ECE3411 – Spring 2019

Lecture 12

Task-Based Programming

Sung Yeul Park
Department of Electrical & Computer Engineering
University of Connecticut
Email: sung_yeul.park@uconn.edu

Copied from Lecture 3b, ECE3411 – Fall 2015, by
Marten van Dijk and Syed Kamran Haider

Based on the Atmega328PB datasheet

UCONN

Important Date for Mini Projects

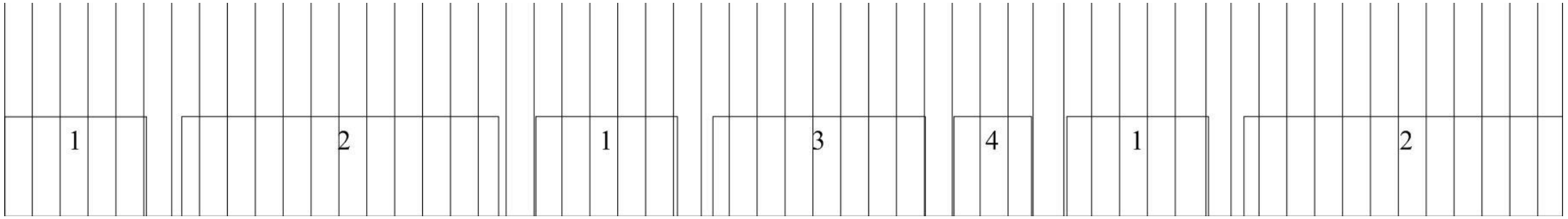
Date	To do List
4/8	- Form a team with two students - Pick up one RedBot kit
4/10	- Complete to form a team with two students - Practice basic RedBot functions
4/24	RedBot demo and racing: - Task based programming code - Straight line and S curve line - Three fastest team will get 20% bonus points - Need to submit codes as well
5/6	3:30~ BNO055 based RedBot moving - Three fastest team will get 20% bonus points - Need to submit codes as well

The Importance of Organizing Code

- The organized and neat code is highly preferable to inefficient and confusing code.
- As a general rule of thumb, the more that you write, the more debugging you will need to do later.
- We also want to reduce the amount of code sitting in our ISRs because 1) it's more difficult to debug, 2) Long ISRs may keep interrupts disabled so long that the MCU misses other events.
- We also want to cram more and more features/peripherals into our programs without hurting our performance.
- How can we do this???

Task-Based Programming

- Suppose that we have, 1. ADC, 2. Math Function #1, 3. UART, 4. Math Function #2
- Each task takes a certain amount of time to finish, so it's more efficient if we periodically space them out and repeat them at specific intervals.



Task Based Programming

- Ex. According to the datasheet, after the initial conversion, ADC will take ~13 ADC Clock cycles. This gives you insight as to how much time the 'task' should take. Doing this estimation for all tasks can give us a rough idea of how to schedule all of our features.

- Suppose that each task, 1, 2, 3, and 4 have software timer variables,

```
int ADC_Timer
int Math_1_Timer
int UART_Timer
int Math_2_Timer
```

Task Based Programming

- Let's say that our ISR occurs every 1mS, ADC takes 2mS, Math_1 takes 4mS, UART takes 3mS, and Math_2 takes 5mS. Initializing our timers with 1mS in between them,

```
int ADC_Timer    = 18; // Waits 0mS
int Math_1_Timer = 21; // Waits 3mS
int UART_Timer   = 26; // Waits 8mS
int Math_2_Timer = 30; // Waits 12mS
```

- The whole cycle, repeating our four functions will take 18mS. With a 1mS ISR we decrement our variables at this frequency,

```
ISR(TIMERn_COMPN_Vect)
  {
       if(ADC_Timer > 0)        ADC_Timer--;
       if(Math_1_Timer > 0) Math_1_Timer--;
       if(UART_Timer > 0)     UART_Timer--;
       if(Math_2_Timer > 0) Math_2_Timer--;
  }
```

Task Based Programming

- The while(1) loop can now keep track of the status of our timer/counters. Once they've been decremented to zero, we reset the timers and then call our functions. Since we've staggered them already, they can be reset to the same values. You can space them out whatever you like. However, remember that the timer ISR will keep counting during the execution of your functions, and you may have two timers == 0 at the same time!!

```
while(1)
{
        if(ADC_Timer == 0{
                ADC_Timer=18; ADC();
        }

        if(Math_1_Timer == 0){

        Math_1_Timer=18; Math1();
        }
        if(UART_Timer == 0){
                UART_Timer=18;
                        UART();
        }
        if(Math_2_Timer == 0){
                Math_2_Timer=18;
                Math2();
        }
}
```

Task Based Programming

- In summary, Task Based Programming allows us to organize and schedule many functions without reducing our performance.
- It also reduces the amount of code we keep in our timer ISR, and provides a basis for us to add things like tickets, variable scheduling times, permissions/priority (the basics of an operating system).

Example: How are the tasks scheduled?

```
while (1)
{
        if (task1_timer == 0)          // if task1_timer is not already equal to 0,
                                       // it is being decremented every 1 millisecond
                                       // during a timer ISR
        {
                task1_timer = t1;
                task1();               // task1 takes m1 milliseconds
        }

        if (task2_timer == 0)          // if task2_timer is not already equal to 0,
                                       // it is being decremented every 1 millisecond
                                       // during a timer ISR
        {
                task2_timer = t2;
                task2();               // task2 takes m2 milliseconds
        }
}
```

Example Cont'd

- Suppose t1=5, m1=1, t2=10, and m2=15
- What is the frequency f1 in Hz at which task1() is called?
- What is the frequency f2 in Hz at which task2() is called?

- Answer:
  - Since both task1_timer and task2_timer are decremented to 0 during the execution of task2(), task1() and task2() alternate.
  - Therefore, f1=f2 = 1 every 16 ms which is equal to 1000/16 Hz.

Example Cont'd

- Suppose t1=20, m1=1, t2=10, and m2=15
- What is the frequency f1 in Hz at which task1() is called?
- What is the average frequency f2 in Hz at which task2() is called?

Answer:

- Since task2_timer is decremented to 0 during the execution of task2(), task2() is called as often as possible.
- When it is task1()'s turn to be executed, it takes more than one and less than two executions of task2_timer to get task1() decremented to 0.
- Therefore, the execution pattern converges to a repetition of task2() (takes 15 ms), task2() (takes 15 ms), task1() (takes 1 ms) giving
  - a frequency f_1=1000/31 Hz and
  - an average frequency f_2=2 * 1000/31.

Example Cont'd

- Suppose t1=20, m1=1, t2=25, and m2=15
- What is the frequency f1 in Hz at which task1() is called?
- What is the frequency f2 in Hz at which task2() is called?

- Answer:
  - During the time that task2() is executed (which takes 15 ms), task1_timer (which initial value is 20) is decremented to a value v<=5.
  - The MCU will be idle for v ms after which task2_timer is decremented to 25-15-v and task1_timer just turned into 0.
  - So, after v ms task1() is executed taking 1ms during which task1_timer reduces to 19 and task2_timer reduces by 1 to 9-v.
  - The MCU will be idle for another 9-v ms after which task1_timer is equal to 10+v and task2_timer just turned into 0.
  - Now task2() is executed (which takes 15 ms) after which task1_timer is equal to 0 and task2_timer is equal to 10.
  - The same argument is now repeated for v=0 showing that the execution pattern converges to a repetition of task2() (takes 15 ms), task1() (takes 1 ms), idle time (takes 9 ms) giving
    - a frequency f_1=f_2=1000/25 Hz.

Example Cont'd

- Suppose t1=4, m1=1, t2=8, and m2=4.
- Assume initially task1_timer = 0 and task2_timer = t2
- What is the average frequency f1 in Hz at which task1() is called?
- What is the average frequency f2 in Hz at which task2() is called?

- Answer:
  - Task 1 executes during the intervals [12n,12n+1], [12n+5,12n+6], for integers n>=0.
  - Task 2 executes during intervals [12n+8,12n+12] for integers n>=0.
  - This gives frequencies f_1=1000*2/12 Hz and f_2=1000/12 Hz.