ECE341

**Lab5 - Interrupts and ISRs**

Report

**Seth Cram**

**10/12/2021**

**Introduction:**

Goal:

Explore the detection of events utilizing interrupts with nested interrupt management for preemptive scheduling.

Background Information:

We will refer to tasks in the instruction service routine (ISRs) as foreground tasks, and instructions in the while(1) loop as background tasks since they’re only completed when foreground tasks don’t need to be. An ISR accepts no arguments, returns no arguments, and can’t be called from code. So, it makes sense that we don’t need a prototype for our ISRs. They must be associated with a specific source, and use an incredibly specific syntax to tie an ISR to an interrupt flag. The interrupt controller is what puts the ISR’s memory address in the program counter (PC) to get the CPU to vector off to completing tasks within an ISR. It also gives the processor the priority level of the interrupt and the shadow set number. The contents of an ISR include the prolog, user code, and epilogue. Both the prolog and epilogue are added by the compiler. The prolog stores the context of the processor from where it vectored off of completing background tasks, while the epilogue restores it. Saving processor context consists of the vectored off from memory address, and usually necessary processor registers. These are all things the processor needs to resume its normal functioning once the ISR is complete.

We can assign priority to interrupts (1-7) so that higher priority interrupts can occur during lower priority ISRs, but only during the user code section. Interrupts also have settable subgroup priority and non-settable inherent priority to resolve ties. We can’t service an ISR during the prolog or epilogue since interrupts are disabled. This possibility of servicing other ISRs during a lower priority ISR introduces non deterministic latency involved when servicing an ISR. Using ISRs will also eliminate our need for “polling” the timer1 interrupt flag. We’ll also have less overall latency because hardware will be monitoring our events instead of us polling them. Other sources of latency include the interrupt controller and the prolog.

Because of the shallow register involved in the change notice interrupt flag triggering, we will need to read the port to clear the pre-existing difference. The Change notice interrupt has many pins associated with it. We will use two of them in this lab. Each pin has its own enable, so we’ll have to enable the ones we use. All the change notice pins do share a single change notice interrupt flag though. Button 1 connected to PortG pin 6 is also attached to change notice bit 8, and button 2 connected to PortG pin 7 is also attached to change notice bit 9. So, these are the change notice pins we’ll need to enable.

Another new aspect of this lab is our usage of global variables. Using these is the method introduced to us in order to pass information back from ISRs, since they can’t directly return information.

*– The majority (if not all) of the discussion in this section should be on interrupts:*

*∗ How are interrupts invoked? Can you “call” an ISR like a typical C function? Why not? What happens if you try to?*

Interrupts are ‘invoked’ by the interrupt controller putting the corresponding ISR’s starting memory address in the program counter. No, you can’t ‘call’ an ISR like a normal C function because they aren’t C functions. They don’t have a return type, take any arguments, and therefore can’t be called from code. When I tried to call an ISR from my code like a typical C function, I tried building the program and it spit out an error saying how there were “conflicting types” for the called ISR. This is most likely because for a function to be called from code, it can’t be specified as the ISR type.

*∗ Why is it necessary to include the special syntax for declaring an ISR?*

So the interrupt controller knows that it’s an interrupt service routine, which source it's associated with, and the priority of the source. So, when building the program it also isn’t allowed to be called from code with its special ‘ISR’ typing.

*∗ If multiple interrupts are triggered at the same time, what determines which is processed first?*

Their priorities. As stated earlier in this section, we can assign priority to interrupts (1-7) so that higher priority interrupts can occur during lower priority ISRs, but only during the user code section. Interrupts also have settable subgroup priority and non-settable inherent priority to resolve priority ties.

*– Explain when a section of code must be protected from interrupts. What is this section called? How do you implement this protection? What might be the consequences of such a section if it is entered after system initialization? ∗ Hint: Think about a system that has multiple priorities, and a lower-priority ISR enters one of these protected sections. Will the higher-priority ISR execute at the right time?*

A section of code must be protected from interrupts when it's saving, or restoring the processor’s context. We can’t service an ISR during the prolog or epilogue since interrupts are disabled and they’re storing and restoring context respectively. This interrupt protection can be implemented by disabling all interrupts at the start of a section, and re-enabling them at its end. The higher-priority ISR will still be executed first over a lower-priority one, but neither will run until the protected code section is exited.

*\* What is the term used for the situation when a lower-priority task “blocks” a higher-priority task?*

This is referred to as “priority inversion” and arises as an issue when non-nested interrupt schemes are used.

*The background information came from the lecture notes and “Lab5” handout.*

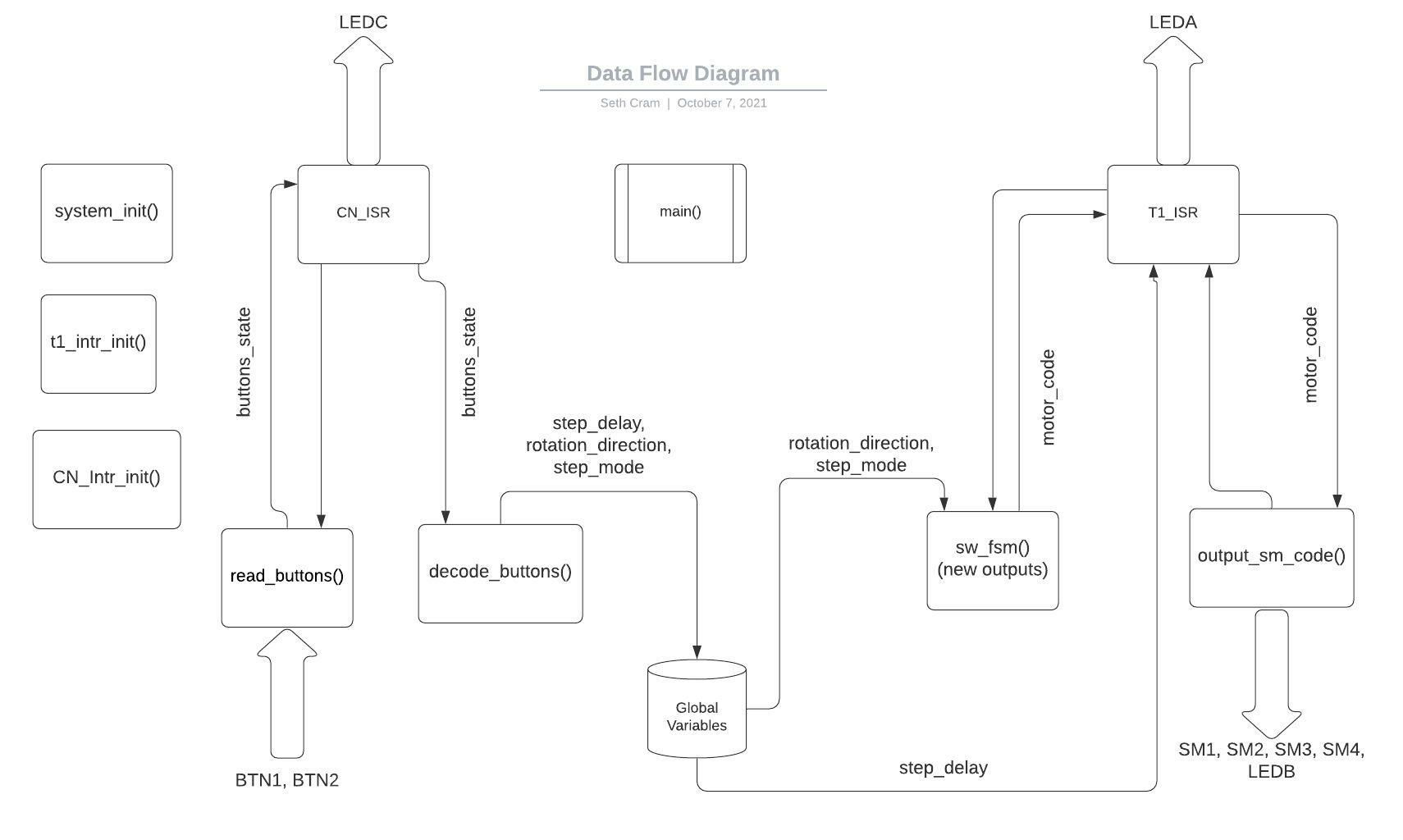
Plan:

Firstly, we’ll need to implement the timer1 and change notice initialization functions given to us in the lab5 handout. We only call these after system\_init(), but before the empty while(1) loop. Since events are now detected by interrupt instead of polling with a software flag, our code previously within it will need to move outside of the while(1) loop. In the change notice ISR, we’ll “debounce” the buttons by delaying for 20ms with a software delay, then read and decode our buttons, and finally clear the change interrupt flag. In the Timer 1 ISR, we’ll decrement the step counter and take a step if it’s time to by running our sw\_fsm(), outputting that code to the stepper motor, and finally resetting the motor\_cntr to the step\_delay we calculated in decode\_buttons(). Also in the Timer1 ISR, we’ll clear the timer1 flag at its end.

Retrieving step\_delay from decode\_buttons() is only possible through making it a global variable. In order to run sw\_fsm(), we also need our stepper motor direction and mode variables set in decode\_buttons, so these also need to be global variables. Along that same vein, in order for our motor\_cntr to be decremented every time the timer1 ISR is run, we’ll need to make this a static variable initialized at zero. Otherwise, it’ll be reset between ISR runs.

**Implementation Discussion:**

Before implementation, I designed a data flow diagram to get a visual of what functions I’d need to design or modify.



As seen above, this data flow diagram varies greatly from any we’ve done in the past. As a result of using interrupts and an empty while(1) loop, no data is passed to/from main(). All data is passed between the ISR’s, their functions, and global variables. From the prelab, I was able to minimize my number of global variables from 5 to just 3. I also implement two new initialization functions, which I’ll go through in my listings after the new header file.

**Listing 1. Proj5.h:**

I re-added the software delay macro found during lab 3 to implement the 20ms delay necessary to debounce our buttons before reading them. I also turned this 20ms delay into a macro for better readability. Then, I specified the three global variables I’ll need to pass information in-between the ISRs. Finally, I added in all the prototypes necessary for my new functions.

*/\* Software timer const \*/*

*#define COUNTS\_PER\_MS 8889*

*//Debounce btns const:*

*#define DEBOUNCE\_TIME 20 //debounce btns for 20ms*

*//global variables:*

*unsigned int step\_delay, dir, mode;*

*//interrupts initialization functs:*

*void t1\_intr\_init(void);*

*void cn\_intr\_init(void);*

*void sw\_msDelay(unsigned int mS);*

**Listing 2. New Structure of Main():**

Firstly, I deleted any variables moved from main to the header file to specify them as global. That leaves the only local variable as the status of the buttons. As usual, we’ll initialize the system, but then we’ll call the timer1 and change notice interrupt initialization functions. Otherwise, we won’t be able to correctly service the interrupts. Then, we use the status of the buttons to store the read buttons, and code them using them and our global variables. The reason we need to read and decode the buttons before servicing any ISR is because without decoding the buttons, the step\_delay variable won’t be set until a button is pressed. The result of not initializing the step\_delay is that we never take a step and move the motor since the reset value is zero.

*//setup timer1:*

*t1\_intr\_init();*

*//setup change notice intrs for btn1 and btn2:*

*cn\_intr\_init();*

*//read and decode the btns so 'step\_delay' set:*

*btns = read\_buttons();*

*decode\_buttons( btns, &step\_delay, &dir, &mode );*

This simple initialization was difficult to model in my data flow diagram, so I left it out. After this, our while(1) loop is empty since we only have foreground tasks for lab 5.

**Listing 3. Addition to system\_init():**

Before enabling either flag as the source of an interrupt, we need to globally enable interrupts and set our interrupt handling to the multi-vectored interrupt scheme.

*//interrupt setup:*

*INTEnableSystemMultiVectoredInt();*

*INTEnableInterrupts();*

The mutli-vectored interrupt scheme allows for nested-interrupts using priority, rather than sequential interrupt execution.

**Listing 4. Interrupt Initialization Functions:**

Although I got most of this code straight from the lab 5 handout, I’ll explain its functionality and list it out. Firstly, for setting up timer1, we opened the timer the same as in lab 4, but we also set it as the source of an interrupt. We then used macros to set timer 1’s interrupt priority to 2, its sub-group priority to 0, and enabled it.

*//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\_TICK-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*

Then for the change notice ISR, we open it by turning it on and enabling change notice bits 8 and 9, which correspond to buttons 1 and 2 respectively. We then set its interrupt priority to 1, so it can be interrupted by the timer1 interrupt. Its sub-group priority is set to zero since we won’t use it. Finally, we have to read the port to clear the difference present on the shallow register. We do this with a dummy variable and finally clear and enable the interrupt flag to be sure of its correct initialization.

*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*

**Listing 5. Timer1 ISR:**

Now that we properly set it up, we can delve into the internals of the Timer1 ISR. Firstly, ISRs have a special type used to declare them and with that comes the source interrupt of the ISR and its priority level. Next, we define a static variable and set it to zero so we step at the correct time. We then flip LEDA for instrumentation every ms. Just like in the previous lab, we then check if the motor counter is less than or equal to zero, and if so run the FSM and output ot the stepper motor. Newly added, after outputting to the stepper motor we toggle LEDB for instrumentation. Finally, we reset the motor counter and clear the timer 1 interrupt flag for future timer 1 ISR servicing.

*//ISR for timer1: (triggered once every 1ms bc PR1 val)*

*void \_\_ISR(\_TIMER\_1\_VECTOR, IPL2) Timer1\_ISR(void)*

*{*

*static int motor\_cntr = 0; //declared static bc only needed in this ISR, zero so initialized below*

*LATBINV = LEDA; //toggle LEDA every ms*

*unsigned int sm\_code; //local var*

*motor\_cntr = motor\_cntr - 1; //decrement sm cntr*

*if( motor\_cntr <= 0 ) //if time to set motor*

*{*

*sm\_code = sw\_fsm( dir, mode );*

*output\_sm\_code( sm\_code );*

*LATBINV = LEDB; //toggle LEDB every motor write*

*motor\_cntr = step\_delay; //reset cntr*

*}*

*mT1ClearIntFlag();*

*}*

**Timer6. Change Notice ISR:**

First, we set LEDC high and use our software delay for 20ms to allow the button values to settle. Then, we read the buttons, decode them, clear LEDC, and clear the change notice interrupt flag. So, LEDC is only high for how long the ISR is running, which should be around 20ms.

*LATBSET = LEDC; //start of CN ISR*

*unsigned int btns; //local var*

*//debounce the btns for 20 ms:*

*sw\_msDelay( DEBOUNCE\_TIME );*

*btns = read\_buttons();*

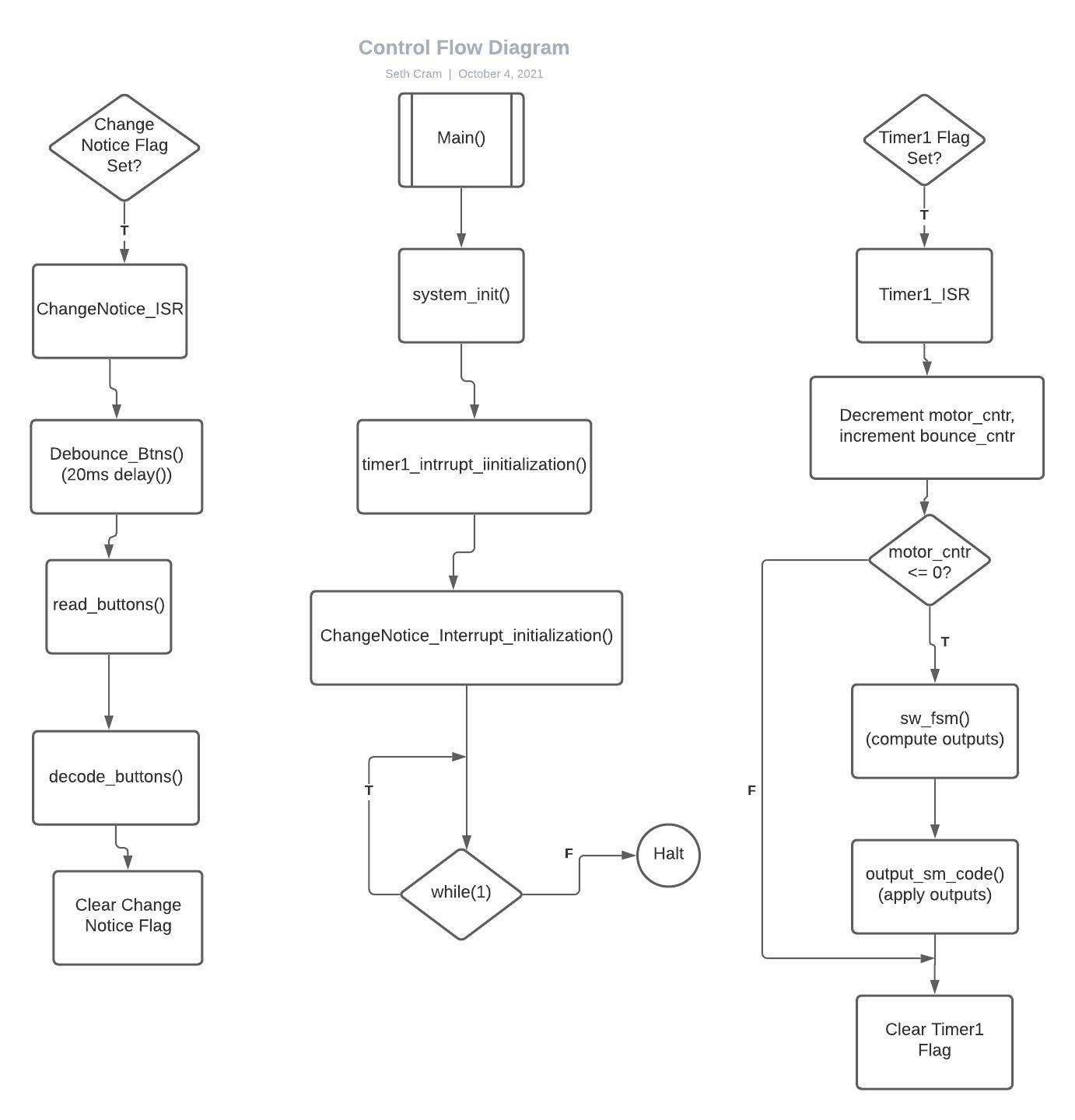
*decode\_buttons( btns, &step\_delay, &dir, &mode );*

*LATBCLR = LEDC; //end of CN ISR*

*mCNClearIntFlag();*

Global variables are essential here since we set them in this ISR, and use their values in the timer1 ISR.

Finally, my control flow diagram models the behavior of the above specified listings:



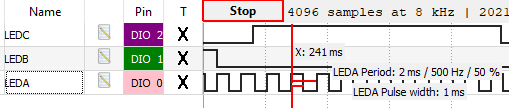
As a result of our use of preemptive scheduling, there’s no link between main() and the ISRs since either ISR can vectored off to at any point after interrupts are initialized. Although the diagram seems to model parallel processing, only one of these three paths can be executed at a time.

**Testing and Validation:**

For testing, I showcased to the TA how the stepper motor operates the same as in lab 4.

*– Use oscilloscope screen capture(s) to show the interrupt priority levels. Explain how the capture(s) indicate the correct nesting scheme. Hint: Trigger on the LED that is set during the CN ISR and change the button state; observe the behavior of the LED toggled in the Timer1 ISR.*

The Timer1 ISR has a higher priority, so I’ll trigger the Timer1 flag during the change notice ISR, and the processor should vector off to servicing the Timer1 ISR before finishing the change notice ISR. This is shown below in how even when the change notice ISR is run (LEDC is set and cleared), the Timer1 ISR is still run (shown with how LEDA is still toggled every ms):



*– Compared to the previous lab (Lab 4), is the stepper motor running at a more accurate speed? Why or why not?*

Strictly determined from the instrumentation LEDs, the stepper motor is running at the same accuracy for speed. Looking at the code, in lab 4 the buttons were always serviced before the stepper motor code was calculated and output. So, the buttons took priority here. Meanwhile, during this lab, the timer1 ISR took priority with a level 2 priority over the change notice ISR. In theory, this should increase our accuracy since the stepper motor code will be executed over the top of reading and decoding the buttons.

**Conclusion:**

Overall, this was an incredibly informative lab that taught about the implementation of preemptive scheduling using interrupts and interrupt service routines. I learned how to implement a counter in an ISR without using a global variables, and how nested interrupt schemes function.

A limitation of my design is the lack of precision for the debouncing of the buttons. I used the pure software delay when I could have just as easily used the timer1 interrupt flag like we did in lab 4. The reason for the pure-software delay is that I understood its internals better so I thought I’d be able to debug it quicker if an issue arose.

*- Discuss the advantages and disadvantages of a real-time control system that uses polling versus an interrupt-based system. When might you choose one over the other?*

An interrupt-based system is less deterministic regarding its latency, but it allows the processor to execute background tasks until an interrupt is serviced. Meanwhile, polling requires the processor’s full attention, but it doesn’t require use of more hardware like interrupts do since polling can be purely software-based. Polling is also far more deterministic. So therefore, when timing needs to be known with absolute certainty, polling should be used. Polling should also be used on smaller projects since the complexity interrupts introduce isn’t trivial. Otherwise, interrupts should normally be used if we have access to the interrupt controller.

*– What is the worst-case latency for a system that uses polling? What about an interrupt-based system?*

The worst case latency for a system using polling is incredibly small since we’re constantly checking to see if the event is done. This makes polling deterministic and accurate since we know how long it’ll take with far more accuracy. In an interrupt-based system, the sources of latency include the interrupt controller, the prolog of the ISR, and the time taken by higher priority interrupts, which is unknown.