Section 1: General Lab Info */0.5 points*

Microprocessor Systems II & Embedded Systems

EECE.4800 – 201

Laboratory 2: Interfacing with a Sensor Device on an Embedded Computer System

Professor Yan Luo

Group #10

Hans-Edward Hoene

29 October 2017

Due Date: 29 October 2017

Section 2: Contributions */1 points*

1. Group Member 1 – Hans-Edward Hoene (Me)
   - Worked on PIC microcontroller code
   - Set up timing diagram and implemented communication protocol on both devices
   - Worked on Galileo user interface
   - Helped debug Galileo GPIO pin errors

2. Group Member 2 - Derek A Teixeira
   - Set up hardware
   - Researched operation of and worked on Galileo GPIO pins
   - Made diagrams for this laboratory report
   - Helped debug errors on the Galileo GPIO pins

3. Group Member 3 - Kyle W Marescalchi
   - Coded PIC PWM functions
   - Worked on Galileo pin functions
   - Debugged problems in the final setup of the circuit such as the strobe pin
   - Helped debug Galileo GPIO pin errors

Section 3: Purpose */0.5 points*

The purpose of this laboratory was to interface the code from the first laboratory from a computer. The first laboratory contained a light intensity sensing device and code to move the PWM, but this code was self-contained in the PIC program logic. In this laboratory, PIC functions from the first laboratory were authorised by an interfacing computer, the Intel Galileo board. To be specific, the Galileo was sending commands via a strobe protocol to the PIC. The explicit purpose of this laboratory is to use a program from the command line on the Galileo to communicate with the PIC and send commands as well as receive responses.

Section 4: Introduction /0.5 points

Interfacing an embedded system means providing a protocol and/or method of communication allowing other computers to access the system. In general, embedded systems, such as microcontrollers, are not user-friendly. There is no command line or graphical user interface. Instead, engineers design systems in a way such that they do what they need to do, but they allow accesses to data and/or ability to change parameters via interfaces. In this laboratory, the PIC microcontroller was interfaced through an Intel Galileo board via a command line program. The interface uses a strobe protocol where the Galileo is the master and the PIC is the slave. The Galileo, at any time, can interrupt the PIC in order to start a communication protocol. In this communication protocol, the Galileo sends a command code, which the PIC will execute, and the PIC will send back an appropriate response. The available commands that the Galileo may send the PIC are as follows: reset, ping, get, and turn. Resetting and pinging the PIC are self-explanatory. Sending a "get" command allows the user to get the latest command read from the analogue-to-digital converter (ADC), and sending a "turn" command allows the user to turn the senso-motor via pulse-width modulated (PWM) signals.

Section 5: Materials, Devices and Instruments /0.5 points

Device Name	Model Number	Description
PICkit	3	Programs microcontroller
PIC Microcontroller	PIC16F18857	Is a microcontroller. Runs code. Operating at 3.3 V in this laboratory.
Servo Motor	SG90	Motor arm controlled by PWM signals. 6 V
LED	N/A	Light-emitting diode
LDR sensor	N/A	Light sensor
GW Instek	GPD-3303D	Power Supply
Galileo Gen2	Intel Quark x1000	Will boot Linux and run terminal code to interface PIC

Section 6: Schematics */0.5 points*

Figure 1
Pin diagram for the Intel Galileo and PIC microcontroller (made by Derek)

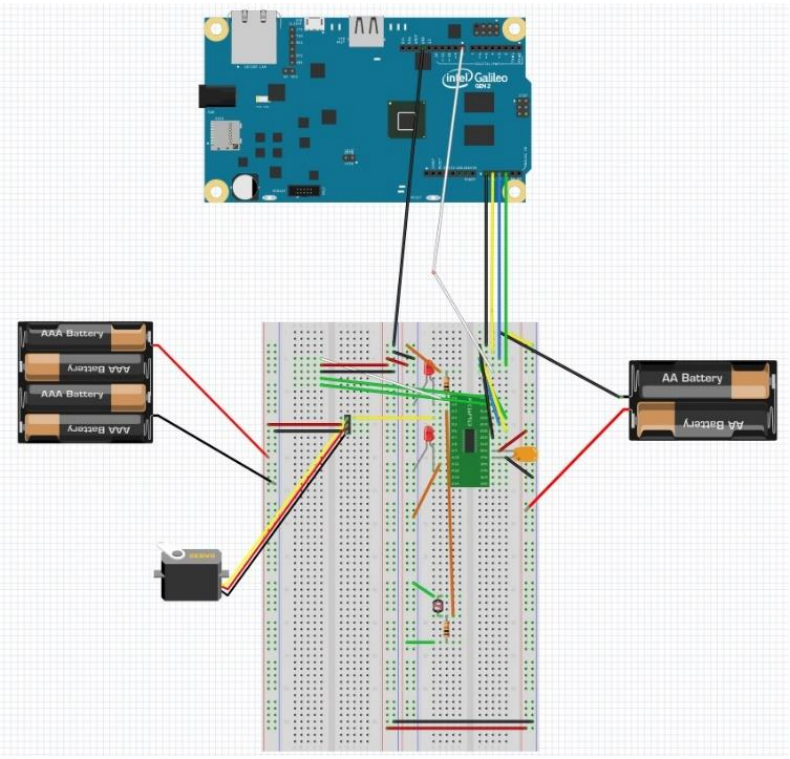
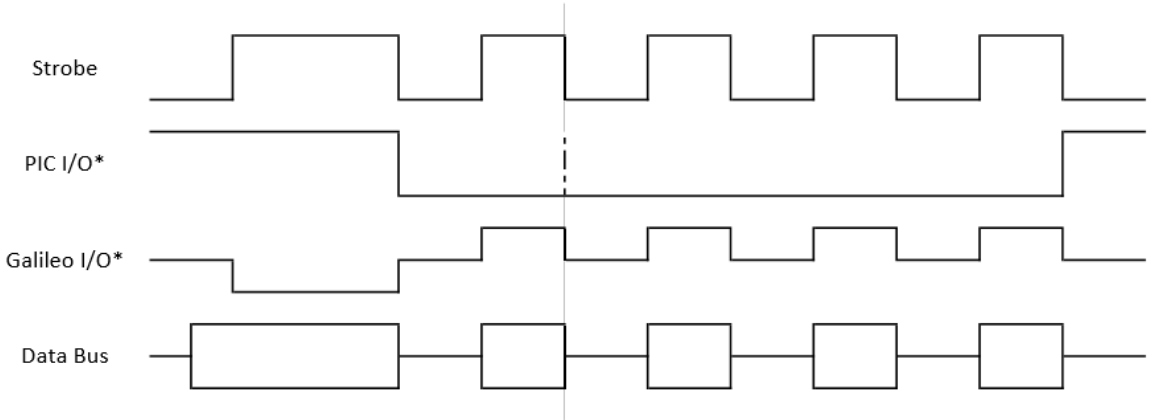


Figure 2
Timing Diagram



Note: dotted line only passed when GET request.
The more solid line is the path that would be followed if the request is **not** GET.

Figure 3
Flow chart of software (made by Derek)

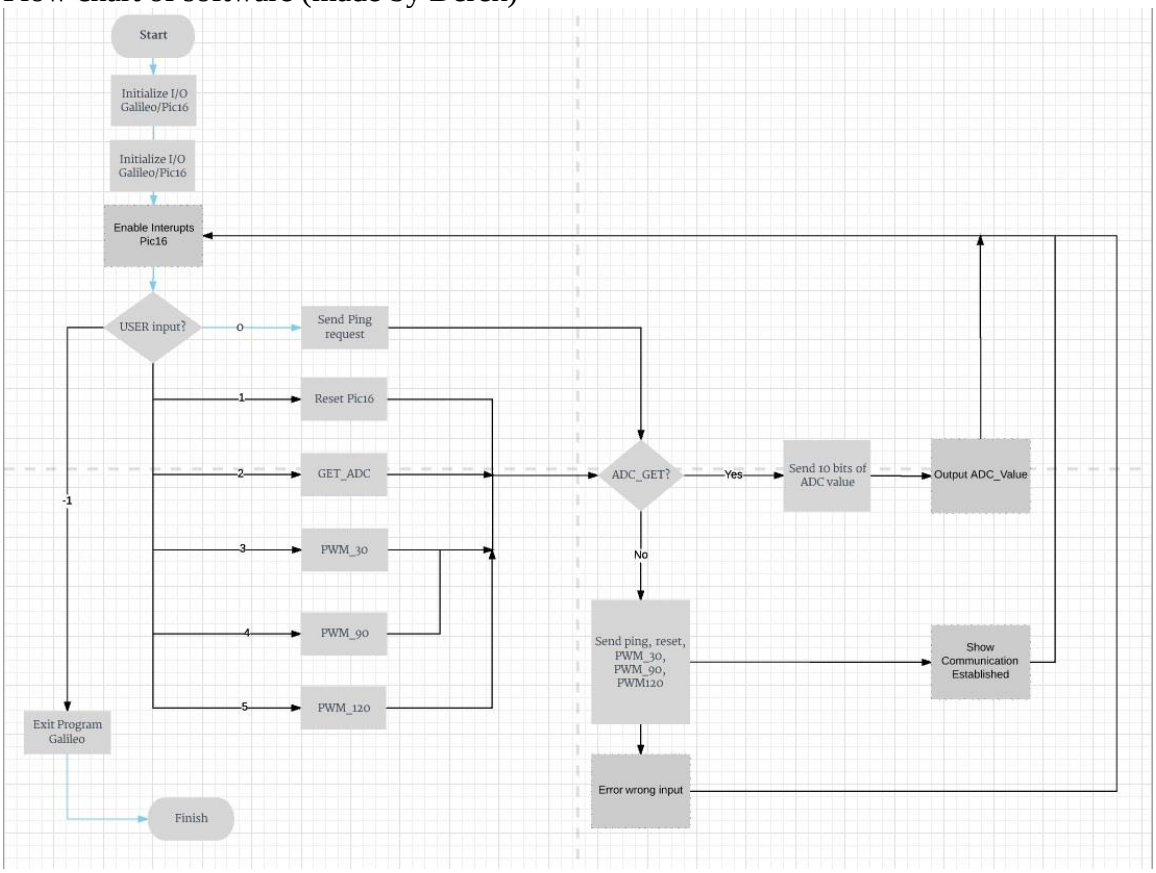
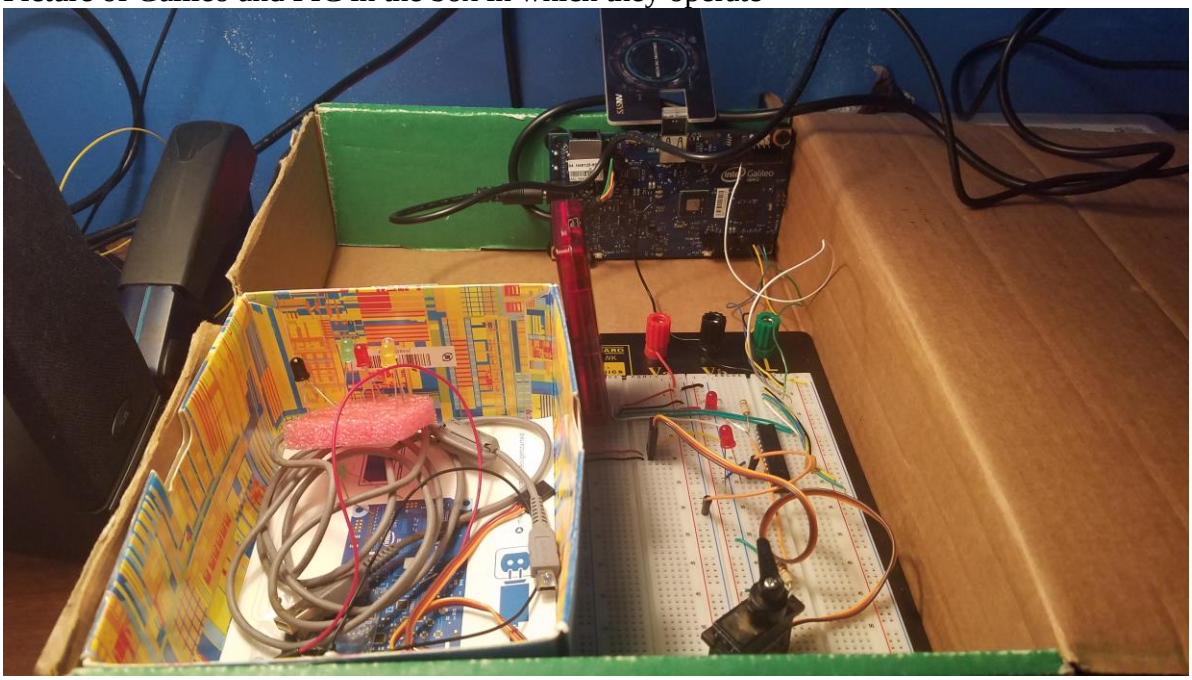


Figure 4
Picture of Galileo and PIC in the box in which they operate



Software design:

In this laboratory, a strobe protocol was designed so that the Galileo could interrupt the PIC microcontroller to send nibbles, or 4-bit packets of data, as commands and receive nibbles back in response. First, the protocol was designed. The PIC and Galileo board had to agree to a set of command and response codes so that they would each understand each other. Then, the timing of the strobe was designed. Refer to Figure 2 for the timing diagram. First, the Galileo puts a nibble on the data bus. The Galileo must put its pins as outputs and the PIC should already have its pins as inputs. Next, the Galileo raises a strobe signal, which interrupts the PIC microcontroller. At this point, the PIC microcontroller has ten milliseconds to read the nibble on the data bus before exiting the interrupt and returning to normal execution. After the ten milliseconds expires, the Galileo lowers the strobe signal, which triggers another interrupt on the PIC microcontroller. This time, the PIC processes the command, which it read from the data bus, and puts the command in the instruction queue if it is a command that takes too long to process immediately. For example, a PWM motor command may take several milliseconds, so it is best to handle those movements in the regular program execution rather than in an interrupt. After adding the instruction to the queue, the proper response is determined before exiting the interrupt. The Galileo will leave at least two milliseconds for all of this to occur. The Galileo will also switch its pins to inputs at this moment. Next, the Galileo will flip the strobe high again triggering yet another PIC interrupt, and the Galileo will give the PIC two milliseconds to move data to the bus before the Galileo reads from the bus. Moving the data to the bus requires the PIC to switch its pins to outputs. If the Galileo expects the PIC to send back an ADC value, this reading process repeats another three times. At the end, the Galileo flips the strobe low again and the PIC makes its pins inputs and resumes normal execution. Notice in this protocol that never at any point are both the Galileo pins and PIC pins both outputs to the bus. This prevents short circuits from occurring, which could potentially damage equipment. In addition, all interrupts on the PIC side are debounced in order to prevent voltage surges or noise from triggering false interrupts. The communication on the PIC side is located in Appendix A1; for the Galileo, Appendix A2.

In normal execution, the Galileo simply waits for user command line input to begin the next communication protocol. The PIC, on the other hand, is constantly polling in an infinite loop. If the execution queue is not empty, the PIC will remove from the queue and execute the desired instruction. These instructions all have come from the Galileo via previous instances of the communication. Also, in the infinite loop, the PIC will poll and increment a counter. Whenever the counter rolls over at a desired value, an analogue-to-digital (ADC) conversion will begin and the value will be stored. The ADC value will be used to determine whether an LED is turned on or off. The threshold is determined dynamically by keeping track of the maximum and minimum ADC values obtained. The threshold will be half way in between the minimum and maximum values obtained thus far. Also, the LED is debounced to prevent it from flickering. The threshold to turn the LED on is higher than the threshold to turn the LED off. The main infinite loopfor the PIC is located in Appendix A3.

Hardware design:

The hardware design of this circuit is based heavily on the software requirements.

The specification wanted an analogue-to-digital converter on the PIC microcontroller to read and save values that are related to the light intensity over a photo-resisting light sensor. To do this, we used a voltage divider like in the first laboratory. The voltage divider was between a photo-resistor in series with a fixed resistor and voltage source. The operating voltage source for the entire circuit was 3.3 V. As the light intensity over the photo-resistor increased, the resistance of the photo-resistor decreased and the input analogue voltage was smaller. The vice versa is true as the light intensity decreases. Additionally, the light-emitting diode (LED) was hooked up as another output. The LED was turned on or off based on the light intensity over the sensor.

The specification also requires that a pulse-wave modulated (PWM) signal be generated by the PIC when appropriate to move a senso-motor arm. The actual senso-motor operated on almost six volts, but the signal coming from the PIC ranged from zero to 3.3 volts.

The specification also required the Intel Galileo board to communicate with the PIC microcontroller in nibbles (4-bits). This required five pins on each device. One pin was always an output for the Galileo and always an input for the PIC; this is where a strobe signal is connected. The strobe protocol is also discussed in the software design. The other four pins switch between inputs and outputs on each device and they act as a data bus.

A drawing of the pin connections can be found in Figure 1 in the schematics section.

Section 8: Trouble Shooting /1 points

Issue 1:

The first significant issue that we encountered was with the PWM code. In this laboratory, Kyle coded three functions – one to move the senso-motor to thirty degrees, one for ninety degrees, and one for one hundred twenty degrees. We were able to get each of the three functions to generate square waves; however, the senso-motor was not moving in the way in which we expected. We reviewed the specification and discovered that the senso-motor was actually supposed to operate at six volts. We were concerned that if we operated the motor at six volts, but the PIC at 3.3 V, then the motor would not respond, but we tested this and found that it worked. Afterwards, Kyle went through and changed some of the delay values in order to get the motor to turn to the proper angles.

Issue 2:

The next major issue was with the Galileo GPIO pin input setting. Derek and I were able to get the GPIO pins to function properly as outputs, and we were able to get the GPIO pins to open as inputs. However, the inputs were not reading valid values. In test programs, the GPIO pins would read a one or zero once before reading odd numbers such as twenty-nine. We discussed the issue and concluded that it was likely that GPIO pins needed to be consistently re-opened as inputs in order to read multiple values. We made this change to the test program, and afterwards, everything functioned as expected.

Issue 3:

The most time-consuming error occurred in the final stages of the project. We were testing both the PIC and the Galileo trying to get them to communicate with each other. The protocol would never finish though. The Galileo user-interface was outputting that a connection error occurred meaning that no valid acknowledgement had been received from the PIC microcontroller. We added very long delays to parts of the protocol in order to analyse portions of the strobe protocol. During our tests, we observed that the PIC was reading proper values from the Galileo, but the Galileo was not receiving anything back. With some further testing, Kyle noticed with a multimeter that the strobe pin was high at the end of the protocol, which is not right. I went through the Galileo and PIC communication protocol implementations and found our fatal error: on the functions that were supposed to set the strobe pin low again, a variable “Strobe” was being used rather than a different variable called “strobe” (lowercase). This means that in the beginning of a protocol, the strobe pin would go high and the PIC would read the data. Afterwards though, the strobe would remain high and the PIC would never add the instruction to the execution queue nor generate an acknowledgement code response. Meanwhile, the Galileo would continue communicating as if there was no problem and it would read data off of an empty data bus from the PIC, which caused the error message. Once this coding error was changed, the program worked immediately!

Section 9: Results */0.5 points*

Figure 5
Screenshot of terminal-interfacing program on Galileo

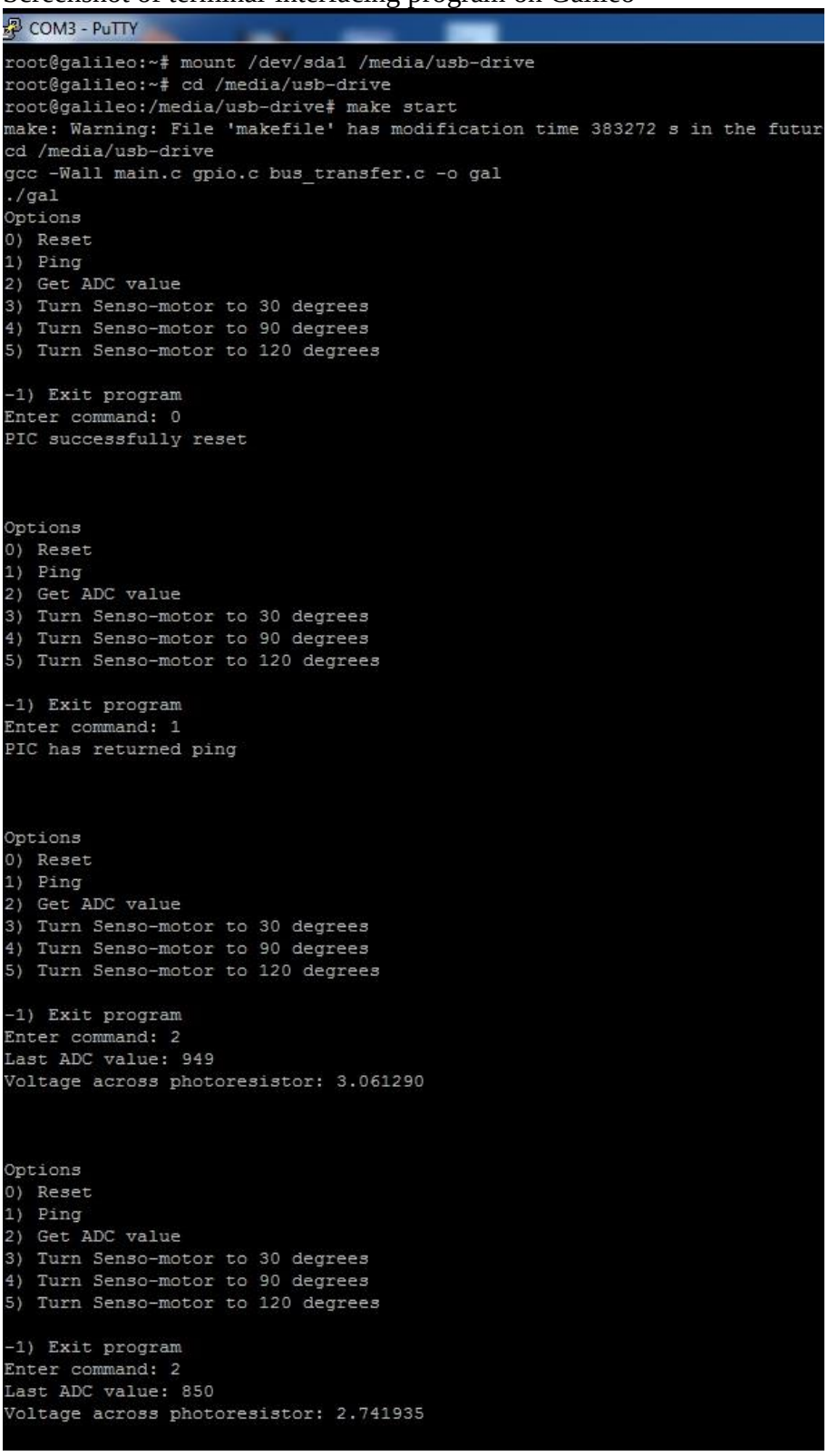


Figure 6
Screenshot of PWM signal used to turn senso-motor to thirty degrees

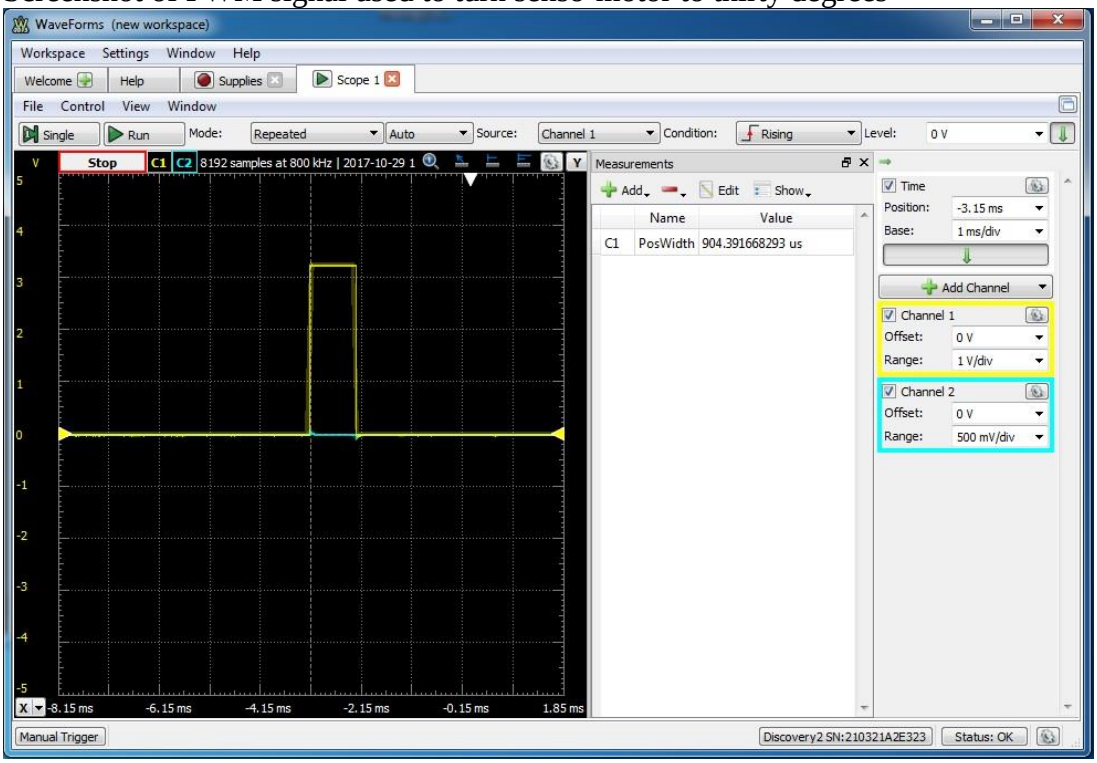


Figure 7
Screenshot of PWM signal used to turn senso-motor to ninety degrees

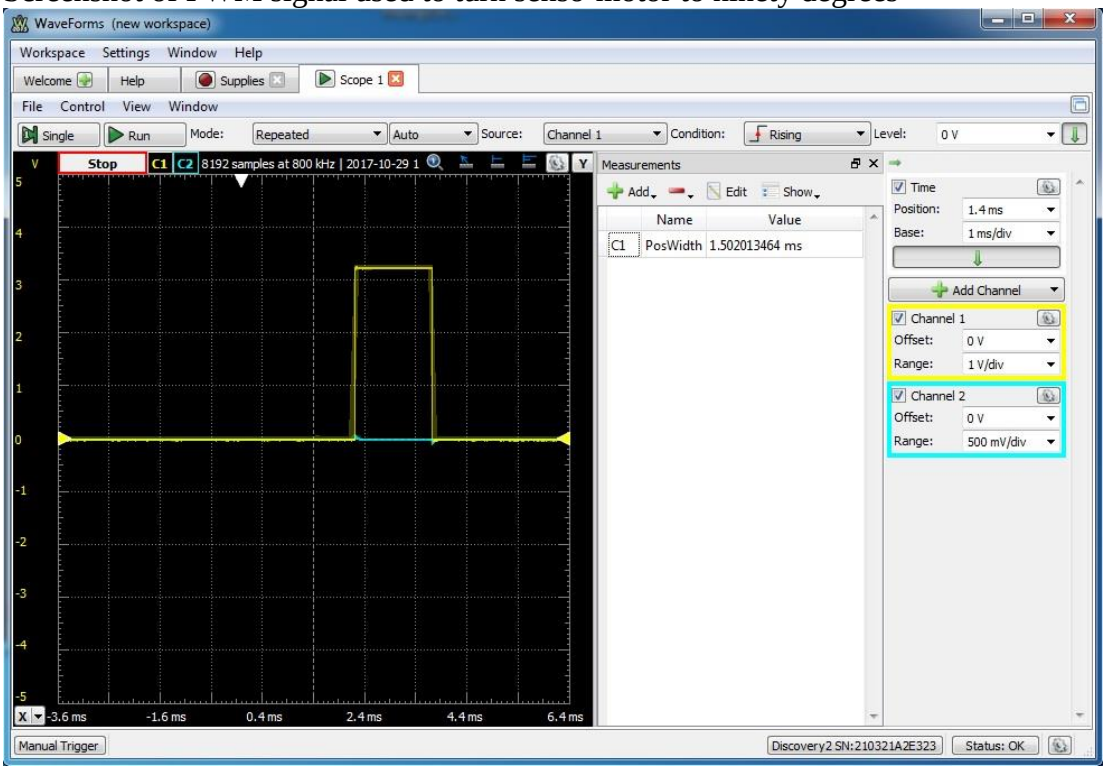
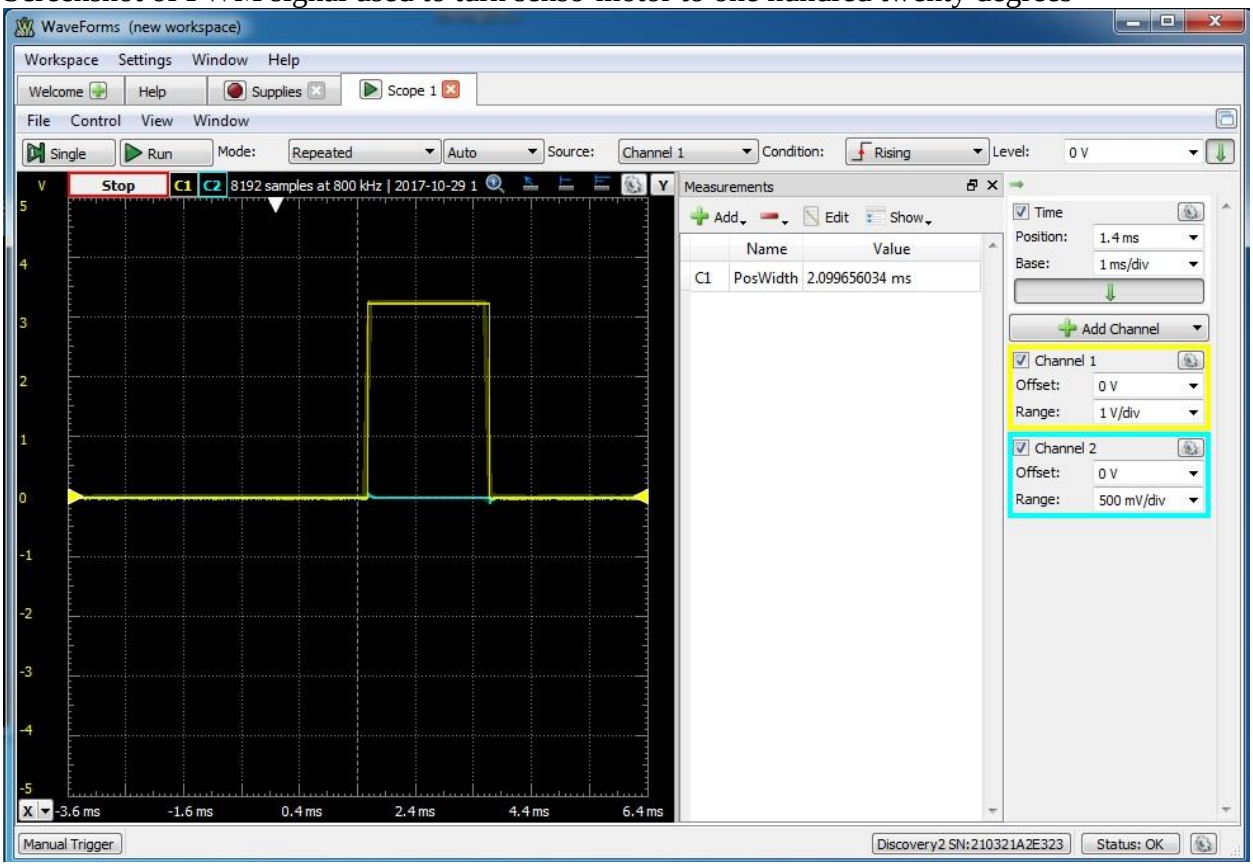


Figure 8
Screenshot of PWM signal used to turn senso-motor to one hundred twenty degrees



A1.

```
/*this switch statement is called every interrupt.
communication_counter represents that phase of the communication.*/
switch (communication_counter) {
case 0:
	// first interrupt, read value from GPIO bus
	// GPIO bus pins should already be set as inputs
	instruction = read();
	++communication_counter;
	break;
case 1:
	// computer is done outputting signal
	// start processing and set up outputs already
	if (instruction == MSG_GET) {
		write((adc_value >> 8) & 0x3);
	} else {
		enqueue(&execution_queue, instruction);
		write(MSG_ACK);
	}
	++communication_counter;
	break;
case 2:
	// computer raises signal
	// reading has begun
	TRISB &= 0xE1;		// set up outputs
	++communication_counter;
	break;
case 3:
	// computer done reading value
	if (instruction == MSG_GET) {
		// write bit again
		write((adc_value >> 4) & 0xF);
		++communication_counter;
	} else {
		TRISB |= 0x1E;  // back to inputs (high impedance)
		communication_counter = 0;	// next edge will be new command
	}
	break;
case 4:
	// only get will come this far
	// reading has begun
	++communication_counter;
	break;
case 5:
	// computer done reading
	// write one more value!
	write(adc_value & 0xF);
	++communication_counter;
case 6:
	// reading has begun
	++communication_counter;
	break;
case 7:
	// reading done
	write(MSG_ACK);
	++communication_counter;
	break;
case 8:
	++communication_counter;
	// reading
	break;
case 9:
	// all done w/ everything
	TRISB |= 0x1E;
	communication_counter = 0;	// next edge is new instruction
	break;
default:
	// handle error
	communication_counter = 0;	// next edge is new instruction
	break;
}
```

A2.

```
/*
START STEP 1
1) open all pins as outputs
2) put data on bus
3) flip strobe on
4) Give pic 10ms to read data
*/

// 1
data[0] = openGPIO(GP_4, GPIO_DIRECTION_OUT);
data[1] = openGPIO(GP_5, GPIO_DIRECTION_OUT);
data[2] = openGPIO(GP_6, GPIO_DIRECTION_OUT);
data[3] = openGPIO(GP_7, GPIO_DIRECTION_OUT);

writeBus(input & 0xF, data);             // 2
writeGPIO(strobe, HIGH);          // 3
usleep(10000);                                  // 4


                                                        /*END STEP 1*/

                                                        /*STEP 2 -- read data from PIC*/
flag = 0;
response = 0;
while ((flag < 4 && input == MSG_GET) || flag < 1) {
       // if msg_get, read 4 times
       // else, just read response

       /*
       READ FROM PIC
       1) bring strobe low
       2) remove data from bus
       3) make pins inputs after closing them
       4) give PIC some auxiliary some extra time to generate response
       5) raise strobe high
       6) give PIC time to convert pins from inputs to outputs
       7) read bus
       */

       writeGPIO(strobe, LOW);                          // 1
       writeBus(0, data);                                      // 2

                                                                              // 3
       closeGPIO(GP_4, data[0]);
       closeGPIO(GP_5, data[1]);
       closeGPIO(GP_6, data[2]);
       closeGPIO(GP_7, data[3]);
       data[0] = openGPIO(GP_4, GPIO_DIRECTION_IN);
       data[1] = openGPIO(GP_5, GPIO_DIRECTION_IN);
       data[2] = openGPIO(GP_6, GPIO_DIRECTION_IN);
       data[3] = openGPIO(GP_7, GPIO_DIRECTION_IN);

       usleep(2000);                                    // 4
       writeGPIO(strobe, HIGH);                 // 5
       usleep(2000);                                    // 6

                                                                     // 7
       if (input == MSG_GET) {
              response += readBus(data) << (4 * (3 - flag));  // 7 + extra
       } else {
              response = readBus(data);
       }

       ++flag;

}
/*END STEP 2*/

/*START STEP 3 -- just switch strobe to low to indicate that communication is over, and close pins*/
writeGPIO(strobe, LOW);
closeGPIO(GP_4, data[0]);
closeGPIO(GP_5, data[1]);
closeGPIO(GP_6, data[2]);
closeGPIO(GP_7, data[3]);
/*END STEP 3*/
```

A3.

```
while (1) {
	/*START LIGHT SENSOR AND LED PART*/
	if (led_counter < 0) {
		// adc conversion is running
		// only if conversion is done,
		// update LED based on LED_THRESHOLD (pre-defined)
		if (ADCON0bits.GO == LOW) {
			// conversion = done
			// update led
			adc_value = (ADRESH << 8) + ADRESL;
			if (adc_value > max) {
				max = adc_value;
				threshold = (max + min) >> 1;
			}
			if (adc_value < min) {
				min = adc_value;
				threshold = (max + min) >> 1;
			}
			// debounce
			if (LATAbits.LATA0 == HIGH) {
		/*LED is off; to turn on, adc_value must go below the lower offset*/
				if (adc_value < threshold - OFFSET) {
					LATAbits.LATA0 = LOW;
				}
			} else {
				if (adc_value > threshold + OFFSET) {
					LATAbits.LATA0 = HIGH;
				}
			}
			led_counter = 0;	// start counter over again
		}
	} else if (led_counter >= LED_ROLLOVER) {
		// it is time to start an adc conversion
		ADCON0bits.GO = HIGH;   // start conversion
		led_counter = -1;   // indicates that conversion is running
	} else {
		// neither so just update counter
		++led_counter;
	}
	/*END LIGHT SENSOR AND LED PART*/
	/*Queue execution*/
	if (!isEmpty(&execution_queue)) {
		switch (dequeue(&execution_queue)) {
		case MSG_RESET:
			min = 1023;
			max = 0;
			threshold = 0;
			break;
		case MSG_TURN30:
			PWM_Turn30();
			break;
		case MSG_TURN90:
			PWM_Turn90();
			break;
		case MSG_TURN120:
			PWM_Turn120();
			break;
		}
	}
} // end infinite loop
```