University of British Columbia
Electrical and Computer Engineering
ELEC291/ELEC292 Winter 2024
Instructor: Dr. Jesus Calvino-Fraga
Section 201

# Project 2 - Metal Detector Robot

**Group: B9**

| Student # | Student Name | % Points | Signature |
|---|---|---|---|
| 13720222 | Abhishek Raghuwanshi | 99 | Abhishek |
| 30064273 | Daniel Kim | 114 | Daniel |
| 44123719 | Jason Xia | 96 | Jason |
| 33647371 | Jay Kim | 99 | Jay |
| 92210558 | Lance Yao | 99 | Lance |
| 94665239 | Noah Seo | 93 | Noah |

Date of Submission: April 9th, 2024

**TABLE OF CONTENTS**

# 1.0 INTRODUCTION

The Metal Detector Robot Project's objective is to create a robot capable of detecting metal objects, utilizing two microcontrollers from different families for its operation. This robot is controlled by an external remote, which can send instructions to the robot and receive the strength of metal detected from it.

The project specifications include:

- The use of two different microcontrollers from distinct families; one for the robot and another for the remote control.
- Both the robot and the remote control should be battery-operated.
- The programming language for the entire project must be C.
- The robot's motion, including moving forwards or backwards and turning, should be managed by Metal Oxide Semiconductor Field Effect Transistors (MOSFETs).
- A metal-detecting inductor must be installed on the robot, enabling it to identify any type of metal, specifically all types of Canadian coins in circulation.
- The remote control should have an LCD screen showing the metal detector's signal strength and a buzzer with increasing frequency of sound generated as the robot detects larger amounts of metal.
- Communication between the robot and the remote should be facilitated by two JDY-40 radio modules.

- The robot should be capable of executing smooth movements to form patterns such as figure eight or a square with adjustable speed and direction.

In our project, we chose the ATMega328p microcontroller for the robot and the PIC32 for the remote, aligning with the requirements by selecting these controllers from the megaAVR and MIPS families, respectively.

# 2.0 INVESTIGATION

## 2.1 IDEA GENERATION

The project started with the team recognizing the requirements of the project. Given the necessity to select two microcontrollers based on our specific goals and requirements, we examined the features and benefits of the microcontrollers at our disposal. Subsequently, we engaged in a group discussion to determine the most suitable pair of microcontrollers for our project. Since we had experience using both the PIC32 and ATmega, we settled on using those microcontrollers for our project.

For one of our extra features, we used an idea a member of our team had thought of for a previous lab involving a servo motor. This eventually became our special feature of an analog inductance meter.

## 2.2 INVESTIGATION DESIGN

Testing was necessary in various design segments, including:

- **Inductor Coil Loop**: To verify the inductor coil loop's approximate value of 1mH, we connected the coil's ends to the lab inductance meter
- **Metal Detector Precision**: To ensure the metal detector's measurements were accurate and consistent, we conducted tests using coins of five different sizes to observe if the quantity and type of metal detected and the time measurements displayed a regular pattern.

## 2.3 DATA COLLECTION

- **Inductor Coil Loop**: The inductance of the detector's inductor coil was measured using an inductance meter, and the values were documented. We repeated the procedure of winding the wire around the wheel until we achieved an inductance value that would detect metal accurately.
- **Metal Detector Accuracy**: We conducted multiple tests for each coin size and metal type and documented the observations.

## 2.4 DATA SYNTHESIS

- **Inductor Coil Loop**: We aimed to adjust the inductor coil's measured value to be near 1mH. We managed to get about 0.85mH, which was sufficiently low for our calculations and thus used it for our actual robot.
- **Metal Detector Accuracy**: From the measurements we had taken, we established the estimated period range for each coin size. We then used the formula $I = T$^(½) to calculate the inductance for each measurement. Our analysis of the relationship between the metal size (the coin) and inductance led

us to conclude that introducing more metal decreased both the period and the inductance.

## 2.5 ANALYSIS OF RESULTS

Before starting with our circuit assembly, we conducted tests on each component to ensure their values aligned with our expectations. The inductance of the coil was below the anticipated 1mH due to looping issues; however, we deemed the resulting inductance sufficient for our design needs and saw no need to change it. Ultimately, we verified that all individual components, including the metal detector, motors, period function, and joystick functioned seamlessly.

# 3.0 DESIGN

## 3.1 USE OF PROCESS

Our team adhered to the standard engineering design procedure, initiating our project by defining its objectives and requirements. Following this, we strategized on how to address the specific needs presented. Evaluating the materials available and the tasks that needed to be accomplished, we determined the most suitable equipment for each aspect of the project. To ensure a structured approach, we organized our tasks into sequential steps, aiming to complete each segment of the design systematically and logically.

## 3.2 NEED AND CONSTRAINT IDENTIFICATION

In our project, we pinpointed requirements across various design dimensions, including:

- **Motors**: We wanted the physical navigation of the robot to be straightforward and simple for the user, and this was achieved through coding the behaviour of the wheels depending on joystick values. Therefore, we added rotation on the spot as part of the robots movement.
- **Remote**: We aimed to make the remote control user-friendly. Recognizing that users might wish to maneuver the robot at varying speeds and directions, we designed the remote with capabilities to adjust the robot's speed and movement angle.
- **Robot**: To make adjustments and fix the robot easier for all parties involved, we made sure to keep the circuitry organized. This made it so that if there were any changes needed, they would be quick and concise.

## 3.3 PROBLEM SPECIFICATION

We pinpointed specific design needs in the following areas:

- **Motors**: Our evaluation revealed that the two motors did not rotate uniformly due to stiffness, necessitating a refinement of the gears within the motors to ensure smoother operation at consistent speeds.
- **Remote Control**: It was imperative for the remote to not only have an ergonomic size but also for the joystick to be positioned optimally to enhance user comfort during operation.
- **Robot Circuit**: The constrained space available required us to devise a circuit layout that was as compact as possible, addressing the challenge of fitting all necessary components within limited space.

## 3.4 SOLUTION GENERATION

We developed solutions for several design challenges as follows:

- Motors: To ensure smoother operation of the motors, we applied WD-40 lubricant oil to the gears until achieving uniform smoothness in both motors. Additionally, we adjusted screws that were excessively tightened at the factory to further facilitate smoother rotation.
- Remote Control: Opting for dual breadboards, we positioned the joystick in the top left section of the circuit, aiming to enhance operational convenience. Further details on the remote circuit layout are expanded in Section 3.x.
- Robot Circuit: A compact circuit arrangement was engineered, with specifics provided in Section 3.6.x.

- Master-Slave Communication: Initial attempts to establish a master-slave radio communication configuration were unsuccessful due to reliability and consistency issues.
- Movement: The joystick was configured to control the robot's speed and direction, utilizing different voltage readings from the joystick's x and y parameters.
- Auditory Feedback: We integrated a buzzer programmed to emit varying frequencies as auditory feedback, indicating the quantity of metal detected.

## 3.5 SOLUTION EVALUATION

With the implemented solutions, we achieved the outcomes we aimed for. By lubricating the gears with oil, we ensured that both motors operated uniformly, eliminating any previous discrepancies in their rotation. The adoption of a dual-breadboard configuration for the remote, incorporating the joystick in a strategic location, led to agreement among our team that this setup maximized operational ease and efficiency. This arrangement allowed for intuitive control, enhancing the user experience by providing manipulation of the robot's speed and direction.

Furthermore, the design approach we adopted for the robot's circuitry proved to be effective, resulting in efficient electrical layout. This planning and execution meant that our circuit was free from errors, contributing significantly to the reliability and performance of our project. Through careful consideration and application of these solutions, we not only met our initial design objectives but also enhanced the overall functionality and user interaction with the robot.

## 3.6 DETAILED DESIGN

### 3.6.1 METAL DETECTOR

To enable the robot to detect metal, we incorporated a metal detector circuit within its main circuitry. This circuit is illustrated in Figure 9 and features a basic yet effective design, comprising a CMOS inverter, a resistor, two capacitors, and the terminations of the inductor coil loop.

The CMOS inverter, a critical component of this setup, utilizes two types of MOSFETs: an FQU13N06LS N-Channel MOSFET and an FQU8P10 P-Channel MOSFET. These components work together to amplify the signal generated by the presence of metal near the inductor coil, enabling the robot to sense metal objects effectively. Our specific configuration of the metal detector demonstrates our practical application of these components to create a responsive and reliable metal detection mechanism within the robot.

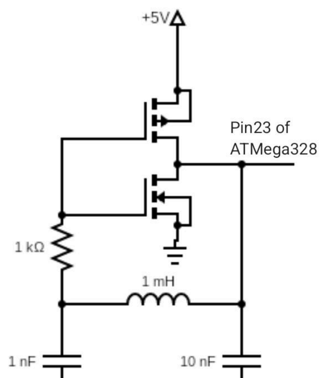

Figure 3. Metal Detector Circuit Diagram.

## 3.6.2 H-BRIDGE WITH OPTOCOUPLER

We used a combination of N-channel and P-channel Mosfets to make an “H” circuit together with optocouplers to connect the wheel motors to the ATMega microcontroller. The optocouplers are used so that we could include the two motors into a complete circuit with the microcontrollers, while having two different voltage sources. The motors required a higher voltage source of 6 volts, and the microcontroller uses a lower voltage. This meant that we also needed two different grounds for the two power sources.

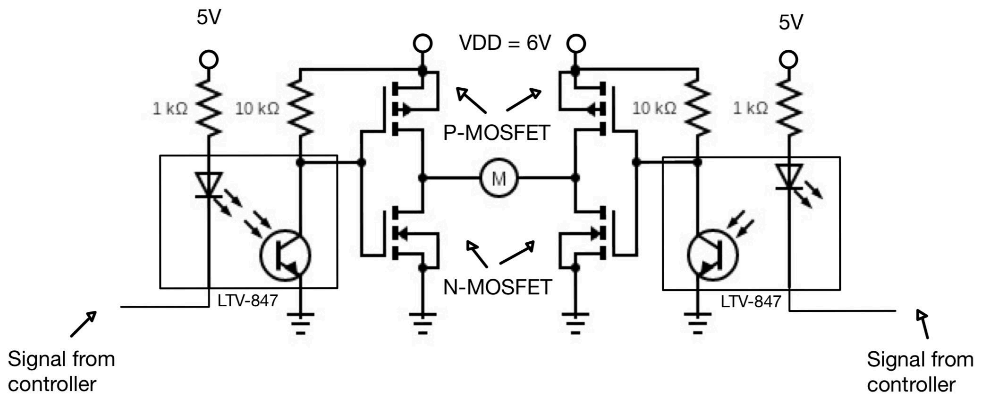

Figure 4.1 H-bridge

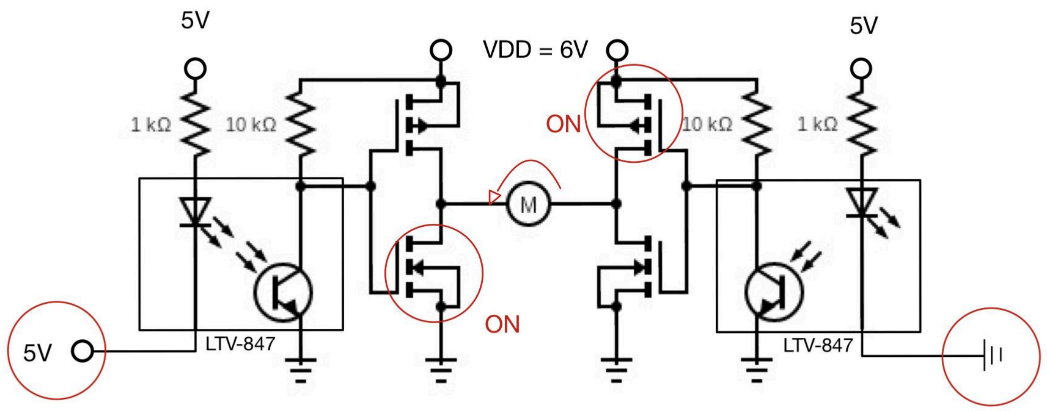

Figure 4.2 H-bridge CCW

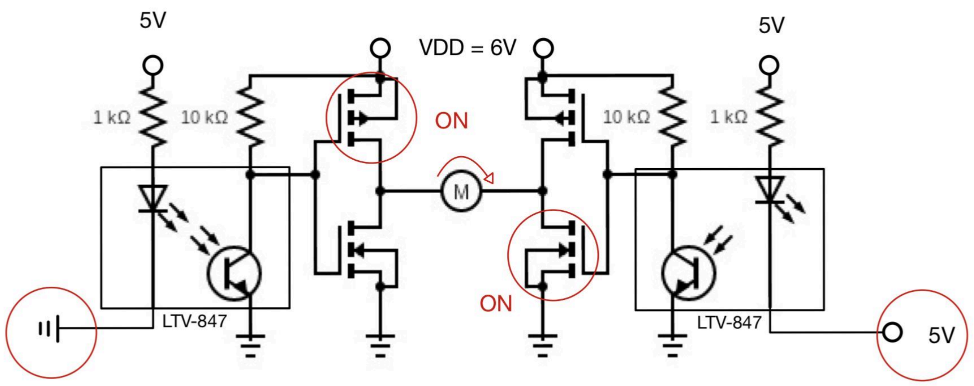

Figure 4.3 H-bridge CW

## 3.6.3 RADIO

We employed the JDY-40 radio modules to facilitate communication between the remote and the robot, installing one radio in each circuit. This setup enabled the remote to transmit Pulse Width Modulation (PWM) signals to the robot, dictating its speed and direction based on the joystick movements. Furthermore, we leveraged the radio's capabilities to switch between different operational modes, such as remote control and autopilot, through distinct signal patterns.

On the robot's end, the communication was primarily outbound, focusing on sending back the calculated period value, which represents the detected metal's inductance, to the remote. This allowed for real-time metal detection feedback. The design ensured that each message, whether from the robot or the remote, was transmitted and received within a 1ms timeframe. This rapid exchange guaranteed that, without any significant signal interference, the robot's responses to remote commands were executed with virtually no delay, ensuring smooth and responsive control over its movements and functions.

## 3.6.4 JOYSTICK

The primary function of the joystick in our project is to dictate the speed and direction of the robot. Manipulating the joystick will create variable resistance values that can be read using the PIC32’s ADC pins. These ADC readings can be turned into voltage values and then further converted into a range of integers for various levels of speed and direction control. From there, these integers are sent to the robot through the JDY40 and converted into PWM signals for the servos.

However, an issue we had was the joystick's poor voltage range when compared to its actual movement range. Because of this, a small physical movement might involve a large voltage change and thus a large jump in speed. Further compounding this was the ADC's inability to read values above 3.3V and thus we had to supply our joystick with 3.3V rather than the typical 5V. Our solution was simply to move the joystick very carefully and reduce the number of integer values to maintain a level of perceptible speed control.

## 3.6.5 ROBOT SOFTWARE

**JDY40**

Initially, the robot waits for something to be received from the remote. When something is received, the robot checks what the character is. If an exclamation mark (!) is received, the robot stores a new string sent by the remote following the exclamation mark. The string contains 2 characters. One for the X value of the joystick, and one for the Y value of the joystick. These characters are converted into integer values to be used for the PWM. If the character received is instead a commercial at (@), the frequency is calculated and sent to the remote. We added a 5ms delay before send function, SendString1, to match the timing with the remote.

**PWM**

We employed PWM to power and control the speed of the wheels. This included adding code in the ISR (Interrupt Service Routine) to implement the PWM, and a function that decides the PWM values based on the joystick coordinates received from the remote. For the two motors, we had 4 unique PWM values, two for each motor with one in the backward and one in the forward direction. We assigned flat values to the PWM variables based on the Y joystick value, and then used the X joystick value to adjust those flat values to turn in the desired direction.

## Frequency

To obtain the frequency, we used the function, GetPeriod, from the sample code provided. The interrupt service routine (ISR) had to be disabled before using GetPeriod, so it was disabled and enabled immediately following the function call. The value from GetPeriod had to be divided by the clock period, 16MHz, and the number of periods we input in the function, 100. Then, by taking the inverse of the period, we obtained the frequency to send to the remote.

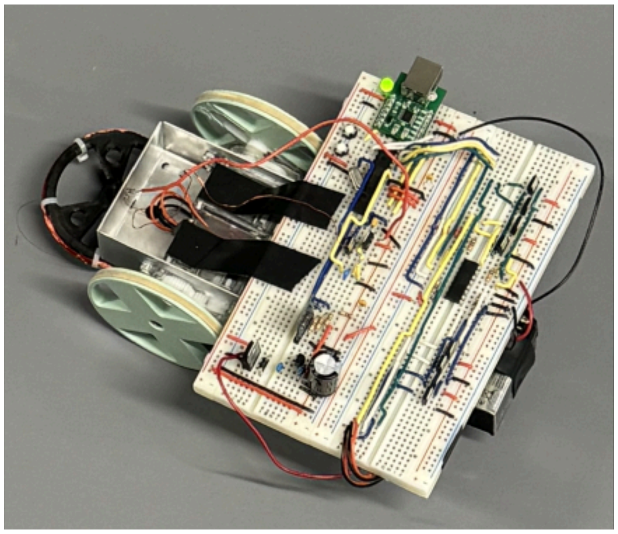

Figure 5. Robot and robot hardware

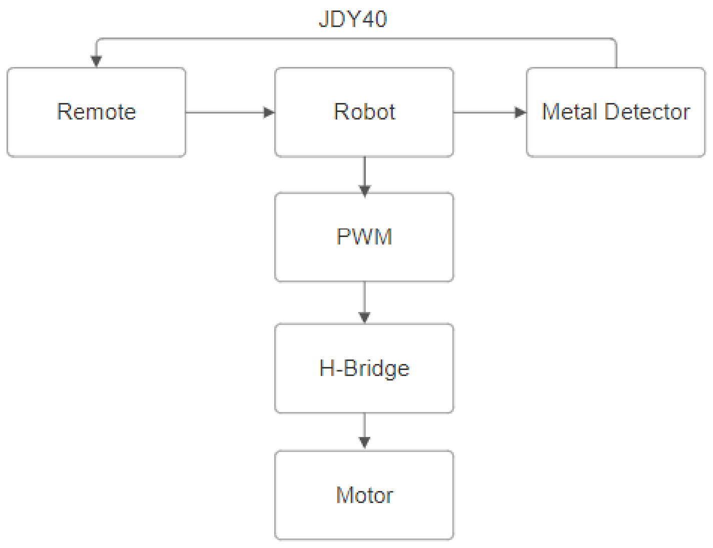

Figure 6. Robot Hardware Block Diagram

## 3.6.6 REMOTE SOFTWARE

**Joystick**

The software component of the remote serves as an interaction between the physical control of the joystick and the robot. Using the ADCread function, we read the ADC value of the x and y dimension of the joystick. The read ADC value is then converted to voltages to make the comparison between the ADC voltage and the VDD. For each range of ADC voltage, we assign a value from 0 to 6 so that the speed control is possible. The x value of the joystick, which ranges from 0 to 6, is contained in the variable joy_x and the y value of the joystick is contained in the variable joy_y. The x and y values are then contained to the buffer[0] and buffer[1], respectively.

**Transmission**

For the communication between the robot and the remote, the remote acts as a master, therefore the transmission happens only when the remote wants to. The detailed explanation of our communication is in section 3.6.5. In a nutshell, the remote sends a buffer, the x and y values of the joystick, to the robot using the SerialTransmit function. The remote then sends '@' to the robot using putc function to tell the robot that the remote wants something from the robot. Using a timeout loop, we wait for the U1STAbits.URXDA flag to be 1, which is the flag that is set to 1 when the message has arrived. Here, we receive frequency information that is detected from the robot.

**Buzzer**

We have made a Buzzer function such that if we input an integer value from 1, it outputs a square wave that is used to generate a sound in our speaker. With testing and experimentation, we have made distinct frequency ranges for each metal that is being detected. For the low frequencies, we create a sound that has a low note and high note for the high frequencies.

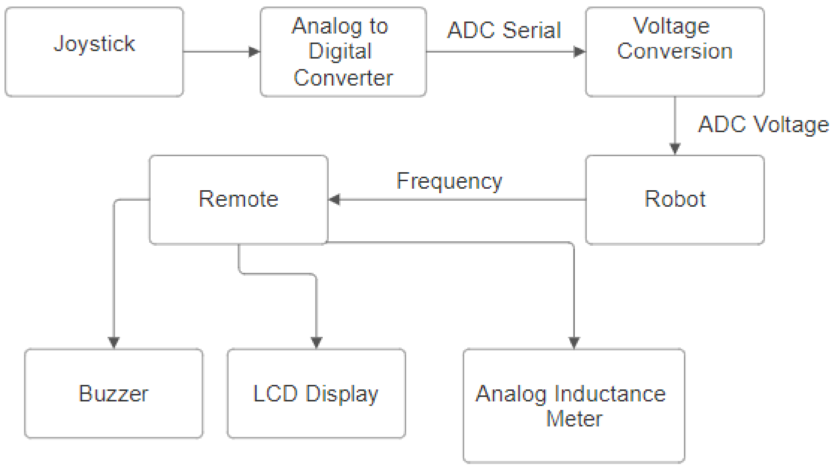

Figure 7. Remote Hardware Block Diagram

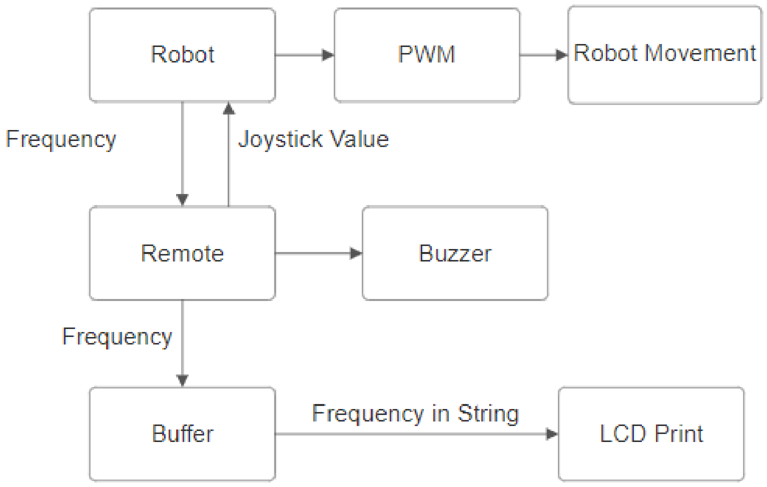

Figure 8. Software Block Diagram

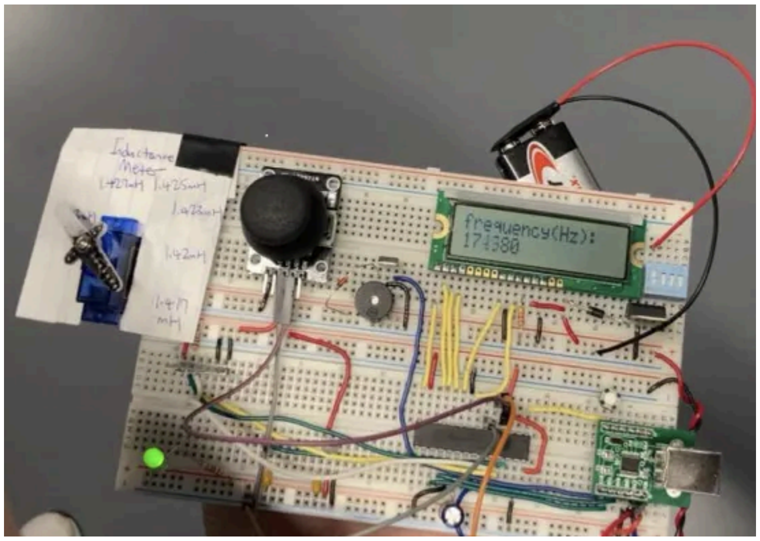

Figure 9. Remote and remote hardware

## 3.7 SOLUTION ASSESSMENT

Following selection processes and testing, we finalized the design of both the robot and its remote control, achieving our desired outcomes. We verified each feature integrated into the robot, ensuring it performs efficiently as a unit. The robot now executes commands without delay in optimal conditions and navigates smoothly, whether turning or moving straight. The remote control has been thoughtfully developed, enhancing its usability with features such as exact frequency data display and metal inductance analog dial display using a servo motor, beyond basic controls. The servo utilises timer 2 of the PIC32 microcontroller as the first timer is used for the buzzer.

Despite these advancements, the robot's operation remains somewhat dependent on its surroundings. Excessive ambient signals can slow down signal reception and occasionally disrupt the basic connection, leading to the need for multiple restarts. While we've attempted to mitigate these issues by simplifying the signal transmission and enhancing the remote with additional components for stability, some problems persist. We also attempted to use a third microcontroller(STM32) to connect to the robot as a “software remote” connecting to Python, where we can send instructions to the robot with the keys on a computer's keyboard, such as using “WASD”keys for motion and “Shift+WASD” keys for moving fast. In the same python script, we utilized the matplot module to plot a graph of the incoming frequency against time so we could determine from the peaks and ignore any disturbances. Using the same inputs, it would also be possible to map the approximate location of the robot from the starting point and reverse it back into position by simply recording and sending the same commands in a backwards order. Unfortunately we ran into issues with the hardware initialisation that couldn’t be fixed in time to demo the project, but can definitely use the knowledge and experience in future projects. For future enhancements, we aim to further reduce the risk of damaging circuit components during operation.

# 4.0 LIVE LONG LEARNING

The Metal Detector Robot project provided various learning experiences, notably in bridging theoretical knowledge with practical application. Specifically, it underscored the importance of integrating diverse skills in microcontroller programming, breadboarding, and wireless communication to construct a functional metal-detecting robot. CPSC 259 and APSC 160 were the two most important courses in the programming of the microcontrollers. Without having a good understanding of C, this project would have been fundamentally impossible. Embedded C programming is quite a daunting challenge but it was greatly helped by a good understanding of memory and variable types learned in these courses.

This project not only filled a knowledge gap by offering hands-on experience in applying electronics and programming concepts to a tangible engineering challenge but also highlighted the critical roles of iterative testing, teamwork, and problem-solving in the engineering design process. From this, we all managed to gain relevant and credible experience towards engineering in a professional environment. Through addressing specific needs and constraints, such as motor uniformity and reliable communication, the project exemplified the practical challenges and solutions encountered in the field of electrical engineering.

# 5.0 CONCLUSIONS

In the end, we successfully built a robot that can detect metal and is controlled by a remote, all programmed in C. The remote uses a joystick to control the robot, and they communicate through a radio module. We have a buzzer that increases in frequency with the strength of the metal detected and a servo that acts as an inductance meter for the coil of wire. We have variable speed control based on joystick inputs and can neutral steer our robot when needed. Our robot is able to detect all Canadian coins in circulation reliably.

We struggled to figure out how to send the correct PWM signal to the motors to spin the wheels. Our batteries would drain too quickly so we had to buy new ones constantly. We had issues with the JDY-40 transmission but used examples to solve them. We had issues making the ISR for the buzzer work and had to slowly debug it. Lastly, our frequency calculation for the metal detector was often inaccurate or varied which we solved through battery replacement. Altogether, our team spent around 95 hours completing this project throughout all hours of the day.

## References

[1] ATmega328P datasheet.
https://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-7810-Automotive-Microcontrollers-ATmega328P_Datasheet.pdf

[2] PIC32MX130 datasheet.
https://ww1.microchip.com/downloads/en/DeviceDoc/PIC32MX1XX2XX%20283644-PIN_Datasheet_DS60001168L.pdf

[3] Sample C code and instructions provided by Professor Jesus Calviño-Fraga for the ELEC291 course at UBC.

[4] Calviño-Fraga, J., "Project 2 – Metal Detector Robot", University of British Columbia, Electrical and Computer Engineering, ELEC291/ELEC292, Department of Electrical and Computer Engineering, UBC.

[5] Calviño-Fraga, Jesús. "Project 2 - Metal Detector Robot Slide." University of British Columbia, Electrical and Computer Engineering, March 15, 2024.

[6] JDY40 manual
https://w.electrodragon.com/w/images/0/05/EY-40_English_manual.pdf

[7] STM32L051 datasheet
https://www.st.com/resource/en/datasheet/stm32l051c6.pdf

## Bibliography

Alexander, C. K., and Sadiku, M. "Fundamentals of Electric Circuits." McGraw-Hill, 4th ed., 2009.

Dally, William J., and R. Curtis Harting. "Digital Design: A Systems Approach." Cambridge University Press, 2012.

# APPENDIX

## APPENDIX A: ROBOT SOURCE CODE

```
#define CHARS_PER_LINE 16
#define F_CPU 16000000UL
#include <avr/io.h>
#include <avr/interrupt.h>
#include <stdio.h>
#include <stdlib.h>
#include <util/delay.h>
#include "usart.h"
#include "Software_UART.h"
#include <math.h>
```

```
#define BUFFER_SIZE 80
#define PIN_PERIOD (PINC & 0b00000001)
#include <string.h>
/* Pinout for DIP28 ATMega328P:

                           -------
     (PCINT14/RESET) PC6 -|1    28|- PC5 (ADC5/SCL/PCINT13)
       (PCINT16/RXD) PD0 -|2    27|- PC4 (ADC4/SDA/PCINT12)
       (PCINT17/TXD) PD1 -|3    26|- PC3 (ADC3/PCINT11)
      (PCINT18/INT0) PD2 -|4    25|- PC2 (ADC2/PCINT10)
 (PCINT19/OC2B/INT1) PD3 -|5    24|- PC1 (ADC1/PCINT9)
    (PCINT20/XCK/T0) PD4 -|6    23|- PC0 (ADC0/PCINT8)
                     VCC -|7    22|- GND
                     GND -|8    21|- AREF
(PCINT6/XTAL1/TOSC1) PB6 -|9    20|- AVCC
(PCINT7/XTAL2/TOSC2) PB7 -|10   19|- PB5 (SCK/PCINT5)
   (PCINT21/OC0B/T1) PD5 -|11   18|- PB4 (MISO/PCINT4)
 (PCINT22/OC0A/AIN0) PD6 -|12   17|- PB3 (MOSI/OC2A/PCINT3)
      (PCINT23/AIN1) PD7 -|13   16|- PB2 (SS/OC1B/PCINT2)
  (PCINT0/CLKO/ICP1) PB0 -|14   15|- PB1 (OC1A/PCINT1)
                           -------
*/
unsigned int cnt = 0;

void wait_1ms(void)
{
    unsigned int saved_TCNT1;

    saved_TCNT1=TCNT1;

    while((TCNT1-saved_TCNT1)<(F_CPU/1000L)); // Wait for 1 ms to pass
}

void waitms(int ms)
{
    while(ms--) wait_1ms();
}

void SendATCommand (char * s)
{
    char buff[40];
    printf("Command: %s", s);
    PORTD &= ~(BIT4); // 'set' pin to 0 is 'AT' mode.
    _delay_ms(10);
    SendString1(s);
    GetString1(buff, 40);
    PORTD |= BIT4; // 'set' pin to 1 is normal operation mode.
    _delay_ms(10);
    printf("Response: %s\r\n", buff);
}
/*
void Configure_Pins(void)
{
    DDRC|=0b11111110; // PC0 is input.
    DDRD|=0b11110000; // PD3, PD4, PD5, PD6, and PD7 are outputs.
}
*/
// GetPeriod() seems to work fine for frequencies between 30Hz and 300kHz.
int GetPeriod (int n)
{
    int i, overflow;
    unsigned int saved_TCNT1a, saved_TCNT1b;

    overflow=0;
    TIFR1=1; // TOV1 can be cleared by writing a logic one to its bit
location.  Check ATmega328P datasheet page 113.
    while (PIN_PERIOD!=0) // Wait for square wave to be 0
    {
        if(TIFR1&1) { TIFR1=1; overflow++; if(overflow>5) return 0;}
    }
    overflow=0;
    TIFR1=1;
    while (PIN_PERIOD==0) // Wait for square wave to be 1
    {
        if(TIFR1&1) { TIFR1=1; overflow++; if(overflow>5) return 0;}
    }

    overflow=0;
    TIFR1=1;
    saved_TCNT1a=TCNT1;
    for(i=0; i<n; i++) // Measure the time of 'n' periods
    {
        while (PIN_PERIOD!=0) // Wait for square wave to be 0
        {
            if(TIFR1&1) { TIFR1=1; overflow++; if(overflow>1024) return
0;}
        }
        while (PIN_PERIOD==0) // Wait for square wave to be 1
        {
            if(TIFR1&1) { TIFR1=1; overflow++; if(overflow>1024) return
0;}
        }
    }
    saved_TCNT1b=TCNT1;
    if(saved_TCNT1b<saved_TCNT1a) overflow--; // Added an extra overflow.
Get rid of it.

    return overflow*0x10000L+(saved_TCNT1b-saved_TCNT1a);
}

void int_to_char(int num, char result[]) {
    // Handle negative numbers
    int i;
    int index = 2;
    if (num < 0) {
        result[index++] = '-';
        num = -num;
    }

    // Handle zero explicitly, otherwise empty string is printed
    if (num == 0) {
        result[index++] = '0';
    }

    // Convert individual digits to characters
    char temp[20]; // Temporary array to store digits
    int temp_index = 0;
    while (num != 0) {
        int digit = num % 10;
        temp[temp_index++] = '0' + digit; // Convert digit to character
        num /= 10;
    }

    // Reverse the temporary array and copy it to the result array
    for (i = temp_index - 1; i >= 0; i--) {
        result[index++] = temp[i];
    }

    // Add null terminator to indicate end of string
    result[index] = '\0';
}

//const int QRD_pin = PC1;

/*void adc_init(void)
{ADMUX = (1<<REFS0);
ADCSRA = (1<<ADEN)|(1<<ADPS2)|(1<<ADPS1)|(1<<ADPS0);
}
void loop()
{int proximityADC = adc_read(PC1);
 float proximityV = (float)proximityADC*5.0/1023.0;
 printf(proximityV);
 _delay_ms(100);
 } */


//void PWM_direc_shift(int PWM5, int PWM4, int PWM3, int PWM2) //if
//{


void PWM_to_motor(int xval, int yval)
{
    //y val - speed backwards or forwards
    if(yval==6){ //want full speed ahead
        PWM5=80;        //left for
        PWM4=0;         //left back
        PWM3=80;        //right for
        PWM2=0;         //right back
    }
    else if(yval==5){ // reduced speed forward
        PWM5=65;        //left for
        PWM4=0;         //left back
        PWM3=65;        //right for
        PWM2=0;         //right back
    }
    else if(yval==4){ // reduced speed forward
        PWM5=50;        //left for
        PWM4=0;         //left back
        PWM3=50;        //right for
        PWM2=0;         //right back
    }
    else if(yval==3){ // no movement
        PWM5=0;     //left for
        PWM4=0;         //left back
        PWM3=0;     //right for
        PWM2=0;         //right back
    }
    else if(yval==2){ // reduced speed backward
        PWM5=0;     //left for
        PWM4=50;            //left back
        PWM3=0;     //right for
        PWM2=50;            //right back
    }
    else if(yval==1){ // reduced speed backward
        PWM5=0;     //left for
        PWM4=65;            //left back
        PWM3=0;     //right for
        PWM2=65;            //right back
    }
    else if(yval==0){ // full speed backward
        PWM5=0;     //left for
        PWM4=80;            //left back
        PWM3=0;     //right for
        PWM2=80;            //right back
    }


    //now adjusting direction with x value
    if(xval==6){ //big turn right
```

```
    if(yval<=3){ //if going backwards, left back turns more
        if(yval==3){ //if y=0, then wheel selecion changes

        PWM5+=70;       //left for
        PWM2+=70;           //right back
        }
        else{
        PWM4+=20;       //left back
        PWM2-=20;       //right back
        }
    }
    //else, turning forward
    else{
    PWM5+=20;
    PWM3-=20;
    }
}

else if(xval==5){ // reduced speed right turn
    if(yval<=3){ //if going backwards, left back turns more
        if(yval==3){ //if y=0, then wheel selecion changes

        PWM5+=60;       //left for
        PWM2+=60;           //right back
        }
        else{
        PWM4+=15;       //left back
        PWM2-=15;       //right back
        }
    }
    //else, turning forward
    else{
    PWM5+=15;
    PWM3-=15;
    }
}

else if(xval==4){ // reduced speed right turn
    if(yval<=3){ //if going backwards, left back turns more
        if(yval==3){ //if y=0, then wheel selecion changes
        PWM5+=50;       //left for
        PWM2+=50;           //right back
        }
        else{
        PWM4+=10;       //left back
        PWM2-=10;       //right back
        }
    }
    //else, turning forward
    else{
    PWM5+=10;
    PWM3-=10;
    }
}
//else if(xval==3){ // no movement - no change, no consideration

//}

else if(xval==2){ // reduced speed left
    if(yval<=3){ //if going backwards, right back turns more
        if(yval==3){ //if y=0, then wheel selecion changes

        PWM4+=50;       //left back
        PWM3+=50;           //right for
        }
        else{
        PWM4-=10;       //left back
        PWM2+=10;       //right back
        }
    }
    //else, turning forward
    else{
    PWM5-=10;   //left front
    PWM3+=10;   //right front
    }
}

else if(xval==1){ // reduced speed left turn
    if(yval<=3){ //if going backwards, right back turns more
```

```
            if(yval==3){ //if y=0, then wheel selecion changes

            PWM4+=60;       //left back
            PWM3+=60;           //right for
            }
            else{
            PWM4-=15;       //left back
            PWM2+=15;       //right back
            }
        }
        //else, turning forward
        else{
        PWM5-=15;   //left front
        PWM3+=15;   //right front
        }
    }
    else if(xval==0){ // full speed left turn
        if(yval<=3){ //if going backwards, right back turns more
            if(yval==3){ //if y=0, then wheel selecion changes

            PWM4+=70;       //left back
            PWM3+=70;           //right for
            }
            else{
            PWM4-=20;       //left back
            PWM2+=20;       //right back
            }
        }
        //else, turning forward
        else{
        PWM5-=20;   //left front
        PWM3+=20;   //right front
        }
    }

}

void main (void)
{
    char buff[80];
    int cnt=0;
    char c;
    int xval;
    int yval;
    int count;
    float T, C;
    float f;
    unsigned long int f1;
    usart_init();   // configure the hardware usart and baudrate
    Init_Software_Uart(); // Configure the sorftware UART
    _delay_ms(500); // Give putty a chance to start before we send
information...
    printf("\r\nJDY-40 test\r\n");

    // We should select an unique device ID.  The device ID can be a hex
    // number from 0x0000 to 0xFFFF.  In this case is set to 0xABBA
    SendATCommand("AT+DVID1337\r\n");

    // To check configuration
    SendATCommand("AT+VER\r\n");
    SendATCommand("AT+BAUD\r\n");
    SendATCommand("AT+RFID\r\n");
    SendATCommand("AT+DVID\r\n");
    SendATCommand("AT+RFC\r\n");
    SendATCommand("AT+POWE\r\n");
    SendATCommand("AT+CLSS\r\n");

    cnt=0;
    while(1)
    {
        if(RXD_FLAG==1) // Something has arrived
        {
            c=GetByte1();

            if(c=='!') // Master is sending message
            {
                GetString1(buff, 80);
                xval=buff[0]-'0'; //char to int
                yval=buff[1]-'0';
                PWM_to_motor(xval, yval);
                printf("Master says: %s\r\n", buff);

            }
            else if(c=='@') // Master wants slave data
            {
                cli();
                count = GetPeriod(100);
                sei();
                T = count / (F_CPU * 100.0);
                f = 1 / (T) / 10;
                f1 = round(f);
                sprintf(buff, "%05d\n", f1);
                cnt++; // is this 16-bits?
                _delay_ms(5); // The radio seems to need this delay...
                SendString1(buff);
            }
        }
    }
}
```

## APPENDIX B: REMOTE SOURCE CODE

```
#include <XC.h>
#include <sys/attribs.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "lcd.h"

/* Pinout for DIP28 PIC32MX130:
                                          --------
                                   MCLR -|1     28|- AVDD
  VREF+/CVREF+/AN0/C3INC/RPA0/CTED1/RA0 -|2     27|- AVSS
        VREF-/CVREF-/AN1/RPA1/CTED2/RA1 -|3     26|-
AN9/C3INA/RPB15/SCK2/CTED6/PMCS1/RB15
   PGED1/AN2/C1IND/C2INB/C3IND/RPB0/RB0 -|4     25|-
CVREFOUT/AN10/C3INB/RPB14/SCK1/CTED5/PMWR/RB14
```

```
  PGEC1/AN3/C1INC/C2INA/RPB1/CTED12/RB1 -|5     24|-
AN11/RPB13/CTPLS/PMRD/RB13
   AN4/C1INB/C2IND/RPB2/SDA2/CTED13/RB2 -|6     23|- AN12/PMD0/RB12
     AN5/C1INA/C2INC/RTCC/RPB3/SCL2/RB3 -|7     22|-
PGEC2/TMS/RPB11/PMD1/RB11
                                    VSS -|8     21|-
PGED2/RPB10/CTED11/PMD2/RB10
                     OSC1/CLKI/RPA2/RA2 -|9     20|- VCAP
                OSC2/CLKO/RPA3/PMA0/RA3 -|10    19|- VSS
                         SOSCI/RPB4/RB4 -|11    18|-
TDO/RPB9/SDA1/CTED4/PMD3/RB9
         SOSCO/RPA4/T1CK/CTED9/PMA1/RA4 -|12    17|-
TCK/RPB8/SCL1/CTED10/PMD4/RB8
                                    VDD -|13    16|-
TDI/RPB7/CTED3/PMD5/INT0/RB7
                    PGED3/RPB5/PMD7/RB5 -|14    15|- PGEC3/RPB6/PMD6/RB6
                                          --------
*/

// Configuration Bits (somehow XC32 takes care of this)
#pragma config FNOSC = FRCPLL       // Internal Fast RC oscillator (8 MHz)
w/ PLL
#pragma config FPLLIDIV = DIV_2     // Divide FRC before PLL (now 4 MHz)
#pragma config FPLLMUL = MUL_20     // PLL Multiply (now 80 MHz)
#pragma config FPLLODIV = DIV_2     // Divide After PLL (now 40 MHz)

#pragma config FWDTEN = OFF         // Watchdog Timer Disabled
#pragma config FPBDIV = DIV_1       // PBCLK = SYCLK
#pragma config FSOSCEN = OFF

// Defines
#define SYSCLK 40000000L
#define DEF_FREQ 16000L
#define FREQ 100000L // We need the ISR for timer 2 every 10 us
#define Baud2BRG(desired_baud)( (SYSCLK / (16*desired_baud))-1)
#define Baud1BRG(desired_baud)( (SYSCLK / (16*desired_baud))-1)

volatile int ISR_pw=100, ISR_cnt=0, ISR_frc;

void __ISR(_TIMER_1_VECTOR, IPL5SOFT) Timer1_Handler(void)
```

```
{
    LATBbits.LATB6 = !LATBbits.LATB6; // Desired frequency on RB6
    IFS0CLR = _IFS0_T1IF_MASK; // Clear timer 1 interrupt flag, bit 4 of
IFS0
}

void __ISR(_TIMER_2_VECTOR, IPL5SOFT) Timer2_Handler(void)
{
    IFS0CLR=_IFS0_T2IF_MASK; // Clear timer 2 interrupt flag, bit 8 of
IFS0

    ISR_cnt++;
    if(ISR_cnt<ISR_pw)
    {
        LATBbits.LATB1 = 1;
    }
    else
    {
        LATBbits.LATB1 = 0;
    }
    if(ISR_cnt>=2000)
    {
        ISR_cnt=0; // 2000 * 10us=20ms
        ISR_frc++;
    }
}


void SetupTimer1(void)
{
    // Explanation here:
    // https://www.youtube.com/watch?v=bu6TTZHnMPY
    __builtin_disable_interrupts();
    PR1 = (SYSCLK / (DEF_FREQ * 2L)) - 1; // since SYSCLK/FREQ =
PS*(PR1+1)
    TMR1 = 0;
    T1CONbits.TCKPS = 0; // Pre-scaler: 1
    T1CONbits.TCS = 0; // Clock source
    T1CONbits.ON = 1;
    IPC1bits.T1IP = 5;
    IPC1bits.T1IS = 0;
    IFS0bits.T1IF = 0;
    IEC0bits.T1IE = 1;

    INTCONbits.MVEC = 1; //Int multi-vector
    __builtin_enable_interrupts();
}

unsigned int SerialReceive(char* buffer, unsigned int max_size)
{
    unsigned int num_char = 0;

    /* Wait for and store incoming data until either a carriage return is
received
     *   or the number of received characters (num_chars) exceeds max_size
*/
    while (num_char < max_size)
    {
        while (!U2STAbits.URXDA);   // wait until data available in RX
buffer
        *buffer = U2RXREG;          // empty contents of RX buffer into
*buffer pointer

        while (U2STAbits.UTXBF);    // wait while TX buffer full
        U2TXREG = *buffer;          // echo

        // insert nul character to indicate end of string
        if (*buffer == '\r')
        {
            *buffer = '\0';
            break;
        }

        buffer++;
        num_char++;
    }

    return num_char;
}
```

```
void UART2Configure(int baud_rate)
{
    // Peripheral Pin Select
    U2RXRbits.U2RXR = 4;    //SET RX to RB8
    RPB9Rbits.RPB9R = 2;    //SET RB9 to TX

    U2MODE = 0;         // disable autobaud, TX and RX enabled only, 8N1,
idle=HIGH
    U2STA = 0x1400;     // enable TX and RX
    U2BRG = Baud2BRG(baud_rate); // U2BRG = (FPb / (16*baud)) - 1

    U2MODESET = 0x8000;     // enable UART2
}


void ADCConf(void)
{
    AD1CON1CLR = 0x8000;    // disable ADC before configuration
    AD1CON1 = 0x00E0;       // internal counter ends sampling and starts
conversion (auto-convert), manual sample
    AD1CON2 = 0;            // AD1CON2<15:13> set voltage reference to
pins AVSS/AVDD
    AD1CON3 = 0x0f01;       // TAD = 4*TPB, acquisition time = 15*TAD
    AD1CON1SET = 0x8000;      // Enable ADC
}

void delay_ms(int msecs)
{
    int ticks;
    ISR_frc = 0;
    ticks = msecs / 20;
    while (ISR_frc < ticks);
}


int ADCRead(char analogPIN)
{
    AD1CHS = analogPIN << 16;    // AD1CHS<16:19> controls which analog
pin goes to the ADC
```

```
    AD1CON1bits.SAMP = 1;         // Begin sampling
    while (AD1CON1bits.SAMP);      // wait until acquisition is done
    while (!AD1CON1bits.DONE);     // wait until conversion done

    return ADC1BUF0;              // result stored in ADC1BUF0
}


void ConfigurePins(void)
{
    // Configure pins as analog inputs
    ANSELBbits.ANSB2 = 1;    // set RB2 (AN4, pin 6 of DIP28) as analog pin
    TRISBbits.TRISB2 = 1;    // set RB2 as an input
    //ANSELBbits.ANSB3 = 1;    // set RB3 (AN5, pin 7 of DIP28) as analog
pin
    ANSELBbits.ANSB0 = 1;    // set RB0 (AN2, pin 4 of DIP28) as analog pin
    TRISBbits.TRISB0 = 1;    // set RB0 as an input

    // Configure digital input pin to measure signal period
    ANSELB &= ~(1 << 6); // Set RB6 as a digital I/O (pin 15 of DIP28)
    TRISB |= (1 << 6);   // configure pin RB6 as input
    CNPUB |= (1 << 6);   // Enable pull-up resistor for RB6

    // Configure output pins
    TRISAbits.TRISA0 = 0; // pin  2 of DIP28
    TRISAbits.TRISA1 = 0; // pin  3 of DIP28
    //TRISBbits.TRISB0 = 0; // pin  4 of DIP28
    TRISBbits.TRISB1 = 0; // pin  5 of DIP28
    TRISAbits.TRISA2 = 0; // pin  9 of DIP28
    TRISAbits.TRISA3 = 0; // pin 10 of DIP28
    TRISBbits.TRISB4 = 0; // pin 11 of DIP28
    INTCONbits.MVEC = 1;
}

// Needed to by scanf() and gets()
int _mon_getc(int canblock)
{
    char c;

    if (canblock)
    {
        while( !U2STAbits.URXDA); // wait (block) until data available in 
RX buffer
        c=U2RXREG;
        while( U2STAbits.UTXBF);    // wait while TX buffer full
        U2TXREG = c;          // echo
        if(c=='\r') c='\n'; // When using PUTTY, pressing <Enter> sends 
'\r'.  Ctrl-J sends '\n'
        return (int)c;
    }
    else
    {
        if (U2STAbits.URXDA) // if data available in RX buffer
        {
            c=U2RXREG;
            if(c=='\r') c='\n';
            return (int)c;
        }
        else
        {
            return -1; // no characters to return
        }
    }
}

/////////////////////////////////////////////////////////
// UART1 functions used to communicate with the JDY40  //
/////////////////////////////////////////////////////////

// TXD1 is in pin 26
// RXD1 is in pin 24

int UART1Configure(int desired_baud)
{
    int actual_baud;

    // Peripheral Pin Select for UART1.  These are the pins that can be 
used for U1RX from TABLE 11-1 of '60001168J.pdf':
    // 0000 = RPA2
    // 0001 = RPB6
    // 0010 = RPA4
    // 0011 = RPB13
    // 0100 = RPB2

    // Do what the caption of FIGURE 11-2 in '60001168J.pdf' says: "For
input only, PPS functionality does not have
    // priority over TRISx settings. Therefore, when configuring RPn pin
for input, the corresponding bit in the
    // TRISx register must also be configured for input (set to ??."

    ANSELB &= ~(1<<13); // Set RB13 as a digital I/O
    TRISB |= (1<<13);   // configure pin RB13 as input
    CNPUB |= (1<<13);   // Enable pull-up resistor for RB13
    U1RXRbits.U1RXR = 3; // SET U1RX to RB13

    // These are the pins that can be used for U1TX. Check table TABLE
11-2 of '60001168J.pdf':
    // RPA0
    // RPB3
    // RPB4
    // RPB15
    // RPB7

    ANSELB &= ~(1<<15); // Set RB15 as a digital I/O
    RPB15Rbits.RPB15R = 1; // SET RB15 to U1TX

    U1MODE = 0;         // disable autobaud, TX and RX enabled only, 8N1,
idle=HIGH
    U1STA = 0x1400;     // enable TX and RX
    U1BRG = Baud1BRG(desired_baud); // U1BRG = (FPb / (16*baud)) - 1
    // Calculate actual baud rate
    actual_baud = SYSCLK / (16 * (U1BRG+1));

    U1MODESET = 0x8000;     // enable UART1

    return actual_baud;
}

void putc1 (char c)
{
    while( U1STAbits.UTXBF);    // wait while TX buffer full
    U1TXREG = c;                // send single character to transmit buffer
}

int SerialTransmit1(const char *buffer)
{
    unsigned int size = strlen(buffer);
    while(size)
    {
        while( U1STAbits.UTXBF);    // wait while TX buffer full
        U1TXREG = *buffer;          // send single character to transmit
buffer
        buffer++;                   // transmit next character on
following loop
        size--;                     // loop until all characters sent
(when size = 0)
    }

    while( !U1STAbits.TRMT);        // wait for last transmission to
finish

    return 0;
}

unsigned int SerialReceive1(char *buffer, unsigned int max_size)
{
    unsigned int num_char = 0;

    while(num_char < max_size)
    {
        while( !U1STAbits.URXDA);   // wait until data available in RX
buffer
        *buffer = U1RXREG;          // empty contents of RX buffer into
*buffer pointer

        // insert nul character to indicate end of string
        if( *buffer == '\n')
        {
            *buffer = '\0';
            break;
        }

        buffer++;
        num_char++;
    }

    return num_char;
}

void wait_100us(void)
{
    _CP0_SET_COUNT(0); // resets the core timer count

    // get the core timer count
    while ( _CP0_GET_COUNT() < ( (SYSCLK)/(2*10000) ) );
}


// Use the core timer to wait for 1 ms.
void wait_1ms(void)
{
    unsigned int ui;
    _CP0_SET_COUNT(0); // resets the core timer count

    // get the core timer count
    while ( _CP0_GET_COUNT() < (SYSCLK/(2*1000)) );
}

void delayms(int len)
{
    while(len--) wait_1ms();
}

void SendATCommand (char * s)
{
    char buff[40];
    printf("Command: %s", s);
    LATB &= ~(1<<14); // 'SET' pin of JDY40 to 0 is 'AT' mode.
    delayms(10);
    SerialTransmit1(s);
    SerialReceive1(buff, sizeof(buff)-1);
    LATB |= 1<<14; // 'SET' pin of JDY40 to 1 is normal operation mode.
    delayms(10);
    printf("Response: %s\n", buff);
}

void SetupTimer2(void)
{
    __builtin_disable_interrupts();
    PR2 = (SYSCLK / FREQ) - 1; // set period register
    TMR2 = 0; // reset timer
    T2CONbits.TCKPS = 0; // set prescaler to 1
    T2CONbits.TCS = 0; // set internal clock source (peripheral clock)
    T2CONbits.ON = 1; // turn on Timer 2
    IPC2bits.T2IP = 5; // set interrupt priority
    IPC2bits.T2IS = 0; // set subpriority
    IFS0bits.T2IF = 0; // clear interrupt flag
    IEC0bits.T2IE = 1; // enable interrupt
    INTCONbits.MVEC = 1; // enable multi-vector interrupts
    __builtin_enable_interrupts();
}

void int_to_char(int num, char result[], int start_index) {
    // Handle negative numbers
    int index = start_index;
    int i;
    if (num < 0) {
        result[index++] = '-';
        num = -num;
    }

    // Handle zero explicitly, otherwise empty string is printed
    if (num == 0) {
        result[index++] = '0';
    }

    // Convert individual digits to characters
    char temp[20]; // Temporary array to store digits
    int temp_index = 0;
    while (num != 0) {
        int digit = num % 10;
        temp[temp_index++] = '0' + digit; // Convert digit to character
        num /= 10;
    }

    // Reverse the temporary array and copy it to the result array
    for (i = temp_index - 1; i >= 0; i--) {
        result[index++] = temp[i];
    }

    // Add null terminator to indicate end of string
    result[index] = '\0';
}

void buzzer(int newF)
{
    unsigned long reload;

    reload = (SYSCLK / (newF * 2L))/1000 - 1;
    //SerialTransmit("\r\nFrequency set to: ");
                //PrintNumber(SYSCLK / ((reload + 1) * 2L), 10, 1);
    T1CONbits.ON = 0;
    PR1 = reload;
    T1CONbits.ON = 1;
}

void main(void)
{
    char buff[80];
    int cont1=0, cont2=100;
    int cnt = 0;
    int joy_x, joy_y;
    int adcvalX;
    int adcvalY;
    int timeout_cnt;
    float X;
    float Y;
    volatile unsigned long t = 0;
    int frequency;
    char freqLCD[6];
    int adcval;
    long int vx, vy;
    unsigned long int count, f;
    unsigned char LED_toggle = 0;
    unsigned int rx_size;
    int newF;
    unsigned long reload;


    DDPCON = 0;
    CFGCON = 0;

    UART2Configure(115200);  // Configure UART2 for a baud rate of 115200
    UART1Configure(9600);  // Configure UART1 to communicate with JDY40
with a baud rate of 9600
    LCD_4BIT();
    ConfigurePins();
    ADCConf(); // Configure ADC
    LCDprint("frequency(Hz):", 1, 0);


    delayms(500); // Give putty time to start before we send stuff.
    printf("JDY40 test program. PIC32 behaving as Master.\r\n");

    //Initialize pin 5 or RB1 for square wave
    TRISBbits.TRISB6 = 0;
    LATBbits.LATB6 = 0;
    INTCONbits.MVEC = 1;
    SetupTimer1();
    SetupTimer2();
    //SetupTimer1(frequency);

    // RB14 is connected to the 'SET' pin of the JDY40.  Configure as
output:
    ANSELB &= ~(1<<14); // Set RB14 as a digital I/O
    TRISB &= ~(1<<14);  // configure pin RB14 as output
    LATB |= (1<<14);    // 'SET' pin of JDY40 to 1 is normal operation
mode

    // We should select an unique device ID.  The device ID can be a hex
    // number from 0x0000 to 0xFFFF.  In this case is set to 0xABBA
    SendATCommand("AT+DVID1337\r\n");

    // To check configuration
    SendATCommand("AT+VER\r\n");
    SendATCommand("AT+BAUD\r\n");
    SendATCommand("AT+RFID\r\n");
    SendATCommand("AT+DVID\r\n");
    SendATCommand("AT+RFC\r\n");
    SendATCommand("AT+POWE\r\n");
    SendATCommand("AT+CLSS\r\n");

    delay_ms(500); // wait 500 ms

    printf("\x1b[2J\x1b[1;1H"); // Clear screen using ANSI escape
sequence.
    printf("Servo signal generator for the PIC32MX130F064B.  Output is in
RB6 (pin 15).\r\n");
    printf("By Jesus Calvino-Fraga (c) 2018.\r\n");
    printf("Pulse width between 60 (for 0.6ms) and 240 (for 2.4ms)\r\n");

    //SetupTimer1(4048);
    while(1)
    {
        //buzzer(1048);

        adcvalX = ADCRead(2); // note that we call pin AN4 (RB2) by it's
analog number
        vx = (adcvalX * 3290L) / 1023L; // 3.290 is VDD

        if (vx <= 3290 && vx >= 2741)
        {
            joy_x = 6;
        }
        else if (vx < 2741 && vx >= 2192)
        {
            joy_x = 5;
        }
        else if (vx < 2192 && vx>1643)
        {
```

```
    joy_x = 4;
}
else if (vx < 481 && vx >= 0)
{
    joy_x = 0;
}
else if (vx < 1443 && vx >= 962)
{
    joy_x = 2;
}
else if (vx < 962 && vx >= 481)
{
    joy_x = 1;
}
else if (vx <= 1643 && vx >= 1443)
{
    joy_x = 3;
}

adcvalY = ADCRead(4);
vy = (adcvalY * 3290L) / 1023L; // 3.290 is VDD

if (vy <= 3290 && vy >= 2775)
{
    joy_y = 6;
}
else if (vy < 2775 && vy >= 2260)
{
    joy_y = 5;
}

else if (vy < 2260 && vy>1746)
{
    joy_y = 4;
}
else if (vy < 515 && vy >= 0)
{
    joy_y = 0;
}
else if (vy < 1546 && vy >= 1030)
```

```
        {
            joy_y = 2;
        }
        else if (vy < 1030 && vy >= 515)
        {
            joy_y = 1;
        }
        else if (vy <= 1746 && vy >= 1546)
        {
            joy_y = 3;
        }

        // Send a message to the slave. First send the 'attention'
character which is '!':
        putc1('!');
        // Wait a bit so the slave has a chance to get ready
        delayms(10);
        // Construct a message
        //sprintf(buff, "%03d,%03d\n", cont1, cont2);
        sprintf(buff, "%d%d\n", joy_x, joy_y);
        //int_to_char(joy_x, buff, 0);
        //int_to_char(joy_y, buff, 1);
        // Send the message
        SerialTransmit1(buff);
        // Increment test counters for next message
        if(++cont1>200) cont1=0;
        if(++cont2>200) cont2=0;

        delayms(5);
        // Request a message from the slave
        //if(U1STAbits.URXDA) SerialReceive1(buff, sizeof(buff)-1);

        putc1('@');

        // Wait up to 10ms for the repply
        timeout_cnt=0;
        while(1)
        {
            if(U1STAbits.URXDA) break; // Something has arrived
            if(++timeout_cnt>100) break;
            wait_100us(); // 100us*100=10ms
        }

        if(U1STAbits.URXDA) // Something has arrived from slave
        {
            rx_size = SerialReceive1(buff, sizeof(buff)-1); // wait here
until data is received

            freqLCD[0] = buff[0];
            freqLCD[1] = buff[1];
            freqLCD[2] = buff[2];
            freqLCD[3] = buff[3];
            freqLCD[4] = buff[4];
            freqLCD[5] = '0';
            printf("frequency is %s", freqLCD);

            LCDprint(freqLCD, 2, 0);
            LCDprint("      ", 2, 6);
            frequency = atoi(freqLCD);

            if (rx_size > 0)
            {
                newF = frequency;
                //printf("%d", newF);
                if (newF > 200000L)
                {
                    //SerialTransmit("Warning: High frequencies will cause
the interrupt service routine for\r\n"
                    //    "the timer to take all available processor time.
Capping to 200000Hz.\r\n");
                    newF = 200000L;
                    //buzzer(1);
                }

                if (newF < 172600)
                {
                    buzzer(100);
                    ISR_pw = 60;

                }
                else if (newF >= 172600 && newF < 173600)
                {
                    reload = (SYSCLK / (newF * 2L)+999)/1000 - 1;
                    buzzer(2);
                    ISR_pw = 100;
                }

                else if (newF >= 173600 && newF < 174600)
                {
                    buzzer(3);
                    ISR_pw = 140;
                }

                else if (newF >= 174600 && newF < 175600)
                {
                    buzzer(4);
                    ISR_pw=180;
                }

                else if (newF >= 175600 && newF < 176000)
                {
                    buzzer(5);
                    ISR_pw=240;
                }
                else
                {
                    buzzer(100);
                }

            }
            printf("Slave says: %s\r\n", buff);

        }
        else
        {
            printf("NO RESPONSE\r\n", buff);
        }
```

```
        delayms(50);  // Set the pace: communicate about every ~50ms
    }
}
```

## APPENDIX C: STM32 SOURCE CODE - Extra feature

```
#include "../Include/stm32l051xx.h"
#include "../Include/serial.h"

// Serial comms routine for the stm32l051 microcontroller.
// Makes use of USART1.  Pins PA9 and PA10 are used for
transmission/reception.
// Defines a new version of puts: e(mbedded)puts and egets
// Similar to puts and gets in standard C however egets checks the size
// of the input buffer.  This could be extended to include a timeout quite
easily.
// Written by Frank Duignan and modified by Jesus Calvino-Fraga
//

// define the size of the communications buffer (adjust to suit)
#define MAXBUFFER   64
typedef struct tagComBuffer{
    unsigned char Buffer[MAXBUFFER];
    unsigned Head,Tail;
    unsigned Count;
} ComBuffer;

ComBuffer ComRXBuffer, ComTXBuffer;

int PutBuf(ComBuffer  *Buf,unsigned char Data);
unsigned char GetBuf(ComBuffer  *Buf);
unsigned GetBufCount(ComBuffer  *Buf);
int ReadCom(int Max,unsigned char *Buffer);
int WriteCom(int Count,unsigned char *Buffer);
```

```
void usart_tx (void);
void usart_rx (void);
unsigned ComOpen;
unsigned ComError;
unsigned ComBusy;

int ReadCom(int Max,unsigned char *Buffer)
{
    // Read up to Max bytes from the communications buffer
    // into Buffer.  Return number of bytes read
    unsigned i;
    if (!ComOpen)
        return (-1);
    i=0;
    while ((i < Max-1) && (GetBufCount(&ComRXBuffer)))
        Buffer[i++] = GetBuf(&ComRXBuffer);
    if (i>0)
    {
        Buffer[i]=0;
        return(i);
    }
    else {
        return(0);
    }
}

int WriteCom(int Count, unsigned char *Buffer)
{
    // Writes Count bytes from Buffer into the the communications TX
buffer
    // returns -1 if the port is not open (configured)
    // returns -2 if the message is too big to be sent
    // If the transmitter is idle it will initiate interrupts by
    // writing the first character to the hardware transmit buffer
    unsigned i;
    if (!ComOpen) return (-1);

    // Wait for transmission to complete before sending more
    if(Count<MAXBUFFER)
    {
        while ( (MAXBUFFER - GetBufCount(&ComTXBuffer)) < Count );
    }
    else
    {
        return (-2);
    }

    for(i=0; i<Count; i++) PutBuf(&ComTXBuffer,Buffer[i]);

    if ( (USART1->CR1 & BIT3)==0) // transmitter was idle, turn it on and
force out first character
    {
        USART1->CR1 |= BIT3;
        USART1->TDR = GetBuf(&ComTXBuffer);
    }
    return 0;
}

void initUART(int BaudRate)
{
    int BaudRateDivisor;
    //int j;

     __disable_irq();
    ComRXBuffer.Head = ComRXBuffer.Tail = ComRXBuffer.Count = 0;
    ComTXBuffer.Head = ComTXBuffer.Tail = ComTXBuffer.Count = 0;
    ComOpen = 1;
    ComError = 0;

    // Turn on the clock for GPIOA (usart 1 uses it)
    RCC->IOPENR |= BIT0;

    BaudRateDivisor = 32000000; // assuming 32MHz clock
    BaudRateDivisor = BaudRateDivisor / (long) BaudRate;

    //Configure PA9 (TXD for USART1, pin 19 in LQFP32 package)
    GPIOA->OSPEEDR  |= BIT18; // MEDIUM SPEED
    GPIOA->OTYPER   &= ~BIT9; // Push-pull
    GPIOA->MODER    = (GPIOA->MODER & ~(BIT18)) | BIT19; // AF-Mode
    GPIOA->AFR[1]   |= BIT6 ; // AF4 selected
    //Configure PA10 (RXD for USART1, pin 20 in LQFP32 package)
    GPIOA->MODER    = (GPIOA->MODER & ~(BIT20)) | BIT21; // AF-Mode
    GPIOA->AFR[1]   |= BIT10;  // AF4 selected

    RCC->APB2ENR    |= BIT14; // Turn on the clock for the USART1
peripheral

    USART1->CR1 |= (BIT2 | BIT3 | BIT5 | BIT6); // Enable Transmitter,
Receiver, Transmit & Receive interrupts.
    USART1->CR2 = 0x00000000;
    USART1->CR3 = 0x00000000;
    USART1->BRR = BaudRateDivisor;
    USART1->CR1 |= BIT0; // Enable Usart 1

    /* Test to see if baud rate is configured correctly.  Mesure the time
of one bit.
       The inverse of that time is the baud rate.*/
    //for(j=0; j<1000; j++)
    //{
    //    USART1->ISR &= ~BIT7;
    //  USART1->TDR = 'U';
    //      while ( (USART1->ISR & BIT7)==0);
    //}

    NVIC->ISER[0] |= BIT27; // Enable USART1 interrupts in NVIC
    __enable_irq();
}

void USART1_Handler(void)
{
    // check which interrupt happened.
    if (USART1->ISR & BIT7) // is it a TXE interrupt?
        usart_tx();
    if (USART1->ISR & BIT5) // is it an RXNE interrupt?
        usart_rx();
}

void usart_rx (void)
{
    // Handles serial comms reception
    // simply puts the data into the buffer and sets the ComError flag
    // if the buffer is fullup
    if ( PutBuf(&ComRXBuffer, USART1->RDR) )
    {
        ComError = 1; // if PutBuf returns a non-zero value then there is
an error
    }
}

void usart_tx (void)
{
    // Handles serial comms transmission
    // When the transmitter is idle, this is called and the next byte
    // is sent (if there is one)
    if (GetBufCount(&ComTXBuffer)) USART1->TDR=GetBuf(&ComTXBuffer);
    else
    {
        // No more data, disable the transmitter
        USART1->CR1 &= ~BIT3;
        if (USART1->ISR & BIT6) USART1->ICR |= BIT6; // Write TCCF to
USART_ICR
        if (USART1->ISR & BIT7) USART1->RQR |= BIT4; // Write TXFRQ to
USART_RQR
    }
}

int PutBuf(ComBuffer *Buf, unsigned char Data)
{
    //while ( (Buf->Head==Buf->Tail) && (Buf->Count!=0));  /* OverFlow?
Wait... */
    if ( (Buf->Head==Buf->Tail) && (Buf->Count!=0)) return(1);  /*
OverFlow */
    __disable_irq();
    Buf->Buffer[Buf->Head++] = Data;
    Buf->Count++;
    if (Buf->Head==MAXBUFFER) Buf->Head=0;
    __enable_irq();
    return(0);
}
```

```
unsigned char GetBuf(ComBuffer *Buf)
{
    unsigned char Data;
    if ( Buf->Count==0 ) return (0);
    __disable_irq();
    Data = Buf->Buffer[Buf->Tail++];
    if (Buf->Tail == MAXBUFFER) Buf->Tail = 0;
    Buf->Count--;
    __enable_irq();
    return (Data);
}

unsigned int GetBufCount(ComBuffer *Buf)
{
    return Buf->Count;
}

int eputs(char *s)
{
    // only writes to the comms port at the moment
    if (!ComOpen) return -1;
    while (*s) WriteCom(1,s++);
    return 0;
}

void eputc(char c)
{
    WriteCom(1,&c);
}

char egetc(void)
{
    return GetBuf(&ComRXBuffer);
}

int egets(char *s, int Max)
{
    // read from the comms port until end of string
    // or newline is encountered.  Buffer is terminated with null
    // returns number of characters read on success
    // returns 0 or -1 if error occurs
    // Warning: This is a blocking call.
    int Len;
    char c;
    if (!ComOpen) return -1;
    Len=0; // Until initialization code is put in init.s, we have to do
this
    c = 0;
    while ( (Len < Max-1) && (c != NEWLINE) )
    {
        while (!GetBufCount(&ComRXBuffer)); // wait for a character
        c = GetBuf(&ComRXBuffer);
        s[Len++] = c;
    }
    if (Len>0)
    {
        s[Len]=0;
    }
    return Len;
}

char egetc_echo(void)
{
    char c;
    c=egetc();
    eputc(c);
    return c;
}

int egets_echo(char *s, int Max)
{
    // read from the comms port until end of string
    // or newline is encountered.  Buffer is terminated with null
    // returns number of characters read on success
    // returns 0 or -1 if error occurs
    // Warning: This is a blocking call.
    int Len;
    char c;
    if (!ComOpen) return -1;
    Len=0; // Until initialization code is put in init.s, we have to do
this
    c = 0;
    while ( (Len < Max-1) && (c != NEWLINE) )
    {
        while (!GetBufCount(&ComRXBuffer)); // wait for a character
        c = GetBuf(&ComRXBuffer);
        eputc(c);
        s[Len++] = c;
    }
    if (Len>0)
    {
        s[Len]=0;
    }
    return Len;
}
```

## APPENDIX D: PYTHON SOURCE CODE - Extra feature

```
import keyboard
import serial
import matplotlib.pyplot as plt
from collections import deque

# Function to print direction based on keyboard input
def print_direction(event):
    if event.event_type == keyboard.KEY_DOWN:
        if keyboard.is_pressed('shift') and keyboard.is_pressed('w'):
            ser.write(b'6\n')
    elif event.name == 'w':
       ser.write(b'4\n')
    elif event.name == 'a':
        ser.write(b'1\n')

    elif event.name == 's':
        ser.write(b'0\n')
    elif event.name == 'd':
       ser.write(b'5\n')
```

```
# Register the key events
keyboard.on_press(print_direction)

# Set up the serial connection
ser = serial.Serial('COM5', 115200, timeout=1)  # Adjust port and baud
rate as needed
ser.flushInput()

# Create deque to store data for frequency plotting
#max_len_freq = 100  # Maximum number of data points to display
#frequency_data = deque(maxlen=max_len_freq)

# Initialize lists to store position data
#x_data = []
#y_data = []

# Create plots for frequency and position
plt.figure(figsize=(10, 5))

# Create plot for frequency
plt.subplot(1, 2, 1)
plt.title('Frequency Plot')
plt.xlabel('Time')
plt.ylabel('Frequency')
line_freq, = plt.plot(frequency_data)
plt.ylim(0, 100)  # Adjust y-axis limits as needed

# Create plot for position
#plt.subplot(1, 2, 2)
#plt.title('Position Plot')
#plt.xlabel('X Axis')
#plt.ylabel('Y Axis')
#plt.gca().set_aspect('equal', adjustable='box')
#position_plot, = plt.plot(x_data, y_data, 'bo-')

# Function to read serial data and update plots
def read_serial_and_plot():
    while True:
        data = ser.readline().decode('utf-8').rstrip()
        if data:
            try:
                # Assume data format is 'frequency,x,y'
                freq, x, y = map(float, data.split(','))
                # Update frequency plot
                frequency_data.append(freq)
                line_freq.set_ydata(frequency_data)
                # Update position plot
                x_data.append(x)
                y_data.append(y)
                position_plot.set_data(x_data, y_data)
                plt.draw()
                plt.pause(0.01)
            except ValueError:
                print("Invalid data format:", data)

# Start reading from serial and plotting
read_serial_and_plot()

# Close the serial connection when the program exits
ser.close()
```