EE 472 Lab 3

Winter 2015

Interrupts

University of Washington - Department of Electrical Engineering

Mitch Ishihara, Joshua Fromm, Tien Lee

Lab Objectives:

In this lab, you will learn to understand and work with hardware interrupts and use them to implement the distance sensor of the RoboTank system. You will also update your keypad code to run more efficiently by replacing polling with interrupt driven software.

The demo for this lab will be to first demonstrate that your keypad software has been updated and remains functional. You will also need to show that as your distance sensor approaches an object, the onboard LED blinks at an increasing speed.

Background:

As you learned in lecture, the majority of embedded systems rely on interrupt driven software. This is because embedded systems are integrated very closely with peripheral hardware and often want to respond only when a certain event occurs (a user pushing a button for example). If the processor were polling the button, it would consume power and processor cycles that could otherwise be used to do other work or go into a lower power mode. Instead what is typically done is the signal from the button is made an interrupt and the processor goes into sleep mode until it sees that interrupt. The processor handles the interrupt with a designated interrupt handler routine and then goes back to sleep. Although we won't be putting our system to sleep, interrupts are still important for allowing us to do multiple things concurrently.

As preparation for this lab, each individual in a group is expected to actively participate but it is encouraged for each member of your group to own and drive expertise on one of the following parts of the LM3S8962 datasheet:

1. Section 3: Cortex-M3 Peripherals, with a specific focus on the NVIC.
2. Section 9: General Purpose Timers.
3. Section 11: Analog-to-Digital Converter.

If your group is collectively comfortable with these sections, the lab should be pretty relaxed.

Step 1: Timer Interrupts

To start, we will implement a new way to blink the onboard LED. Rather than sit in a while loop and call the toggle led function, you must have an empty main loop, instead calling led toggle from a function that is periodically called by a timer interrupt service routine (ISR).

1.1: Interrupt Vector Table

Before we even begin setting up registers, it is important to understand exactly how a processor responds to an interrupt event. Each type of interrupt that can occur has a vector number associated with it (see LM3S8962.pdf table 2.9 on page 86). When an interrupt occurs the processor gets the vector number and goes to the vector table. The vector table is basically an array of pointers to special functions (in this case ISRs). The processor reads the address from the vector table for the associated vector number and jumps to that address.

To find the right function for a given vector number, the processor simply does something like this:

```
// declare a function pointer to interrupt_handler that
// takes no parameters and returns no parameters
void (*interrupt_handler)(void);

// assign the function pointer interrupt_handler the address of the ISR from the vector table
interrupt_handler = vector_table[vector_number];

// the processor calls ISR – note the processor calls the ISR, not your C code
interrupt_hander();
```

Of course this is all done automatically in hardware rather than software. What you need to do is initialize the vector table by storing the address of the ISR in the vector number slot for the peripheral's interrupt.

To do this, check out startup.c (included in the lab3 folder) and modify it so that the timer0 interrupt points to the address of a function you will write called Timer0IntHandler (remember to extern the function so that it can be called from outside the scope of the file).

1.2: Timer Setup and Handler:

Now that we've initialized the vector table, we'll move on to setting up the software we need for timer0. Begin by writing a function called timer0_init. This function needs to set up all the registers needed for timer0 to trigger interrupts at a continuous periodic interval. These registers include SYSCTL_RCGC1, TIMER0_CFG, TIMER0_TAMR, TIMER0_CTL, TIMER0_IMR, and TIMER0_TAILR. Additionally, you will need to tell the core that you want to trigger timer0 interrupts by setting the appropriate bit in the NVIC_EN0 register. Finally you can use the built in CPUcpsie() function to globally enable interrupts (note you can use CPUcpsid() to globally disable interrupts). With timer0_init written, you can now write Timer0IntHandler, which needs to simply toggle the LED pin and clear the timer interrupt using the TIMER0_ICR register. Note that if you do not clear the interrupt, the timer handler will be perpetually called **without** any delay.

Step 2: Distance Sensor:

2.1: Setup:

Now that you have set up a timer interrupt we'll use it to periodically sample the output of the distance sensor. Start by wiring up your sensor. The red and black wires are of course power (5 V) and ground respectively. The white wire should be plugged into the ADC0 header on the Stellaris board. The voltage on white wire will start at 0 when no object is in front of the sensor and increase as an object approaches. Although the ADC module of the CPU can be used in fairly complicated ways, we'll be sampling as simply as possible. Rather than have ADC interrupts, we'll simply request a sample and then read the value a little later. Write a function called ADC_init that sets up ADC0 to do this. You need only write to SYSCTL_RCGC0 and ADC0_ACTSS to set it up properly.

Next modify your timer event handler so that each timer event alternates between requesting ADC data and reading that data. To request data simply write to the ADC0_PSSI register. Data can then be read from the ADC0_SSFIFO0 register. Note that the value read will range between 0 and 1024, with 0 indicating a voltage of 0V and 1024 indicating a voltage of 3.3V. Modify your handler so that the LED turns on when a value of 512 or higher is read from the ADC (hand is in front of the sensor) and test that your code works.

2.2: Adding a second Timer Interrupt:

If you’ve made it to this point you’ve implemented a simple binary distance sensor that indicates if an object is close or not. Next, we will modify your code so that the rate the LED blinks indicates **how close** an object is. To do this, we will need one timer that samples the ADC at a fixed rate and another that blinks the LED at a variable rate. Repeat step 1 for Timer 1, including the creation of a Timer1EventHandler function.

Then, modify your Timer0EventHandler function so that it only deals with the ADC. Use the value read from the ADC to calculate a suitable value to write to TIMER1_TAILR to change how often Timer1EventHandler is called. Finally write the code needed for Timer1EventHandler to blink the LED.

Step 3: Updating Keypad:

Using what you’ve learned about timer interrupts, go back to your code from lab 2 and modify the keypad functions so that they use timer interrupts rather than delay loops. Your keypad functions and main loop should not have any delay loops and instead use timer events. Note that is ok to have delays in your display functions for now. To implement this properly, it is likely you will need shared variables (since they are persistent across interrupts).

Grading Guidelines

You are expected to demo your work to a TA, showing that it does all the required features along with anything extra you coded. Be prepared to explain how the software or hardware components used in the lab works as well as any issues encountered and solutions to the issues you encountered.

You will need to turn in a report comprised of the content described in this lab. Clearly indicate the contributions of each team member in the report. As always, you should also turn in your **thoroughly** documented source code.