University of British Columbia
Electrical and Computer Engineering
ELEC291/ELEC292 Winter 2022
Instructor: Dr. Jesus Calvino-Fraga
Section 201

Magnetic Field Controlled Robot

Team Members: Alvina Gakhokidze, Idara Nkono, Jasia Azreen, Nafeesa Nawar, Ruth Tau, Sikher Sinha

Date of Submission: April 13, 2023

Table of Contents

Introduction **3**

Investigation **6**

Idea generation 6

Investigation design 6

Data Collection 7

Data synthesis 8

Analysis of result 9

Design **10**

Use of Process 10

Need and Constraint Identification 11

Problem Specification 12

Solution evaluation 15

Safety And Professionalism 17

Detailed design 17

Transmitter 18

Transmitter Circuit Power Source 19

Signal Generation 20

Nunchuk & Command Signals 20

Safety Mechanism - Temperature Sensor 21

Receiver 21

Solution Assessment 23

Transmitter Circuit 23

Receiver Circuit 24

Livelong Learning **25**

Conclusion 26

References **28**

Bibliography **28**

Appendix A - Transmitter Code **29**

Appendix B - Receiver Code 39

Introduction

This project report introduces the Magnetic Field Controlled Robot, a remote controlled autonomous robot that operates using a varying magnetic field [1]. The basic functionalities of this project can be split into the magnetic transmitter and the receiver circuits, both of which have microcontrollers programmed in C language. The transmitter is built using the PIC32 microcontroller system [2] and produces electromagnetic signals at a constant frequency. These signals are processed by the receiver using inductive sensors to determine the modes of operation of the robot. The robot itself is battery operated and its motors are controlled using MOSFETs. The body of the robot also holds the receiver circuit using STM32 microcontroller system [3], which dictates its movements. It has two modes of operation: the track mode and the command mode. In the track mode, the robot keeps a fixed distance of at least 50 cm from the transmitter and changes position accordingly when the transmitter is moved towards or away from it. In the

command mode, the robot moves forward, backward, left or right, depending on the command received from the controller via magnetic field. In addition to these basic four commands, our robot is also able to make a U-turn, stop and move backward to the left or right upon request. Each of these commands are employed using a nunchuck on the transmitter (an extra feature, as opposed to using push buttons on the transmitter circuit) which produces signals with different off times, allowing the receiver to distinguish between each instruction. Plus, we implemented a temperature sensor for the transmitter inductor as a safety feature to warn users if it gets too hot. An overview of the system can be seen in **Figures 1, 2** and **Appendix B**.

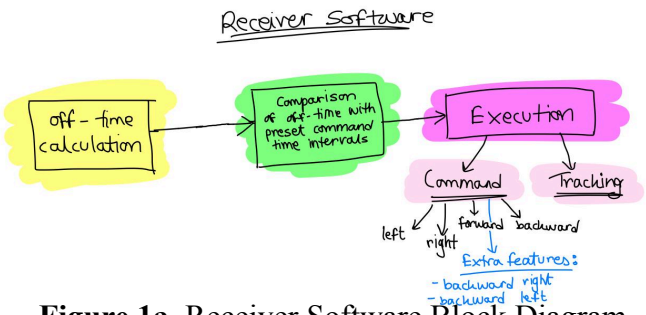


Figure 1a. Receiver Software Block Diagram

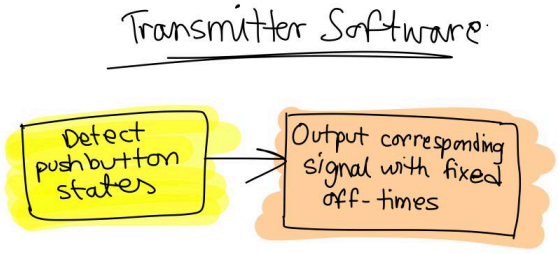


Figure 1b. Transmitter Software Block Diagram

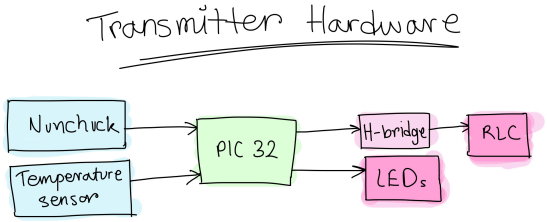


Figure 2a. Transmitter Hardware Block Diagram

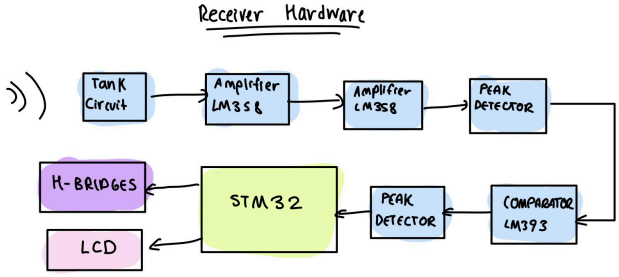


Figure 2b. Receiver Hardware Block Diagram

Investigation

Idea generation

We started by identifying key functions from the project slides [1][4] and studying the PIC32 and STM32 datasheets [2][3]. Our group decided to first assemble the robot body together [5] and then split into two subgroups, one working on the receiver and the other on the

transmitter. We focused on building the hardware first, initially setting up different parts of the circuits on individual breadboards (e.g. H-bridge, peak detector+amplifier+comparator) and later integrating them together. This way, it was much easier to spot mistakes and allowed us to plan before we started building our main circuits.

Our team used the provided example codes [6][7] as a starting point for making the software. For instance, the OffTime_One() function that calculates the off time of the received signal utilizes timers for implementation but was inspired by the GetPeriod(n) function from the STM32 makefiles on canvas. The rest of the receiver code involves a series of 'if' statements that determine which pins to be set on/off to make the robot carry out the desired command. Similarly, the transmitter code was mainly based off of the "toner.mk" [8] makefile. The idea was to measure the signal from the inductors and transmit it to the receiver.

Investigation design

Multiple datasheets, lab 5 and project slides from canvas [1][2][3][4][5][9][10] were the key resources used to ensure proper assembly of the circuits and robot body. The motors controlling the wheels are connected to the receiver circuit via a H bridge. The H bridge has four transistors which act like switches and can be turned on/off to control its motion. Reversing the polarity of voltage across the motor makes it rotate in the opposite direction. The different PWM signals are generated on the receiver by varying the pulse width and transmitted when designated nunchuck buttons are pressed.

During the hardware testing phase, we used a DMM to check the voltage across different components, and an oscilloscope to verify if correct signals were being outputted. This helped us identify faulty MOSFETs and debug other issues with our circuits. For example, we realized that having the two inductors (on receiver circuit) on the same level was vital in ensuring a stable

voltage reading. To do this, we inserted a chopstick to the robot body to hold them at the same height and orientation. Additionally, we noticed that while implementing the comparator [9] for our receiver circuit the output voltage from the op-amp [10] going into the non-inverting node of the comparator is always greater than the inverting node of the comparator, which is connected to ground. To deal with this issue we created a voltage divider at the inverting node of the comparator which developed a small voltage signal at that node. This was done since we were continuously comparing the voltage at the non-inverting node of the comparator to zero. Furthermore, in order to increase the range of detection in the tracking mode, we experimented with different resistor values to change the gain of the non-inverting op amp. In the end, we decided on a voltage gain of around 60, with resistors valued at 10k and 150 Ohms.

To make the software debugging process simpler, we created isolated test codes that ensured code for each part of the circuit worked before we ran them all together. This saved us from the hassle of having to look over the entire code to determine what the source of error was.

Data Collection

The project data was gathered utilizing a range of advanced tools such as the oscilloscope, multimeter and LCD display. Below is an outline of the procedures employed and the tools utilized to obtain the data:

- *Voltage Outputs on DMM:* To ensure proper functionality, the output voltage of specific components was assessed using the multimeter. For instance, the output of the op-amp was measured to validate its outputting the expected voltage.
- *Oscilloscopes*:
    1. Used to ensure correct off time was produced when each nunchuck button was pressed and that the signal was being received by the receiver

2. Used to verify if the peak to peak voltage of the inductor was in fact 200V
3. Used to check the output frequency from the transmitter microcontroller

- *LCD display:* we used the LCD to display outputs when we tested our isolated codes. For example, when we tested the H bridge, we programmed the test code so that the LCD displayed what was supposed to happen with each type of received signal e.g. ‘left’ when the off time was 36ms or ‘forward’ when it was 12ms. This ensured the program was entering the correct ‘if’ statement when a certain nunchuck button was pressed on the transmitter (i.e. signal with a certain off time was generated).

Data synthesis

Our group made sure the correct conclusions were reached by thorough debugging, testing and comparing obtained results with expected ones. This involved meeting two main criteria:

- Checking the LCD was displaying a drop in voltage as the distance between the robot and the receiver decreased
- Checking that signals with the correct off time was being transmitted, received and that the robot was moving accordingly

Analysis of result

We confirmed the reliability and precision of our transmitter circuit by measuring its peak-to-peak voltages using an oscilloscope. The circuit's electromagnetic signal was sent at the desired voltage and was strong and stable enough for the robot's inductive sensors to accurately detect. To control the robot's movement, we used Mosfet drivers and an H-bridge and checked each component's functionality using a multimeter. We also tuned the transmitter's frequency to

16000 Hz to increase the inductive sensors' range and precision. We adjusted the receiver circuit's resistance and tested it with a multimeter. After integrating the entire system, we thoroughly tested the robot to ensure it accurately tracked the signal from the transmitter and received commands to move in different directions. By taking these steps, we built a reliable and effective system capable of functioning in various situations.

Design

Use of Process

Throughout the project, we built the transmitter and receiver circuits using the RLC circuit provided on Piazza and the slides for class, and we used the microcontroller code provided by our professor as the basis for our own code. We encountered a major challenge in understanding how the timer works in the transmitter and determining the optimal frequency for our circuit. However, after some trial and error, we were able to determine that the optimal frequency for our transmitter was 16103 Hz, which we verified using an oscilloscope to measure the frequency of the electromagnetic signal transmitted by the circuit.

To help us troubleshoot any issues we encountered, we also made use of tools such as Putty and the oscilloscope to check the peak-to-peak voltage and frequency of the signal being transmitted. Despite these challenges, we successfully built and tested a functional magnetic field-controlled robot by leveraging the resources and processes provided in class and persevering through setbacks.

Additionally, we faced a challenge in obtaining the correct peak-to-peak voltage for our circuit. Our circuit behaved unexpectedly due to an incorrect peak-to-peak voltage, so we experimented with adjusting the resistor values until we achieved the desired voltage. Through this process, we gained a deeper understanding of circuit design and learned how to troubleshoot

circuit performance issues. Overall, we successfully completed the project and developed valuable skills in circuit design and troubleshooting.

Need and Constraint Identification

We started with collecting the requirements for this lab such as having a tracking and command mode, tracking distance at least 50 cm and having forward, backwards, left and right commands. These requirements give us a better understanding on how we approach the transmitter and receiver circuit design. For example, the voltage across the inductor has to be big enough to produce an induced emf on the receiver. Hence, the voltage generated from the RLC circuit must be larger than 200 volts PK-PK.

Due to magnetic field intensity decreases as distance between the source and object increases, our transmitter and receiver will have a maximum transmission length. Additionally we had to ensure that the microcontrollers utilized for both the transmitter and receiver circuit belonged to different families. Therefore the microcontrollers that we utilized included STM32(receiver) and PIC32 (transmitter), and due to space constraints on the transmitter circuit, we could not have many extra features on the car. The robot should be able to communicate quickly with the transmitter circuit through the tank circuits which are present on the receiver circuit. The code for our entire lab was required to be completely in the language of C. Also for the whole project to work, the receiver and transmitter circuit were required to operate on the same frequency. Another important point to note is that the whole project is battery operated. In order to control the motors we were supposed to utilize the NMOS and PMOS transistors provided to us. However the microcontroller cannot interact directly with the MOSFET's , so we had to use opto-isolators to isolate the microcontroller from the motors.

Problem Specification

To improve the range of detection we tried several methods but ultimately decided to tweak the gain of the amplifiers by shifting around resistors between the input and output of the op-amp. For the peak detector we were required to find the right capacitor and resistor value which would result in negligible ripple voltage at the operating frequency of our robot. Additionally we were required to tune the robot sensors of the tank circuit to ensure smooth and reliable communication between the transmitter and the receiver circuit. It is important to note that the inductors of our tank circuit hanging from the receiver had to be stable at all times. This was done through with the help of chopsticks as seen in Figure 3.

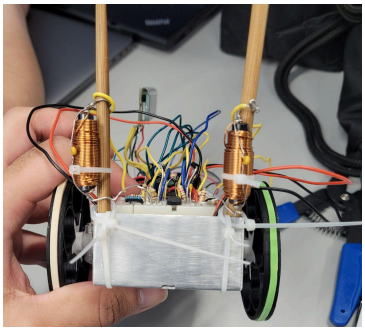


Figure 3. Inductors attached to chopsticks

Lastly, as a part of our extra feature we added a nunchuck to our transmitter circuit which allowed us to control the robot and operate it with relative ease.

One constraint our group faced during the development of the transmitter circuit was limited space and power availability, which restricted the additional features we could add to it. For instance, we opted for a PIC32 microcontroller in the transmitter, which had limited pins and prevented us from adding an LCD.

One potential feature that may have been limited by the constraints of the transmitter circuit is the ability to control the robot over long distances. The range of the electromagnetic signal produced by the transmitter is limited, depending on the specific design of our circuit. This limits our ability to control the robot from a greater distance, which is a desirable feature in certain applications.

Keeping project constraints in mind is important when considering additional features for transmitter and receiver circuits. Despite limitations, a functional circuit can still be achieved by

leveraging available tools and processes, as seen in the addition of LEDs that correspond to different commands.

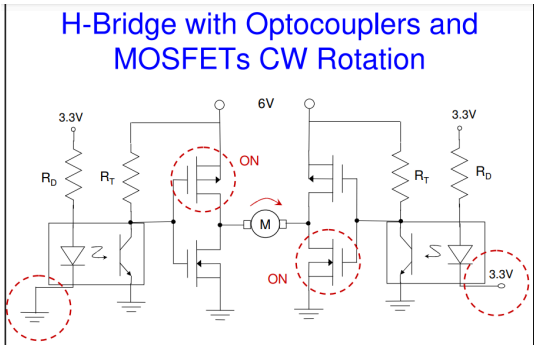
Solution Generation

One of the primary design specifications stated that the distance between the transmitter and the receiver should be maintained at a constant 50 cm. Initially we struggled with this requirement as the strength of our signal was quite weak at around 40 cm. We changed the gain of our cascaded amplifiers, producing a gain of 2500.

This helped but the signal strength was still weak around the 50cm mark. Upon further investigation, we concluded that the receiver inductors were not stable and not aligned properly with the transmitter inductor. To solve this, we placed the inductors horizontally balanced across a chopstick, providing stability to them when moving our receiver circuit. Additionally, our resonant frequency, which we fine tuned for the tank circuit, could also be a possible cause for weak signals detected by the inductors. We soon found that the signal received by our right inductor was weaker than that of the left inductor. So upon tweaking the gain of the amplifier, providing a stable and aligned platform for the inductors and fine tuning the frequency of the tank circuit to match with that of the receiver circuit we were able to meet the design specification. Our signal was strong up to a distance of 56-57 cm.

To tune our robot sensors, we taped the sensors of our receiver's tank circuit to a chopstick and positioned both the inductors equidistant from the transmitter. Next, we adjusted the frequency of the transmitter until both inductors detected signals with the same peak-peak voltage amplitude. The margin of error allowed was around 2mV. This is the frequency at which we get the best reception.

To move the car, we controlled the inputs to the H bridge. For clockwise and anti-clockwise rotations we referred to the figures above which were obtained from the lecture slides. It helped us determine which MOSFETS were required to be switched on/off respectively.



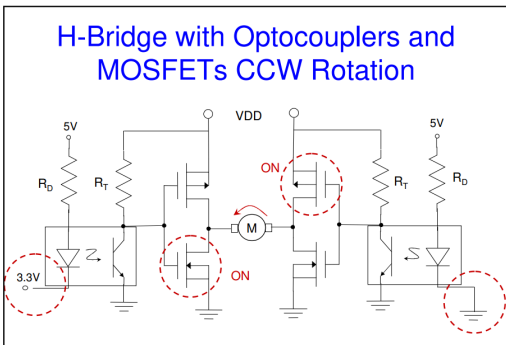


Figure 4a. H-Bridge for CW rotation

Figure 4b. H-Bridge for CW rotation

Solution evaluation

We had several design concepts for some of the project criteria specified in Table 1 below.

Battery Type:

- Design Concept 1: ***Using 9V batteries.*** Although using 9V batteries for both the transmitter and receiver circuits would make the design more compact, the battery life is shorter than the other available option.
- Design Concept 2: ***1.5V batteries in series.*** Opting for 1.5V batteries in series for powering both the transmitter and receiver circuits has the advantage of longer battery life, but the downside of being heavier and less compact compared to the 9V battery option.
- Chosen Design:We opted for ***1.5V batteries connected in series*** for longer-lasting power, which also helps maintain the peak voltage generated by the transmitter circuit's inductor over time.

Amplifier:

- Design Concept 1: ***Non-inverting Amplifier.*** Using a non-inverting amplifier would keep the circuit simple and compact, but it also limits the maximum distance the robot can receive signals from the transmitter. With this setup, the robot can only receive signals up to a maximum distance of 50cm from the transmitter.
- Design Concept 2: ***Differential Amplifier.*** Although we expected the differential amplifier to provide higher gain, its performance was not significantly better than the non-inverting amplifier. With this setup, the robot's maximum signal receiving distance remained at around 50cm.
- Design Concept 3: ***Cascading Amplifier.*** Using a cascading amplifier involves adding an extra circuit board, which reduces the overall compactness of the design, but expands the robot's signal receiving range to 56-57cm.
- Chosen Design: We decided on using the ***Cascading Amplifier*** since it provided the greatest range for our robot.

Inductor Position:

- Design Concept 1: ***Position Horizontally.*** This position made it difficult for the robot to receive signals from the transmitter.
- Design Concept 2: ***Position Vertically.*** In this position the robot seemed to receive the signals better.
- Chosen Design: We chose to ***position the inductors vertically*** because this orientation provided the best signal reception. To hold the inductors vertically on both the receiver and transmitter circuits, we used chopsticks and zip-ties.

Safety And Professionalism

The voltage generated through the RLC circuit on both the receiver and transmitter was above 200 V PK-PK. Interacting with such voltages is a high-risk situation and touching the inductor may result in electrical shocks. To deal with this, we ensured our fingers were always placed away from the inductors and wore safety goggles when probing the voltages of the inductors with an oscilloscope.

Additionally for the whole project to work and in order to not damage the chips, we ensured that a constant voltage of 5V was only passed to the H-Bridge in order for it to function.

Also , as mentioned later in the solution generation, one of the methods we employed to fix the range of detection involved replacing the resistors lying behind the PMOS and NMOS transistors present in the transmitter circuit. We reduced the resistors to values below 440 ohms and noticed that the chips started melting. This obviously also affected the communication between the receiver and the transmitter. So we stuck to our original plan and kept the resistors at values above 440 ohms.

Detailed design

Initially, we had to make a decision regarding the selection of two microcontrollers for our project. After considering various options, including the STM32, PIC32, and LPC824 microcontrollers, we opted to choose the STM32)for the receiver/robot) and PIC32(for the transmitter) . Our team's prior experience with these microcontrollers, coupled with their extensive pinouts, which allow for greater integration of additional features, were the primary factors behind this decision.

Transmitter

The transmitter serves as a remote control for the robot and generates an electromagnetic signal at a constant frequency. It transmits commands such as tracking mode, command mode, left, right, forward, and backward (with added features for turning backwards right and backwards left) via PWM signals:

• 12ms - move forward • 28ms - move backward • 44ms - move left • 60ms - move right	• 76ms - tracking mode • 92ms - command mode • 124ms - backwards left • 140ms - backwards right

The transmitter also includes several extra features:

- ***Safety Mechanism***: A temperature sensor (LM335) monitors the inductor's temperature, which can become excessively hot. In the event of high temperatures, the signal sent from the microcontroller is turned off for roughly 15 seconds before being restored. While the signal is off, the inductor does not generate an electromagnetic signal, and a white LED illuminates to indicate this state.
- ***Command LEDs***: For each of the eight commands, one or a combination of red, green, yellow, blue, and white LEDs are lit up.
- ***Nunchuk***: Serves as a command signal input device.
    - C-button: Utilized for command mode.
    - Z-button: Utilized for tracking mode.

Figure 5. Nunchuck [11]

- Joystick Directions:

• Joystick up (+y direction): forward. • Joystick down (-y direction): backward. • Joystick left (n- x direction): left.	• Joystick right (+ x direction): right. • Joystick up + Z : backward left. • C + Z: backward right.

Further details regarding the hardware and software aspects of each transmitter component are discussed in subsequent sections.

Transmitter Circuit Power Source

To power both the H-bridge and the microcontroller, a 9V battery source is needed for the H-bridge, while the microcontroller requires a 3.3V power source. In our design, we used an LM3940 voltage regulator to step down the 9V battery source to 5V, and then a MCP1700 voltage regulator to further step it down to 3.3V for the microcontroller.

Signal Generation

The transmitter generates an electromagnetic signal by sending a square wave at a constant frequency using the microcontroller. The output square wave signal is set to a frequency of 15500 Hz and assigned to pins RB6 (pin 15) and RB0 (pin 4) in the variable declaration of the code. This is then initialized in the Timer 1 Setup and toggled in the Timer 1 ISR (see Appendix).

We conducted tests on a range of frequencies between 15000 - 16000 Hz and selected 15500 Hz as the defined frequency since it produced the greatest inductor peak voltage. The output pins (RB6 & RB0) are connected to the H-Bridge , which transfers the square wave signal to the inductor and capacitor. This results in an electromagnetic signal with an inductor voltage ranging between 250 - 350 V.

Nunchuk & Command Signals

The nunchuck is connected to the PIC32's digital I/O pins RB2 (pin 6) and RB3 (pin 7). The microcontroller reads the position of the joystick's x and y axis, as well as the state of the 'C' and 'Z' buttons through the input analog pins. Based on these readings, the microcontroller sends a command in the Timer 1 ISR.

For instance, if the x-axis reading is greater than 60, the y-axis reading is 0, and both push buttons are not pressed (i.e. their reading is 1), the signal is turned off for 60 ms to execute the command for turning right. Similarly, there are multiple if-else statements assigned in the Timer 1 ISR for all commands (see **Appendix A - Timer 1 ISR**). The LEDs connected to the microcontroller's digital output pins A2 , B14, B13, B12, and B10 (pins 9, 25, 24, 23, and 21 respectively) are also lit up in the Timer 1 ISR depending on the command(see **Appendix A - Timer 1 ISR**).

Safety Mechanism - Temperature Sensor

The LM335 temperature sensor is placed in close proximity to the inductor to measure its temperature directly. It is connected to the microcontroller via pin B15 (pin 26) as an analog input, and the temperature value is obtained through the PIC32 ADC. In the Timer 1 ISR, this temperature value is compared to a predefined value, and if the temperature exceeds that value, a flag is set to 1.

When the flag is set, the while loop in the main function toggles the white LED and turns off Timer 1 to prevent any signal from being generated. Another flag, flag2, is set to 1 and Timer 1 remains off until 15 seconds have passed. The 15-second delay is implemented by setting a count value of 150, delaying by 100 milliseconds at the end of each loop iteration, and subtracting the count value by one each time the loop is executed (see **Appendix A)**

Receiver

As shown in the above receiver hardware diagram, the main logic on how the receiver receives and interprets the signal sent by the transmitter was to pass the signal through a tank circuit, which contains inductors strapped with a capacitor of 0.1uF that produce a resulting voltage. Then, the signal is passed through two cascaded LM358 amplifiers, with gains of 10 and 250, which will magnify the signal to around 2500 times, as the received voltage can be very small. Thus, it is critical to amplify the voltage in order for the peak detector to interpret “commands" afterwards. Additionally, this provides a distinguishable reading as the receiver will be placed more than 50 cm away from the transmitter. The amplified signal is then sent to the peak detector which determines when signals represent “1” for “High” or “0” for “Low”. However, because the amplifiers are so strong, the comparators generate a square wave that has the same frequency as the original transmitted sin wave. Thus, we need to place more peak detectors at the output. Using these peak detectors, we can determine the period or length of time that the signal sent was “On” or “Off”. Our team used these “off times” to send out “commands", which means the transmitter will constantly send out a “High” signal to the receiver, but when a “Low” signal is sent, the comparator will start the systick timer in STM32 and count the length of the “Low” signal. The time recorded is then translated into “commands" sent by the transmitter. For example, if the “Low” signal lasted for 94ms, the receiver reads that and interprets it as a “move backwards” command.

Each motor moving the wheels is controlled by an H bridge, which has two inputs (corresponding to two pins) from the STM32 microcontroller, as shown in **Figures 4a and 4b**. Hence, by powering on and off of the pins, which provide 3.3V or GND, we can turn the motors in different directions.

Following the previous example, after STM32 received the command “move backwards”, it will set one input high, and another low to rotate the wheel in one direction that will move the car backwards.

Since we needed to implement 2 modes, command and tracking mode, we also had two separate commands (with different off times) to transmit to the car and communicate that we wanted to switch modes. Then, based on the offtime that the receiving microcontroller reads,, the program will perform different “condition” statements.

In tracking mode, if the left inductor reads a larger value, then the car will turn right, moving the left inductor further away from the transmitter, similarly for the right side, it is constantly adjusting. If both inductors read a similar voltage and are lower than a threshold voltage at 50cm, which means it's too far away from the transmitter, it will move backwards, similarly to moving forward. In command mode, the receiver will read an “off” time, then output on and off for the servo motors depending on the command being transmitted.

Solution Assessment

Transmitter Circuit

The transmitter circuit was designed to transmit an electromagnetic signal at a constant frequency and send commands to the robot using PWM signals during command mode. We tested the circuit's frequency stability by connecting the inductor to an oscilloscope while the car was in use.

To confirm that the transmitter was correctly sending signals by turning off the signal for a certain period of time, we also connected it to an oscilloscope and tested the commands. Initially, we observed that holding down the pushbutton/nunchuk resulted in the signal being turned off continuously instead of sending multiple command signals (see **Figure 6a**). This posed

a problem as it could be interpreted as a different signal or no signal at all. To address this, we added an on-time for each command signal after the off-time. This modification resulted in the transmitter producing multiple identical commands when the nunchuk/pushbutton was held down for any specific command (see **Figure 6 b**).

(a) **(b)**

Figure 6. Output signal after holding down push button/nunchuck for command, where 't' is the off time for the command that will be read and translated by the robot.

Receiver Circuit

The robot was required to track the transmitter when they were both separated by at least 50 cm. We set a goal for the robot to at least be able to track up to a distance of 60cm. Unfortunately, we were not able to meet this requirement and the maximum we were able to accomplish was the 56-57cm mark. We also adjusted the gain of our amplifiers to improve the range of detection, and we changed the resistors lying behind the N-MOSFET and P-MOSFETS on the transmitter. However the latter resorted to the transistor chips melting so the second method did not help us achieve the specification.

For the receiver circuit we encountered an issue in our tracking mode where the right inductor would not read a signal well. To fix this issue we changed the gain of the amplifier for the right inductor. This seemed to work as our right wheel was able to move when required in tracking mode and at large distances as well.

Another requirement of our project included finding the resonant frequency of both the receiver and the transmitter circuit. To tune, we had to find a transmitting frequency on our

PIC32 that would result in the two receiving inductors reading a similar voltage. The only other issue with this part was that the voltage signal read by the right inductor was pretty weak, even despite tuning, and fluctuated much more frequently than the left inductor. This could be due to many factors, some of which include the magnetic permeability of the core on each inductor of our tank circuit, or perhaps one of the inductors had more wire coils. Due to the shortage of inductors we had to work our way around this problem in order for some of the commands in the tracking and command mode to work. The solutions are mentioned in the solution generation circuit.

Livelong Learning

The main concept of this project is utilizing inductors, generating magnetic fields to transmit and receive signals, which are interpreted as commands. Concepts of magnetic field, induced voltage, coils and resonant frequency were taught in first year PHY 157, more in depth teaching in ELEC 202 and ELEC 211. Many wireless devices such as radio communication, RFID, wireless charging technology are only possible by using induced current generated by magnetic fields. Therefore, understanding and having hands-on experience with this project could benefit us on how to implement this technology in real life.

Additionally we learnt how to deal with broken MOSFETS. This was because at times we were driving the MOSFET directly or reduced the value of the resistances which resulted in the chips melting.

Moreover, our team used PIC32 and STM32 controllers, which are the most common microcontrollers in the current tech industries. This project allowed us to learn how to program or configure them and gave us a better glimpse of the capability of these microcontrollers.

Furthermore, having hands-on experience with these microcontrollers allow us to apply them in future projects or internships.

Conclusion

In short, the magnetic field-controlled robot designed and constructed has been successful in meeting the specifications of tracking a transmitter when separated by at least 50cm. The design and construction involved overcoming various challenges, which were resolved through a systematic approach.

The receiver hardware was designed to receive and interpret the signal sent by the transmitter by passing the signal through a tank circuit, amplifying the signal, and determining the length of time that the signal was on or off. The robot was designed to operate in two modes, command and tracking mode, and performed different actions based on the modes selected. The use of ADC and Timer One helped to improve the accuracy of signal readings, and the tolerance value helped to balance out the difference between the two readings of inductors, resulting in a more accurate calculation.

The robot was designed to move left or right based on the voltage read by the left or right inductor and move forward or backward if both inductors read similar voltages and were lower than a threshold voltage at 50cm. The challenges faced during the project included issues with resistor values and voltage tolerance, which were resolved by altering the resistor values.

All in all, the magnetic field-controlled robot designed and constructed demonstrates the successful implementation of the magnetic field-controlled tracking system, which can be useful in environments where other wireless technologies are not practical. The project highlights the importance of a systematic approach to the design and construction of robotics systems, and the potential applications of such systems in various fields.

References

[1] Calviño-Fraga, J., Lecture slides: Project 2: Magnetic Field Controlled Robot, 2023.

[2] Microchip, PIC32MX Family Data Sheet, microchip.com, 2008.

[3] STMicroelectronics, Datasheet- STM32F205xx, STM32F207xx, st.com, July, 2020.

[4] Calviño-Fraga, J., Lecture slides: Lab 5- Measuring Phasors, 2023.

[5] Calviño-Fraga, J., Robot Body Assembly, 2023.

[6] Calviño-Fraga, J., PIC32.zip, 2023

[7] Calviño-Fraga, J., STM32L051.zip, 2023

[8]

[9] Texas Instruments, LM193-N, LM2903-N, LM293-N, LM393-N, ti.com, 2018.

[10] Texas Instruments,Industry-Standard Dual Operational Amplifiers, ti.com, 2022.

[11] You-Buy.ca. (n.d.). NC Wii Nunchuck Controller Joystick Gamepad Compatible with Nintendo Wii/Wii U Video Game Gamepads [Photograph]. Retrieved April 13, 2023, from https://www.you-buy.ca/en/product/35BI5CO-nc-wii-nunchuck-controller-joystick-gamepad-compatible-with-nintendo-wii-wii-u-video-game-gamepads-w

Bibliography

Glisson, T. H.., Introduction to Circuit Analysis and Design, Springer, 2012.

Appendix A - Transmitter Code

Variable Declaration

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

// 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

// Defines
#define SYSCLK 40000000L
#define DEF_FREQ 15500L
#define Baud2BRG(desired_baud)( (SYSCLK / (16*desired_baud))-1)
volatile int pw = 0;
volatile int count = 0;
volatile int joy_x;
volatile int joy_y;
volatile char but1;
volatile char but2;

//ADC
volatile int adcval;
volatile float voltage;
volatile float measure;
volatile float temp;
volatile int flag =0;

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
}
```

ADC Functions

```
//ADC
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
```

```
51.    AD1CON3 = 0x0f01;       // TAD = 4*TPB, acquisition time = 15*TAD
52.    AD1CON1SET=0x8000;      // Enable ADC
53.  }
54.
55.  int ADCRead(char analogPIN)
56.  {
57.    AD1CHS = analogPIN << 16;    // AD1CHS<16:19> controls which analog pin goes
       to the ADC
58.    AD1CON1bits.SAMP = 1;        // Begin sampling
59.    while(AD1CON1bits.SAMP);     // wait until acquisition is done
60.    while(!AD1CON1bits.DONE);    // wait until conversion done
61.    return ADC1BUF0;             // result stored in ADC1BUF0
62.  }
63.
64.
65.  //END ADC
66.
67.
68.  // Needed to by scanf() and gets()o
69.  int _mon_getc(int canblock)
70.  {
71.    char c;
72.
73.    if (canblock)
74.    {
75.       while( !U2STAbits.URXDA); // wait (block) until data available in RX buffer
76.       c=U2RXREG;
77.       while( U2STAbits.UTXBF);    // wait while TX buffer full
78.       U2TXREG = c;          // echo
79.       if(c=='\r') c='\n'; // When using PUTTY, pressing <Enter> sends '\r'.  Ctrl-J sends
       '\n'
80.       return (int)c;
81.    }
82.    else
83.    {
84.       if (U2STAbits.URXDA) // if data available in RX buffer
85.       {
86.          c=U2RXREG;
87.          if(c=='\r') c='\n';
88.          return (int)c;
89.       }
90.       else
91.       {
92.          return -1; // no characters to return
93.       }
94.    }
95.  }
96.
97.  void wait_1ms(void)
98.  {
99.    unsigned int ui;
100.   _CP0_SET_COUNT(0); // resets the core timer count
101.
102.   // get the core timer count
103.   while ( _CP0_GET_COUNT() < (SYSCLK/(2*1000)) );
```

Timer 1 ISR

Nunchuck configuration & signal outputs

```
104.}
105.
106.void delayms(int len)
107.{
108.   while(len--) wait_1ms();
109.}
110.
111.
112.void __ISR(_TIMER_1_VECTOR, IPL5SOFT) Timer1_Handler(void)
113.{
114.   //ADC
115.      //ADC temp
116.      adcval = ADCRead(9); // note that we call pin AN4 (RB2) by it's analog number
117.      measure=adcval*3.316/1023.0;
118.      voltage = (measure - 2.73)*100.0;
119.
120.      if(voltage > 20.0)
121.      {
122.         temp = voltage;
123.      }
124.      if(temp >= 60.0)
125.      {
126.         flag = 1;
127.      }
128.      else if(temp < 60.0)
129.      {
130.         flag = 0;
131.      }
132.
133.   //ADC_END
134.
135.   LATBbits.LATB6 = !LATBbits.LATB6;
136.   LATBbits.LATB0 = !LATBbits.LATB0;
137.
138.   if(count <= 0)
139.   {
140.   //first button - move forward
141.   if ((joy_y > 55) && (but1 == 1) && (but2 == 1) && (joy_x < 3) &&(joy_x > -3)){
142.      pw = 12;
143.         if(pw > 0)
144.         {
145.            LATBbits.LATB6 = 0;
146.            LATBbits.LATB0 = 1;
147.            LATBbits.LATB14=1;
148.            delayms(pw);
149.            LATBbits.LATB14=0;
150.            count = 4000;
151.
152.         }
153.      }
154.   //second button - move backward
155.      if ((joy_y < -55) && (but1 == 1) && (but2 == 1) && (joy_x < 3) &&(joy_x >
      -3)){
156.      pw = 28;
157.         if(pw > 0)
```

```
158.          {
159.             LATBbits.LATB6 = 0;
160.             LATBbits.LATB0 = 1;
161.             LATBbits.LATB13=1;
162.             delayms(pw);
163.             LATBbits.LATB13=0;
164.             count = 4000;
165.          }
166.       }
167.
168.    //third button - move left
169.       if ((joy_x < -55) && (but1 == 1) && (but2 == 1) && (joy_y < 3) &&(joy_y -3)){
170.       pw = 44;
171.          if(pw > 0)
172.          {
173.             LATBbits.LATB6 = 0;
174.             LATBbits.LATB0 = 1;
175.             LATBbits.LATB12=1;
176.             delayms(pw);
177.             LATBbits.LATB12=0;
178.             count = 4000;
179.          }
180.       }
181.
182.    //fourth button - move right
183.       if (joy_x > 55 && (but1 == 1) && (but2 == 1) && (joy_y < 3) &&(joy_y -3)){
184.       pw = 60;
185.          if(pw > 0)
186.          {
187.             LATBbits.LATB6 = 0;
188.             LATBbits.LATB0 = 1;
189.             LATBbits.LATB10=1;
190.             delayms(pw);
191.             LATBbits.LATB10=0;
192.             count = 4000;
193.          }
194.       }
195.
196.    //fifth button - tracking mode - z button
197.       if (((but1) == 0) && (joy_y == 0) &&(but2 == 1)){
198.       pw = 76;
199.          if(pw > 0)
200.          {
201.             LATBbits.LATB6 = 0;
202.             LATBbits.LATB0 = 1;
203.             LATBbits.LATB14=1;
204.             LATBbits.LATB10=1;
205.             delayms(pw);
206.             LATBbits.LATB14=0;
207.             LATBbits.LATB10=0;
208.             count = 4000;
209.          }
210.       }
211.
212.    //sixth button - command mode - c button
```

```
213.     if ((but2 == 0) && (but1 == 1) && (joy_y == 0)){
214.     pw = 92;
215.        if(pw > 0)
216.        {
217.           LATBbits.LATB6 = 0;
218.           LATBbits.LATB0 = 1;
219.           LATBbits.LATB13=1;
220.           LATBbits.LATB12=1;
221.           delayms(pw);
222.           LATBbits.LATB13=0;
223.           LATBbits.LATB12=0;
224.           count = 4000;
225.        }
226.     }
227.
228.     //seventh button RB9 - extra feature 1 - u-turn i think
229.     if ((but1 == 0) && (joy_y > 55)){
230.        pw = 124;
231.        if(pw > 0)
232.        {
233.           LATBbits.LATB6 = 0;
234.           LATBbits.LATB0 = 1;
235.           //blink w
236.           LATAbits.LATA2=1; // One pin set to one
237.           delayms(pw);
238.           LATAbits.LATA2=0;
239.           count = 4000;
240.        }
241.     }
242.
243.  //eighth button - extra feature 2 - stop i think
244.     if ((but2 == 0) && (but1 == 0)){
245.        pw = 140;
246.        if(pw > 0)
247.        {
248.           LATBbits.LATB6 = 0;
249.           LATBbits.LATB0 = 1;
250.           LATBbits.LATB14=1;
251.           LATBbits.LATB13=1;
252.           LATBbits.LATB12=1;
253.           LATBbits.LATB10=1;
254.           LATAbits.LATA2=1;
255.           delayms(pw);
256.           LATBbits.LATB14=0;
257.           LATBbits.LATB13=0;
258.           LATBbits.LATB12=0;
259.           LATBbits.LATB10=0;
260.           LATAbits.LATA2=0;
261.           count = 4000;
262.        }
263.     }
264.
265.
266.  }
267.
```

Timer Setup	268. count--;
	269.
	270. IFS0CLR=_IFS0_T1IF_MASK; // Clear timer 1 interrupt flag, bit 4 of IFS0
	271.
	272.
	273.}
	274.
	275.void SetupTimer1 (void)
	276.{
	277. // Explanation here:
	278. // https://www.youtube.com/watch?v=bu6TTZHnMPY
	279. __builtin_disable_interrupts();
	280. PR1 =(SYSCLK/(DEF_FREQ*2L))-1; // since SYSCLK/FREQ = PS*(PR1+1)
	281. TMR1 = 0;
	282. T1CONbits.TCKPS = 0; // Pre-scaler: 1
	283. T1CONbits.TCS = 0; // Clock source
	284. T1CONbits.ON = 1;
	285. IPC1bits.T1IP = 5;
	286. IPC1bits.T1IS = 0;
	287. IFS0bits.T1IF = 0;
	288. IEC0bits.T1IE = 1;
	289.
	290. INTCONbits.MVEC = 1; //Int multi-vector
	291. __builtin_enable_interrupts();
	292.}
	293.
	294.
	295.// Use the core timer to wait for 1 ms.
	296.void Init_I2C2(void)
	297.{
	298. // Configure pin RB2, used for SDA2 (pin 6 of DIP28) as digital I/O
	299. ANSELB &= ~(1<<2); // Set RB2 as a digital I/O
	300. TRISB |= (1<<2); // configure pin RB2 as input
	301. CNPUB |= (1<<2); // Enable pull-up resistor for RB2
	302. // Configure pin RB3, used for SCL2 (pin 7 of DIP28) as digital I/O
	303. ANSELB &= ~(1<<3); // Set RB3 as a digital I/O
	304. TRISB |= (1<<3); // configure pin RB3 as input
	305. CNPUB |= (1<<3); // Enable pull-up resistor for RB3
	306.
	307. I2C2BRG = 0x0C6; // SCL2 = 100kHz with SYSCLK = 40MHz (See table 24-1 in page 24-32, 830 in pdf)
	308. I2C2CONbits.ON=1; // turn on I2C
	309.}
	310.int I2C_byte_write(unsigned char saddr, unsigned char maddr, unsigned char data)
	311.{
	312. I2C2CONbits.SEN = 1;
	313. while(I2C2CONbits.SEN); //Wait till Start sequence is completed
	314.
	315. I2C2TRN = (saddr << 1);
	316. while(I2C2STATbits.TRSTAT); // Check Tx complete
	317.
	318. I2C2TRN = maddr;
	319. while(I2C2STATbits.TRSTAT); // Check Tx complete
	320.
	321. I2C2TRN = data;

```
322.   while(I2C2STATbits.TRSTAT); // Check Tx complete
323.
324.   I2C2CONbits.PEN = 1; // Terminate communication with stop signal
325.   while(I2C2CONbits.PEN); // Wait till stop sequence is completed
326.
327.   return 0;
328.}
329.
330.int I2C_burst_write(unsigned char saddr, unsigned char maddr, int byteCount, unsigned
      char* data)
331.{
332.   I2C2CONbits.SEN = 1;
333.   while(I2C2CONbits.SEN); //Wait till Start sequence is completed
334.
335.   I2C2TRN = (saddr << 1);
336.   while(I2C2STATbits.TRSTAT); // Check Tx complete
337.
338.   I2C2TRN = maddr;
339.   while(I2C2STATbits.TRSTAT); // Check Tx complete
340.
341.   for (; byteCount > 0; byteCount--)
342.   {
343.      I2C2TRN = *data++; // send data
344.      while(I2C2STATbits.TRSTAT); // Check Tx complete
345.   }
346.
347.   I2C2CONbits.PEN = 1; // Terminate communication with stop signal
348.   while(I2C2CONbits.PEN); // Wait till stop sequence is completed
349.
350.   return 0;
351.}
352.
353.int I2C_burstRead(char saddr, char maddr, int byteCount, unsigned char* data)
354.{
355.   // First we send the address we want to read from:
356.   I2C2CONbits.SEN = 1;
357.   while(I2C2CONbits.SEN); //Wait till Start sequence is completed
358.
359.   I2C2TRN = (saddr << 1);
360.   while(I2C2STATbits.TRSTAT); // Check Tx complete
361.
362.   I2C2TRN = maddr;
363.   while(I2C2STATbits.TRSTAT); // Check Tx complete
364.
365.   I2C2CONbits.PEN = 1; // Terminate communication with stop signal
366.   while(I2C2CONbits.PEN); // Wait till stop sequence is completed
367.
368.   // Second: we gatter the data sent by the slave device
369.   I2C2CONbits.SEN = 1;
370.   while(I2C2CONbits.SEN); //Wait till Start sequence is completed
371.
372.   I2C2TRN = ((saddr << 1) | 1); // The receive address has the least significant bit set to
      1
373.   while(I2C2STATbits.TRSTAT); // Check Tx complete
374.    for (; byteCount > 0; byteCount--)
```

```
375.  {
376.     I2C2CONbits.RCEN=1;
377.     while(I2C2CONbits.RCEN); // Wait for a byte to arrive
378.     *data++=I2C2RCV;
379.     if(byteCount==0) // We are done, send NACK
380.     {
381.        I2C2CONbits.ACKDT=1; // Selects NACK
382.        I2C2CONbits.ACKEN = 1;
383.        while(I2C2CONbits.ACKEN); // Wait till NACK sequence is completed
384.     }
385.     else // Not done yet, send an ACK
386.     {
387.        I2C2CONbits.ACKDT=0; // Selects ACK
388.        I2C2CONbits.ACKEN = 1;
389.        while(I2C2CONbits.ACKEN); // Wait till ACK sequence is completed
390.     }
391.  }
392.
393.  I2C2CONbits.PEN = 1; // Terminate communication with stop signal
394.  while(I2C2CONbits.PEN); // Wait till stop sequence is completed
395.
396.  return 0;
397.}
398.
399.void nunchuck_init(int print_extension_type)
400.{
401.  unsigned char buf[6];
402.
403.  I2C_byte_write(0x52, 0xF0, 0x55);
404.  I2C_byte_write(0x52, 0xFB, 0x00);
405.
406.  // Read the extension type from the register block.
407.  // For the original Nunchuk it should be: 00 00 a4 20 00 00.
408.  I2C_burstRead(0x52, 0xFA, 6, buf);
409.  delayms(10);
410.  if(print_extension_type)
411.  {
412.     printf("Extension type: %02x  %02x  %02x  %02x  %02x  %02x\r\n",
413.        buf[0],  buf[1], buf[2], buf[3], buf[4], buf[5]);
414.  }
415.
416.  // Send the crypto key (zeros), in 3 blocks of 6, 6 & 4.
417.  buf[0]=0; buf[1]=0; buf[2]=0; buf[3]=0; buf[4]=0; buf[5]=0;
418.
419.  I2C_byte_write(0x52, 0xF0, 0xAA);
420.  I2C_burst_write(0x52, 0x40, 6, buf);
421.  I2C_burst_write(0x52, 0x40, 6, buf);
422.  I2C_burst_write(0x52, 0x40, 4, buf);
423.}
424.
425.void nunchuck_getdata(unsigned char * s)
426.{
427.  unsigned char i;
428.
429.  // Start measurement
```

```
430.   I2C_burstRead(0x52, 0x00, 6, s);
431.   delayms(10);
432.
433.   // Decrypt received data
434.   for(i=0; i<6; i++)
435.   {
436.       s[i]=(s[i]^0x17)+0x17;
437.   }
438.}
439.
440.void main(void)
441.{
442.   char buf[32];
443.   int newPw, reload;
444.   unsigned char rbuf[6];
445.   int off_x, off_y, acc_x, acc_y, acc_z;
446.   int count = 0;
447.   int flag2 = 0;
448.   CFGCON = 0;
449.   Init_I2C2(); // Configure I2C2
450.
451.   // Configure RB15 as temp
452.   ANSELBbits.ANSB15 = 1;
453.   TRISBbits.TRISB15 = 1;
454.
455.   ADCConf(); // Configure ADC
456.   delayms(200);
457.   //int pw = 0;
458.
459.   DDPCON = 0;
460.   CFGCON = 0;
461.   TRISBbits.TRISB0 = 0;
462.   LATBbits.LATB0 = 0;
463.   TRISBbits.TRISB6 = 0;
464.   LATBbits.LATB6 = 0;
465.   INTCONbits.MVEC = 1;
466.
467.   LATBbits.LATB0=1; // One pin set to one
468.   LATBbits.LATB6=0; // The other pin set to zero
469.
470.   //initialize led W
471.   TRISAbits.TRISA2 = 0;
472.   LATAbits.LATA2 =0;
473.
474.   //LED G
475.   TRISBbits.TRISB14= 0;
476.   LATBbits.LATB14 =0;
477.
478.   //LED Y
479.   TRISBbits.TRISB13= 0;
480.   LATBbits.LATB13 =0;
481.
482.   //LED R
483.   TRISBbits.TRISB12 = 0;
484.   LATBbits.LATB12 =0;
```

```
485.
486.   //LED B
487.   TRISBbits.TRISB10= 0;
488.   LATBbits.LATB10 =0;
489.
490.   SetupTimer1();
491.    UART2Configure(115200);  // Configure UART2 for a baud rate of 115200
492.   delayms(500); // Give putty time to start before we send stuff.
493.   printf("Frequency generator for the PIC32MX130F064B.  Output is in RB5 and RB6
      (pins 14 and 15)\r\n");
494.   printf("By Jesus Calvino-Fraga (c) 2018.\r\n\r\n");
495.
496.   nunchuck_init(1);
497.   delayms(100);
498.   nunchuck_getdata(rbuf);
499.
500.   off_x=(int)rbuf[0]-128;
501.   off_y=(int)rbuf[1]-128;
502.   printf("Offset_X:%4d Offset_Y:%4d\r\n", off_x, off_y);
503.
504.   while (1)
505.   {
506.      count--;
507.      if(flag == 1) //blinking W LED
508.      {
509.         LATAbits.LATA2 =!LATAbits.LATA2;
510.         T1CONbits.ON = 0;
511.         count = 150;
512.         flag2 = 1;
513.         flag = 0;
514.
515.      }
516.      if(flag2 == 1)
517.      {
518.         T1CONbits.ON = 0;
519.      }
520.      if (count <= 0)
521.      {
522.         T1CONbits.ON = 1;
523.         flag2 = 0;
524.         LATAbits.LATA2 = 0;
525.      }
526.      nunchuck_getdata(rbuf);
527.
528.      joy_x=(int)rbuf[0]-128-off_x;
529.      joy_y=(int)rbuf[1]-128-off_y;
530.      but1=(rbuf[5] & 0x01)?1:0;
531.      but2=(rbuf[5] & 0x02)?1:0;
532.
533.      //print in putty
534.      printf("\rCurrent Temperature: %7.4fC %3.4f\r", temp, measure);
535.      fflush(stdout);
536.      delayms(100);
537.
538.   }
```

```
539.}
```

Appendix B - Receiver Code

```
1.  #include "../Common/Include/stm32l051xx.h"
2.  #include <stdio.h>
3.  #include <stdlib.h>
4.  #include "../Common/Include/serial.h"
5.  #include "adc.h"
6.  #include "lcd.h"
7.
8.  #define F_CPU 32000000L
9.  #define DEF_F 100000L // 10us tick
10.
11.
12. volatile int PWM_Counter = 0;
13. volatile unsigned char ISR_pwm1 = 100, ISR_pwm2 = 100;
14.
15.
16. void wait_1ms(void)
17. {
18.   // For SysTick info check the STM32l0xxx Cortex-M0 programming manual.
19.   SysTick->LOAD = (F_CPU / 1000L) - 1;  // set reload register, counter rolls over
    from zero, hence -1
20.   SysTick->VAL = 0; // load the SysTick counter
21.   SysTick->CTRL = SysTick_CTRL_CLKSOURCE_Msk |
    SysTick_CTRL_ENABLE_Msk; // Enable SysTick IRQ and SysTick Timer */
22.   while ((SysTick->CTRL & BIT16) == 0); // Bit 16 is the COUNTFLAG.  True when
    counter rolls over from zero.
23.   SysTick->CTRL = 0x00; // Disable Systick counter
24. }
25.
26.
27.
28. void Configure_Pins(void)
29. {
30.   RCC->IOPENR |= BIT0; // peripheral clock enable for port A
31.
32.   // Make pins PA0 to PA5 outputs (page 200 of RM0451, two bits used to configure:
    bit0=1, bit1=0)
33.   GPIOA->MODER = (GPIOA->MODER & ~(BIT0 | BIT1)) | BIT0; // PA0
34.   GPIOA->OTYPER &= ~BIT0; // Push-pull
35.
36.   GPIOA->MODER = (GPIOA->MODER & ~(BIT2 | BIT3)) | BIT2; // PA1
37.   GPIOA->OTYPER &= ~BIT1; // Push-pull
38.
39.   GPIOA->MODER = (GPIOA->MODER & ~(BIT4 | BIT5)) | BIT4; // PA2
40.   GPIOA->OTYPER &= ~BIT2; // Push-pull
41.
42.   GPIOA->MODER = (GPIOA->MODER & ~(BIT6 | BIT7)) | BIT6; // PA3
43.   GPIOA->OTYPER &= ~BIT3; // Push-pull
```

```
44.
45.    GPIOA->MODER = (GPIOA->MODER & ~(BIT8 | BIT9)) | BIT8; // PA4
46.    GPIOA->OTYPER &= ~BIT4; // Push-pull
47.
48.    GPIOA->MODER = (GPIOA->MODER & ~(BIT10 | BIT11)) | BIT10; // PA5
49.    GPIOA->OTYPER &= ~BIT5; // Push-pull
50.  }
51.
52.
53.
54.  void Hardware_Init(void)
55.  {
56.    RCC->IOPENR |= (BIT1 | BIT0);          // peripheral clock enable for ports A and B
57.
58.    // Configure the pin used for analog input: PB0 and PB1 (pins 14 and 15)
59.    GPIOB->MODER |= (BIT0 | BIT1);  // Select analog mode for PB0 (pin 14 of
       LQFP32 package)
60.    GPIOB->MODER |= (BIT2 | BIT3);  // Select analog mode for PB1 (pin 15 of
       LQFP32 package)
61.
62.    initADC();
63.
64.    // Configure the pin used to measure period
65.    GPIOA->MODER &= ~(BIT16 | BIT17); // Make pin PA8 input
66.    // Activate pull up for pin PA8:
67.    GPIOA->PUPDR |= BIT16;
68.    GPIOA->PUPDR &= ~(BIT17);
69.
70.    GPIOA->MODER &= ~(BIT26 | BIT27); // Make pin PA13 input
71.  // Activate pull up for pin PA13:
72.    GPIOA->PUPDR &= ~(BIT26);
73.    GPIOA->PUPDR &= ~(BIT27);
74.
75.
76.    GPIOA->MODER &= ~(BIT24 | BIT25); // Make pin PA12 input
77.    // Activate pull down for pin PA12:
78.    GPIOA->PUPDR &= ~(BIT24);
79.    GPIOA->PUPDR &= ~(BIT25);
80.
81.
82.    // Configure the pin connected to the pushbutton as input
83.    GPIOA->MODER &= ~(BIT28 | BIT29); // Make pin PA14 input
84.    // Activate pull up for pin PA8:
85.    //GPIOA->PUPDR |= BIT28;
86.    //GPIOA->PUPDR &= ~(BIT29);
87.
88.    GPIOA->PUPDR &= ~(BIT28);
89.    GPIOA->PUPDR |= (BIT29);
90.
91.    // Configure some pins as outputs:
92.    // Make pins PB3 to PB7 outputs (page 200 of RM0451, two bits used to configure:
       bit0=1, bit1=0)
93.    GPIOB->MODER = (GPIOB->MODER & ~(BIT6 | BIT7)) | BIT6;    // PB3
94.    GPIOB->OTYPER &= ~BIT3; // Push-pull
95.    GPIOB->MODER = (GPIOB->MODER & ~(BIT8 | BIT9)) | BIT8;    // PB4
```

```
96.     GPIOB->OTYPER &= ~BIT4; // Push-pull
97.     GPIOB->MODER = (GPIOB->MODER & ~(BIT10 | BIT11)) | BIT10; // PB5
98.     GPIOB->OTYPER &= ~BIT5; // Push-pull
99.     GPIOB->MODER = (GPIOB->MODER & ~(BIT12 | BIT13)) | BIT12; // PB6
100.    GPIOB->OTYPER &= ~BIT6; // Push-pull
101.    GPIOB->MODER = (GPIOB->MODER & ~(BIT14 | BIT15)) | BIT14;  // PB7
102.    GPIOB->OTYPER &= ~BIT7; // Push-pull
103.
104.    // Set up timer
105.    RCC->APB1ENR |= BIT0;  // turn on clock for timer2 (UM: page 177)
106.    TIM2->ARR = F_CPU / DEF_F - 1;
107.    NVIC->ISER[0] |= BIT15; // enable timer 2 interrupts in the NVIC
108.    TIM2->CR1 |= BIT4;      // Downcounting
109.    TIM2->CR1 |= BIT7;      // ARPE enable
110.    TIM2->DIER |= BIT0;     // enable update event (reload event) interrupt
111.    TIM2->CR1 |= BIT0;      // enable counting
112.
113.    __enable_irq();
114.}
115.
116.// A define to easily read PA8 (PA8 must be configured as input first)
117.#define PA8 (GPIOA->IDR & BIT8)
118.
119.
120.void PrintNumber(long int val, int Base, int digits)
121.{
122.    char HexDigit[] = "0123456789ABCDEF";
123.    int j;
124.#define NBITS 32
125.    char buff[NBITS + 1];
126.    buff[NBITS] = 0;
127.
128.    j = NBITS - 1;
129.    while ((val > 0) | (digits > 0))
130.    {
131.        buff[j--] = HexDigit[val % Base];
132.        val /= Base;
133.        if (digits != 0) digits--;
134.    }
135.    eputs(&buff[j + 1]);
136.}
137.
138.
139.#define PB3_0 (GPIOB->ODR &= ~BIT3)
140.#define PB3_1 (GPIOB->ODR |=  BIT3)
141.#define PB4_0 (GPIOB->ODR &= ~BIT4)
142.#define PB4_1 (GPIOB->ODR |=  BIT4)
143.#define PB5_0 (GPIOB->ODR &= ~BIT5)
144.#define PB5_1 (GPIOB->ODR |=  BIT5)
145.#define PB6_0 (GPIOB->ODR &= ~BIT6)
146.#define PB6_1 (GPIOB->ODR |=  BIT6)
147.#define PB7_0 (GPIOB->ODR &= ~BIT7)
148.#define PB7_1 (GPIOB->ODR |=  BIT7)
149.
150.#define PA14 (GPIOA->IDR & BIT14)
```

```
151.
152.// note to myself: add pin to to protect inductor
153.
154.
155.// will want to check voltage constantly
156.// but we only want to do this in tracking mode
157.
158.
159.int tracking_mode = 0;
160.int command_mode = 1;
161.int init_mode_03m = 2;
162.int init_mode_05m = 3;
163.
164.int tracking_time = 76;
165.int forward_time = 12;
166.int backward_time = 28;
167.int left_time = 44;
168.int right_time = 60;
169.int command_time = 92;
170.int extra_feature = 108;
171.float time_tolerance = 8; // in ms
172.int command;
173.int forward_com = 0;
174.int backward_com = 1;
175.int left_com = 2;
176.int right_com = 3;
177.int voltage_tolerance_right = 0;
178.int voltage_tolerance_left = 0;
179.
180.
181.// extra features for backwards left and right
182.int backward_right_time = 124;
183.int backward_left_time = 140;
184.
185.
186.// need to make sure we configure the correct pins to output and input
187.
188.
189.
190.#define PIN_PERIOD1 (GPIOA->IDR&BIT12)
191.#define PIN_PERIOD2 (GPIOA->IDR&BIT13)
192.
193.long int OffTime_one(void)
194.{
195.  int i;
196.  unsigned int saved_TCNT1a, saved_TCNT1b;
197.
198.  //**the 1st 2 lines RESET the timer
199.  SysTick->LOAD = 0xffffff;  // 24-bit counter set to check for signal present
200.  SysTick->VAL = 0xffffff; // load the SysTick counter
201.  SysTick->CTRL = SysTick_CTRL_CLKSOURCE_Msk |
      SysTick_CTRL_ENABLE_Msk; // Enable SysTick IRQ and SysTick Timer */
202.  while (PIN_PERIOD1 != 0) // Wait for square wave to be 0
203.  {
204.      if (SysTick->CTRL & BIT16) return 0;
```

```
205.   }
206.   //**the following lines STOPS the timer
207.   SysTick->CTRL = 0x00; // Disable Systick counter
208.
209.   SysTick->LOAD = 0xffffff;  // 24-bit counter set to check for signal present
210.   SysTick->VAL = 0xffffff; // load the SysTick counter
211.   SysTick->CTRL = SysTick_CTRL_CLKSOURCE_Msk |
      SysTick_CTRL_ENABLE_Msk; // Enable SysTick IRQ and SysTick Timer */
212.   // not to self: might need to make this a comparison and not an equals
213.   while (PIN_PERIOD1 == 0) // Wait for square wave to be 1
214.   {
215.      if (SysTick->CTRL & BIT16) return 0;
216.   }
217.   SysTick->CTRL = 0x00; // Disable Systick counter
218.
219.   SysTick->LOAD = 0xffffff;  // 24-bit counter reset
220.   SysTick->VAL = 0xffffff; // load the SysTick counter to initial value
221.   SysTick->CTRL = SysTick_CTRL_CLKSOURCE_Msk |
      SysTick_CTRL_ENABLE_Msk; // Enable SysTick IRQ and SysTick Timer */
222.
223.   while (PIN_PERIOD1 != 0) // Wait for square wave to be 0
224.   {
225.      if (SysTick->CTRL & BIT16) return 0;
226.      if (((0xffffff - SysTick->VAL) * 1000) / (F_CPU) > (command_time * 2)) return
      1000;
227.      // making sure to return 0 if we're waiting longer than longest command
228.   }
229.   SysTick->CTRL = 0x00; // Disable Systick counter
230.
231.   SysTick->LOAD = 0xffffff;  // 24-bit counter reset
232.   SysTick->VAL = 0xffffff; // load the SysTick counter to initial value
233.   SysTick->CTRL = SysTick_CTRL_CLKSOURCE_Msk |
      SysTick_CTRL_ENABLE_Msk;
234.
235.   while (PIN_PERIOD1 == 0) // Wait for square wave to be 1
236.   {
237.      if (SysTick->CTRL & BIT16) return 0;
238.      // we don't need the thing here bc we're never in a state where we're always sending
      a 0 signal
239.   }
240.
241.   SysTick->CTRL = 0x00; // Disable Systick counter
242.
243.   return 0xffffff - SysTick->VAL;
244.}
245.
246.
247.
248.int main() {
249.   // note to self: need to test
250.   long int count2;
251.   int T, f;
252.   int j, v;
253.   int off_time;
254.   int signal[] = {};
```

```
int right_inductor, left_inductor;
int right_ind_05_val;
int left_ind_03_val;
int right_ind_03_val;
int left_ind_05_val;
int distance;
long int count, count1;
char line1[17];
char line2[17];
int mode = 0;
T = 1; //not accurate!!! it's actually basically 0 milliseconds
// also it's a float but im giving it an integer value


Hardware_Init();
Configure_Pins();
waitms(500); // Wait for putty to start.

LCD_4BIT();

// automatically in tracking mode
mode = 0;

PB3_0;
PB4_0;
PB5_0;
PB6_0;
PB7_0;

eputs("initialization");
sprintf(line1, "initializing");
waitms(2000);


LCDprint(line1, 1, 1);

mode = init_mode_05m;

while (mode == init_mode_05m) {
    j = readADC(ADC_CHSELR_CHSEL8);
    left_ind_05_val = (j * 33000) / 0xfff;

    j = readADC(ADC_CHSELR_CHSEL9);
    right_ind_05_val = (j * 33000) / 0xfff;

    // pin is pulled high unless button pulls it low i believe
    if (PA14) {
        waitms(300);
        if (PA14) {
            mode = init_mode_03m; // will default to tracking mode unless otherwise specified
            waitms(1000); //allow for slow human
        }

    }
```

```
309.
310.  }
311.
312.  sprintf(line2, "past first button");
313.  LCDprint(line2, 2, 1);
314.  mode = command_mode;
315.
316.  while (mode == init_mode_03m) {
317.
318.
319.      j = readADC(ADC_CHSELR_CHSEL9);
320.      right_ind_03_val = (j * 33000) / 0xfff;
321.
322.      // pin is pulled high unless button pulls it low i believe
323.      if (PA14) {
324.          waitms(300);
325.          if (PA14) {
326.              mode = command_mode; // will default to tracking mode unless otherwise
    specified
327.              waitms(1000); //allow for slow human
328.          }
329.
330.      }
331.
332.  }
333.
334.  int d_scale = (5000000-3000000)/(right_ind_03_val);
335.
336.
337.  sprintf(line2, "past second button");
338.  LCDprint(line2, 2, 1);
339.  mode = command_mode;
340.
341.  while (1)
342.  {
343.      off_time = (OffTime_one() * 1000) / (F_CPU); // should be in this loop until we
    press the button
344.      // the first button should indicate if we're in command or tracking mode
345.
346.      if ((off_time < (command_time + time_tolerance)) && (off_time >
    (command_time - time_tolerance))) {
347.          mode = command_mode;
348.      }
349.      else if ((off_time < (tracking_time + time_tolerance)) && (off_time >
    (tracking_time - time_tolerance))) {
350.          mode = tracking_mode;
351.      }
352.      else if (off_time == 0) {
353.          if (mode == command_mode) {
354.              // make sure motors aren't spinning
355.              PB4_0;
356.              PB3_0;
357.
358.              // left wheel counterclockwise: left_right off and left_left on
359.              PB5_0;
```

```
360.              PB6_0;
361.          }
362.          else if (mode == tracking_mode) {
363.              // do nothing?
364.          }
365.      }
366.
367.      if (mode == tracking_mode) {
368.
369.          while (PIN_PERIOD1 != 0) {
370.              // while in tracking mode, the signal always stays high unless something is
      pressed, and we're gonna stay here
371.              j = readADC(ADC_CHSELR_CHSEL8);
372.              left_inductor = (j * 33000) / 0xfff;
373.
374.              j = readADC(ADC_CHSELR_CHSEL9);
375.              right_inductor = (j * 33000) / 0xfff;
376.
377.              distance = right_inductor*d_scale;
378.
379.
380.              if (right_inductor > right_ind_05_val + voltage_tolerance_right &&
      left_inductor > left_ind_05_val + voltage_tolerance_left) {
381.                  // move forward
382.                  PB4_1;
383.                  PB3_0;
384.
385.                  PB5_0;
386.                  PB6_1;
387.
388.              }
389.              else if (right_inductor < right_ind_05_val - voltage_tolerance_right &&
      left_inductor < left_ind_05_val - voltage_tolerance_left) {
390.                  // move backward
391.
392.                  PB4_0;
393.                  PB3_1;
394.
395.                  PB5_1;
396.                  PB6_0;
397.
398.              }
399.              else if ((right_inductor > right_ind_05_val + voltage_tolerance_right) &&
      (left_inductor < left_ind_05_val - voltage_tolerance_left)) {
400.                  // turn left
401.                  PB4_1;
402.                  PB3_0;
403.
404.                  PB5_0;
405.                  PB6_0;
406.
407.              }
408.              else if ((left_inductor > left_ind_05_val + voltage_tolerance_left)  &&
      (right_inductor < right_ind_05_val - voltage_tolerance_right)) {
409.                  // turn right
```

```
410.                PB4_0;
411.                PB3_0;
412.
413.                PB5_0;
414.                PB6_1;
415.            }
416.            // extra features - backwards left and right for tracking mode
417.            else if ((right_inductor < right_ind_05_val - voltage_tolerance_right)  &&
    (left_inductor > left_ind_05_val + voltage_tolerance_left)) {
418.                // backwards left
419.                PB5_0;
420.                PB6_0;
421.
422.                PB4_0;
423.                PB3_1;
424.            }
425.            else if ((left_inductor < left_ind_05_val - voltage_tolerance_left) &&
    (right_inductor > right_ind_05_val + voltage_tolerance_right)) {
426.                // backwards right
427.                PB4_0;
428.                PB3_0;
429.
430.                PB5_1;
431.                PB6_0;
432.            }
433.
434.        }
435.        // pin_period has gone low, going to be receiving new command (or potentially
    being told we're still in tracking mode)
436.        // so while we're waiting to determine, want to turn wheels off
437.    }
438.    else if (mode == command_mode)
439.    {
440.
441.        count2 = OffTime_one();
442.
443.        if (count2 > 0)
444.        {
445.            off_time = (count2 * 1000) / (F_CPU);
446.            if (off_time > T) {
447.
448.
449.                if ((off_time < forward_time + time_tolerance) && (off_time >
    forward_time - time_tolerance)) {
450.                PB4_1;
451.                    PB3_0;
452.
453.                    PB5_0;
454.                    PB6_1;
455.                }
456.                else if ((off_time < backward_time + time_tolerance) && (off_time >
    backward_time - time_tolerance)) {
457.
458.                    // right wheel counter clockwise: right_right off, right_left on
459.                    PB4_0;
```

```
460.                PB3_1;
461.
462.                // left wheel clockwise: left_right on, left_left off
463.                PB5_1;
464.                PB6_0;
465.
466.            }
467.            else if ((off_time < left_time + time_tolerance) && (off_time > left_time -
    time_tolerance)) {
468.                // right wheel clockwise and left wheel off
469.                PB4_1;
470.                PB3_0;
471.
472.                PB5_0;
473.                PB6_0;
474.
475.            }
476.            else if ((off_time < right_time + time_tolerance) && (right_time >
    right_time - time_tolerance)) {
477.                // right wheel off, left wheel counter clockwise
478.                PB4_0;
479.                PB3_0;
480.
481.                PB5_0;
482.                PB6_1;
483.
484.            }
485.            else if ((off_time < tracking_time + time_tolerance) && (off_time >
    tracking_time - time_tolerance)) {
486.                // right wheel off, left wheel counter clockwise
487.                PB4_0;
488.                PB3_0;
489.
490.                PB5_0;
491.                PB6_0;
492.
493.
494.            }
495.            else if ((off_time < command_time + time_tolerance) && (off_time >
    command_time - time_tolerance)) {
496.                // right wheel off, left wheel counter clockwise
497.                PB4_0;
498.                PB3_0;
499.
500.                PB5_0;
501.                PB6_0;
502.
503.                //sprintf(line2, "command");
504.                //LCDprint(line2, 2, 1);
505.            }
506.            else if ((off_time < backward_right_time + time_tolerance) && (off_time >
    backward_right_time - time_tolerance)) {
507.
508.
509.                PB4_0;
```

```
                PB3_0;

                PB5_1;
                PB6_0;

            }
            else if ((off_time < backward_left_time + time_tolerance) && (off_time >
    backward_left_time - time_tolerance)){
            // backwards left  - PB5_0 , PB6_0, PB4_0; PB3_1;

            PB5_0;
            PB6_0;

            PB4_0;
            PB3_1;
        }

        }

      }
    }

  }
```

Appendix B - Our Design Process