ECE2560 Spring 16 Final Exam/Project

Alarm Clock with Wakup Light – Part II

In the course project you have completed the first step of a real time alarm clock: your program should be able to tick a real time clock (whose time information is stored as hr, min, sec in the RAM) and setting up an alarm. However, the course program was constantly pulling information in a loop and therefore wasting lots of power. We would like to modify the program so that the alarm will “sound” automatically using a hardware peripheral (Timer). Beyond that, our goal is to build a prototype of an alarm clock capable of a “wake up process”, meaning that a wakeup light (simulated with the green LED connected to P1.6 in our project) will turn on gradually for a certain period of time till the set alarm time is reached. The gradual turning on of the wakeup light will be achieved by using Pulse Width Modulation on the P1.6 connecting to the green LED. Finally, we would like to put the microcontroller into sleep whenever it can to further conserve power.

We will provide you with a skeleton program to work on. This skeleton program is modified upon of what you have done with your course project. Here are some of the additional features in this new skeleton program that you need to know (You do not need to work on these features. They are done for you in the skeleton program):

1. A variable `alarmState` is created in the RAM. Setting BIT0 of `alarmState` indicates that your alarm function is on. Setting BIT1 of the `alarmState` indicates that your wakeup sequence is in progress.
2. TimerA1 peripheral is setup to generate a 1Hz waveform on P1.1 in an interrupt service routine called `TimerA1CCR0_ISR`.
3. TimerA0 counter can be clocked from an external clock source on P1.0. We would like the 1Hz waveform on P1.1 to be able to drive TimerA0 via P1.0. To establish a connection needed, you can remove the jumper connecting P1.0 and the red LED and re-place it as shown by the green circle in the figure below:
4. For debugging purposes, you can connect P2.0 to the red LED to see the red LED toggling every one second.

5. Once your alarm function is turned on (at a button press), TimerA0 would count, in hardware, the number of seconds determined in `timeTillAlarm`. After the count is reached, TimerA0 will initiate the wake up sequence by setting BIT1 of `alarmState` and automatically starting the pulse width modulation on the green LED.

6. Wakeup sequence has two parameters associated with it: `wakeupPeriod` and `wakeupCntDown`. `wakeupPeriod` determines how long before the set alarm time the green LED will start turning on. `wakeupCntDown` keeps track of how far into the wakeup process you are. You can think of `wakeupPeriod` as the period of your pulse width modulation. `WakeupCntDown` therefore determines the pulse width. A smaller `wakeupCntDown` value indicates that you are closer to your set alarm time, and a larger pulse width will be produced within a PWM period, your LED will turn on brighter.

7. When BIT1 of `alarmState` variable is set, the main loop of the program will decrement `wakeupCntDown` by 1 every one second until it reaches a lower threshold.
8. For testing purposes, the skeleton program is written so that when you press on the button, `timeTillAlarm` will be set to a total value 15. A `wakeupPeriod` of 30 seconds and an initial `wakeupCnt` down of 30 seconds will be used for the wakeup sequence. Therefore 15 seconds after pressing the button (P1.3), you should see the green LED starts turning on gradually and then reaches its full intensity after 30 seconds. A second press of the button on P1.3 will turn off the alarm and reset the alarm related timer parameters at any time.

**Final Exam/Project tasks:**

1. Run the skeleton project to verify that it is operating as described above.
2. Answer the following question: what low power mode can we use with this program? State your reasons clearly.

In the skeleton program, we have marked out the parts that you need to modify using the notation “check point #”.

3. Modify check point 2 and TimerA1CCR0_ISR so that the main program loop will be put into the proper low power mode, wake up every 1 second to finish some tasks, and go back to sleep again. Copy the modified part of the code into your final exam submission.
4. At checkpoint 3, write a piece of code that will accomplish the following:
   a. reduce the total `timeTillAlarm` by the value defined in variable `wakeupPeriod`.
   b. If the total `timeTillAlarm` is larger than the `wakeupPeriod`, copy the value at `wakeupPeriod` into `wakeupCntDown`.
   c. If the total `timeTillAlarm` is less than the `wakeupPeriod`, move the total `timeTillAlarm` into `wakeupCntDown`. Then clear `timeTillAlarm`.

   Copy your code into your final exam submission. Include a block diagram of your logic.

5. If the timerA1 is sourced from an external watch crystal via ACLK, what is the highest level of low power mode can you use? State your reasons clearly.

Please turn in your final exam in a word or pdf document to the Carmen final exam dropbox by May 3rd 11:59pm. Include all your group members’ name in your submission.