CMPE 3815 - Microcontroller Systems

University of Vermont

Fall 2024

Final Report

Shea McGregor & Spencer Hart

Submission date: 12/11/2024

GitHub:

https://github.com/spencerh122/CMPE-3815---Microcontroller-Systems-Final-Project.git

TABLE OF CONTENTS

Introduction

The airplane was intended to incorporate a rudimentary fly-by-wire control system using an Arduino microcontroller, with the goal of creating a more easily controlled remote-controlled (RC) aircraft. We sought to accomplish this using an MPU6050 Inertial Measurement Unit (IMU) to feed current position information to the Arduino, allowing it to autonomously adjust the airplane's orientation. This proved far more complicated than anticipated, however, so we decided instead to pursue a direct-control approach in which the pilot directly controls the position of the flight controls to steer the airplane. We explored multiple communication methods to communicate between the pilot and the onboard Arduino, including Bluetooth and radio communication. These methods successfully controlled the flight controls but faced various challenges with unreliability and control surface jitter. We did successfully fly the airplane for a few seconds, but a bad battery caused a fire that destroyed the airplane in its final phases of testing.

Materials

- Arduino Uno (x2)
- Servo (x4)
- Joystick module (x2)
- Three-phase brushless drone motor
- Electronic Speed Controller (ESC)
- 3.7V LiPo battery (x4) (connected in series to make a 14.8V battery)
- 9V battery
- RF transmitter module
- RF receiver module
- Bluetooth module
- 24-gauge antenna wire (x2)
- Jumper wires
- Foam board
- Rubber bands
- Wooden dowels
- Balsa wood
- Plastic propellers
- Stiff metal wire

Part 1: Airplane Construction & Operation

The final airplane was constructed using 5mm-thick foam board, a ⅜” dowel, poster paper, balsa wood, rubber bands, barbecue skewers, and hot glue. The foam board, dowel, and

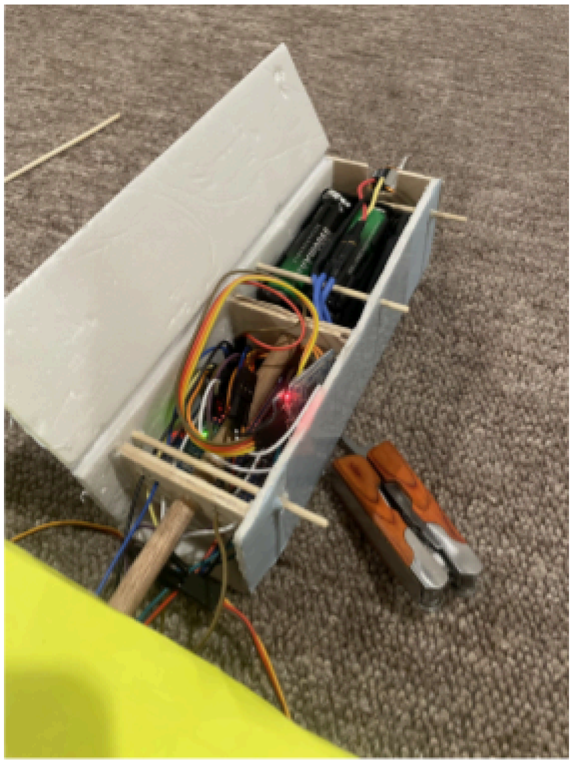
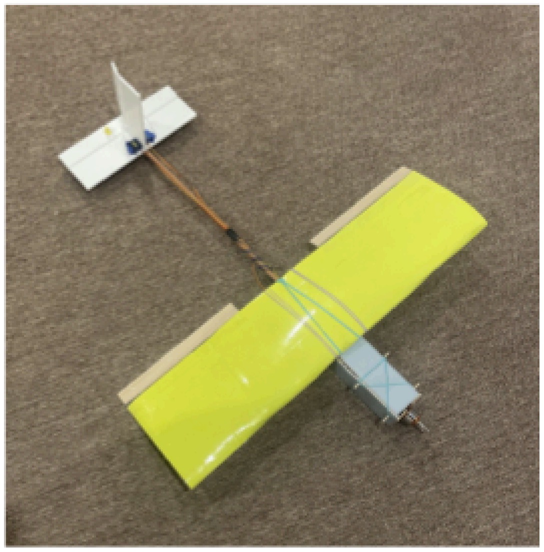

Figure 1: Final aircraft design and Arduino installation.

some of the balsa wood was used to make the fuselage. The rest of the balsa wood was cut into Clark-Y airfoil wing ribs, around which we wrapped the poster paper to create a lightweight wing. The fuselage and wing were both wrapped with clear packing tape for structural integrity and attached to each other with skewers and rubber bands. The tail was made of foam board. Four servos actuate the flight controls. Four servos control the two ailerons, the elevator, and the rudder. We attached the servos’ actuator arms to the control surfaces using stiff wire pushrods attached to 3D-printed control horns, hot glued to each surface. The motor is driven by an electronic speed controller (ESC), which uses a servo-style pulse width modulation input signal to determine the frequency and voltage at which to switch the different phases of the three-phase electric current used to drive the brushless motor.

Part 2: Inertial Measurement Unit

The plan for this airplane was to give it a fly-by-wire control system - that is, the pilot commands a certain state of the aircraft (bank and pitch in degrees) and the Arduino automatically adjusts the flight controls to achieve this state. It would do this by comparing the desired state transmitted by the pilot to the current state of the aircraft, measured using an onboard IMU. Unfortunately, we faced significant difficulties getting the IMU to work properly. Not only is the IMU affected by vibration, but its measurements drift over time. The MPU6050 Inertial Measurement Unit has six sensors: an accelerometer and a gyroscope for the X, Y, and Z axes. Since the raw data from the IMU is useless unless properly smoothed and processed, we attempted to use a Madgwick filter to clean the data and prepare it for use in the flight controller.

A Madgwick filter processes IMU data to estimate an object's orientation using quaternions, mathematical constructs used to quantify 3D rotations. Quaternions, composed of one real and three imaginary components ($q = w + xi + yj + zkq$), are computationally simple and allow for lightweight calculations of orientation given gyroscope and accelerometer data. The filter combines gyroscope data, integrated over time to estimate orientation, with accelerometer readings, which provide a gravity reference to correct drift. The Madgwick filter uses a method called gradient descent to adjust its calculations, aligning the accelerometer's readings with the actual direction of gravity. This helps correct errors and ensures the orientation estimate stays accurate over time. Implementing this filter proved too complex for our project, either because of mathematical complexity or faulty hardware. Even when using an algorithm written in C specifically for the purpose of running MPU6050 data through a Madgwick filter, hardware issues with the IMU made the data unreliable and caused it to periodically cease transmitting. While it could have improved our system's functionality, it exceeded our project's timeline and resources, leaving it as a potential future improvement.

Part 3: Bluetooth control

The Bluetooth control implementation was an alternative communication approach to the radio frequency (RF) transmission method intended to be used for final operation. This system uses the Dabble Bluetooth library to enable remote control of the aircraft through a joystick and several buttons on a smartphone. The Bluetooth communication protocol allows for the simultaneous interpretation of gamepad inputs across multiple control axes.

The Dabble library relies on interrupt pins 2 and 3 being used for the transmit and receive pins between the Bluetooth module and the Arduino. This allows the Arduino to continuously monitor for new inputs. Every iteration of the main loop begins with the Dabble library processing any new inputs, and then checking to determine which inputs were made. Depending on what the control input was, the program then proceeds to adjust the flight controls or throttle accordingly.

The joystick inputs are communicated to the Arduino by the Dabble library system as having a range from -7 to 7 for both horizontal and vertical axes. In this setup, the horizontal axis of the joystick controls the ailerons and the vertical axis controls the elevator. The X and Triangle buttons control power and the Square and Circle buttons control the rudder position.

While experimenting with the custom 14.8V battery pack, we noticed that the ESC was unable to drive the motor above a servo.write( ) command of 170. The ESC also required a minimum servo command of approximately 18 to stay connected to the Arduino while not providing sufficient power to the motor to move it. These values were determined by trial and error. The mapping values for each control channel (throttle, ailerons, elevator, rudder) are as follows:

- Throttle: 18 - 170
- Ailerons: 0 - 180
- Elevator: 0 - 180
- Rudder: 0 - 180

The pilot controls the throttle setting and amount of deflection of each control surface by directly determining the servo position for the corresponding control. In order to roll to the left, the pilot moves the joystick to the left; if the pilot moves it all the way to the left, this transmits a value of -7 to the Arduino using the Dabble library's communication protocol. The code then maps this value to a servo position of 0 for both aileron servos. Since the aileron servos are oriented opposite each other, they move in opposite directions for the same control input. Therefore, the left wing's aileron moves up to a servo position of 0 degrees while the right

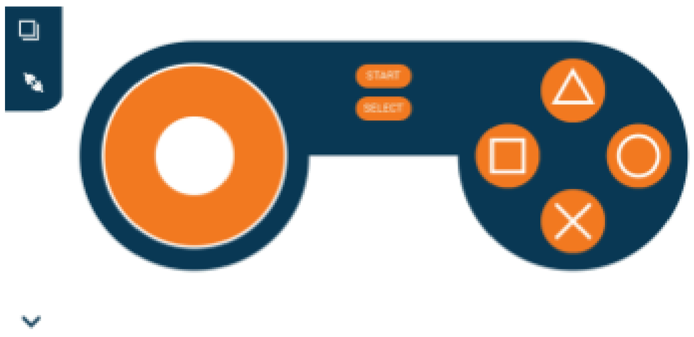

Figure 2: Dabble gamepad control module used for Bluetooth control of airplane

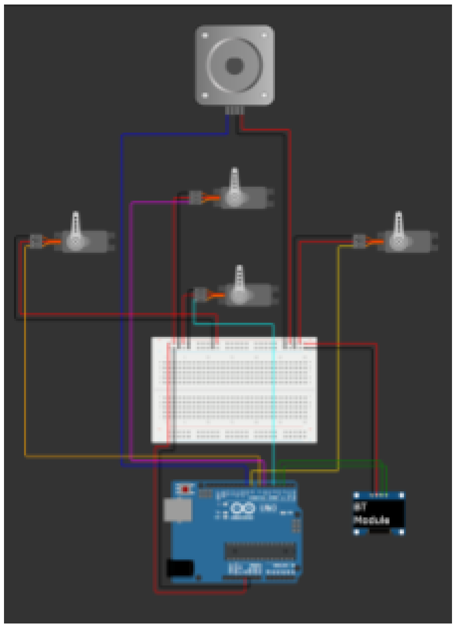

Figure 3: Bluetooth control wiring diagram.

aileron moves down to a servo position of 0 degrees. This creates more lift on the right wing and less lift on the left wing, causing the airplane to roll left.

Unfortunately, the Bluetooth control proved too twitchy to adequately operate the airplane. Any button press or joystick movement causes all of the controls to jump and jitter randomly, no matter what control was pressed. There is also a small but noticeable delay between control input and servo actuation. We did implement some debouncing features that helped to smooth out the jitters, but the greater the smoothing effect, the greater the lag between control inputs and servo actuation. By the time debouncing was effective enough to smooth out most of the jitters, there was a delay of several seconds between control input and response. A more skilled RC pilot may be able to overcome these challenges and still fly the airplane, but due to these hardware limitations, Bluetooth-controlled flight was not possible. It did, however, prove that all of the control surfaces would actuate as desired and the ESC would control the throttle properly and precisely, if provided a clean signal.

Part 4: RF Control

Transmitter Setup

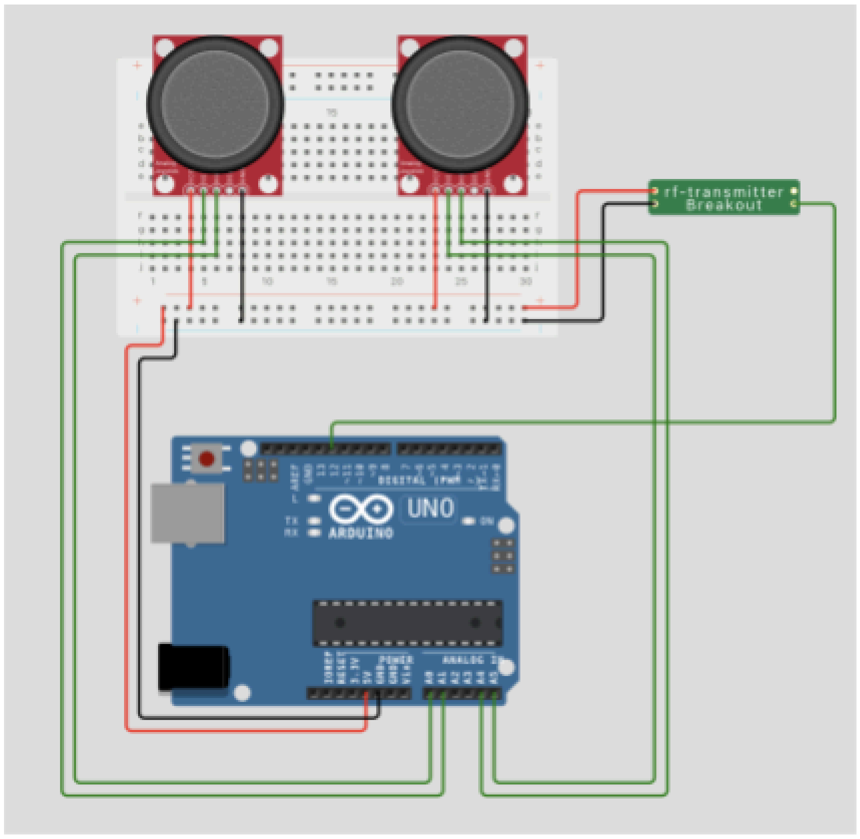

Figure 4: Circuit setup for the transmitter/remote controller. This uses an Arduino Uno, two joysticks, an RF transmitter module, an antenna (not pictured), and a 9V battery (not pictured).

Description

The purpose of the transmitter module is to read inputs from two joysticks and send them to the receiver. We did this using analogRead( ) on each axis of each joystick. Each axis acts as a "channel" of data to be translated into servo positions to control the airplane flaps or throttle value to control the motor. Table 1 summarizes which axis on which joystick corresponds to what channel.

Table 1: Summary of which axis on which joystick corresponds to what channel.

Axis	**Left Joystick**	**Right Joystick**
X	Rudder	Aileron
Y	Throttle	Elevator

Due to the joysticks experiencing jitter—where the neutral position value jumped to different values surrounding the theoretical analog value median ($1023/2 \approx 512$)—we imposed bounds upon each axis of each joystick according to what their true neutral position values were. This ensures that for each joystick the neutral position returns the desired analog value of 512. We then convert the joystick values to servo values, mapping analog values (0-1023) to servo values (0^{o}-180^{o}). Next, using the RadioHead ASK ("Amplitude Shift Keying") RF library, we send the data of each channel to the receiver module. ASK works by representing digital values entering the transmitter as carrier wave amplitudes, and then mapping those wave amplitudes back to digital values in the receiver module. A carrier wave is the signal sent from the transmitter to the receiver. Greater carrier wave amplitude corresponds to a digital value of 1 and lesser carrier wave amplitude corresponds to a digital value of 0.

We had an additional channel whose data came from a simple incrementing counter to be used for packet validation in the receiver. A packet is defined as a “chunk” of data to be transmitted or received that contains data from all five channels: left joystick x-axis, left joystick y-axis, right joystick x-axis, right joystick y-axis, and counter.

We also included an additional antenna to supplement the onboard antenna, hoping to extend the communication range between the transmitter and the receiver. The original range without the additional antenna was roughly 50 feet, which increased to approximately 150 feet upon adding the antenna. The antenna length was designed to be ¼ of the signal wavelength we were sending: 433MHz signal → wavelength ≈ 70cm → antenna length: $70cm \times \frac{1}{4} = 17.5cm$. This ratio allows for better impedance matching between the transmission line and the antenna, resulting in fewer signal reflections and better transmission.

We found these RF modules to be largely unreliable due to high noise susceptibility. Their intended use is in RFID tags and simple remotes for devices like TVs or garages. They do not perform very well for applications requiring high-speed, large-bandwidth communication. Close proximity to the hospital likely contributed to ambient electromagnetic noise that interfered with RF communication outside.

Receiver Setup

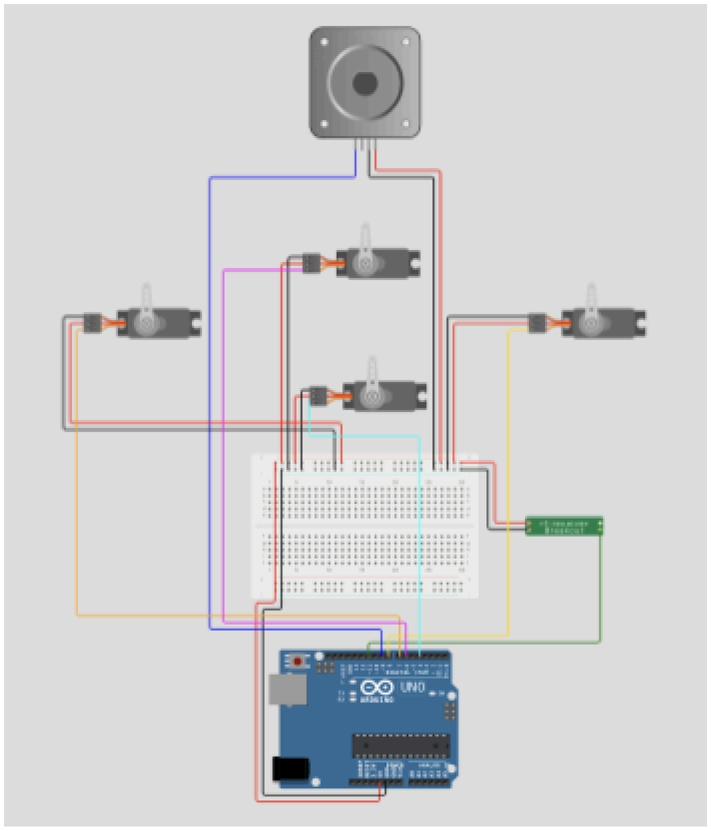
Figure 5. Shows the circuit setup for the receiver/onboard controller, which used an Arduino Uno, four servos, one three-phase drone motor, an ESC (not pictured), an RF receiver module, an antenna (not pictured), and a 14.8V battery (not pictured).

Description

The purpose of the receiver module was to receive packets of channel data from the transmitter and translate them into servo positions and throttle values. We received the channel data using the RadioHead library and assigned each channel's data to a variable corresponding to the specific channel identity. We mapped the values of these variables to values used by the ServoTimer2 library, which was needed while operating with RF in place of the typical servo library. This was due to the fact that both the RadioHead.h and Servo.h libraries rely on the same interrupt vector on the Arduino. These mapped values were then outputted to their respective servo (or ESC for the case of outputting a throttle value).

If the receiver stops detecting data from the transmitter, we will experience “error.” Error occurs when the counter channel value from the transmitter does not match the expected counter value in the receiver. For example, if we assume the transmitter and receiver turn on at the same time, the first transmitter packet should contain a counter value of 0. The receiver expects to receive a value of 0, and its “expected packet” value will increment along with the transmitter’s counter. This scenario would ensure that there is no error.

If we consider the scenario where the transmitter and receiver are out of sync (either the receiver turns on before or after the transmitter), there will be a difference between the expected packet value in the receiver and the current packet value sent from the transmitter. No matter the case, assuming the transmitter has not been turned off permanently (and is therefore no longer transmitting any packets at all), the receiver’s “error rate” counter will increment by 1. After this increment, the receiver will set the expected packet value equal to the current packet value plus one to ensure that the transmitter and receiver packet values are aligned.

If we assume that the receiver stops receiving packets from the transmitter for an extended period of time, the error rate will continuously increase. Once the error rate has reached a value of 10, we set servo positions to the middle and slowly decrease the throttle value. This is a safety precaution to land the plane in the case of flying out of range of the transmitter.

Once again, we used an additional antenna on the receiver to extend the communication distance between the transmitter and the receiver.

Part 4: Issues Encountered

Bluetooth control posed significant challenges due to input instability and communication delays. Joystick movements and button presses often caused random jitters in the servos, making precise control difficult. Attempts to debounce inputs smoothed these jitters but introduced lag, sometimes delaying responses by several seconds. While the system successfully demonstrated proper actuation of control surfaces and throttle, the unpredictable behavior and delayed inputs made it unsuitable for flight. A higher-quality Bluetooth receiver may be able to better filter out the transmission noise and make this a viable method of control.

Implementing the MPU6050 IMU presented significant challenges due to vibration sensitivity, drift, and unreliable data transmission. The IMU's accelerometer and gyroscope measurements require processing to produce meaningful orientation data, but vibrations from the motor introduced noise, complicating this process. Additionally, the gyroscope data suffered from cumulative drift over time, making long-term measurements inaccurate without correction. Although we attempted to use a Madgwick filter to address these issues, its mathematical complexity and implementation difficulty exceeded the scope of our project. Hardware inconsistencies further compounded the problem, as the IMU often stopped transmitting data, rendering it unreliable for use in the flight controller.

The RF link we used proved inconsistent, with reliability varying based on the environment and motion. While it sometimes worked correctly, transmission often became unreliable, particularly when the transmitter or receiver was in motion or when used in certain rooms. At one point we got it to work out to around 150 feet in testing, but a few minutes later it could not transmit successfully even when the antennas were just inches apart. This unpredictability made it unsuitable for an RC plane, which is constantly moving and requires a stable, dependable connection to maintain control during flight. Typical RC plane receivers cost between \$16 and \$20, while transmitters run from \$30 to several hundred dollars. The RF links we used cost less than \$2 apiece, and though they could certainly communicate and transmit data, they would be unable to connect to each other as often as not. With a higher-quality RF link, we would likely have been able to acquire more reliable control of the airplane in flight.

Finally, on the last day of testing, the airplane… caught fire. Since neither the Bluetooth or the radio frequency links worked reliably enough to communicate effectively with the onboard receiver, we tested the airplane with a commercial transmitter and receiver to see if it would actually fly at all, with the receiver linked directly to each servo. This mimicked the setup that we had achieved with the unreliable Bluetooth and RF communication, and was simply a cleaner way of transmitting the data with better equipment. Initially using our 14.8V battery pack, we could fly the plane, but it did not have enough power to stay airborne for more than a few seconds. We had access to an *old* drone battery that could provide between 22.2V and 25V at peak power. We tested this new setup at low power settings without attempting to fly the plane, and everything worked correctly. The motor clearly received more power than it did with the 14.7V battery pack, even at low power settings, leading us to believe that this setup would yield an actually flyable aircraft. However, when we applied maximum throttle during takeoff for the first time with this new battery pack, there was a “pop” and a puff of smoke. The battery, asked to supply high current for the first time in a couple of years, caught fire and burned the entire front of the plane off. It smelled pretty bad.

Conclusion

During this project, we explored the design, construction, and operation of an RC airplane controlled by Arduino systems. Despite the numerous challenges encountered, we were able to explore control system implementation, signal processing, and get perspective on the limitations of some pieces of hardware.

While the IMU-based fly-by-wire system proved too complex for the scope of this project, our exploration of the Madgwick filter and quaternion mathematics introduced us to inertial navigation and data smoothing techniques, which are useful in many applications outside of aerospace. Similarly, while Bluetooth control faced limitations due to jitter and delay, it demonstrated the potential of using alternative communication methods for RC control other than traditional radio frequency data transmission.

The use of the RF modules was slightly difficult to figure out at first but eventually became an exciting way to transfer information. Once communication between the transmitter and receiver was established, and we were able to transmit joystick position values, we then encountered issues trying to control servos with that data. We learned that the servo library we were using was utilizing interrupts on the same Arduino vector that the RadioHead library was using. Once we switched to a different servo library, ServoTimer2, we were able to control the servos, though with some difficulty. ServoTimer2 does not use the highly accurate Timer1 built into the Arduino Uno, but the slightly less reliable Timer2. Although it worked when tested with two servos at the same time, scaling to four servos caused some issues with the new library. The servos jittered and had a limited range of motion, which indicated that the new library was not as capable of controlling the servos as the conventional Servo.h library.

The project had very mixed results. Although we only achieved flight for a few seconds with a commercial transmitter and receiver, this was an ambitious project in which we succeeded in achieving several milestones, including the successful actuation of flight control surfaces as controlled by a pilot via multiple means of communication. The Bluetooth and RF link could both communicate with the onboard Arduino and adjust the flight controls in the same manner that we used to control an RC plane with a commercial receiver and transmitter successfully; the quality of the communication was simply not sufficient to allow for reliable or even practically functional control of the aircraft.

We would like to emphasize that the airplane itself was capable of flight, and we did establish communication with the onboard Arduino that exactly mimicked how commercial transmitters and receivers communicate to control RC airplanes. However, the quality of these communication links was unreliable. With more reliable components and additional development time, many of the obstacles we encountered could be overcome, leading to a fully functional system.

Appendix

All code used in this project can be found in the GitHub repository:

https://github.com/spencerh122/CMPE-3815---Microcontroller-Systems-Final-Project