9/10

ECE 341 Lab 5

Zachary DeLuca

February 21st 2023

1 Introduction

In this lab we accomplish the same thing as the previous lab, but using a different scheduling scheme this time. Last lab, we controlled a motor using a loop that checked the button output each time a certain amount of cycles had been completed and wrote to the motor each time the delay-number of cycles had been completed and delayed the activation of the second if statement based on how fast the motor was meant to spin. This time, there is no loop, just interrupts. The Timer1 ceaselessly counts down creating a ms pulse to be used set the Timer1 interrupt. There are two interrupts in this lab, the time based interrupt and the input hardware interrupt. The time interrupt is responsible for the speed of the motor, and the input, or ”change notice” interrupt is responsible for the system inputs.

✓

Use two "``" in LaTeX.

2 Implementation

There were only a couple of relevant changes to the previous lab top produce this setup. As the main function was nothing but an empty loop and a calling of the initialization functions, it shall be omitted from the listings. The initialization functions are shown below.

Listing 5.1

```
void system_init(void){
     // Setup processor board
     Cerebot_mx7cK_setup();
     PORTSetPinsDigitalOut(IOPORT_G, LED1|LED2|LED3|LED4);
     PORTSetPinsDigitalOut(IOPORT_B, SM_LEDS);
     LATBCLR = SM_LEDS; /* Turn off LEDA through LEDH */

     INTEnableSystemMultiVectoredInt(); //done only once
     INTEnableInterrupts(); //use as needed

     timer1_interrupt_initialize();
     cn_interrupt_initialize();
}
```

Listing 5.1 simply shows that we initialize the board like normal, but now we also have to initialize a couple of other things. The multi-vectored interrupts initialization means that we are enabling interrupts on the board from multiple sources. We then enable the interrupts so they can work. We then run the initialization for the interrupts when this overall function is called in main.

The next two listings are the prepackaged code that initializes both the timer interrupt and the change notice interrupt, enabling the change interrupt on the buttons.

Listing 5.2

```
void timer1_interrupt_initialize(void)
{
    //configure Timer 1 with internal clock, 1:1 prescale,
    //PR1 for 1 ms period
    OpenTimer1(T1_ON | T1_SOURCE_INT |
    T1_PS_1_1, T1_INTR_RATE-1);
    // set up the timer interrupt with a priority of 2,
     //sub priority 0
     mT1SetIntPriority(2); // Group priority range: 1 to 7
     mT1SetIntSubPriority(0); // Subgroup priority range: 0 to 3
     mT1IntEnable(1); // Enable T1 interrupts
}
```

Part of this initialization is setting the interrupt priorities, which is important to keep the motor spinning. If the interrupts were of the same priority, then the timer interrupt would not be able to be invoked during the change notice routine.

Listing 5.3

✓

```
void cn_interrupt_initialize(void)
{
    unsigned int dummy; // used to hold PORT read value
    // BTN1 and BTN2 pins set for input by Cerebot header file
    // PORTSetPinsDigitalIn(IOPORT_G, BIT_6 | BIT7); //
    // Enable CN for BTN1 and BTN2
    mCNOpen(CN_ON,(CN8_ENABLE | CN9_ENABLE), 0);
    // Set CN interrupts priority level 1 sub priority level 0
     mCNSetIntPriority(1); // Group priority (1 to 7)
    mCNSetIntSubPriority(0); // Subgroup priority (0 to 3)
    // read port to clear difference
    dummy = PORTReadBits(IOPORT_G, BTN1 | BTN2);
    mCNClearIntFlag(); // Clear CN interrupt flag
    mCNIntEnable(1); // Enable CN interrupts
    // Global interrupts must enabled to complete the
    //initialization.
}
```

The more interesting part of the code is the ISR's themselves. These are defined like functions, but are not ~~functions themselves~~. They are invoked, not called and because of this they have no function prototypes. As it has no prototype, it would not be able to be called if tried. Because it looks like a function, special syntax is required for the compiler to understand that it is an ISR and not a function.

Listing 5.4

```
void __ISR(_TIMER_1_VECTOR, IPL2) Timer1Handler(void)
{
    LATBINV = LEDB;
    if(!stepWait){
        LATBINV = LEDA;
        shoe = stepperState(dir,mode);
        stepperPush(shoe);
        stepWait = step_delay;
    }
    stepWait--;
    mT1ClearIntFlag(); // Macro function to clear the interrupt flag
}
```

Listing 5.4 shows the timer ISR which houses the stepper motor output code. As the interrupt is brought up every millisecond, there has to be a way of knowing when to write to the motor. For this global variables were used to store the counter variable to see if it was time to spin the motor. This is very reminiscent of the multi-rate scheduling of lab 4.

Listing 5.5

```
void __ISR(_CHANGE_NOTICE_VECTOR, IPL1) CNIntHandler(void)
{
    LATBSET = LEDC;
    Timer1_delay(20);
    literate = readButtons();
    decodeButtons(literate);
    LATBCLR = LEDC;
    mCNClearIntFlag(); // Macro function
}
```

Listing 5.5 shows the same code used in the labs before to poll the buttons and produce code meant for the motor writing functions. The only major difference is that there is a flag to be reset at then end of the routine, and that there is a debounce period incorporated in the routine. The 20s delay keeps the routine from reading erroneous values as the contact plates resolve after bouncing upon being pushed.

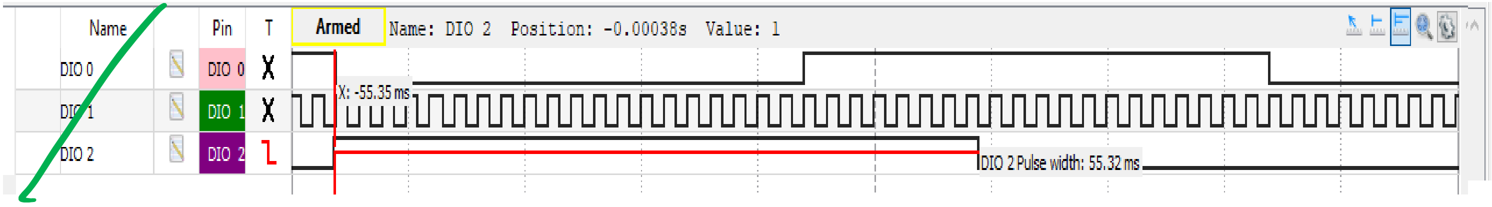
3 Testing and Verification

For the testing and verification part, there were LED's used, motors spun and waveforms analyzed. The LED's used were some on the stepper motor board to show what was being written out, and also the LED's on the cerebot board incorporated in previous labs to verify that the buttons were indeed being pushed.

The last thing done was producing the following waveforms showing the length of each of the pulses listed above the figure.

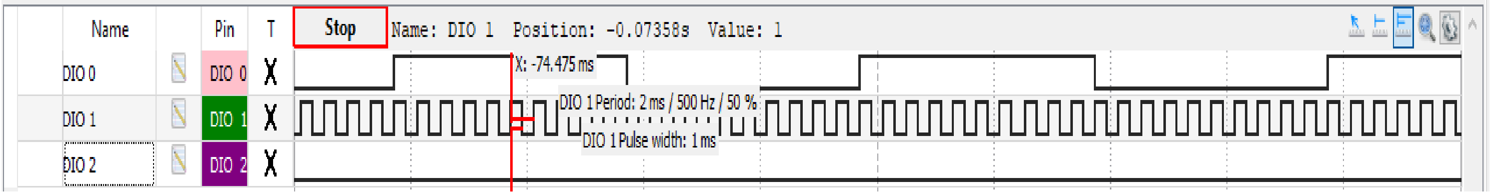
lecker! :-)

Measurement of the change notice process



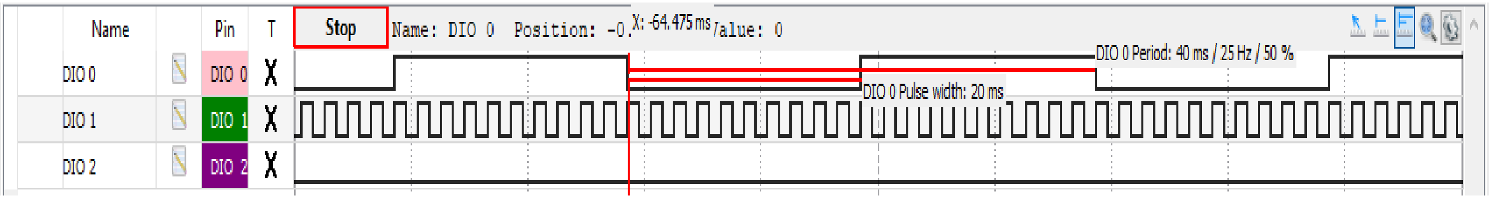
Name: DIO 2 Position: -0.00038s Value: 1

Measurement of the millisecond pulse



Name: DIO 1 Position: -0.07358s Value: 1

Measurement of step duration



Name: DIO 0 Position: -0. Value: 0

Looking at the waveforms for this lab and the lab previous, the accuracy is about the same as the level of precision is low enough that the waveforms always read the same.

4 Conclusion

In this lab we were able to implement an interrupt exclusive stepper motor controller. This foray into interrupts allows us to compare the methods used in this lab as compared to the labs previously. The interrupts were an improvement upon the single and multi rate scheduling in terms of resources and response time, as the constant polling model requires a cycle to complete and return to the polling section before it can utilize the new read values. The interrupts on the other hand, allow the data to only be collected when something has changed and we aren't wasting space in the background processes constantly checking the values of the buttons. Unfortunately, this requires a system that can host multi-vectored interrupts and as I have not worked with other systems I don't know if that is a universal feature, making that something that would detract from portability. The other downside is that the change notice hardware can not distinguish between the inputs and so additional code is required to sift through the buttons that would already be included in the constant polling schedule. For a simple process like this though, the interrupts seem like an improvement.

well, given how small transistors are, I suspect pretty common, but I don't know about Arduinos.