Seth Cram

ECE341

Lab5 Prelab

10/02/2021

**Prelab 5: Interrupts and ISRs**

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 nor 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 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.

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.



