Microcontroller System Design

ECE121-Lab2

Professor Peterson

James Huang

Collaborated with Aaryan Redkar, Peter Le, Justin

03/19/24

1 Introduction

This report only contains information on part 1 of lab 2.

1.1 Hardware Background

To set up the timers in the PIC32, we need to configure the right timer register and consult the datasheet.

Register 14-2: TxCON: Type B Timer Control Register

r-0	r-0	r-0	r-0	r-0	r-0	r-0	r-0
—	—	—	—	—	—	—	—
bit 31							bit 24

r-0	r-0	r-0	r-0	r-0	r-0	r-0	r-0
—	—	—	—	—	—	—	—
bit 23							bit 16

R/W-0	R/W-0	R/W-0	r-0	r-0	r-0	r-0	r-0
ON(1)	FRZ(2)	SIDL(4)	—	—	—	—	—
bit 15							bit 8

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	r-0	R/W-0	r-0
TGATE	TCKPS<2:0>			T32(3)	—	TCS	—
bit 7							bit 0

(Figure 1: Type B timer control register.)

There are two types of timers in the PIC32, type A and type B. Type A timers are 16 bits and type B are 32 bits, where it takes 2 16 bit timers to form. The datasheet provides the hardware control registers for configuring the timers, the TCKPS bits control the prescale for our clock. A prescale is basically determining how fast we want our clock to run. We use PBCLK to drive our timers, which are set at 40MHz, by using the prescale we can change the frequency to our desire. The PIC32 has a TMRx and PRx register for the timers, the TMRx timer will count until the set value of PRx and then it will rollover and start again. Each time TMRx equals PRx, an interrupt is generated. Just like the previous lab, interrupt configuration can be found in the datasheet.

2 Lab2

2.1 Lab Part 1 Introduction

The first part is to initialize the type B timers on the PIC32. We want to be able to count in milliseconds and microseconds, with a function that returns the elapsed time in both units. By having the timers set up, we are able to count accurate time in our program. For example, we can blink an LED every 2 seconds exactly.

2.2 Lab Part 1 Initial Approach

The timer datasheets provided useful information on how to configure the type B timers and set the prescale. We are able to set TMRx and PRx in our timer initialization function, which determines how fast it rolls over. An interrupt must be generated to count time, that means we need an ISR to handle interrupts. This should be straightforward as we can set TMRx and PRx to rollover at desired time, and it generates an interrupt to tell us that time is reached.

2.3 Lab Part 1 Implementation

First thing in our timer initialization, we want to stop and clear current timer registers. Then we set the desired prescale and our TMRx and PRx values. These values will determine the rollover time. For our case, if we use 1:1 prescale, that means the clock runs at 40MHz, and each tick is 25nS. In order to achieve our 1mS rollover time, it would take 40,000 ticks or 40,000 25nS to get to 1mS. That means our PRx value has to be 40,000, and TMRx is 0, the timer would tick 40,000 times and then generate an interrupt which is exactly 1 mS. Then we enable the interrupts and lastly, we turn on the timer.

```
void FreeRunningTimer_Init(void){
    T5CON = 0; //clear timer 5 register
    T5CONbits.ON = 0; //stops clock

    //T5CONSET //stop timer
    T5CONbits.TCKPS = 1; // 1:2 prescale

    TMR5 = 0;
    PR5 = (PBCLK / freq) >> 1; //rollover every mS

    IEC0bits.T5IE = 1; //interrup enbale
    IFS0bits.T5IF = 0; //interrupt flag
    IPC5bits.T5IP = 3; //set priority


    T5CONbits.ON = 1; //turn on clock

    //T5CONSET = 0x8000; //start timer
}
```

(Figure 2: Timer initialization.)

However, so far we have only been able to count 1mS, and not 1microS, in order to do so, we need to do some math. We will have a variable keeping track of how many mS has elapsed, and in order to count accurate microS, we have to take the elapsed mS, convert it to microS and add the remaining microS that is still currently counting in the TMRx register. For example, let's say the mS variable is at 3, and the TMRx is currently counting at 20,000 out of the 40,000 ticks. That means the current time should be 3.5 mS. however the mS variable only increments when a full mS has passed. So we take the mS variable and multiply it by 1000 to get microS, and add it

to the remaining TMRx timer divided by 40. If we have 20,000 in our TMRx register, dividing it by 40 will give us 500, which is 500 microS or 0.5 mS. so the total microS will be 3000 plus 500 and that will be 3500 microS or equal to 3.5 mS.

```
unsigned int FreeRunningTimer_GetMicroSeconds(void){
    return ((mSTimer * 1000) + TMR5 / 20);
}
```

(Figure 3: Microseconds function, but the prescale is 1:2 so it's divided by 20.)
Lastly, the ISR is simply incrementing the mS variable everytime TMRx is equal to PRx and an interrupt is generated.

3 Conclusion

3.1 Summary

I was not able to get far into this lab, however this first timer part was interesting. It felt familiar to other timers we have worked with in ECE13 and its interrupt format. This taught me how to work with the different types of timers in the PIC32 and how to set it up correctly. It also taught me how to convert units with one single running timer, and be able to accurately represent any time units. I wish I had more time to advance to the rest of lab 2.

3.2 Challenges

Overall, this part of the lab wasn't difficult, however the first time I went through it, I did not know that when TMRx is equal to PRx it automatically generates an interrupt. I thought I had to code an interrupt generation. But after clarifying with the TA, itt became so much easier and not hard to understand.