



“Microprocessor Systems II & Embedded Systems”

“EECE.4800”

“Lab2: Interfacing with a Sensor Device and an Embedded Computer System”

“Instructors Yan Luo, TA; Ionnis Smanis”

“Group #10”

“Derek Teixeira”

“Due date October 30, 2017”

“Handed in October 30, 2017”

1. Group Member 1 – Derek Teixeira (Me)

Had the same role as last time, being the team manager and handling all of the equipment and being at all of the team meetings. Main responsibility was to get the Galileo board started by getting it to boot on an SD card and figuring out mounting and unmounting a usb drive. Eventually went to figuring out of how to enable the read and write pins coming from the Galileo and to the pic16 chip. Coding side of things were doing the initializations for the GPIO pins on the Galileo and figuring out how to use and run the sample Galileo code given to the students.

2. Group Member 2 – Hans Edward Heone

Hans was to take care of refining and making our lab 1 code work into this lab 2's code. Worked on most of the pic side of the coding and was able to figure out how to use interrupts on the pic microcontroller in order to handle our strobe pulse from the Galileo. Hans also had a big part in setting up our user commands for the Galileo side of the coding. He was able to get our strobe and 4-bit values correct when entering a command into our interface.

3. Group Member 3 – Kyle Marescalchi

Kyle was the main guy for the PWM module and was able to code and figure out all of the values for the pic side. Kyle helped Derek to start initializing the Galileo board pins and get read and write options from user inputs. Kyle is also our test guy who debugs and spends his time trying to break the circuit. This is actually very useful and full proofs our design and helps us find important bugs in our system.

The Purpose of lab 2 is to control the same pic microcontroller from lab1, but with the use of an intel Galileo Board. This lab shows the student how to interface with the sensors, ADC value and Servo Motor, while using a computer to control their operations. The purpose is to show the student how a CPU is used to control a servant type of chip in the circuit. In a real-world situation Computers sometimes need other devices to manage memory or I/O commands. A CPU needs to be efficient and used for main tasks in a computer while the slave handles other smaller tasks. Being able to use the GPIO pins from the Galileo and setting up pins from the microcontroller are how this lab will be completed to show a master/servant display.

Lab 2 is about using a master(Galileo) and a slave device (pic16 chip). The Galileo will have some basic commands to send to the pic16 and it will do the commands and report back to the Galileo board. Types of commands are reset, ping, get ADC, or turn a certain degree on the servo motor. Lab 2 is going to send data through the Galileo GPIO ports and send data using bus protocol towards the pic16 microcontroller. Being able to time up both the Galileo and Microcontroller are essential in delivering consistent and accurate data back and forth between both devices. Using a timing chart can help to see where there is an issue in communicating between the pic microcontroller and the intel Galileo Board.

- PICKit3 microchip, Model# 10-00424-R7, S/N# BUR142550918, Operating Voltage (1.8-5 Volts) Used to Communicate code and initializations between a PC interface and pic chip.
- Analog Discovery 2, Model# 210-321, S/N# 210321AZE323, Operating Voltage (5 volts) Was sometimes used to check voltages, square wave pulses, and supply voltages.
- Benchtop Power Supply-GWINSTEK GPD-3303P, S/N# EL920014.
Used to power fully circuit. Outputted 3.3 volts to the circuit board to power the pic16 chip and outputted a separate 5-6 volts with a higher current to power the servo motor.
- Breadboard R.S.R Electronics
Used for everything hardware related. Pins for pic16, PICKIT3, servo motor, Photoresistor, LED, and power/ground.
- Pic16f18857, Chip# 1621RW6
Gets user input from Galileo Board and controllers servo motor and ADC conversion for an LED. Will send Acknowledgments back to Galileo Board.
- Micro Servo Motor, Model name Longrunner MG 90s
Operation is to 30 degrees 90 degrees or 120 degrees with user inputted commands.
- Intel Galileo Gen 2 Board, S/N# FZGL50400CW, operating voltage is 3.3 or 5 volts. Used with GPIO ports to send and receive messages from the pic(16) microcontroller.
- USB-to-UART cable, has 6 pins and is used for serial communication between the Galilo and a PC USB port. FTDI drivers needed for communication.

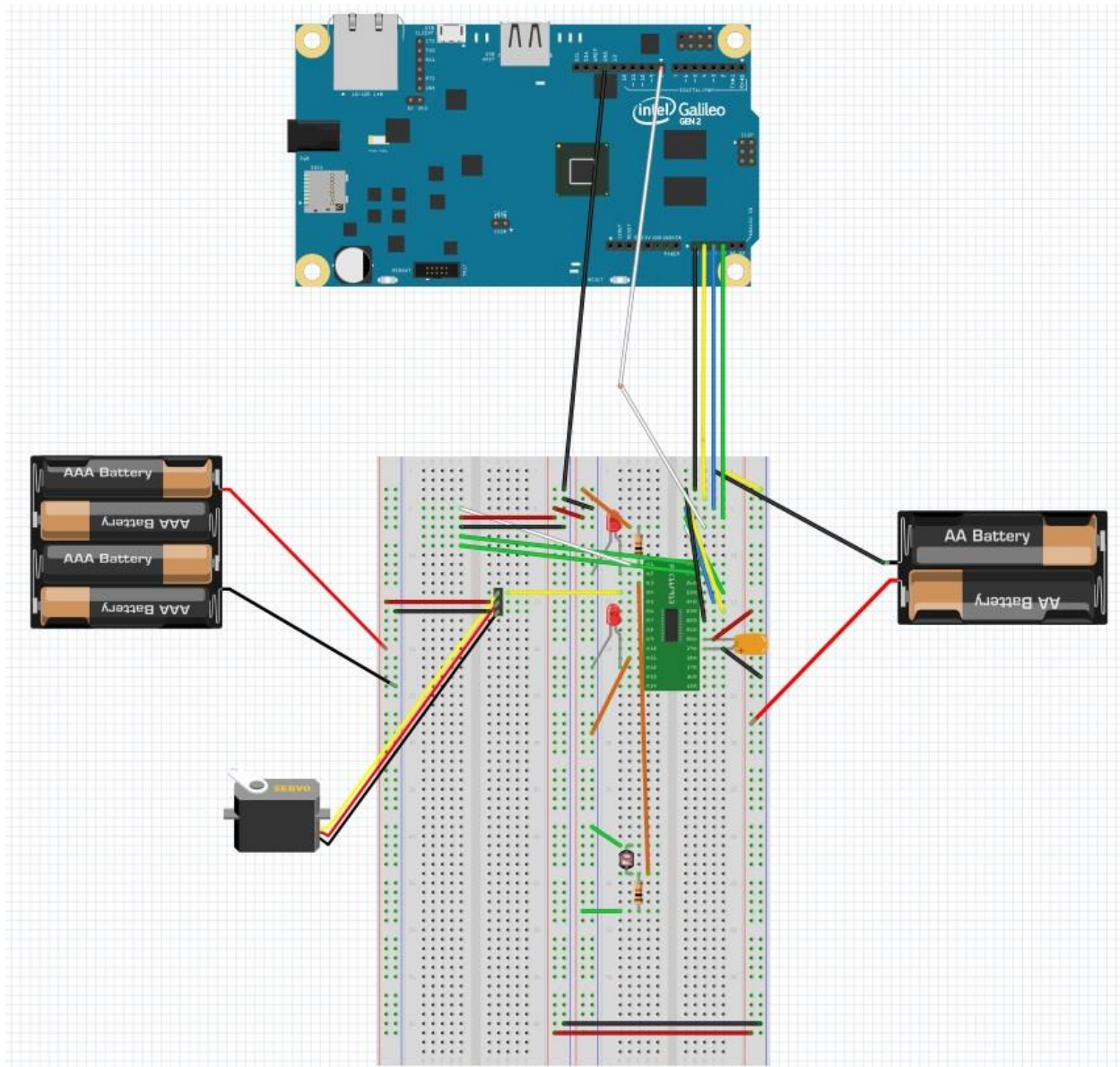


Figure 1: Circuit Schematic Lab2 pic16f18857 and Intel Galileo gen 2 board

Similar Design to that of Lab1. Some changes are using the ports RB1-RB5 to read and write from the Galileo to the Pic microcontroller (port RB5 is our strobe that is used as an interrupt). Another addition is using an LED on RC1 that will hard reset when the wire from RC0 goes high then low. The left power supply can be around 5-6 volts for the servo motor while the right power supply is set at 3.3 for the microcontroller.

Hardware design:

Lab 2 consists of making our own bus protocol that is used to send or receive four bits at a time from the Galileo to the pic microcontroller and vice versa. The group agreed on using the RB ports to read and write on the side of the pic microcontroller. We wanted to use pins RB0-RB4 but RB0 got fried in the first connection between pic and Galileo (see troubleshoot section). Putting us in position to use RB1-RB5. RB1-RB4 are used as the data ports and RB5 is used for the strobe feed. On the Galileo side of things, the group used the same pins given to us by the example Galileo code. These pins are A0-A3 for data and pin 8 for our strobe/interrupt pin.

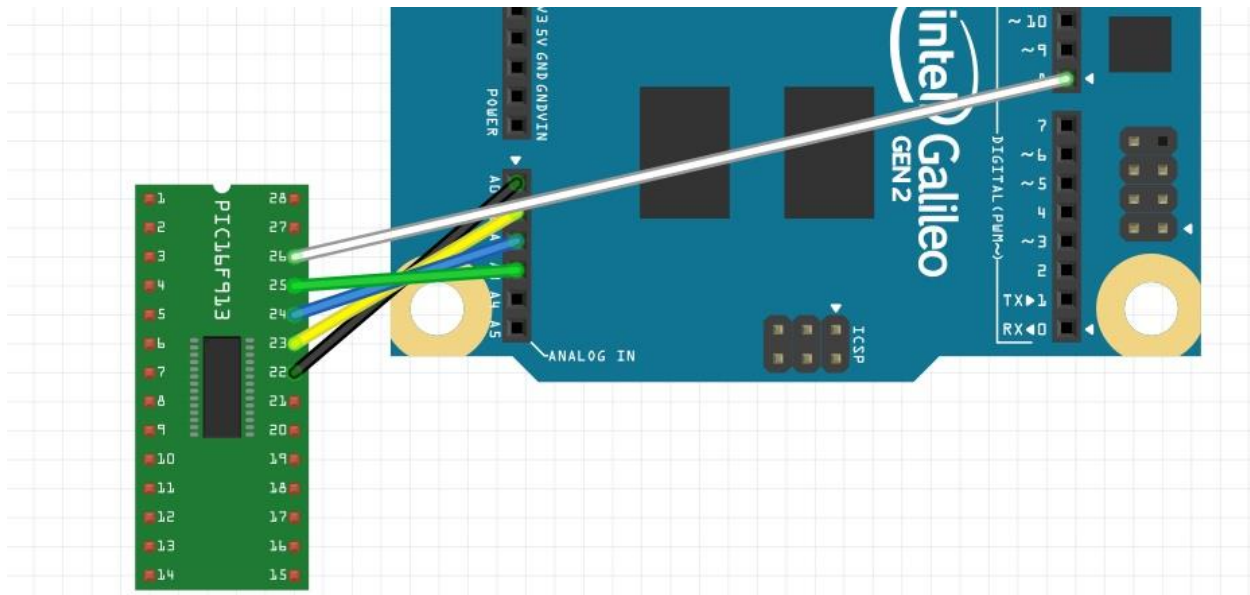


Figure 2: PIN selection for 4-bit bus protocol

This is exactly how we wired up our connections. RB1 goes to A0, RB2 goes to A1, RB3 goes to A2, and RB4 goes to A3. This makes sense on paper, but technically we should have done it backwards in order not to cross the wires. This would also make it less difficult to wire the system bus. The strobe is the white wire coming out of RB5 that goes directly to the pin 8 on the Galileo board. All of the pins match the Galileo template on the Micro2 github page.

The rest of the circuit is exactly the same in lab1 besides one more addition. This is the hard reset that can be done physically without the need to use our coded reset from the Galileo interface. The way the reset works is taking the orange wire and setting it to high and then setting it back to low. The LED will be on when high when hitting the 3.3 voltage, and turn off when grounded, to signal a reset of the pic microcontroller. This was not necessary to the lab2 designed but help incredibly when debugging our circuit.

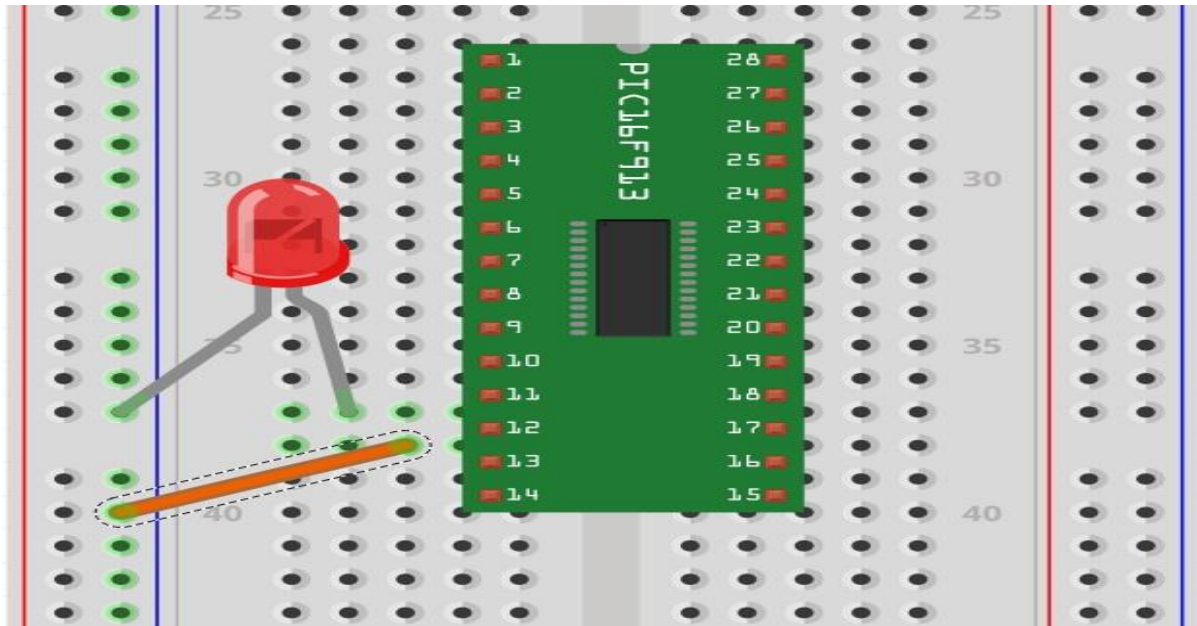


Figure 3: Hard reset using pins RC1 and RC0

Software design:

Once a timing diagram was configured the group was able to see what steps were needed to obtain data in each direction. The Line down the center is only used when the GET_ADC command is used. This command needs to get a 10-bit value from the pic microcontroller while sending 4 bits at a time. When using any other command, the timing diagram ends before the line in the middle of the diagram. When the strobe is high the pic is reading and the galileo is writing. When the strobe goes low the galileo is reading and the pic swaps the writing. When the strobe goes back to high the galileo outputs what data is gets back from the pic16 chip.

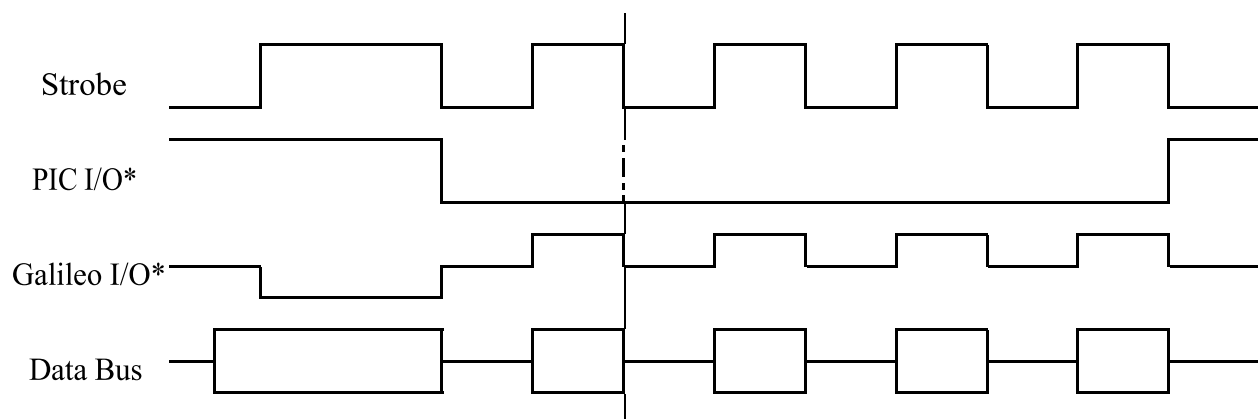


Figure 4: Timing Diagram of communication between pic and Galileo. Line in the middle signals if the GET_ADC function is used.

First priority on the check list was to assign 5 select pins on the Galileo board and the pic16F15887 microcontroller. These sets of pins will be used for both reading and writing in both directions on the Galileo board and the Pic microcontroller. As explained in the hardware section, the pins are RB1-RB5 on the microcontroller and A0-A3 plus pin 8 on the Galileo board. The next piece of coding is how we set up each set of pins with our coding guidelines.

```

5  void GPIO_Init() {
6      communication_counter = 0;
7
8      TRISB = 0xFF;
9      ANSELB = 0x00;                                     // PORT B is digital input
10
19  int read() {
20      return (PORTB >> 1) & 0xF;
21  }
22
23  void write(int instruction) {
24      LATB &= 0xE1;
25      LATB |= (instruction & 0xF) << 1;
26  }

```

Figure 5: Setting portB to read or write function on pic16

```

13  //Linux GPIO number///  Aduino shield Pin Name
14  /*GPIO Pins*/
15  #define Strobe      (40)    //8
16  #define GP_4        (48)    //A0
17  #define GP_5        (50)    //A1
18  #define GP_6        (52)    //A2
19  #define GP_7        (54)    //A3
20
21  /*GPIO Directions*/
22  #define GPIO_DIRECTION_IN    (1)
23  #define GPIO_DIRECTION_OUT   (0)

```

Figure 6: defining pins for the Galileo and direction.


```

6  void writeBus(int value, int *bus) {
7      writeGPIO(bus[0], value & 0x1);
8      writeGPIO(bus[1], (value >> 1) & 0x1);
9      writeGPIO(bus[2], (value >> 2) & 0x1);
10     writeGPIO(bus[3], (value >> 3) & 0x1);
11 }
12
13 int readBus(int *bus) {
14     // readGPIO must return 0 or 1 !!!
15     int value = 0;
16     value |= (readGPIO(bus[0]));
17     value |= (readGPIO(bus[1]) << 1);
18     value |= (readGPIO(bus[2]) << 2);
19     value |= (readGPIO(bus[3]) << 3);
20     return value;
21 }

```

Figure 7: Setting up GPIO pins for read and write functions Galileo board

In order to initialize reading and writing between the two devices, there needed to be a strobe pin that connects pin RB5 (pic side) to pin 8 (Galileo side). The strobe signal is sent to the pic from Galileo and an interrupt occurs on the pic microcontroller to signal when to read and write. The next section of code initializes the read function on the pic side of interrupt then sets it the write when the strobe goes back to low.

```

11     PIE0bits.IOCIE = 1;
12     // PIE0bits.INTE = 1; // I don't need
13     IOCBPbits.IOCBP5 = 1;
14     IOCBNbits.IOCBN5 = 1;
15     INTCONbits.PEIE = 1;
16     INTCONbits.GIE = 1;

```

Figure 8: Setting Interrupt Flags Pic


```

void interrupt ISR() {

    if (IOCCFbits.IOCCF0 == HIGH) {
        IOCCFbits.IOCCF0 = LOW;
        reset();
    }
    // check source
    if (PIR0bits.IOCIF == HIGH && IOCBFbits.IOCBF5 == HIGH) {
        // check if <<Interrupt-on-Change Interrupt Flag bit (read-only); p. 142>> is HIGH
        // indicates that interrupt-on-change caused interrupt
        // check if RB0 was the on-change interrupt (p. 261)
        // by design, only positive-edge change is enabled

        // debounce
        __delay_ms(DEBOUNCE_DELAY);
        if (PORTBbits.RB5 == (communication_counter % 2 ? LOW : HIGH)) {
            // interrupt is legit, so handle it, dumbass

```

Figure 9: Interrupt pin getting enabled to start reading from Galileo pin RB5

When selected what the user can input, setting up the Galileo board needed 4 steps in the process of using the GPIO pins for input or output. They needed to be “export”, “direction”, “read/write”, and “unexport”.

```

1  #include "gpio.h"
2
3  int openGPIO(int gpio, int direction) {
4      int handle;                //file variable
5      char buf[256];
6      int inout;
7
8      //simple command to enable pin A0
9
10     handle = open("/sys/class/gpio/export", O_WRONLY);

```

Figure 10: setting up variables and exporting a GPIO pin

```

18     sprintf(buf, "/sys/class/gpio/gpio%d/direction", gpio);
19
20     handle = open(buf, O_WRONLY);
21
22     // Set out direction
23     switch (direction) {
24     case GPIO_DIRECTION_OUT:
25         write(handle, "out", 3);
26         inout = O_WRONLY;
27         break;
28     case GPIO_DIRECTION_IN:
29     default:
30         write(handle, "in", 2);
31         inout = O_RDONLY;
32         break;

```

Figure 11: setting up direction in(input) or out(output)

```

57 //Read value on the GPIO pins
58 int readGPIO(int handle) {
59     char ret;
60     read(handle, &ret, 1);
61     return (ret - '0');
62 }
63 //write value on the GPIO pins
64 void writeGPIO(int handle, int status_write) {
65
66     // Set GPIO high status
67     if (status_write) {
68         write(handle, "1", 1);
69     } else {
70         write(handle, "0", 1);
71     }
72
73     return;
74 }

```

Figure 12: Setting pins to read or write depending on the direction that is set; Out(write), In(read)

```

76 void closeGPIO(int gpio, int handle) {
77     char BUFFER[255];
78
79     close(handle);
80
81     handle = open("/sys/class/gpio/unexport", O_WRONLY);
82     sprintf(BUFFER, "%d", gpio);
83     write(handle, BUFFER, strlen(BUFFER));
84     close(handle);
85     return;

```

Figure 13: After each pin is set needs to be closed before using same pin again. (unexport)

Before getting to the main code, we need to show how the PWM motor is used for, from the user input. Still not able to figure out how to use the actual PWM module from lab 1, the group went with delays to get specific duty cycles for 30, 90, and 120-degree rotation. These pulses are coming out of our RA2 pin.

```

5 void PWM_Init()
6 {
7     TRISAbits.TRISA2 = 0;    // TRISC pin 2 is output.
8                               // Wi
9     ANSELAbits.ANSA2 = LOW;
10    LATAbits.LATA2 = 0;
11 };
12 void PWM_Turn30()
13 {
14     int i;
15     for (i = 0; i<15; i++) {
16         LATAbits.LATA2 = 1;    // Set PWM Signal HIGH
17         __delay_ms(0.9);
18         LATAbits.LATA2 = 0;
19         __delay_ms(18.2);
20     }
21 };
22 void PWM_Turn90()
23 {
24     int i;
25     for (i = 0; i<15; i++) {
26         LATAbits.LATA2 = 1;    // Set PWM Signal HIGH
27         __delay_ms(1.5);
28         LATAbits.LATA2 = 0;
29         __delay_ms(18.5);
30     }
31 };
32 void PWM_Turn120()
33 {
34     int i;
35     for (i = 0; i<15; i++) {
36         LATAbits.LATA2 = 1;    // Set PWM Signal HIGH
37         __delay_ms(2.1);
38         LATAbits.LATA2 = 0;
39         __delay_ms(17.9);
40     }
41 };

```

Figure 14: Enabling pin RA2 for output of differing degrees for the Servo motor

A queue is used to take care of multiple inputs to our interrupt pin. The queue's job is to take each input, put it inside of a queue, and then activate that input when it is that inputs turn.

```
1  #include "Queue.h"
2  #include "mcc_generated_files/mcc.h" //default library
3  #include <htc.h>
4
5  int isFull(Queue *data) {
6      return (
7          ((data->back + 1) % (BUFFER + 1))
8              == data->front);
9  }
10 int isEmpty(Queue *data) {
11     return (data->back == data->front);
12 }
13 void enqueue(Queue *data, unsigned int item) {
14     if (!isFull(data)) {
15         data->vals[data->back] = item;
16         data->back = (data->back + 1) % (BUFFER + 1);
17     }
18 }
19 unsigned int dequeue(Queue *data) {
20     unsigned int ret;
21     if (!isEmpty(data)) {
22         ret = data->vals[data->front];
23         data->front = (data->front + 1) % (BUFFER + 1);
24     }
25     return ret;
26 }
```

Figure 15: Queue to accept multiple requests in a row.

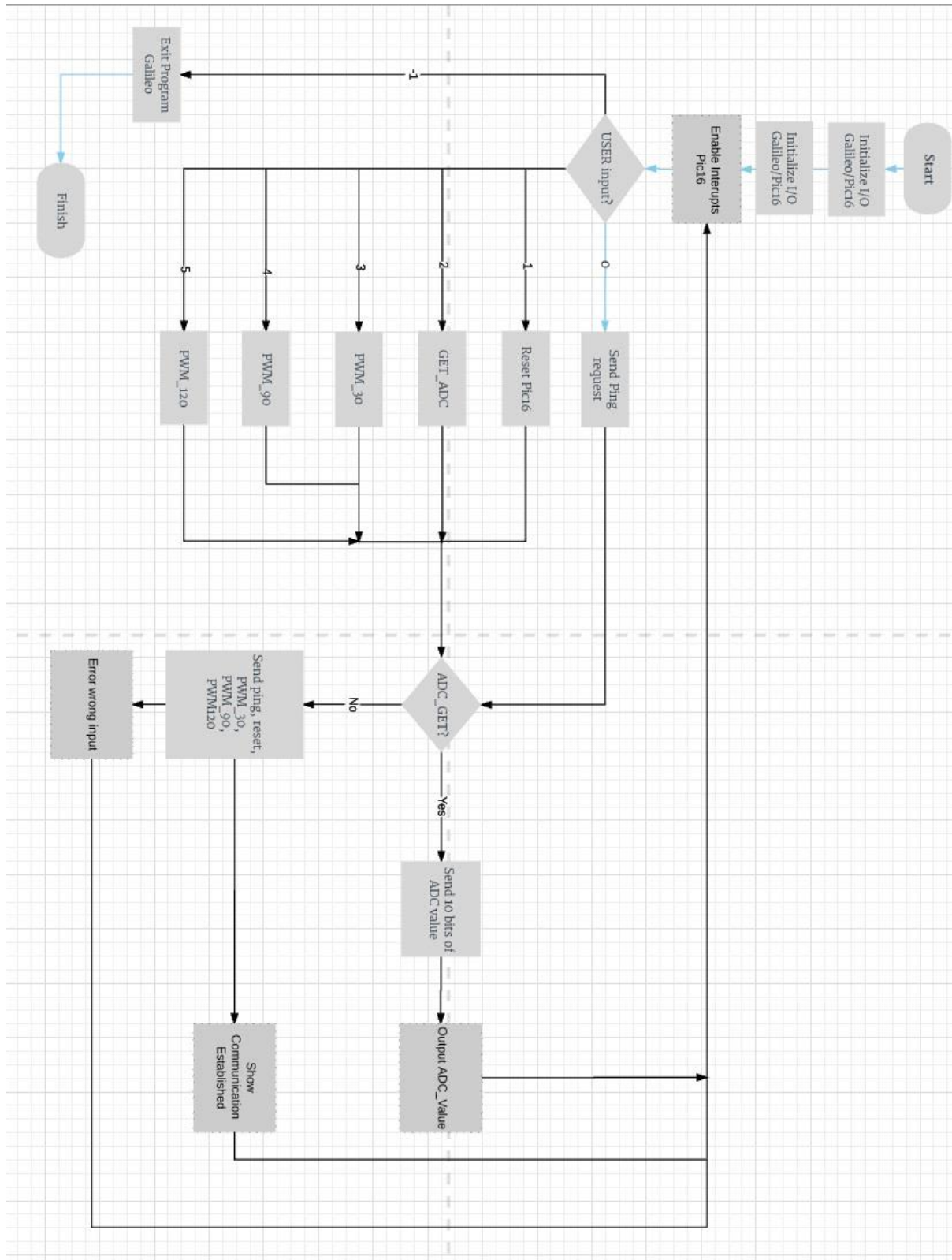


Figure 16: Flowchart showing the steps from start to finish.

Code from A1 is the main Galileo code used in our presentation. It basically starts with giving the user an input command line for what function the user wants to generate. It takes the user input and sets it into the data array that is used to signal which GPIO pins are used for each output. Once the number is entered, the GPIO get exported, direction set, and in or out is selected. If the Galileo code is successful a message appears back to the user with what has happened or it will show that an error has occurred.

Code from A2 is the pic16 side of the coding and involves using a counter to run through our program of what is to be enabled on our breadboard. It counts up by 1 each time and if the result is even it has a high strobe and odd has a low strobe coming out of the Galileo. The only exception is the ADC_GET command which will have to step through each step 4 more times to get the whole adc value. By the end of the counter it will reset our interrupt pin in order to accept new interrupts generated by the Galileo board.

Section 8: Trouble Shooting

/1 points

ISSUE #1

First problem was when the whole group first met to get a game plan on how to attack this project. The first task was as simple as getting the Galileo to boot up Linux. We had the right steps in order but we were not able to successfully get the board to load up Linux and start trying the command prompt. Good thing that this was about 3-4 weeks before lab2 was due and I (Derek) was able to bring the Galileo board home and install a Linux boot to an SD disk. We were having issues with the windisk32 installing on Han's laptop and when I did the SD disk operation at my home desktop, it was able to install successfully to the SD disk. Although not a big issue, having this issue so early on opened our eyes to how much was needed to go into this project.

ISSUE #2

When testing our write and read functions on the Galileo board we made a simple program that would say what GPIOs were enable or which were not enabled. We had no issues when setting our outputs but thought that our inputs were never being established. This was confusing and Hans and I wasted about an hour trying to get the direction in to read.

This was a mistake in my part, because I was only doing the "grep out" command when checking outputs and inputs. I needed to also type in the command "grep in" to see what the inputs for the GPIO were also. This wasn't the biggest of issues, just a small thing that cause us to waste over an hour. Once using the "grep in" command was figured out we were able to start testing reading and writing outputs at the same time.

ISSUE #3/4

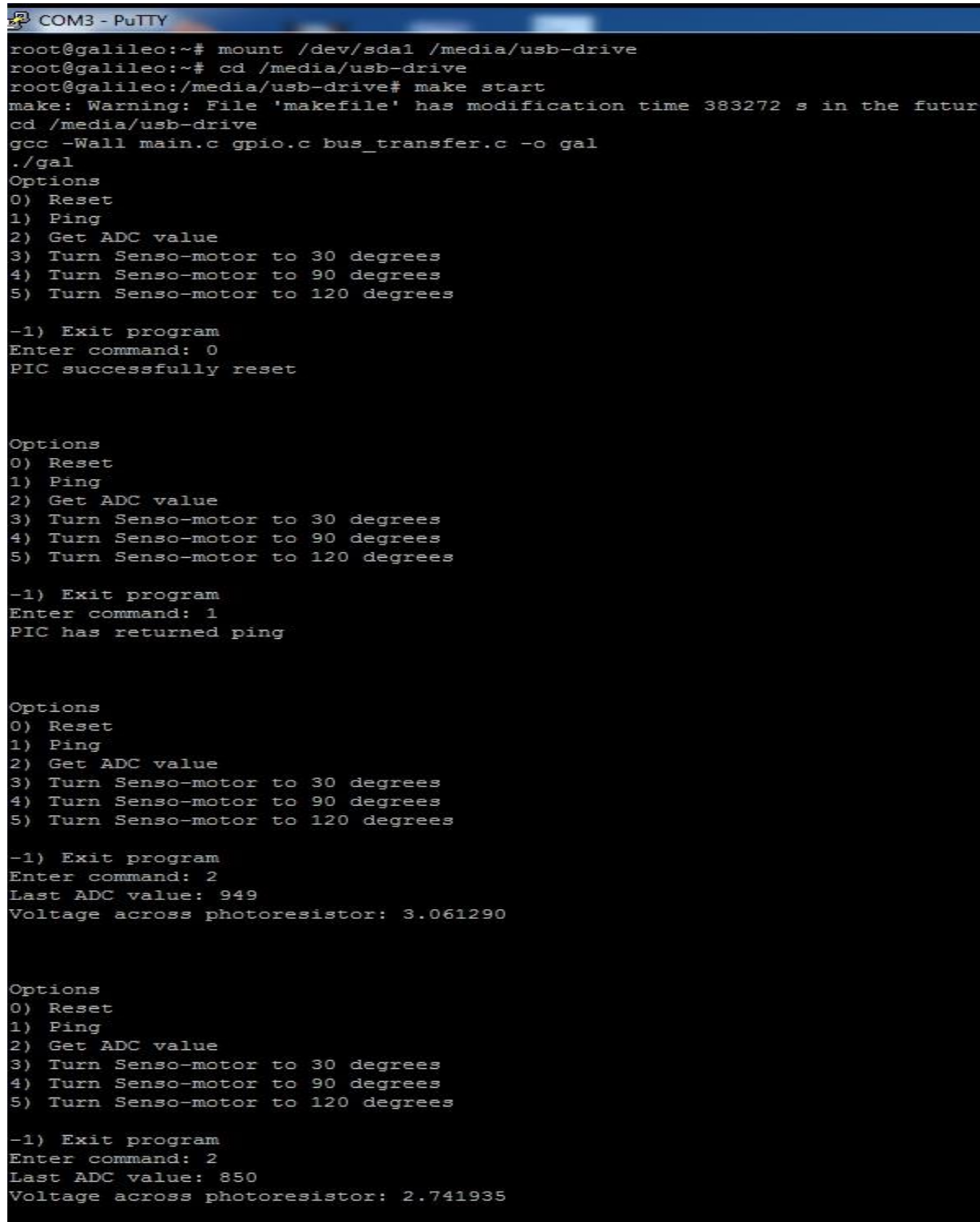
When getting everything put together before our big test from the Galileo to the pic16 chip, we were having an issue with outputting the correct data to the pic16. Kyle who is our test guy and debugger was able to realize that our strobe signal was staying as a high value. Luckily Kyle was able to point this out and Hans figured out that one of our variables was an upper-case Strobelow and not strobelow. This was figured out almost immediately and no time was lost. Without Kyle this problem may have been very difficult to find considering it was a typo and not a function error.

Going along with our testing of connecting the Galileo to the pic16 was having wrong high and lows going from the Galileo to the pic microcontroller. Again, Kyle was able to realize that everything, including the Galileo, need to be grounded together. Once everything was grounded, the outputs were correct and we were able to get the servo motor to output correct rotation.

ISSUE #5

Frying RB0 was done when the first time trying to get an output from the Galileo to activate an interrupt on our pic16 chip. The Galileo board also reset at the same time when the RB0 pin went off. Not much is known of how this happened but our RB0 pin does not work anymore. It could have been both input and output have low impedance levels and shorting something out. To fix this issue we will no longer use this pin in the future labs.

Couple terminal shots from our lab2 project



```
COM3 - PuTTY
root@galileo:~# mount /dev/sda1 /media/usb-drive
root@galileo:~# cd /media/usb-drive
root@galileo:/media/usb-drive# make start
make: Warning: File 'makefile' has modification time 383272 s in the futur
cd /media/usb-drive
gcc -Wall main.c gpio.c bus_transfer.c -o gal
./gal
Options
0) Reset
1) Ping
2) Get ADC value
3) Turn Senso-motor to 30 degrees
4) Turn Senso-motor to 90 degrees
5) Turn Senso-motor to 120 degrees

-1) Exit program
Enter command: 0
PIC successfully reset

Options
0) Reset
1) Ping
2) Get ADC value
3) Turn Senso-motor to 30 degrees
4) Turn Senso-motor to 90 degrees
5) Turn Senso-motor to 120 degrees

-1) Exit program
Enter command: 1
PIC has returned ping

Options
0) Reset
1) Ping
2) Get ADC value
3) Turn Senso-motor to 30 degrees
4) Turn Senso-motor to 90 degrees
5) Turn Senso-motor to 120 degrees

-1) Exit program
Enter command: 2
Last ADC value: 949
Voltage across photoresistor: 3.061290

Options
0) Reset
1) Ping
2) Get ADC value
3) Turn Senso-motor to 30 degrees
4) Turn Senso-motor to 90 degrees
5) Turn Senso-motor to 120 degrees

-1) Exit program
Enter command: 2
Last ADC value: 850
Voltage across photoresistor: 2.741935
```

Figure 17: Options 0-2 used to show reset, ping and ADC Value

```
COM3 - PuTTY
Options
0) Reset
1) Ping
2) Get ADC value
3) Turn Senso-motor to 30 degrees
4) Turn Senso-motor to 90 degrees
5) Turn Senso-motor to 120 degrees

-1) Exit program
Enter command: 2
Last ADC value: 715
Voltage across photoresistor: 2.306452

Options
0) Reset
1) Ping
2) Get ADC value
3) Turn Senso-motor to 30 degrees
4) Turn Senso-motor to 90 degrees
5) Turn Senso-motor to 120 degrees

-1) Exit program
Enter command: 2
Last ADC value: 458
Voltage across photoresistor: 1.477419

Options
0) Reset
1) Ping
2) Get ADC value
3) Turn Senso-motor to 30 degrees
4) Turn Senso-motor to 90 degrees
5) Turn Senso-motor to 120 degrees

-1) Exit program
Enter command: 0
PIC successfully reset

Options
0) Reset
1) Ping
2) Get ADC value
3) Turn Senso-motor to 30 degrees
4) Turn Senso-motor to 90 degrees
5) Turn Senso-motor to 120 degrees

-1) Exit program
Enter command: 3
PIC has queued command to turn senso-motor to 30 degrees

Options
0) Reset
1) Ping
2) Get ADC value
3) Turn Senso-motor to 30 degrees
```

Figure 18: Differing values for ADC when closing box lid, option 3 for 30 degree rotation

```
COM3 - PuTTY

Options
0) Reset
1) Ping
2) Get ADC value
3) Turn Senso-motor to 30 degrees
4) Turn Senso-motor to 90 degrees
5) Turn Senso-motor to 120 degrees

-1) Exit program
Enter command: 4
PIC has queued command to turn senso-motor to 90 degrees

Options
0) Reset
1) Ping
2) Get ADC value
3) Turn Senso-motor to 30 degrees
4) Turn Senso-motor to 90 degrees
5) Turn Senso-motor to 120 degrees

-1) Exit program
Enter command: 5
PIC has queued command to turn senso-motor to 120 degrees

Options
0) Reset
1) Ping
2) Get ADC value
3) Turn Senso-motor to 30 degrees
4) Turn Senso-motor to 90 degrees
5) Turn Senso-motor to 120 degrees

-1) Exit program
Enter command: 1
An unexpected error occurred.

Options
0) Reset
1) Ping
2) Get ADC value
3) Turn Senso-motor to 30 degrees
4) Turn Senso-motor to 90 degrees
5) Turn Senso-motor to 120 degrees

-1) Exit program
Enter command: -1
grep out /sys/class/gpio/gpio*/direction
/sys/class/gpio/gpio47/direction:out
/sys/class/gpio/gpio60/direction:out
grep in /sys/class/gpio/gpio*/direction
/sys/class/gpio/gpio0/direction:in
/sys/class/gpio/gpio57/direction:in
/sys/class/gpio/gpio59/direction:in
/sys/class/gpio/gpio63/direction:in
make: warning: Clock skew detected. Your build may be incomplete.
root@galileo:/media/usb-drive#
```

Figure 19: Command 4 and 5 give 90 and 120 degrees. iniUShows error when power is off on the pic microcontroller, signaling no communication is occurring.

Oscilloscope shots of differing duty cycles for each command

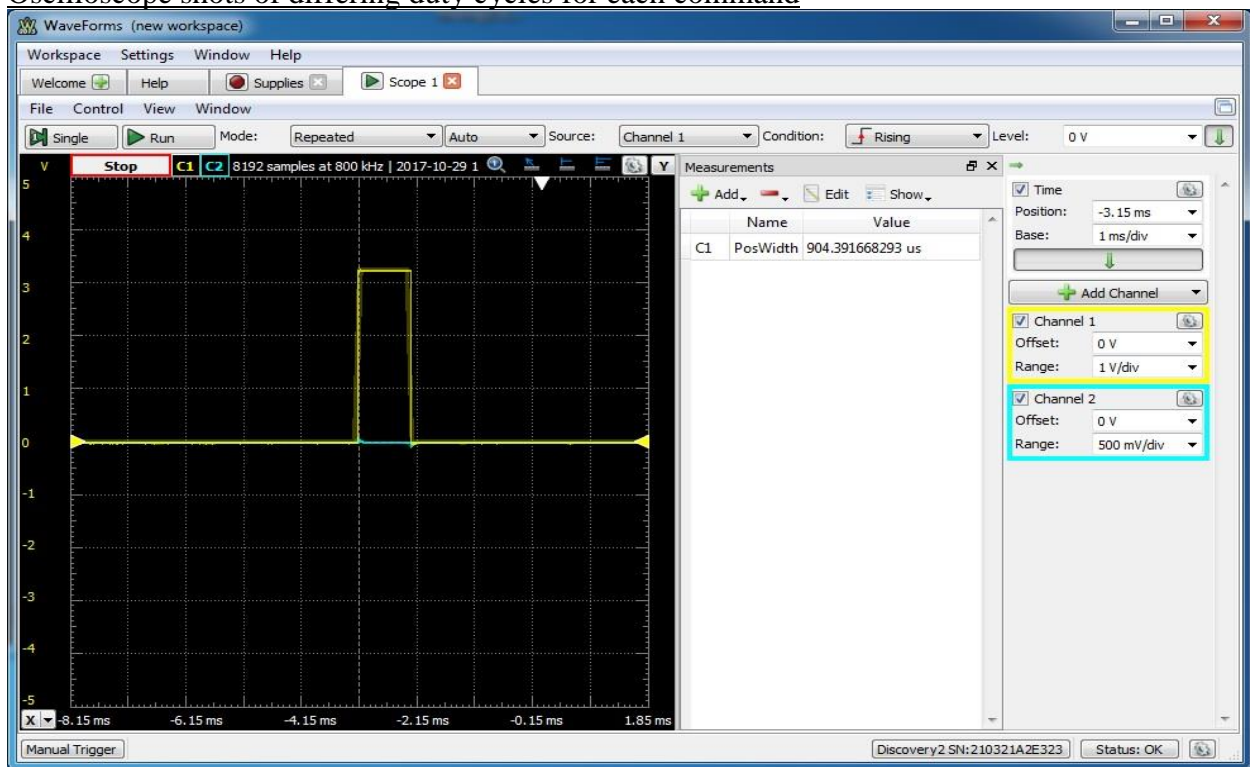


Figure 20: Showing 30 degree command PosWidth = 904.391 us

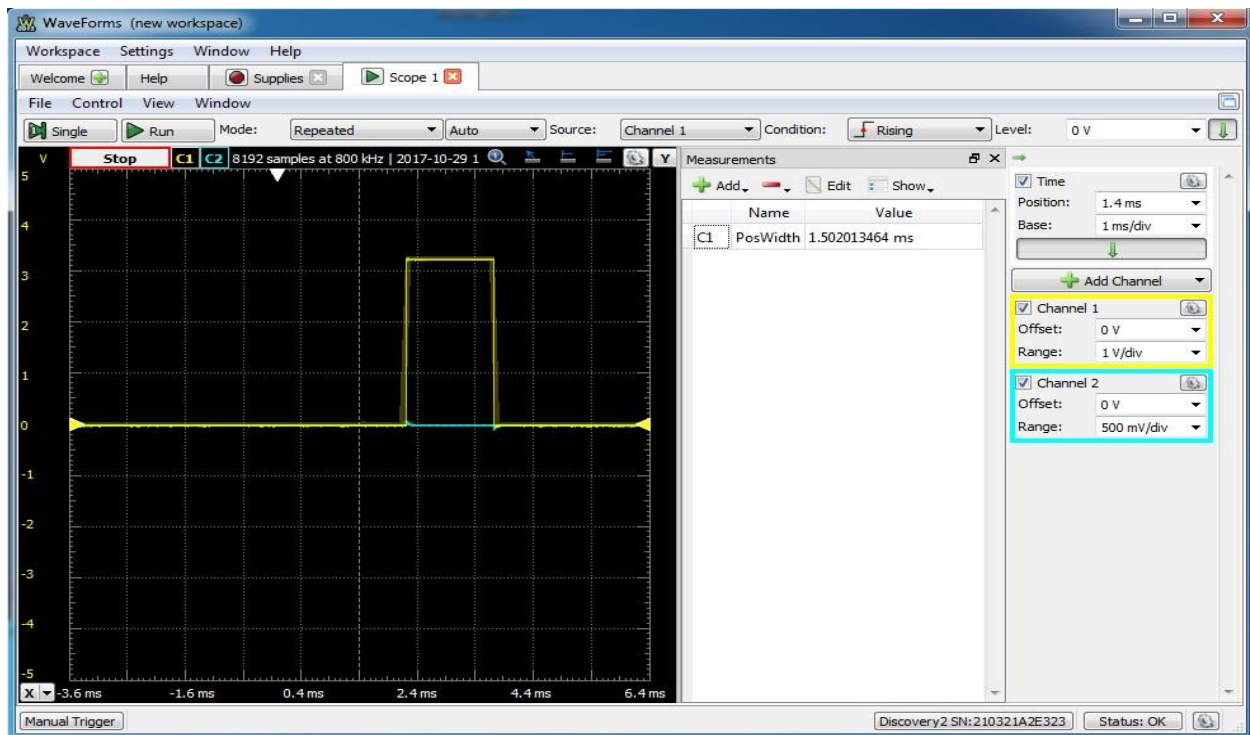


Figure 21: Showing 90 degree command PosWidth = 1.502 ms

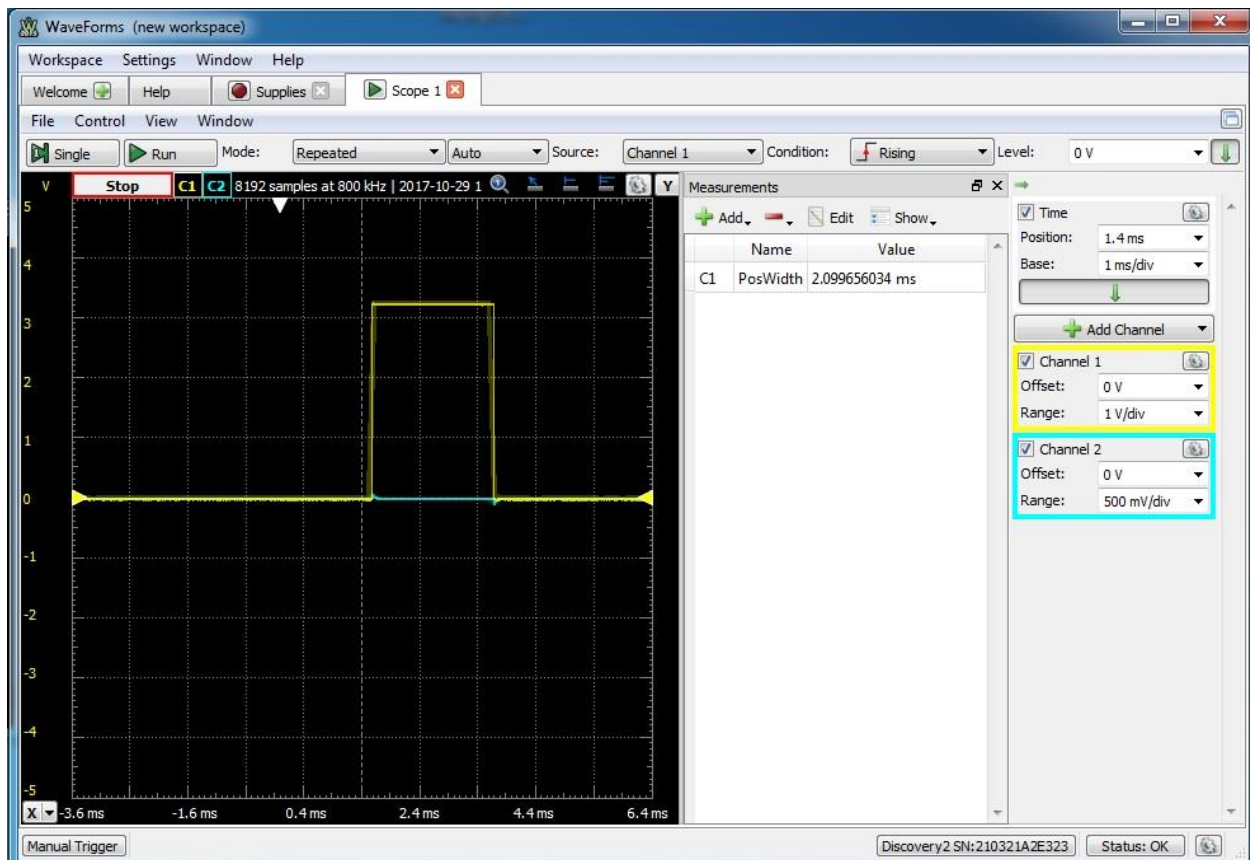


Figure 22: Showing 120 degree command PosWidth = 2.0996 ms

Section 10: Appendix

A1.

```
int main() {

    int input;                // user command input
    int response;             // response from PIC

    int flag;

    int strobe;                // handle for strobe signal
    int data [4];              // handles for data bus pins (A3-A0)

    strobe = openGPIO(Strobe, GPIO_DIRECTION_OUT);    // always out!
    writeGPIO(strobe, LOW);

    while (1) {

        puts("Options");
        puts("0) Reset");
        puts("1) Ping");
        puts("2) Get ADC value");
        puts("3) Turn Senso-motor to 30 degrees");
        puts("4) Turn Senso-motor to 90 degrees");
        puts("5) Turn Senso-motor to 120 degrees\n");
        puts("-1) Exit program");

        do {
            printf("Enter command: ");
            flag = 1 - scanf("%d", &input);

            if (flag) {

                puts("ERROR: Invalid number input\n");

                // must flush line!

            } else if (input < -1 || input > 5) {
                // remember that -1 is valid input
                printf("Error: %d is an invalid option\n",
input);
                flag = -1;
            }

            flushLine();

        } while (flag);

        // exit if input = -1
        if (input < 0) {
            closeGPIO(Strobe, strobe);
            return 0;
        }

        /*
```

```

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*/

    // return status message
    if ((response & 0xF) == MSG_ACK) {
        // good response code
        switch (input) {
            case MSG_RESET:
                puts("PIC successfully reset");
                break;
            case MSG_PING:
                puts("PIC has returned ping");
                break;
            case MSG_GET:
                response = (response >> 4);
                // last 4 bits is MSG_ACK, upper 10 bits is data
                printf("Last ADC value: %d\nVoltage
across photoresistor: %lf\n", response, 3.3 * (double)response / 1023.0);
                break;
            case MSG_TURN30:
                puts("PIC has queued command to turn
senso-motor to 30 degrees");
                break;
            case MSG_TURN90:
                puts("PIC has queued command to turn
senso-motor to 90 degrees");
                break;
            case MSG_TURN120:
                puts("PIC has queued command to turn
senso-motor to 120 degrees");
                break;
        }
    } else {
        // if last 4 bits of response (the response code) are
bad

```

```

        puts("An unexpected error occurred.");R
    }
    puts("\n\n");

}

return 0;
}

```

A2.

```

if (PORTBbits.RB5 == (communication_counter % 2 ? LOW : HIGH)) {
    // interrupt is legit, so handle it, dumbass

    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
outputs
            TRISB &= 0xE1;           // set up

            ++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;
} // end of switch statement

}

IOCBFbits.IOCBF5 = LOW;    // clear flag

} // else if other flags to determine other sources of interrupt

}

```