ECE 3720

Microcomputer Interfacing Laboratory

Section 005

Aaron Bruner

Date Performed: 04 / 01 / 2021

Lab 10

**ABSTRACT**:

A lab designed to demonstrate how to program the PIC32 microcontroller to output a binary count depending on how hard we press a force-sensitive resistor. As we press the resistor harder the binary count increases. Like the previous labs, this will also enhance skills in reading documentations for new and existing parts.

**INTRODUCTION:**

The goal of lab 10 is to understand how to program our PIC32MX to display 8-bit data on LEDs using a force-sensitive resistor (FSR). The theory behind out program is that as will increase pressure on the FSR an interrupt is triggered. The interrupt will update the LEDs based on the voltage value. We are comparing to Vref+ and a Vref-; Vref+ will be the FSR input, 3.3V at idle, and Vref- is 0V, ground. We will be using StaticIO in the WaveForms software to view the output instead of actual LEDs. To perform our lab, we required a variety of materials including:

* A breadboard
* Wires
* Force-Sensitive Resistor (FSR)
* Resistor (x1 5.1k Ω)
* PIC32MX150F128D
* Analog Discovery 2 (AD2)
* Digilent WaveForms
* MPLAB Snap Debugger
* MPLAB X IDE (Programming Software)

The AD2 will power the PIC32 chip along with partially powering the MPLab Snap Debugger so that it can write to the PIC32. The majority of the power for the MPLAB Snap Debugger will come from the Micro USB cord.

**EXPERIMENTAL PROCEDURES:**

We must assume that the individual reproducing this lab has already setup their breadboard in a manner that their PIC32 chip can be programmed using the MPLAB X IDE software.

To begin we discuss how the circuit is setup in Figure 1. The top connection is the force-sensitive resistor setup. Based on the image provided in the lab manual:

Diagram, schematic

Description automatically generated

We applied a 3.3V supply to one connection of the FSR and then connected the other side to A0 (pin 2). There also is a pull-down resistor in the image above, instead of using the microcontroller’s built-in pull-down resistors we opted to use our own resistor with a value of 5.1kΩ. Lower resistor values didn’t give enough voltage which resulted in lower binary numbers so we increased it until we got to 200+ in binary. The next pin is A1 which is our Vref- pin. We simply connect this to ground since it needs to be 0V. The next 8 connections are the LED output. Instead of combining both A pins and B pins we opted to use B0-7 which is pins 4-7, 11, 14, and 15-16. The output from the PIC32 would normally go to physical LEDs with pull down resistors but instead we put it into the DIO pins of the AD2. We used DIO pins 0-7 for our LEDs, then changed it from LEDs to progress bar. This wraps up the setup for our circuit.

The code we wrote was based on the 15 steps of section 17.4 (ADC Module Configuration). We start by including our basic libraries and then disable the secondary oscillator and JTAG modules. We disable the secondary oscillator to that we enable the B4 pin for use. JTAG did not have to be disabled in the way that we did but it was necessary to disable it so that we could use B7 as an output. Next, the interrupt is setup; it is using vector 23 which is useful for our ADC 1 convert done trigger. We initialize an integer, FSR, which will store our binary value for the LEDs. Then, we take 4 buffer values and sum them up and divide by the number of buffers we are summing. This will get a somewhat close average of what our binary representation should be. We bit shift right by two since we are only using 8 bits of the 10-bit register. Lastly, we write the value after it is shifted to the B pins and then reset the interrupt flag. The main function is practically a copy of the steps in section 17.4. We begin by enabling interrupts and then setting up our pins. Pin B12 will be used for the FSR input so we set it to analog using ANSELBbits.ANSB12 = 1. Then, we set pin B12 to input and then the rest of the pins to output using TRISB = 0x1000;. The last step in setting up our pins is making the A pins analog and also input since we will only be using A0 and A1 for both Vref+ and Vref-.

Now, we start the 15 steps to setup the ADC module. We begin by setting up the Vref+ and Vref- for the ADC module. Using CH0SA and CH0NA we can set Vref+ to be the input for the ADC and Vref- to be V\_R-. Next, we set the form of the module to be a 16-bit integer using FORM = 0. Next, we make the internal counter end sampling and start conversion automatically. Next, we configure the reference voltage sources to use the internal sources (V\_DD and V\_SS); to do this we set VCFG = 0. Then, we set CSCNA to 0 so we do not scan inputs. SMPI is set to 4 as requested by the lab TA. This is the number of conversions per interrupt. The BUFM module is set to 0 which means the buffer is configured as one 16-word buffer. Next, ALTS is set to 0 which means that we use MUX A input for multiplexer setting. ADRC is set to 0 which sets the clock to be derived from the peripheral bus clock. SAMC is set to 12 which means we auto-sample time, also requested by the lab TA. We set our conversion clock to all ones which is 0xFF. Next, we turn on the ADC module and finally configure the A/D interrupt. We repeat the same steps we used in previous labs, enable the interrupt, clear the flag, set the priority and sub priority. Lastly, we enable auto sampling mode per requested by the TA.

**RESULTS and DISCUSSION:**

Once the microcontroller was programmed it was apparent that things were working as expected when we started to apply pressure to the FSR the progress bar in StaticIO was progressing towards being full. Then, when we reduced the pressure on the FSR the binary count decreased and as a result the progress bar in StaticIO decreased.

One problem during this experiment was that the 8 LEDs were constantly all lit up and did not change when the FSR was pressed. Even when the FSR was removed the LEDs would not stop being lit up. After multiple rewrites I found out that Vref+ will not be a reference voltage for 3.3V but instead the output voltage from the FSR. Thus, after removing the 3.3V line from the Vref+ input (pin 2) the LEDs turned off. Then, when I connected the FSR output to Vref+ the LEDs lit up correctly.

This week we demonstrated how to utilize a force sensitive resistor and the idea of a analog to digital conversion. The ADC converted the voltage from the FSR into a digital value which was displayed in binary on the microcontroller’s B pins. As pressure was increased the binary number increased.

**CONCLUSION:**

To conclude, lab 10 taught us many useful techniques on how to represent data from a force sensitive resistor on LEDS. We also continue to study how to implement interrupts on the PIC32 microcontroller. As with many other labs, this one teaches us how to debug our circuit to check if everything is working. This lab also taught us more complex ways to analyze problems and how to solve them.

Diagram, schematic

Description automatically generated**FIGURES AND TABLES:**

**Figure 1: Wiring for lab 10 (Pin connections described in experimental procedures)**

**CODE:**

#include <xc.h>

#include <sys/attribs.h>

#pragma config FSOSCEN = OFF //Disable secondary Oscillator (enable RA4 and RB4)

#pragma config JTAGEN = OFF //Disable JTAG (Enable RB7-RB9 & RB11)

void \_\_ISR(23)FSRinterrupt(void){ // Interrupt for FSR

int FSR = 0;

FSR = (ADC1BUF0 + ADC1BUF1 + ADC1BUF2 + ADC1BUF3) / 4;

LATB = FSR >> 2; //shifts bits to only get upper bits

IFS0bits.INT0IF = 0;

}

int main(void) {

INTCONbits.MVEC = 1;

\_\_builtin\_enable\_interrupts();

ANSELBbits.ANSB12 = 1; // Analog for FSR

TRISB = 0x1000; // LED Outputs & FSR Input

TRISA = 1; // Analog for Vref+ and Vref-

ANSELA = 1; // Vref+ & Vref-

AD1CHSbits.CH0SA = 0; // Make Vref+ the input for the ADC

AD1CHSbits.CH0NA = 0; // Set Vref- to be the V\_R-

AD1CON1bits.FORM = 0; // 16-bit integer

AD1CON1bits.SSRC = 7; // internal counter ends sampling and starts conversion (auto convert)

AD1CON2bits.VCFG = 0; // Use internal reference voltages (V\_DD and V\_SS)

AD1CON2bits.CSCNA = 0; // Module does not scan inputs

AD1CON2bits.SMPI = 4; // Number of conversions per interrupt

AD1CON2bits.BUFM = 0; // Buffer configured as one 16-word buffer

AD1CON2bits.ALTS = 0; // Always use MUX A input multiplexer setting

AD1CON3bits.ADRC = 0; // Clock is derived from peripheral bus clock

AD1CON3bits.SAMC = 12; // Set auto-sample time

AD1CON3bits.ADCS = 0xFF; // Set the conversion clock

AD1CON1bits.ON = 1; // Turn on the ADC module

IEC0bits.AD1IE = 1; // Enable A/D interrupt

IFS0bits.AD1IF = 0; // Clear A/D Interrupt Flag

IPC5bits.AD1IP = 1; // A/D Priority

IPC5bits.AD1IS = 1; // A/D Sub priority

AD1CON1bits.ASAM =1; // Auto sample mode

while(1){ // Run forever

}

}