑ Port 23..16 □ Port 15..8 □ Port 7..0 □

Advanced settings
□ Ignore packets with no SYNC
□ Overwrite CYCCNT

And the System and Thread Viewer looks like this:



Item	Value
Timer Number:	0
Tick Timer:	0.010 mSec
Round Robin Timeout:	0.050 mSec
Stack Size:	2044
Tasks with User-provided Stack:	0
Stack Overflow Check:	Yes
Task Usage:	Available: 6, Used: 5
User Timers:	Available: 0, Used: 0

ID	Name	Priority	State	Delay	Event Value	Event Mask	Stack Load
255	os_idle_demon	0	Ready				3%
6	Tsk_LED	1	Ready				3%
5	Tsk_LED	1	Ready				3%
4	Tsk_LED	1	Ready				3%
3	Tsk_LED	10	Running				0%
2	Tsk_Dispatcher	9	Ready				3%

The content is self-evident and you can monitor the states of tasks in real time. Please notice that at any given time, there will be only one Running task.

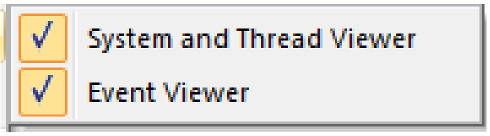
Try to give a simple example when the System and Thread Viewer can help in detecting deadlock situations.

Two or more tasks in WAIT_SEM while there is only the os_idle_demon always in Running state.

6. Now, try to repeat Q2 and Q3 with the System and Thread Viewer feature on, is it intrusive? Does the timing suffer from having this debug feature?

   The effect is not obvious. The timing is still around 0.3s. So you can just assume it is non-intrusive. RTX will store related information in a memory area that is accessible by the debugger so that this feature can be enabled.

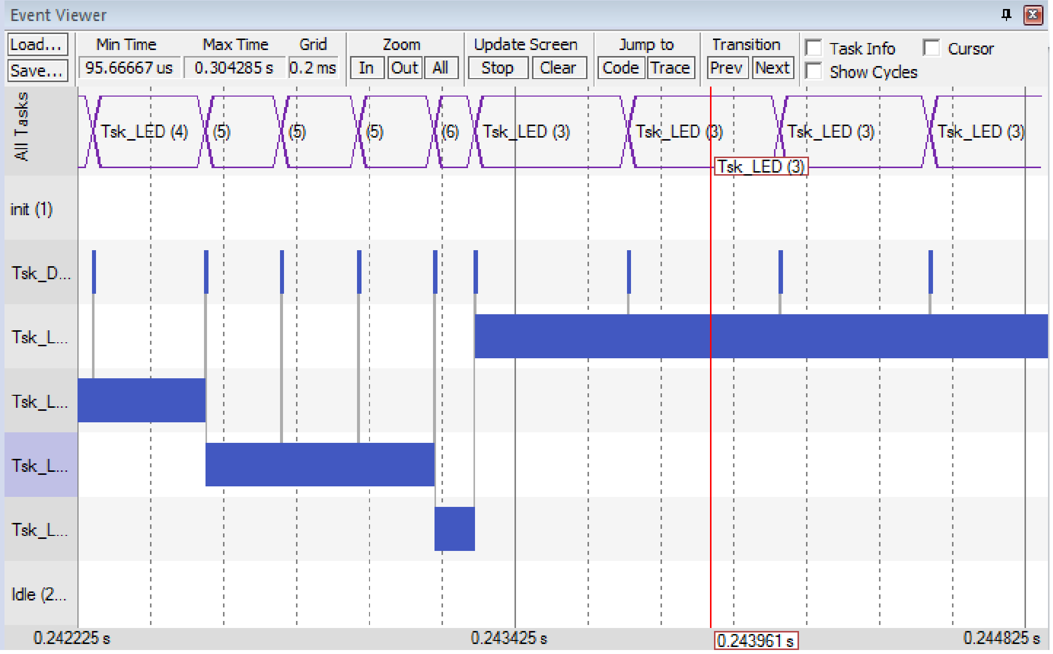
7. Another OS-aware debugging feature is the Event Viewer, (you could have noticed from Q5). Enable that in the debug mode, you will see the Event Viewer (much similar to the Logic Analyzer). Enable that debugging feature.



☑ System and Thread Viewer

☑ Event Viewer

And the Event Viewer looks like this:



Event Viewer

Min Time: 95.66667 us

Max Time: 0.304285 s

Grid: 0.2 ms

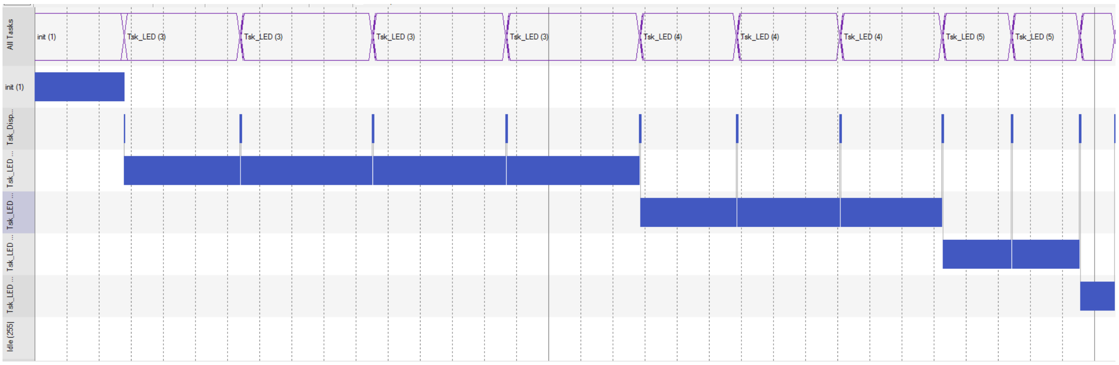
☐ Task Info

☐ Cursor

☐ Show Cycles

You can see it provides a very clear view of what happened, including the transitions between tasks. The number next to the task name is its task ID (OS_TID). Similarly, you can see from the Event Viewer that at any given time, there will be only one task running.

The Event Viewer uses Instrumentation Trace Marcocell (ITM, part of CoreSight Debug technology) and is slightly intrusive, but should not be a problem for this lab. This feature may not be available on some boards.

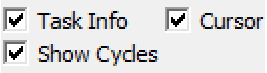


This snapshot captures the activity of the first cycle. Try to explain for whatever happened in time order. Does that correspond to the default LED_priority (4,3,2,1)?

Init was created at first. It then created the dispatcher. After that, it also four LED tasks. After that, it deleted itself and as the dispatcher has a priority of 9, it ran after init.

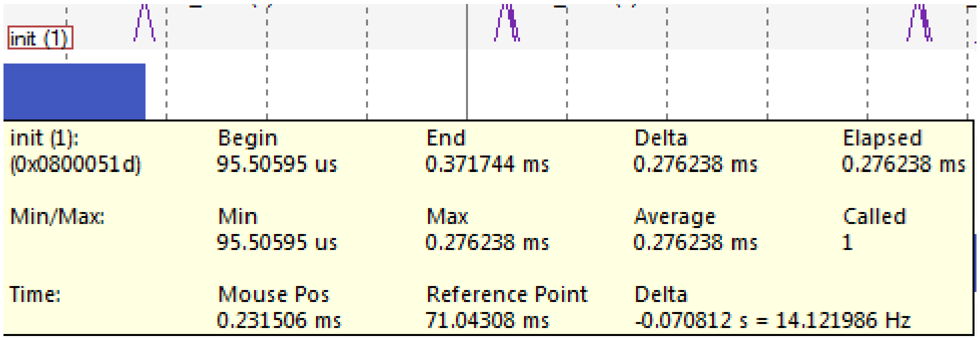
Then the dispatcher dispatched first LED task four times, the second LED task three times, the third LED task two times and the last LED task one time in turn. All other cycles follow the same sequence, but the timing maybe slightly different. This corresponds to the default LED_priority 4,3,2,1.

8. Notice that there are three additional features on the top right corner of the Event Viewer, enable them:



☑ Task Info ☑ Cursor
☑ Show Cycles

Now if you stop the cursor on a task, it will show more related information, such as:



```
init (1):       Begin          End                Delta                          Elapsed
(0x0800051d)    95.50595 us    0.371744 ms        0.276238 ms                    0.276238 ms

Min/Max:        Min            Max                Average                        Called
                95.50595 us    0.276238 ms        0.276238 ms                    1

Time:           Mouse Pos      Reference Point    Delta
                0.231506 ms    71.04308 ms        -0.070812 s = 14.121986 Hz
```

You can also measure the time between two points (Delta in the task info window), and the time for that specific running slice.

The task information also includes call times (Called), which is really useful for optimisation. If a task runs more times than you expected, then there could be a problem. Also, in terms of improving the performance, you will be most interested in the task called by most times as improving them will have highest general effect. You can also calculate the execution time for each task by multiplying the call times and the average execution time. The minimum execution time and the maximum execution time are good indicators of the variability of the task as well.

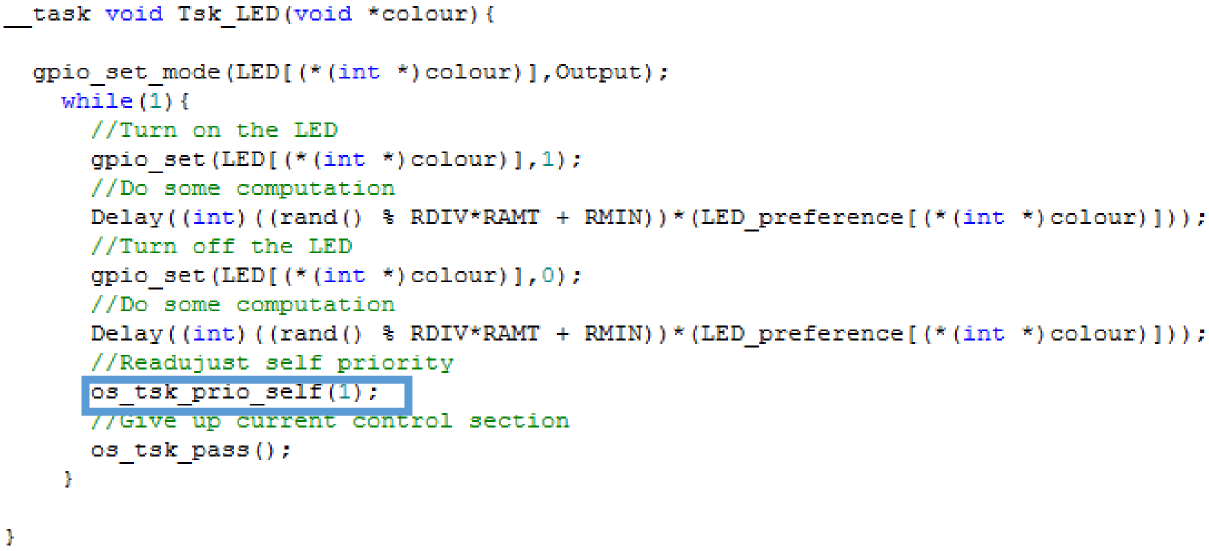
Now run the program and wait until it finishes. Stop the debugger, record call times and calculate the execution times for all tasks and fill in the following table. Do the call times correspond to what you expected? Calculate the sum of execution times for all LED tasks. Assume the execution time of init and dispatcher is negligible - does it approximate to the answer you have for Q2 or Q3? Ideally, we would like to reduce the execution time of the most frequently evoked task, to best optimise the timing performance of our program.

Task	init	Dispatcher	LED1	LED2	LED3	LED4
Average Execution time(ms)			0.4	0.3	0.2	0.1

Call Time	1	1001	400	300	200	100

It corresponds to the priority set up and the number of cycles executed.
0.4*400+0.3*300+0.2*200+0.1*100=300ms. Around 0.3s.

9. It is suggested that the os_tsk_prio_self(1); is not necessary within the LED task:



```
__task void Tsk_LED(void *colour){

  gpio_set_mode(LED[(*(int *)colour)],Output);
    while(1){
      //Turn on the LED
      gpio_set(LED[(*(int *)colour)],1);
      //Do some computation
      Delay((int)((rand() % RDIV*RAMT + RMIN))*(LED_preference[(*(int *)colour)]));
      //Turn off the LED
      gpio_set(LED[(*(int *)colour)],0);
      //Do some computation
      Delay((int)((rand() % RDIV*RAMT + RMIN))*(LED_preference[(*(int *)colour)]));
      //Readujust self priority
      os_tsk_prio_self(1);
      //Give up current control section
      os_tsk_pass();
    }

}
```

Since the following os_tsk_pass() can give up its current section to dispatcher. Is this true or not? Why?

It is not true, see Q10 for more details.

10. Try to run the program without the line that downgrades LED's priority. What has happened? Try to explain Q9 based on the result.

Only one LED task will be constantly running or waiting. As it has the highest RTX priority (10, assigned by the dispatcher), any other tasks with lower RTX priority will not be considered by the scheduler even if the LED task executed os_tsk_pass();, the scheduler will dispatch the LED task after that.

Recall that high priority background tasks in RTX must be self-blocking. Downgrading own priority (os_tsk_prio_self) is a way to achieve this.

11. Come back to the code editing mode and in the main.c tab, put os_tsk_prio_self(1) back, and press F2 to see the bookmarked line of LED_priority. Try to change the sequence of the LED_priority array, for example, LED_priority[4]={4,1,2,3};   rearrange the sequence only. Run the program and check the end_time, is it faster?

Around 0.26s. So around 0.05s faster than the default setting. As a matter of fact, any other priority setting will improve the timing.

12. Is that the optimal priority setting? Try some other settings randomly, record the combinations you have tried and note the best priority setting among them.

    1,4,2,3 around 0.23s

    3,1,2,4 around 0.23s

    3,4,2,1 around 0.29s

    1,2,3,4 around 0.2s

    Actually the optimal setting is 1,2,3,4.

13. At this point, you should be able to see the relation between your priority setting and the LED preference of the program. Try to explain how it works by either checking the code or moving on to the next step.

    The LED task with highest LED preference runs longest. Try to avoid those and run more low LED preference task is a good idea.

14. Now modify the MAX_COUNT constant from 100 to 10000 (should take less than 1 minute to finish). Try the best setting you have from Q12 or Q13; how long does that take to finish the entire program?

    For example, if the best setting you get from Q12 is 1,4,2,3, then the end time is around 24s.

15. Enable the random coefficient generator by setting COEFFICIENT_RAM_GEN to 1. Rerun the program several times with the same setting as you did in Q14 and record the end_time; are they the same or similar? Is your setting still a suitable option with a random generated program preference?

    End_time varies between 20s and 30s. Check the LED_preference through the watch window, you will be able to predict if your setting will perform well or not in that specific run. In most cases, it takes more than 23 seconds to finish, which means this is probably a less good option to tackle the unpredictable preference problem.

16. Now disable the random coefficient generator and enable the ML, which is an adapter algorithm that can help the dispatcher to learn about the preference and dynamically adjust the led priority to the optimal level. You need not concern yourself with the detail of how it works during this lab. Now run the program and monitor the LED_priority array through the watch window. What is the final result of the LED_priority and end_time?

    The learned outcome is {1,2,3,4}, in some cases, may be {1,2,4,3}. End_time is around 22seconds.

17. Now disable the ML and statically assign the learned priority value from Q16 to LED_priority, and rerun the program. Does that improve timing?

It takes around 21 seconds to finish. So the ML program is around 1 second slower than the optimal case. The ML algorithm tries to identify the best priority setting at the expanse of some timing performance.

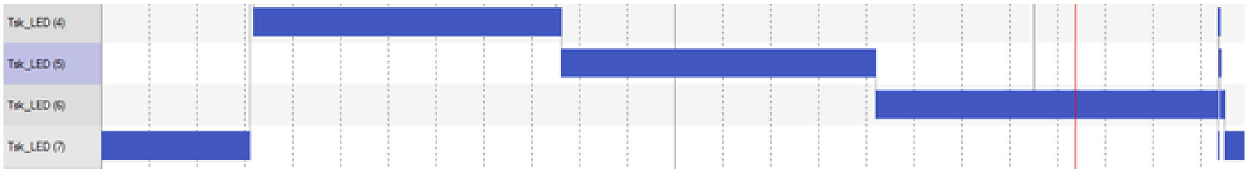
18. Enable both ML and coefficient generator and rerun the program for several times. Check if the performance is consistently close to optimal.

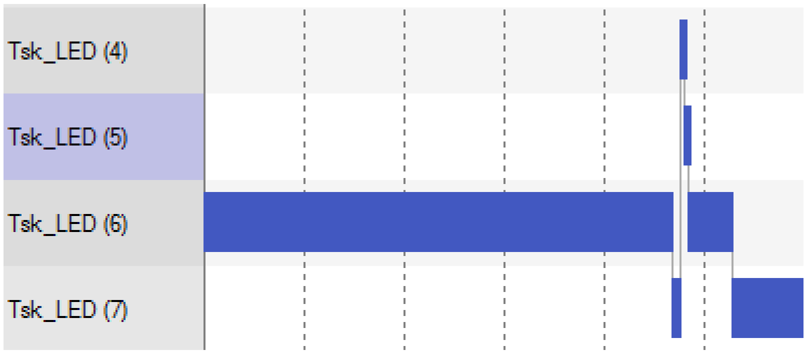
    Yes, the end_time is usually around 22 seconds. This may not be the optimal result, but it can deal with the random situation with a satisfactory performance.

19. (Optional) If your colleague finds that the called times for a LED task is unexpectedly high (for example, should be 600, but actually 834), what would you suggest to help investigate the problem?

    Use the Event Viewer to zoom in the timing behaviour of that task, and pay attention to the transitions from that task to other tasks or from other tasks to that task.

20. (Optional) Your colleague decides to investigate the problem through the Event Viewer and here are two snapshots of what happened:





    What would you suggest to tackle the problem?

    It seems that task LED 6 is being constantly interrupted by other LED tasks and then resumes its own execution. It may be a good idea to start from investigating what causes these transitions; most likely an incorrect way of giving up execution or wrong priority adjustment.