Delft University of Technology

Real-time Systems
IN4343

Assignment 1

Authors:

Jia Shi (5218845)
Kewei Du (5209633)

March 5, 2021

1 Part1: Understand the system

1.1 Question 1

The original frequency of MSP430 is 32768Hz, the source for *Timer_A* is the original frequency divided by 8, 4096Hz.

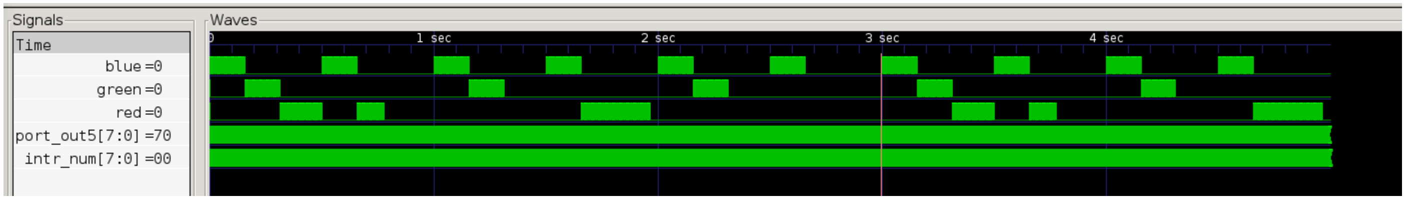

Figure 1: output schedule of part1

From the output picture, we can see that the hyperperiod of this system is 3 seconds. To get more accurate values, we export results from GTKWave, in which we can see the exact length of time period on executing system functionalities (where *intr_num[7:0]* is 0). By adding all the period of time that spent on executing system functionalities, from 3.45 millisecond to 3 seconds, we can see that the time spent on executing system functionalities is about 1.8 second.

1.2 Question 2

To calculate the time more accurately,we exported the result from GTKWave as well. The system is in interrupt when *intr_num[7:0]* is 06, which also means the task is not executing.

We calculated the time that was not spent on executing actual tasks for all the simulation signals, then divided it by the total simulation time. This result is same with the required result since the whole simulation time consists of several hyperperiod.

$$\frac{time\ not\ executing\ actual\ tasks}{simulation\ time} = \frac{1536489170}{4999867251.00} \approx 0.307305993$$

1.3 Question 3

Using the same method in Question 1, we can calculate the overhead of different number of tasks, then get the plot of overhead. The average overhead is 30.7%. The overhead will increase with the increasing number of tasks, in a relationship almost linear.

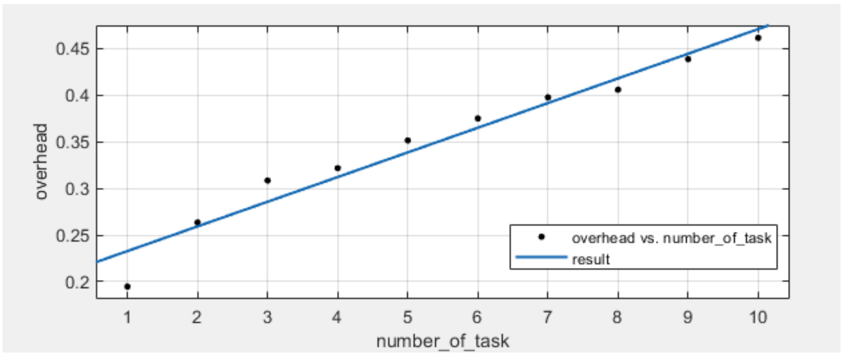

Figure 2: average overhead per unit of time

2 Part2: Improve the System

2.1 Question 1

Please check the code handed in on BrightSpace.

2.2 Question 2

Here list the results of test case *Tst1*.

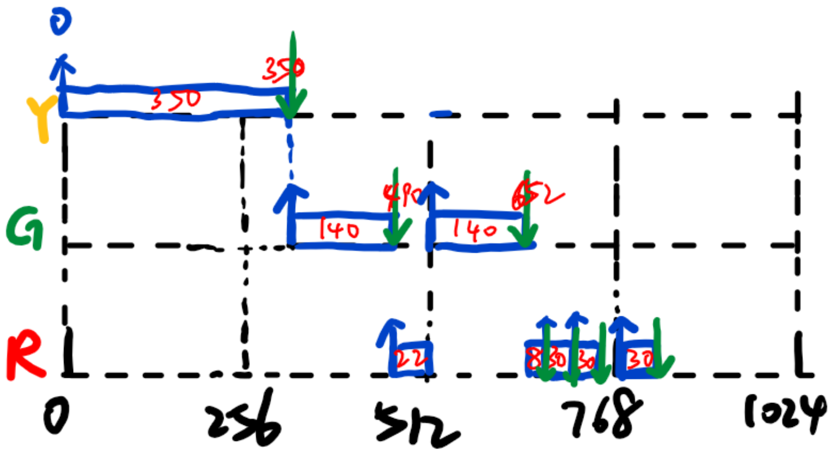

Figure 3: sketch of Tst1 P FP

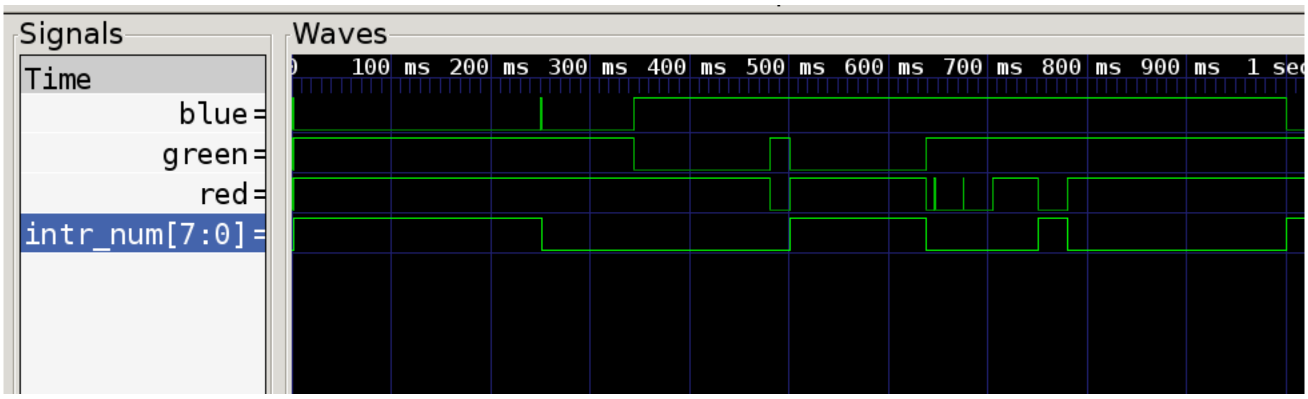

Figure 4: simulation result of Tst1 P FP

The basic idea of the timer interrupter is to trigger itself every time a new task arrives. After triggered, the interrupter handles the event as such:

1. Compare the **NextRelease** time of each task with current **NextInterruptTime** to determine which task indeed triggers the interruption. Once determined, the **NextRelease** will be activated one. more time and added by one period.

2. Determine the **NextInterruptTime** by iterating all the tasks again. The **NextInterruptTime** will be the smallest **NextRelease** among all the tasks.

When done checking, the interrupt function will call scheduler, which is **Scheduler_P_FP** here. **Scheduler_P_FP** simply follows the idea of scheduler in part 1.

1. Once a task is executed, one extra bit in its **Flag** will be set as **ACTIVE**, in function **ExecuteTask()**.

2. In scheduler, it first checks all the tasks to see the task with highest priority being executed, by inspecting the **ACTIVE** flag. Once found, it records the priority and terminates the loop.

3. Next, the scheduler only needs to go through all the tasks with higher priority than current executing task, which are potential tasks that should be executed first. Once found, these tasks with higher priority will goes into function **ExecuteTask()**.
4. Once a new interrupt occurs during the execution, the execution will be suspended, till the interrupt is done.

2.3 Question 3

Overall time		Timer Interrupt		Scheduler	
P1OUT	P2OUT	P3OUT	P4OUT	P1OUT	P2OUT
0C	7B	00	0E	00	02

Table 1: The timer interrupt and scheduler result of *Tst1*

From the table, we can see that the overall interrupt time is 3584 in decimal, the overall time is 31500 in decimal.

$$system\ overhead = \frac{overall\ interrupt\ time}{overall\ time} = \frac{3584}{31500} \approx 11.38\%$$

2.4 Question 4

No, it will not. The method of overhead measuring is by reading the value of **TAR** and save the time of interrupt and execute period into array **Totals**. **TAR** will keep counting as long as the program is executing, so the array **Totals** can calculate the executing and interrupt time with the time slot it getting into the scheduler and getting out of the scheduler (There is only two status: executing task or interrupt). Since the calculation time of subtraction and addition can be ignored compared to execution and interrupt time of the program, the tracking functions will not interfere with the program execution.

2.5 Question 5

From the result in Table 2 and Figure 5, we can observe a generally increasing time spent on interrupt and scheduler.

Number of task	Overall time		Timer Interrupt		Overhead
	P1OUT	P2OUT	P3OUT	P4OUT	
1	7B	05	02	00	0.143%
2	33	0A	06	00	0.230%
3	7B	0C	0E	00	0.438%
4	0C	0E	1A	00	0.723%
5	2C	0E	1E	00	0.827%
6	2C	0E	1E	00	0.827%
7	2C	0E	24	00	0.992%
8	2D	0E	26	00	1.047%
9	06	C4	0E	2D	1.198%
10	EB	0E	2F	00	1.231%

Table 2: Overhead vs number of tasks

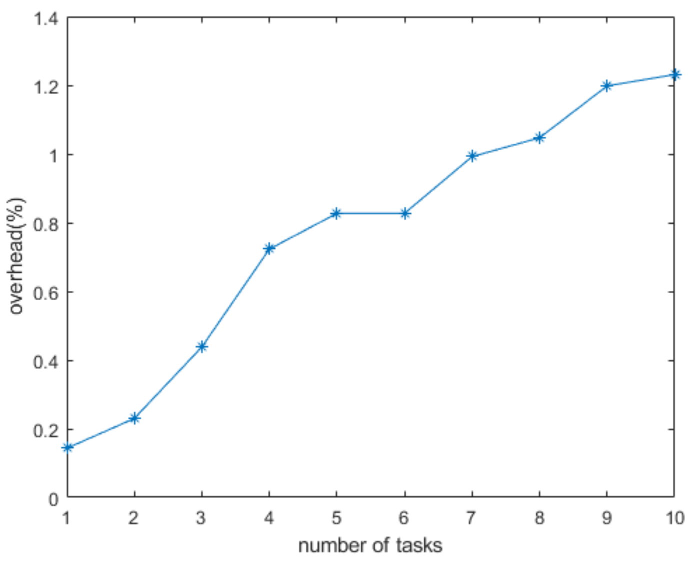

Figure 5: Overhead vs number of tasks

2.6 Question 6

For period, a smaller one indicates a higher frequency of interruption calling, thus a higher time spent on the interrupt overhead, perceiving the overhead for calling interrupt function as a constant.

On the other hand, execution time does not contribute to the overhead in an explicit way. However, the task transition from low priority to high priority often takes longer to finish, as observed in the waveform. If task with lower priority has a long execution time, while higher priority tasks arrives in a high frequency, the frequent transition form low to high will considerably increase the total overhead.

3 Part3: Add new features

3.1 Question 1

Please check the code handed in on BrightSpace.

3.2 Question 2

Sketching and simulation results are listed below w.r.t. different tast case. Overall, the result of sketch and simulation is generally the same.

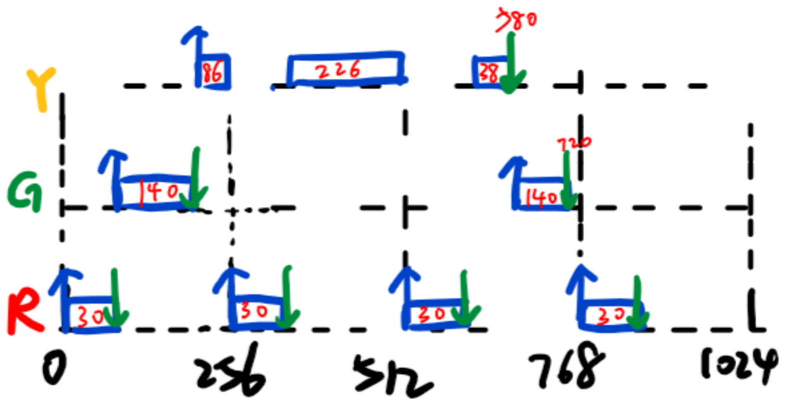

Figure 6: sketch of *Tst1* P EDF

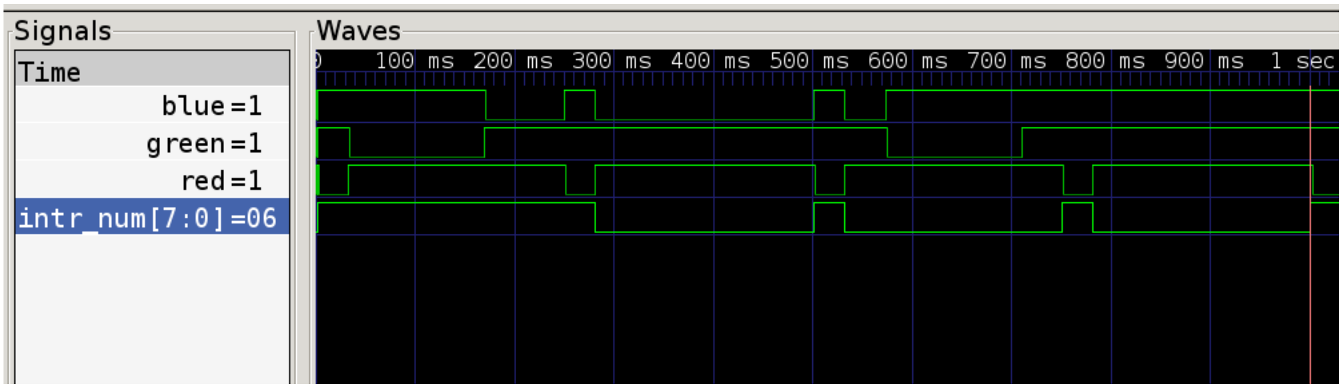

Figure 7: simulation result of *Tst1* P EDF

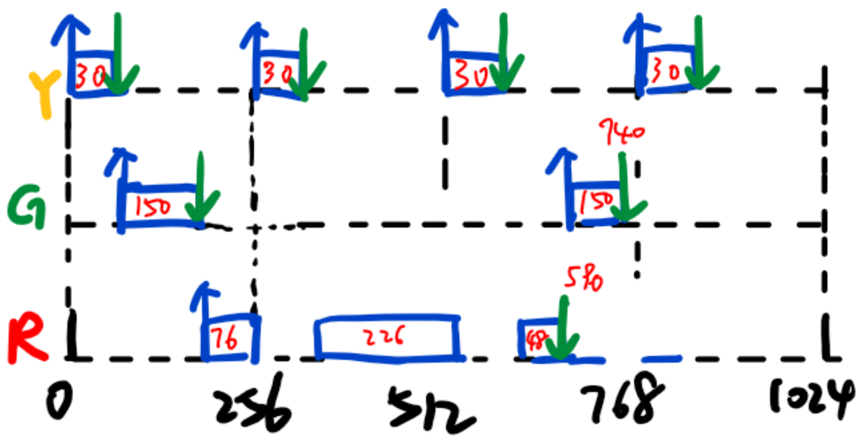

Figure 8: sketch of *Tst2* P EDF

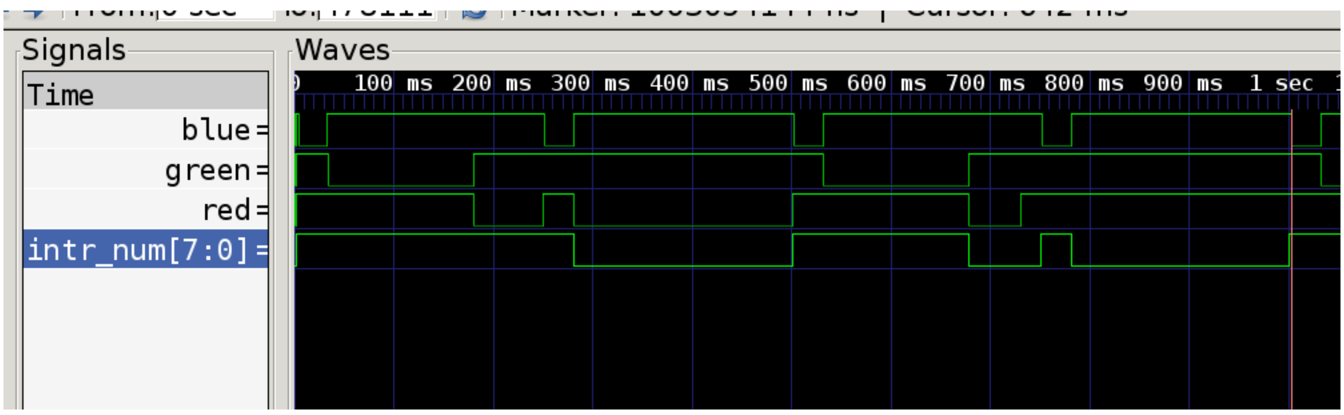

Figure 9: simulation result of *Tst2* P EDF

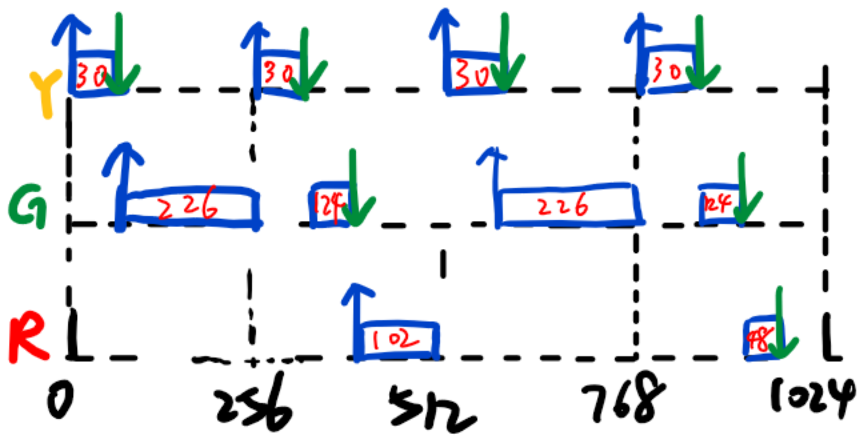

Figure 10: sketch of *Tst2* P EDF

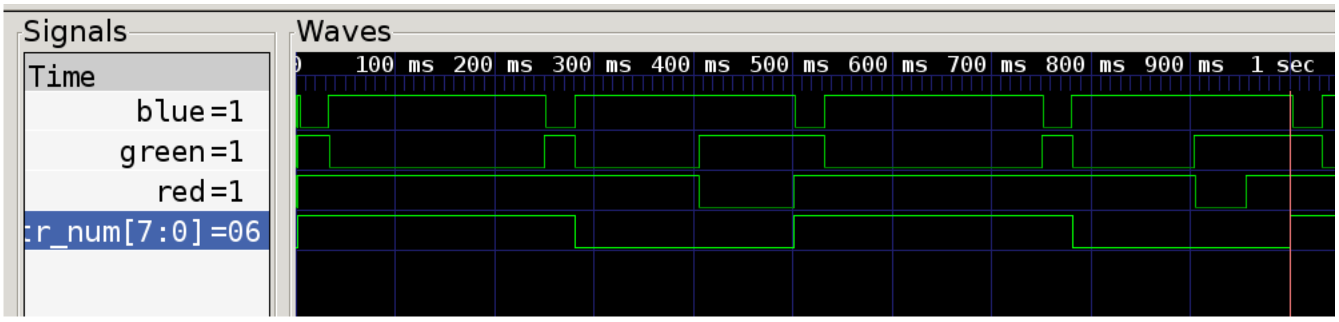

Figure 11: simulation result of *Tst3* P EDF

3.3 Question 3

The implementation of EDF algorithm, in general, requires an additional data structure to store and sort the job to be executed, served as a queue. For easy inserting new-coming tasks and delete finished tasks, we implement a doubly-linked list. Each node in the linked list has two data attributes, one for task index and the other for task deadline. Task index is here for task positioning, and task deadline is for task sorting. When a new task comes, it will first compare the deadline with arranged tasks in the list. Once its deadline is greater or equal to the previous scheduled task node, and less than the deadline of next node, it finds its right position. When two tasks share one common deadline, they will be scheduled based on the priority, i.e., the order in the task array. Then the tasks will be executed following the order in the list.

One special occasion is that, once the task with flag set as active, the execution will cease since this active task is actually be started by some other scheduler already. The rest of tasks will be executed after the active task, following the list order.

3.4 Question 4

The result in Table 3 and Figure 12.

Number of task	Overall time		Timer Interrupt		Overhead
	P1OUT	P2OUT	P3OUT	P4OUT	
1	7F	05	06	00	0.426%
2	35	1A	0F	00	0.2235%
3	7E	0C	21	00	10.319%
4	2E	0D	47	00	21.043%
5	2F	0D	5C	00	27.259%
6	34	0C	58	00	28.169%
7	34	0C	63	00	31.69%
8	30	0E	80	00	35.242%
9	0C	0E	A2	00	45.05%
10	0C	0E	BB	00	52.0%

Table 3: Overhead vs number of tasks

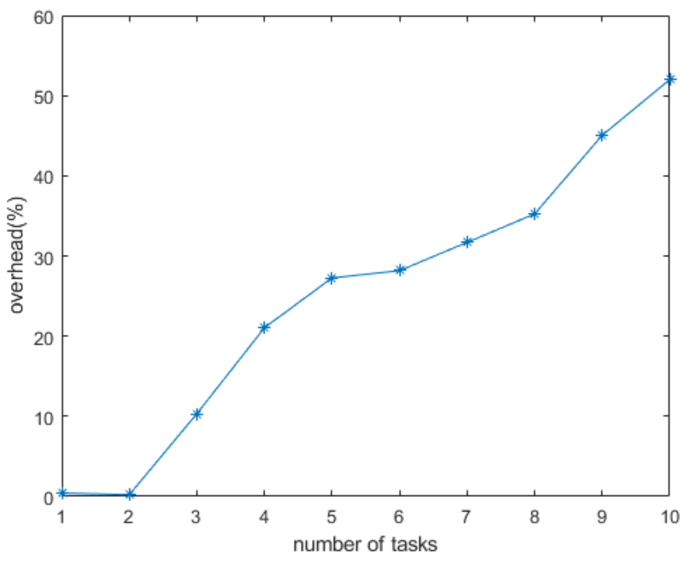

Figure 12: Overhead vs number of tasks