Lab 2: Timer

ECE/CS

Due by Thursday Fed 3, 2011 11:59 PM

1 Problem Statement

In this Lab, you will learn how to program hardware timers. The MSP430 MCU has two built-in timers, which function almost the same. You are going to use one of them to control the LEDs, making them flash in a desired rhythm. To do so, you need to understand how to configure and start a timer, and how to install interrupt service routines (ISR).

Particularly, you are asked to complete the following two tasks:

1. using one timer to control LED1, making it flash in the frequency of 1 Hz (by toggling it on/off every 1 second);
2. as LED1 flashes, using the same timer to make LED2 flash in 2 Hz.

2 Reading Guide

With the knowledge you have learnt in previous Labs, such as toggling LEDs, you may need to further read the chapter 14 of the User's Guide, which describes everything about one of the two built-in timers -Timer A. (Chapter 15 is about Timer B, pretty much the same thing with a few highlights about the difference from Timer A.)

The chapter is not short and you do not have to understand everything to complete your tasks. In your reading, focus on the following contents:

- clock source selection and divider (ACLK clock)
- timer operating mode (Stop, Up, Continuous modes)
- capture/compare blocks (Compare Mode)
- timer interrupts

Following is a high level description of how the timer works, concentrating on the above issues. It aims to give you an overview before you dive into the User's Guide. For more accurate and complete instructions, read the chapter.

Informally, a timer is a 16-bit counter driven by the selected clock. Depending on the frequency of the source clock, the timer can count (by increasing or decreasing the counter by one at the rising edge of a clock signal) in a predictable speed. Therefore, knowing the clock frequency, yon can compute the time it takes to count any number. For example, ACLK clock runs at 32k Hz by default (1k = 1024), so if ACLK is selected as the source clock, the initially zeroed counter will be overflowed (count up to 0xFFFF then 0) in 2 seconds. If you feel the selected clock is too fast, you can set a divider. Then the actual clock frequency will be firstly divided by the divider and the timer will count in the computed frequency. e.g. ACLK with divider set to 2 drives the timer in 16k Hz.

The timer can operate in four modes, of which we only care about Stop, Up, and Continuous. Stop mode is simply that the timer is halted.

Under Up mode, the timer repeatedly counts from 0 to the value of register CCR0 and generates an interrupt (if enabled) at CCR0, as shown in Figure 1. By installing a CCR0-corresponded ISR, you get a periodic task handler.

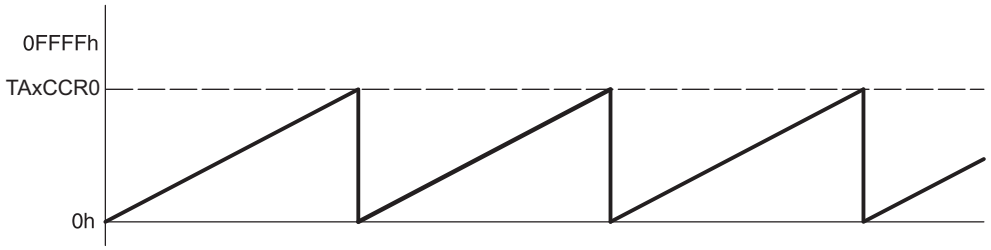

Figure 1: **Up Mode**

Under Continuous mode, instead of counting up to CCR0, the timer counts to 0xFFFF and restarts from 0. During the repetition, the corresponding interrupt will be generated (again if enabled) at the point that the counter counts to CCRn where n can be 0 to 6. Therefore, by using Continuous mode and setting CCRn to desired values, you can get up to 7 periodic interrupts with different time intervals.

Figure 2 shows an example of using Continuous mode. In the example, we get two periodic interrupts with interval of t_0 and t_1 respectively using CCR0 and CCR1. As you may notice, maintaining the desired interval needs you to adjust the associated CCR in the ISR. That is, at each occurrence of an interrupt, e.g. TAxCCR0a, you must adjust the CCR0 accordingly by adding an fixed size offset (counting over which takes t_0 long) to it to make sure the next interrupt occurs at time point TAxCCR0b. The figure is taken from User's Guide subsection 14.2.3.3, where you can find more details.

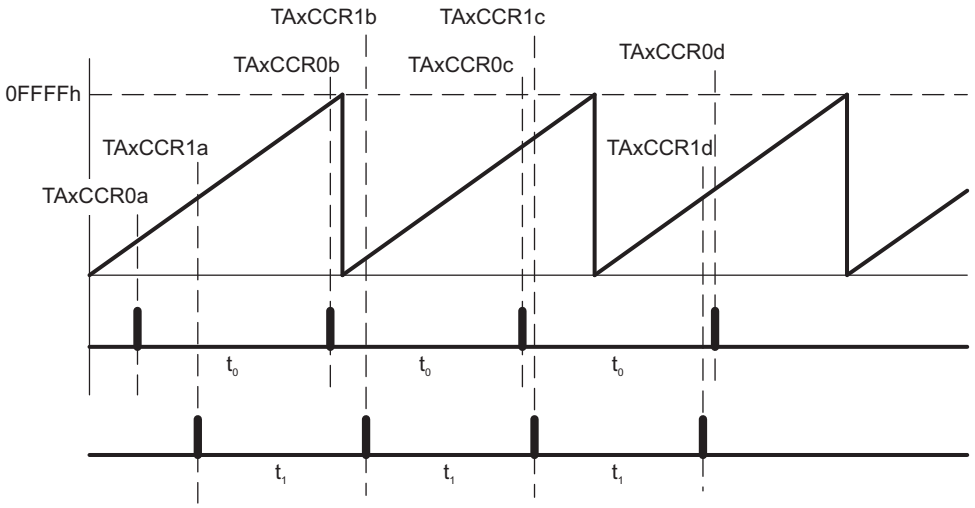

Figure 2: **Continuous Mode Time Intervals**

All the above description are made under the assumption that the CCRs are used under Compare mode. Under this mode, CCRs are used as targets to which the counter is compared. The Capture mode, on the other hand, uses CCRs to record the time when certain events happened by copying the counter value into a CCR. You would not need Capture mode in this Lab.

As mentioned previously, each CCRn can be a source of interrupts. The counter overflow will generate a interrupt as well. Every interrupt can be enabled or disabled independently. There are two interrupt vectors reserved for each timer. One is dedicated to CCR0 interrupt and the other is for all the rest. Using former one is straightforward, while using the latter may

need a bit more work. Essentially, when an interrupt other than CCR0's occurs, your ISR will be invoked and a particular value indicating the interrupt source will be assigned to a register called IV (the timer prefix is omitted, see section 3.1). Therefore, you will be able to know where the interrupt comes from by examining IV in your ISR hence able to handle it properly.

3 Implementation Hints

Here we give some hints about your implementation.

3.1 Timer Instances

As mentioned at the beginning of Chapter 14, Timer A may have multiple instantiations (each uses prefix TAx in its register names). By looking at the header file "msp430f5438.h" (which you can find at directory `CCSV4_HOME/msp430/include`, where `CCSV4_HOME` denotes the directory the Code Composer Studio is install), you will find there are two of them - TA0 and TA1. For Timer B, there is only one instance TB0. Simply pick one of them and make sure you always work on the registers with the same prefix you picked.

3.2 Registers and Bits

Suppose you choose to use TA0, Table 1 shows a SUPERSET of the registers and bits that you need to deal with concerning timer configuration. "Superset" means not all of them are supposed to show up in your program; just pick what you need. For the detailed description, refer to User's Guide subsection 14.3.

Table 1:

Registers	Bits
TAxCTL	TASSEL, ID, MC, TACLR
TAxCCTLn	CAP, CCIE
TAxIV	TAIV
TAxEX0	IDEX
TAxCCRn	all bits

3.3 ISR and Interrupt Vector

The MSP430 C/C++ compiler that comes with Code Composer Studio is a slightly modified version of the IAR one. Here we list some intrinsics, pragmas, and micros that you may use when dealing with interrupts:

To enable global interrupts, use

```
__bis_SR_register(GIE);
```

To install an ISR *THE_ISR* into the interrupt vector *THE_VECTOR*, use

```
#pragma vector=THE_VECTOR
__interrupt void THE_ISR(void)
{
  // your code here
}
```

In the header file "msp430f5438.h", you can find all the defined micros of interrupt vectors. The ones you may be interested in are picked out and listed in Table 2. When you install an ISR to one of the vectors listed in the left column, your ISR will be invoked when the timer counts to the corresponding CCR(s) listed in the right column.

Table 2:

Vector Micros	Triggering Registers
TIMER0_A0_VECTOR	TA0 CCR 0
TIMER0_A1_VECTOR	TA0 CCR 1 - 6
TIMER1_A0_VECTOR	TA1 CCR 0
TIMER1_A1_VECTOR	TA1 CCR 1 - 6
TIMER0_B0_VECTOR	TB0 CCR 0
TIMER0_B1_VECTOR	TB0 CCR 1 - 6

4 Submission

Please zip everything before submission and only upload the package. Do not upload every single file separately. The content of the package should be properly organized. Particularly, you should put everything under a directory named *username*_`lab2`, where *username* is your Blackboard system login id. Under the directory, include

1. `timer1.c`: LED1 flashing program;
2. `timer2.c`: LED1 and LED2 flashing program.