Section 1: General Lab Info /0.5 points

Microprocessors 2

EECE-4800

Lab 1: Sensor Design and Analog Digital Conversion

Instructor: Yan Luo

TA: Ioannis Smanis

Group 10

Kyle Marescalchi

Handed in 10/2/17

Due 10/2/17

Section 2: Contributions */1 points*

1. Group Member 1 – Kyle Marescalchi (Me) …….……

My involvement in this project was largely focused on the pulse width modulation process and, more namely, understanding and adjusting the output of the pulse width module to create the output function. Working together with Hans, he and I came up with the effective way to output the pulse itself. Beyond such, my involvement largely consisted of debugging the code, cleaning the code up and making it consistent, adjusting the functions to be more efficient, and helping with cleaning the circuit and implementing the motor.

2. Group Member 2 – Derek Teixera …….……

Section 3: Purpose */0.5 points*

Derek handled working on the electronic portions of the circuit, working alongside Hans as they worked largely on the ADC part of the lab. He developed and explained the process in which the hardware would work, and had major involvement in debugging, design, and creating the ADC coding.

3. Group Member 3 – Hans Hoene…….…….

Hans' primary focus was constructing the base program and the functions related to the ADC. Through this, he developed the effective code and came up with the idea of using the counters to handle the cycles for both the LED and the PWM. Working together with Hans, we effectively implemented these into the design of the lab.

The purpose of this lab was to understand the process of creating a physical sensor using traditional electronics knowledge, and using the input from the sensor with an electronic kit. The lab demonstrated the process of approaching a microprocessor, adjusting its internal hardware to prepare the ADC conversion, then outputting signals to represent a pulse width module – a case of interfacing with an outside piece of equipment. Further, the overall goal of this lab was to learn how to design a microcontroller environment and sensor to act in tandem.

Section 4: Introduction */0.5*

This lab uses a photosensitive resistor to adjust a sensed voltage in a voltage divider; in using this voltage divider it was then sensed by the input of an ADC on the PIC-kit microcontroller. This ADC had to be initialized and adjusted in the program itself, which was programmed into the microcontroller. Depending on what voltage had been sensed by the ADC, the microcontroller then would send an active HIGH or an active LOW signal to an LED to turn it on or off. All throughout this time, a pulse width motor would actively cycle between facing left or right. This servo was controlled by pulse widths of either one millisecond or two milliseconds in duration, out of a 50hz frequency.

Section 5: Materials, Devices and Instruments */0.5*

- PIC-16F18857 Microcontroller operating at 3.3V
- PIC-Kit interface
- LDR Sensor (Photoresistor)
- Red LED
- 2x 10k-Ohm Resistors
- Servo Motor
- Breadboard
- Oscilloscope
- Analog Discovery
- Benchtop Power Supply
- Wires

Section 6: Schematics */0.5*

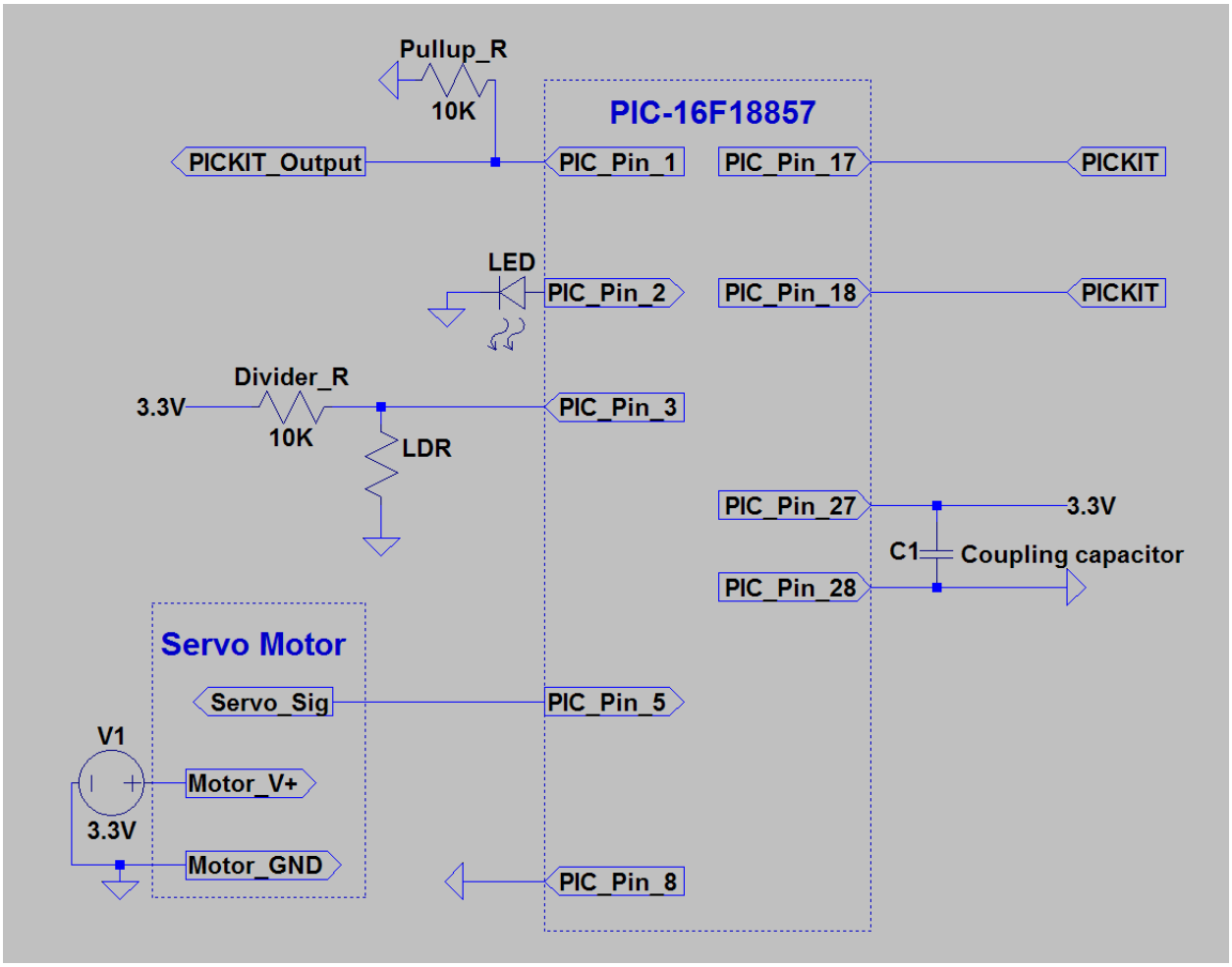

Figure 1: Circuit schematic.

Section 7: Lab Methods and Procedure /2

Hardware design:

The most basic principal in this circuit was that of a voltage divider, which was an inherent need in the construction of the sensor itself. The photoresistor is a device that, as light shines on the resistor element, its resistance lowers. To understand what this meant it was implemented into a circuit and tested for varying light levels – dark or bright. It was seen that the lower voltage across the resistor meant that the light levels were high. This led to a circuit design that was active high – when the voltage was high, the LED would turn on.

The LED was simply connected to pin two of the PIC16F18857 microcontroller, as it would not output a voltage or current high enough to short the circuit—leading to a more effective use of hardware. The servo motor was connected to an independent 3.3V power supply and ground, similar to the PIC microcontroller that was on its own power supply as well. A power coupling capacitor ensured for stable power to the kit.

The PIC-Kit itself had a grouping of wires that led to a separate series of 'ports' that were hard-wired to their corresponding pins on the microcontroller. This not only made for an easier setup, but also led to a neater and more clearly understood circuit. A 10k pullup resistor was also attached to pin 1 of the PIC-16F18857, serving to increase the voltage to the pin itself and more properly embed data.

Software design:

The flowchart for the software can be seen below. The program focuses largely on the use of counters (utilizing the internal clock) of the circuit to test when the programs should begin. The most important feature of the circuit is the analog to digital conversion. In order for the ADC to function, it was first initialized in the function ADC_Init(). This function can be seen in appendix A1. The purpose of this code is to initialize the Analog to Digital Converter built into the microchip, which begins by disabling all ADC ports except for ADCON1, the one specifically used. It was adjusted to be left-justified (meaning that ADRESH is used in comparing), the reference voltages were all set to zero, and the conversion module was set as accepted. Finally, the ADC-positive channel was selected to properly initiate.

In the main module, the ADC_Init function was called, variables were set, and the pins on the microchip itself were adjusted to be outputs or inputs. Using TRISXbits.TRISXY to adjust pins RA0 to output, RA1 to input, and RA3 to output, the microchip was properly 'attached' to the physical circuit itself. Then, the rest of the function is in a while loop – as the circuit was directed to act infinitely. In the PWM section, which came first, it checks to see if the rollover was matched. If so, it will then switch direction and reset the counter. If not, it remains to a low output and increments the duty counter. Following such is the LED cycle, which checks to see if a conversion is running. If it is, it skips the process and leds the conversion finished. If it is not, it checks the value of ADRESH compared to the threshold, and will adjust the output to pin 0 accordingly. This process loops infinitely.

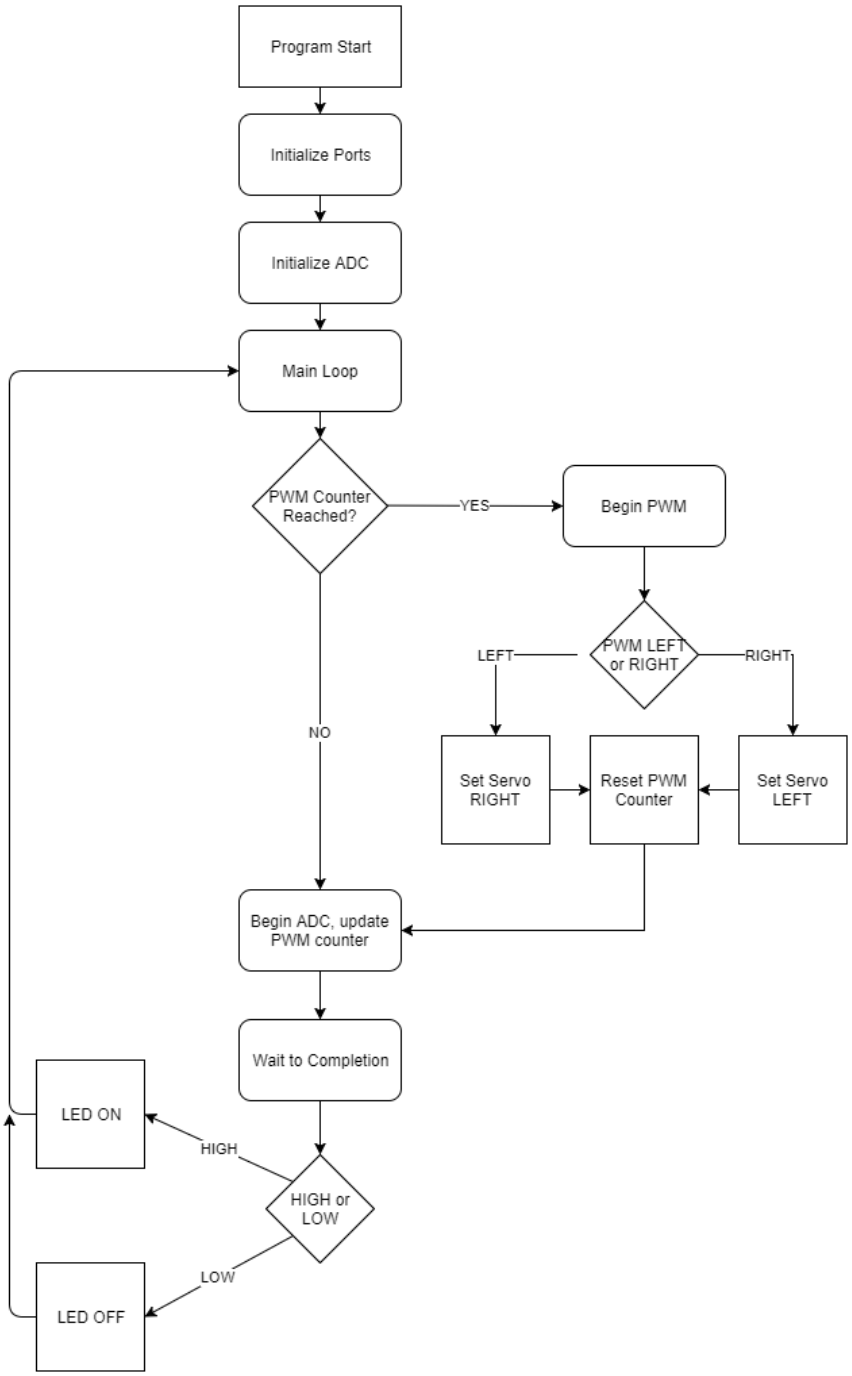

Figure 2: Schematic Flowchart

Section 8: Trouble Shooting */1*

Issue 1: Our primary issue was getting the ADC to initiate. We could not properly get the ADC to take in a value and begin the process of converting it, and this was found to be an issue with the internal register values set.

In the ADC_Init function, this issue was resolved by initiating the ADREFbits to set a V-Reference, selecting the channel in ADPCHbits, and adjusting the value for ADCON0bits to enable the ADC, supporting the clock to the Fosc, and disabling continuous operation. Through these settings the ADC properly functioned.

Issue 2: Another very large issue that we had was with the PWM module. Regardless of what our group did to initiate the PWM module, we would constantly receive a sinusoidal signal out on the oscilloscope – and it was not a clean one either. This led to the believe that the PWM module was not properly initiated. Regardless of what our attempts were (all three of us working on different solutions,) we did not come to a proper resolution.

This is what led to the use of the rollover function and a simple on/off of an output to represent the pulse width output. In doing this, the function became much simpler to perform. The values LEFT, CENTER, and RIGHT were given to determine the width of the signal. These need further tuning for further use, however, for as of right now they are only crudely working – the signal widths are not proper.

Issue 3: After testing the independent solution for the PWM output, there was a large issue where the ADC would not begin. This meant that only either the PWM or the ADC would be functioning at any given time.

This issue was solved by adjusting the main program. It was found that there were nested ‘while’ loops, which both resulted in a forever-situation that would disable one of the other counters under a certain circumstance of counters. These two statements – the PWM or the ADC blocks – needed to be both set into the same loop with independent counters.

Section 9: Results */0.5*

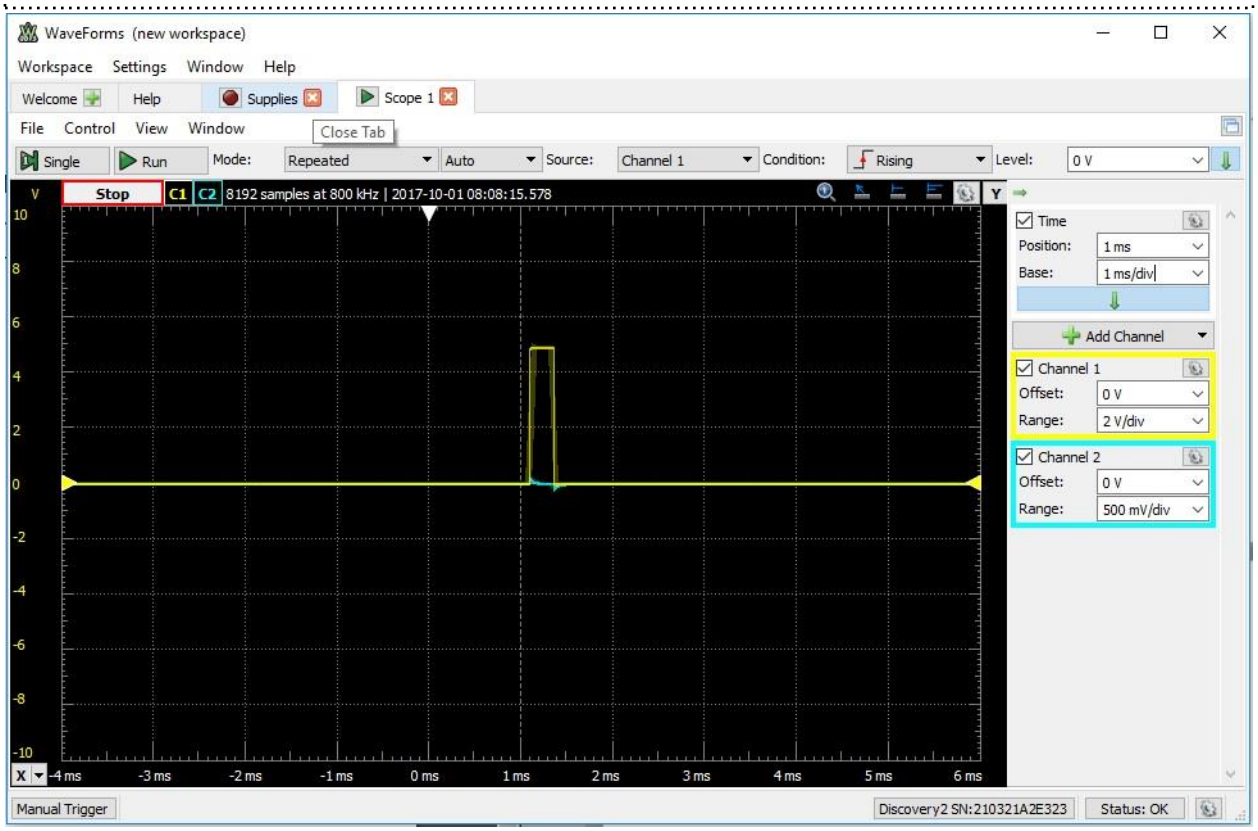

Figure 3: 1% pulse width.

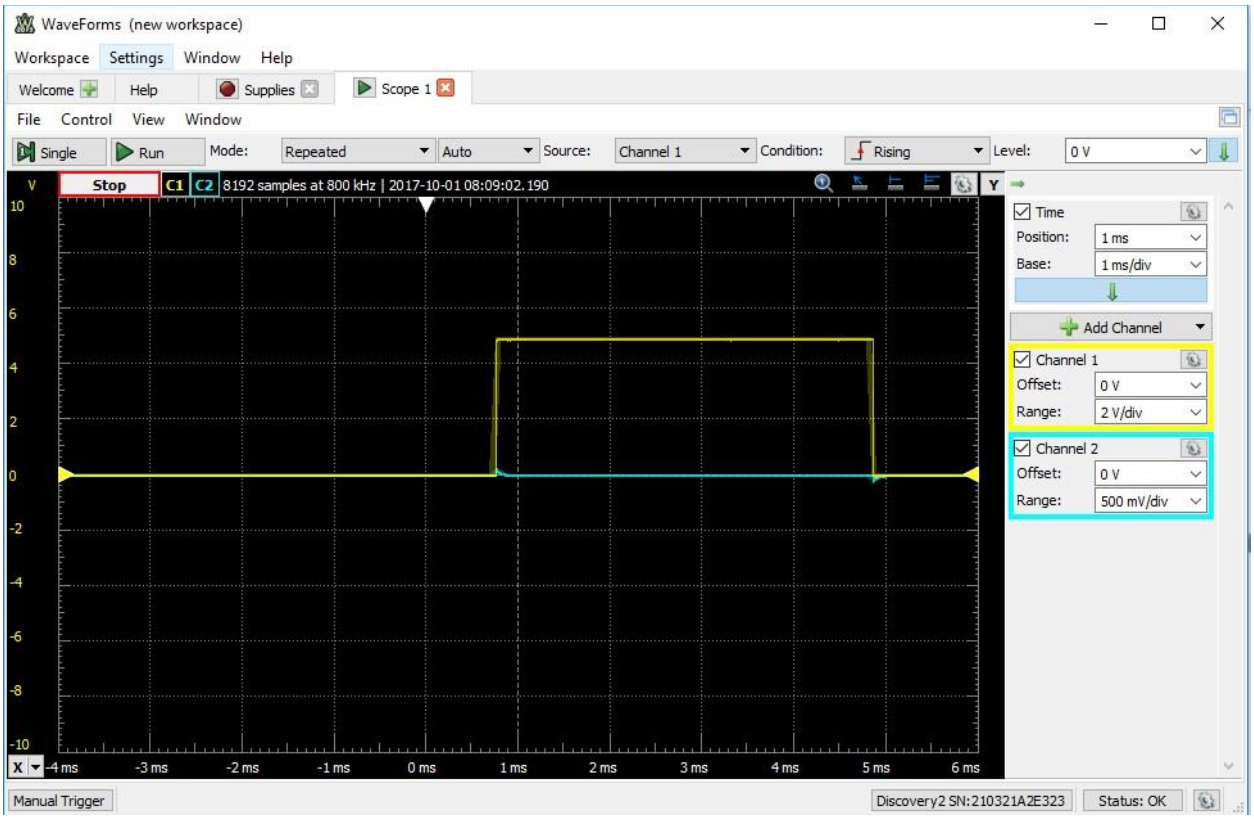

Figure 4: 15% pulse width.

It is important to note that these are *not* the pulse widths given to us by the lab materials, which called for either 10% or 20%. From 1% to 15% we were able to achieve the full range of motion, and in higher values we did not achieve a full range of motion – it was wavering from left to right instead of making the full sweep.

Section 10: Appendix

A1. ADC_Init(void)

```
  void ADC_Init(void) {
       //ADCON0 = 0;
    ADCON1 = 1;
    ADCON2 = 0;
    ADCON3 = 0;                     // set adc threshold reg to 0
    ADACT = 0;                      // disable adc auto conversion trigger
(is this how????????????)
                                    // 5 already disabled, genius
    ADSTATbits.ADAOV = 0;                    // I hope this works?
    ADCAP = 0;                      // 7
    ADPRE = 0;                      // 8

    ADCON0bits.ADFM = 0;                    // left justified alignment?
    // maybe ^ = ADCON0bits.ADFRM0 = 0???
    ADCON0bits.ADON = 1;                // adc enable
    ADCON0bits.ADCS = 0b101;                //  Clock supplied by FOSC,
divided according to ADCLK register? (0) vs. Clock supplied from FRC
dedicated oscillator
    ADCON0bits.ADCONT = 0;              // disable continuous operation

    //ADNREF = 0;                 //negative voltage reference: Vss
    //ADPREF1 = 0;                //positive voltage reference: Vdd
    //ADPREF0 = 0;                //positive voltage reference

    //ADCON0 = ADCON0 | 0x44;
    //ADCON0 = ADCON0 & 0XE7;

    //ADCON0bits.VCFG = 0;
    ADREFbits.ADNREF = 0;
    ADREFbits.ADPREF0 = 0;
    ADREFbits.ADPREF1 = 0;          //V references

    ADSTATbits.ADSTAT0 = 1;
    ADSTATbits.ADSTAT1 = 1;         //ADC conversion module
    ADSTATbits.ADSTAT2 = 0;

    ADPCHbits.ADPCH0 = 1;
    ADPCHbits.ADPCH1 = 0;           //ADC positive channel select
    ADPCHbits.ADPCH2 = 0;
    ADPCHbits.ADPCH3 = 0;
    ADPCHbits.ADPCH4 = 0;
    ADPCHbits.ADPCH5 = 0;
}
```