Seth Cram

ECE341

Lab4 Prelab

09/28/2021

**Prelab 4: PIC32 Timers and Multi-rate Scheduling**

Goal:

To understand PIC32 timers, their interrupt flag, and its use in how they can be used for multi-rate scheduling.

Background Information:

Timers operate at 10MHz since they are connected to the peripheral bus clock (PBCLK). Since Timer1 is a Type A timer, it’s a 16-bit timer, like the other timers, but with prescale values 1, 8, 64, and 256. A higher prescale value allows us to divide the incoming clock frequency more. This results in the timer only counting 1 tick every ‘x’ clock cycles, with ‘x’ as the prescale value. After turning them on, timers will continue counting until the value loaded into their period register (PRx) is reached. At this point, they’ll set their corresponding timer interrupt flag (TxIF) and start counting again from zero. Care needs to be taken so that the period register is loaded with one less than the number of timer ticks needed, since all timers start counting at zero. Care also needs to be taken to reset the timer flag every time we acknowledge it as set.

Multi-rate processing is when all tasks aren’t executed on the same interval. It’s accomplished by using a timer that counts at a set rate and we execute tasks based on the rate at which its corresponding interrupt flag is triggered. When these interrupt flags are triggered, we decrement counters we use to time events.

Plan:

Making use of the code from our previous lab, I’ll modify it to use the PIC32 interrupt flag from Timer1 and counter variables for multi-rate scheduling. First, I’ll need to OpenTimer1(). I placed this in system\_init(), since it’s a part of initializing the system for this project. I made sure to turn the timer on with a prescale value of 1, and with a period register value of 9999. I got the 9999 from the Lab4 handout, where it was calculated using toggles per second, the prescale value, and the peripheral clock frequency. Instead of hardcoding 9999 in, I’ll calculate it in my header file as a constant for readability. Next, we should initialize our two counting variables to zero, so they enter our below discussed conditionals and their values are set from the get-go.

With our system setup, we can enter the while(1) loop. First, we should check if the button counter has reached zero, if so we’ll read and decode the buttons, toggle LEDB, and reset our button counter to 100. Our ‘read\_buttons()’ function is the exact same from the last lab. Within ‘decode\_buttons()’, we’ll have to set the step delay differently. In the lab handout, it specified the different speed of the motor for each button combination. So, we’ll use the formula T\_delay (ms/step) = 60000 (ms/min) / (X rev/min \* 100 steps/rev \* MODE) when MODE is 1 for full stepping and 2 for half stepping, to calculate what we need to set the step delay to in each case. This step delay value is available in main(), since we passed it to decode buttons by reference.

Next in the while(1), if our motor counter is equal to zero, we’ll use our finite state machine and motor output functions the same as we had in the previous lab, toggle LEDC, and reset the motor counter to the step delay found in ‘decode\_buttons()’.

Finally, at the bottom of the while(1) loop, we’ll call ‘Timer1\_delay()’ and pass both our counter variables to it by reference. This function will wait for 1 ms, which is until the Timer1 interrupt flag is triggered, and then clear the interrupt flag, toggle LEDA, and decrement both counter variables. Since we passed in the counter variables by reference, dereferencing them and assigning them a new value will change their value back in main() as well.

This will have the overall effect of reading the buttons once every 100ms, and outputting code to the stepper motor once every 20-40ms depending on the mode and speed of the stepper motor. Therefore, our goal of implementing multi-rate scheduling using Timer1 and its interrupt flag will have been met.



