IN4343
REAL TIME SYSTEMS
Assignment - II

March 21, 2018

Chinmay Pathak (4740327)
Himanshu Shah (4739779)

1. Overhead of timer handler and event latency for SchedulerNPBasic.c

1. *Measure the execution time of the timer-interrupt handler as a function of the number of tasks NUMTASKS (i) from 1 up to and including 10 and (ii) for 15, 20, 24, and 25. Present your results in a table and a graph.*

   Ideally, the execution time of the timer-interrupt handler should be constant if there is a change in the number of maximum tasks. While measuring the values below, we comment out the tasks registers under the function.

NUMTASKS	EXECUTION TIME(ms)
1	40.5
2	115
3	153.1
4	195.3
5	230.8
6	263.1
7	307.8
8	344.3
9	380.3
10	423.3
15	606.5
20	793.01
24	942.5
25	979.8

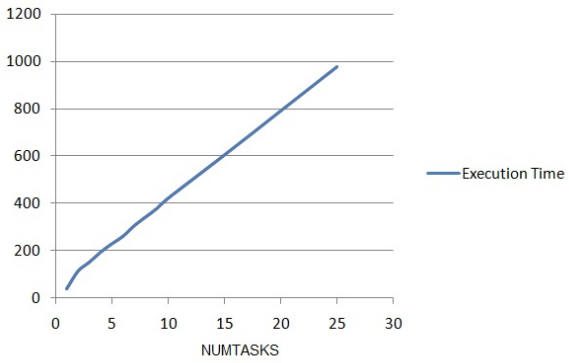


Figure 1: Execution times for various Tasks

2. *For which value of NUMTASKS will the interrupt handler become longer than the length T_{Clk} of the period of the timer-interrupt?*

   For NUMTASKS = 25, the period of the interrupt handler becomes longer than the length T_{Clk} of the period of the timer-interrupt.

3. *The overhead of the timer handler is not entirely linear in the number of tasks (although it may be approximated as such). What could be a reason for these deviations from a linear relationship, in particular for NUMTASKS = 1?*

   Since in case of NUMTASKS=1, the while loop is only executed once the compiler optimize the loop into normal statements using loop unrolling which causes the execution time to be less.

4. *What do you recommend to reduce the overhead of the timer handler (without changing SchedulerNPBasic.c)? Suggest 2 techniques that a system designer can use to alleviate the overheads of interrupt handling.*

   One way to alleviate is to reduce the number of entries of NUMTASKS array to 3 instead of 10 as there are only three tasks scheduled. This can reduce the overhead by a factor of 3.

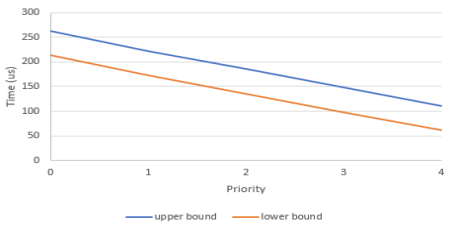
   Another technique can be to employ different optimization techniques used by the compiler.

5. *Before looking at the specific question, formulate a hypothesis concerning the event latency as a function of the task priority.*

   Ideally there should be no event latency but stacking of activated events, setting the variables do take some amount of time which then causes systems to have some event latency. As the activated tasks are registered into an array in our case, it is possible for the system to keep the event latency for highest priority task to be minimum.

   Measure both an upper and a lower bound for the event latency as a function of the task priority for NUMTASKS = 5. Present your results in a table as well as a graph

Priority	Upper bound	Lower bound
0	262 μ sec	213 μ sec
1	222 μ sec	173 μ sec
2	186 μ sec	136 μ sec
3	149 μ sec	98 μ sec
4	110 μ sec	61 μ sec



2: Non-pre-emptable task execution (SchedulerNP.c)

When the timer interrupt is active, all the tasks are checked for (priority wise).
If they are actually registered or not.
If its time for them and the period matches.
After these conditions are met, the higher priority task sets the pending flag, which also leads the MSP430 to exit the low power mode*.
The waiting task 'HandleTasks' can now execute and activate all the tasks that were

waiting (priority wise).

[*]The low power mode is ideal for MSP, since our tasks are short to execute. The processor will have nothing to do after finishing these small tasks, and it can go to sleep instead of doing 'HandleTasks'.

6. *Looking at the timer-interrupt line and the start of the task executions, how can you observe the new behavior in the simulation results?*

   The difference between the behaviour can be observed at a moment where the tasks are being invoked. In the Lab1 the tasks were invoked in the timer interrupt only, while in this case tasks are invoked by the handle tasks routine. This change makes sure that on period of the timer interrupt is for timer interrupt only, while the off period is for handle tasks where, the tasks are being invoked. This adds some delay for hte execution of the tasks but provides better hadling of the tasks in case of delay.

7. *SchedulerNP.c impacts the fluctuations in the execution time of the timer-interrupt handler as compared to SchedulerNPBasic.c*

   (a) *The fluctuation in the execution time of the interrupt handler*

   SchedulerNP.c: 512 - 462 = 50 μs
   SchedulerNPBasic.c: 575 - 462 = 113 μs

   (b) *The execution time of the interrupt handler for those cases where no task is activated.*

   SchedulerNP.c: 462 μs
   SchedulerNPBasic.c: 458 μs (quite similar)

8. *Measure the delay of CountDelay(60000) compared to SchedulerNPBasic.c.*

   The CountDelay(60000) for SchedulerNP.c lasted for a duration of 1248ms whereas CountDelay for SchedulerNPBasic.c was 640ms long. Hence, CountDelay for SchedulerNP.c was approximately 600ms longer

9. *Assume that CountDelay(60000)takes 640 ms when it is executed without any preemptions and without any interruptions of the interrupt handler (e.g. as is the case*

using SchedulerNPBasic.c). Using SchedulerNPBasic.c, 654 interrupts were missed during the execution of CountDelay(60000). Using SchedulerNP.c, these interrupts, plus some additional ones, will interfere with CountDelay(60000). Assume that the execution time C_{Clk} of the interrupt handler is 0.464ms for SchedulerNP.c during the execution of CountDelay(60000) and the period TClk of the interrupt handler is 1/1,1024≈0.977ms.
a) Apply the worst case response time analysis to Compute the expected(worst-case) interval length, in which CountDelay(60000) is executing.

Ans: Period of the interrupt handler is 0.977ms
Execution time of interrupt handler is 0.464ms
Period - Execution time = 0.513ms
Since, CountDelay(60000) takes 640ms and can not execute during the execution of interrupt handler, the number of cycles required by CountDelay(60000) is 640/0.513 ≈ 1247 cycles.
The period of CountDelay(60000) = 1247*0.977ms = 1218.319ms

b) How many timer interrupts are interfering during the computed interval length, and how much time is the execution of CountDelay(60000) delayed due to those interrupts?

Ans: 1247 interrupts are interfering during the computed interval length. The execution time of CountDelay(60000) is delayed by 608ms.

c) The measured delay in Question 8 differs from the computed delay in the previous question. Is the computed worst-case response time a valid upper bound for the actual worst-case response time of CountDelay(60000) ?

Ans: No, it is not a valid upper bound because actual delay is greater than the computed delay.

10. *Apparently, our assumptions on which our calculation is based are not correct, in particular, C_{Clk} is apparently too small. Give two main reasons why C Clk is larger than 0 . 464 ms . Hint : What does the ISR do differently after 2.0s?*

One reason for the C_{Clk} being larger than 0.464ms is because we are considering the time of the false or else case of the if condition of the loop only, but when the if condition evaluates to true the time taken by C_{Clk} is greater than 0.464ms.

```
interrupt (TIMERA0_VECTOR) TimerIntrpt (void)
{
  uint8_t i = NUMTASKS-1;
  do {
    Taskp t = &Tasks[i];
    if (t->Flags & TRIGGERED) { // countdown
      if (t->Remaining-- == 0) {
        t->Remaining = t->Period-1;
        t->Activated++;
    Pending = 1;
      }
    }
  } while (i--);
  if (Pending) ExitLowPowerMode3();
}
```

Figure 2: Unpredictability due to if condition

In the above image when the highlighted 'if' condition evaluates to false C_{Clk} is 0.464ms, but it is larger than 0.464ms when this 'if' condition evaluates to true.

11. *Describe how the efficiency of HandleTasks() can be improved.*

    One of the way that make handle tasks inefficient is that it travels the whole tasks array to check which one task is pending. This causes unnecessary if condition checks which is where the better efficiency can be achieved. To overcome this instead of using binary flag "pending", the priority of the task being activated could be set equal to pending in interrupt handler. This will cut down on the unnecessary traverse time of the handle tasks routine. This way handle tasks will start executing with the highest activated priority taks.

12. *Zoom in around the second falling edge of green , where multiple activations of BlinkGreen() are pending. Looking at the order of the execution of BlinkGreen() and BlinkYellow() , what goes wrong here?*

    In this case, we can see that even though BlinkYellow() has highest priority and

BlinkYellow is pending, BlinkGreen() is executed before BlinkYellow() which should not be the case.

13. *Examine the code of 'HandleTasks()'. The problem can be resolved by strengthening the guard of the inner while-loop. Suggest a replacement for the guard, such that the scheduling anomaly is resolved.*
Solution:

```
void HandleTasks (void)
{
  while (Pending) {
    int8_t i=NUMTASKS-1; Pending = 0;
    while ((i>=0) && (Pending == 0)) {
      Taskp t = &Tasks[i];
      if (t->Activated != t->Invoked) {
        if (t->Flags & TRIGGERED) {
          t->Taskf(); t->Invoked++;
        }
        else t->Invoked = t->Activated;
      }
      else i--;
} } }
```