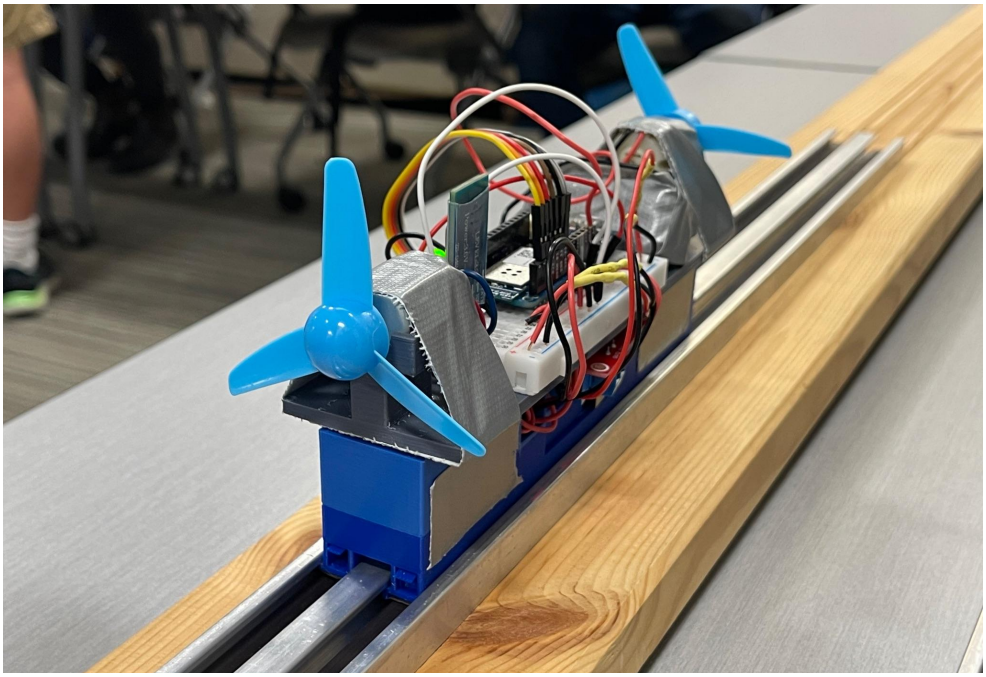
CSCI 1600 Project Report: MagLev Train

Kevin Hsu, David Owoade, Ben Bachmann, Zhongzheng Xu, Michael Kearney



1. Introduction

This report details the development of a magnetically levitated train model for use as an educational toy for high school science students. The primary goal of this project is to design and construct a simple maglev train, measuring approximately 5 inches in length and 1.5 inches in width, which operates on a magnetic railway track system. The train uses magnetic levitation, achieved by magnetic tapes on its underside repelling against magnetic rails, to hover above the track.

The train's movement is powered by a motor-controlled fan propeller, directing speed and motion. A controller FSM reads user input from sensors and sends the information over Control is maintained through a finite state machine (FSM) with three states: STOP, FORWARD, and BACKWARD, utilizing an H-bridge for motor control and PWM for speed adjustment.

Incorporating two Arduino boards, one handles the speed and brake input for the motors, while the other, mounted on the train, receives these inputs via WiFi. This setup provides a responsive control system. The user interface inputs includes a potentiometer for operation, a regular brake button to halt motion, and an emergency button for immediate stoppage. The emergency brake button utilized interrupt service routine (ISR) to ensure the rapid response from the motor controller.

As the prototype of an educational toy for high school science students, we hope that a guided build process and parts kit would allow our users to combine physics and computer science knowledge to make something fun and rewarding. We used common, safe components like hobby motors, 9V batteries, and standard breadboard components. Our kit would include instructions and specialized components like the H-bridge, Bluetooth transmitters, and train and track. The Bluetooth transmitters and H-bridge are versatile and can be reused for a wide range of future projects that building our toy might inspire.

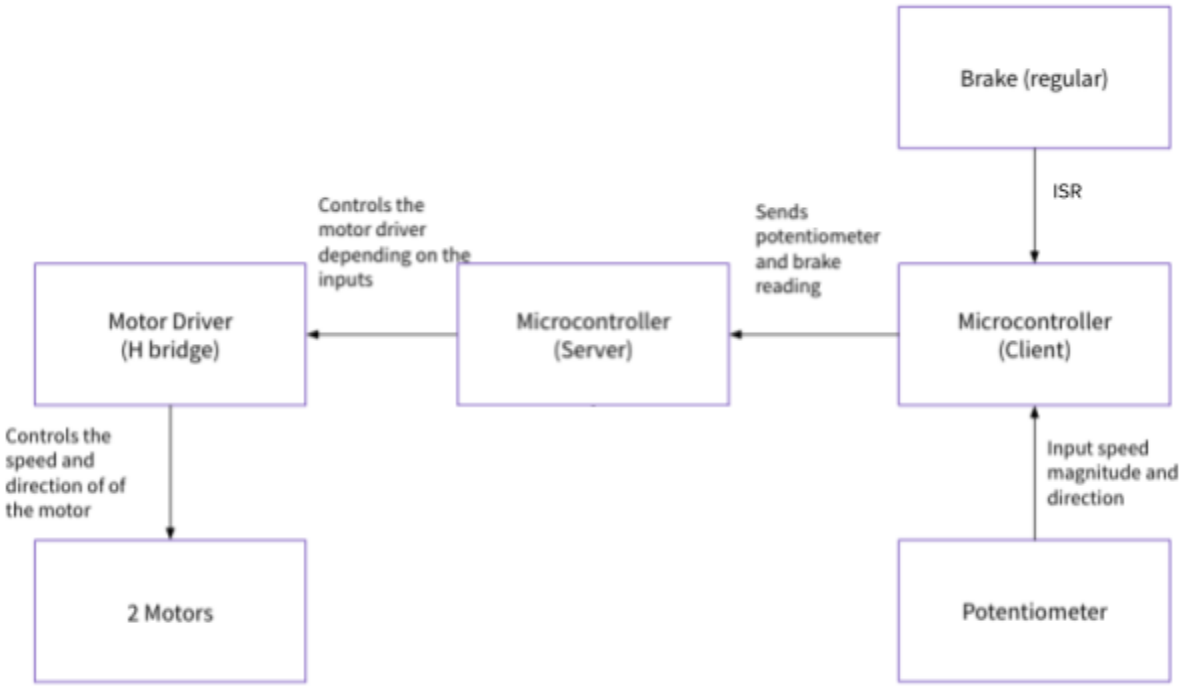
We made some assumptions because this is an educational toy and not a real train. First, we assumed a distance between controller and train of less than 10 meters. Second, we designed a press-and-release brake system that temporarily disables the motors instead of acting as a true emergency brake. Third, we designed safety into the project by using hobby-like components and designing low-current circuits. Our code testing focused on ensuring a reliable educational product, not on meeting the safety-critical requirements associated with trains that carry people.

2. Requirements

The controller will consist of a user-accessible potentiometer to control speed and a user-accessible brake button. The system will meet the following requirements:

- R1: When the potentiometer is changed, the propellers on the train shall update their rotation based on the user input.
- R2: While the brake button is pushed, the system shall not supply power to the propellers.
- R3: If the potentiometer is set and the brake is not pressed, the propellers shall turn forward if the potentiometer is set to the right, backward if the potentiometer is set to the left, and not at all if the potentiometer is set to the middle.
- R4: The propellers should turn at a rate between 0 and their maximum speed that is proportional to the setting of the potentiometer between the middle OFF position and the furthest setting on the side that the potentiometer is set.

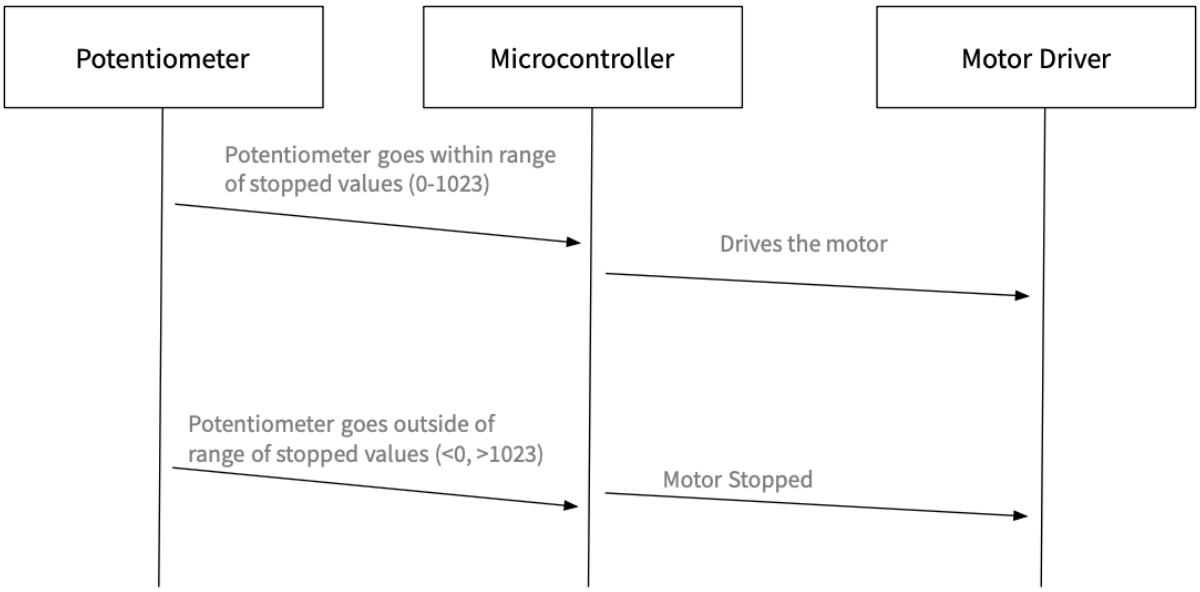
3. Architecture Diagram



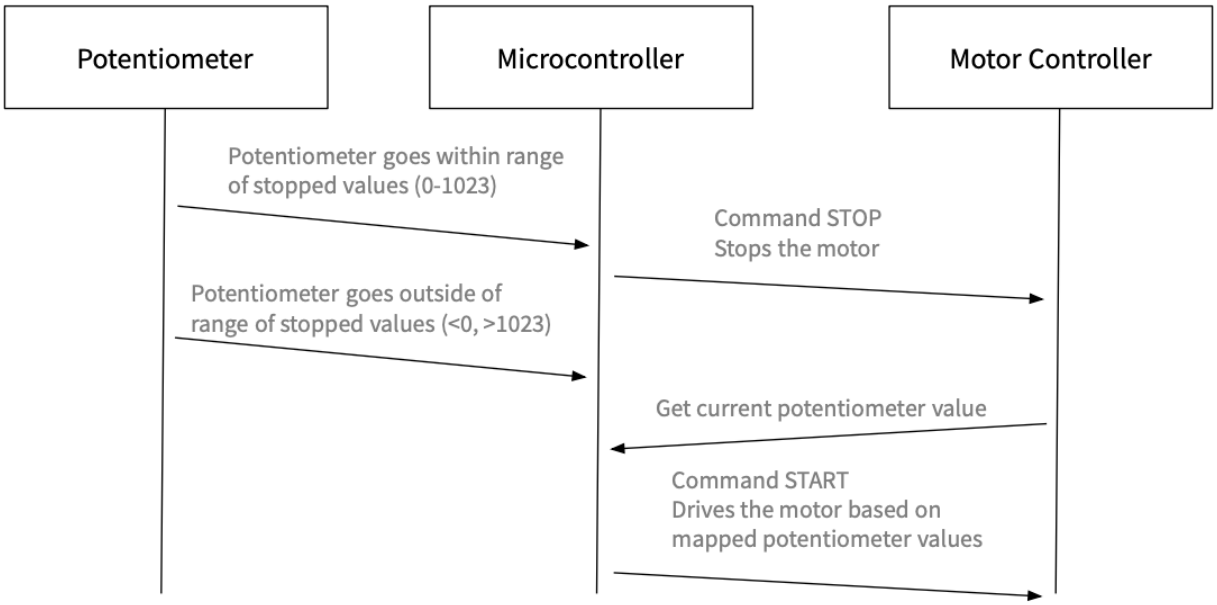
4. Sequence Diagrams

Note that, in accordance with our composite FSM design, we are treating our two MCUs as clones and the communication between them as analogous to the communication within an MCU. As such, we did not include two MCU's in our sequence diagrams, in the same way that we wouldn't include internal MCU logic in a sequence diagram.

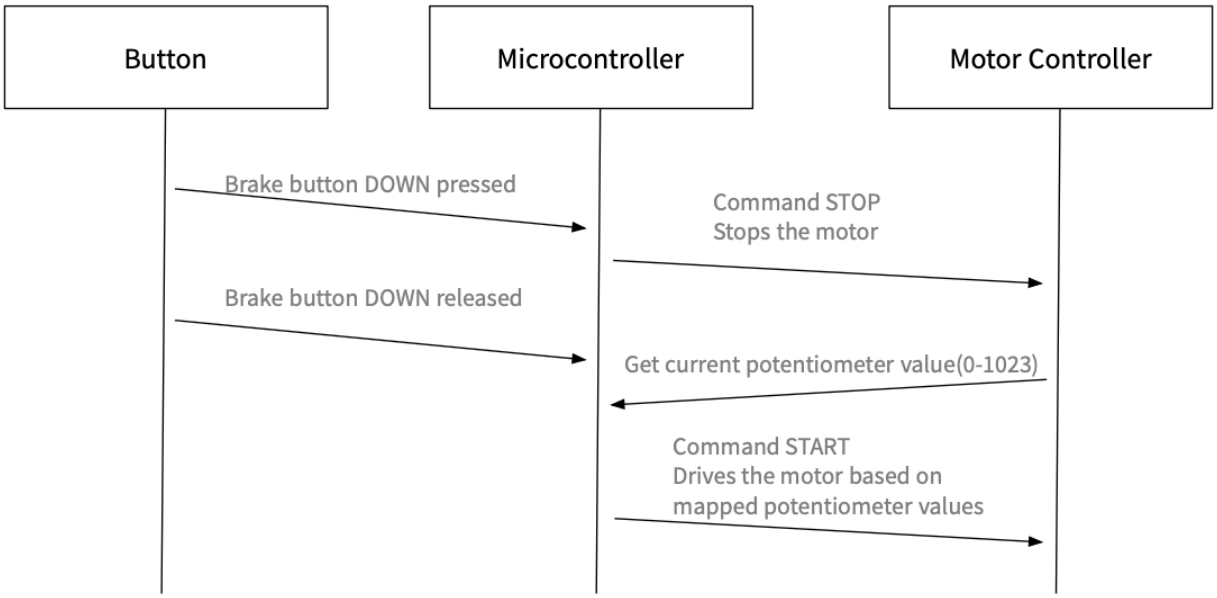
Scenario 1: User starts the train. Turn the potentiometer to the range of acceleration, and the train propeller starts rotating.



Scenario 2: User stops the train by decreasing the speed to zero. Turns potentiometer to specific angle ranges, and train propellers stop.

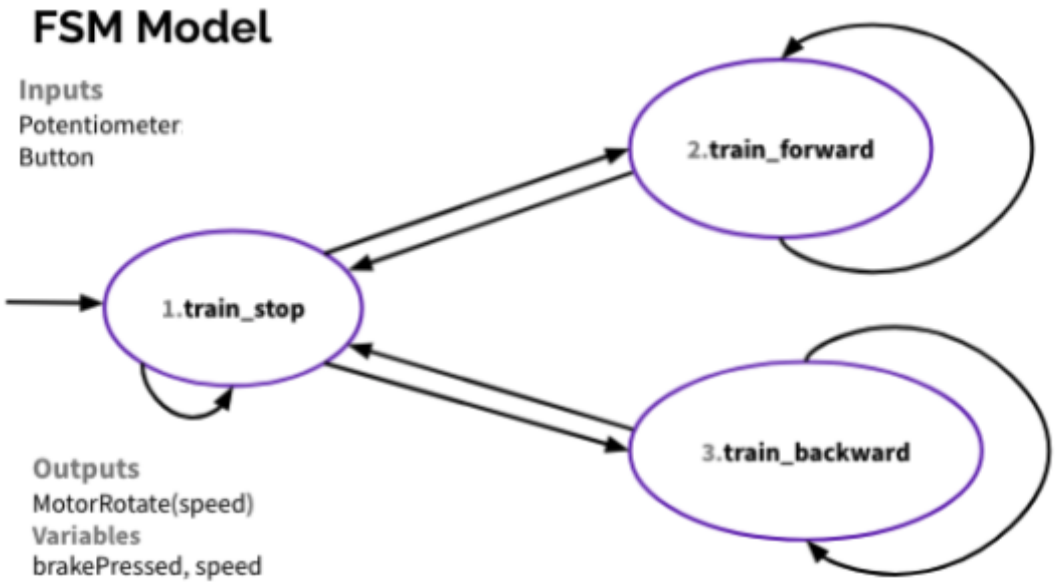


Scenario 3: User stops the train by pressing the emergency brake button. The train propellers stop, and when emergency brake releases, the train starts moving again based on the current potentiometer value.



5. Finite State Machine

Even though we use one MCU on our controller and one MCU on our train, we only have one FSM. We designed our system as a composite FSM. The non-hybrid FSMs for each MCU would be identical because the states, inputs, and outputs are identical. Essentially, our controller and train MCUs are clones of each other, allowing a wireless gap between the user input and the train motors. The difference between the MCUs is not in the states or transitions, but in where our implementation gets the inputs and what it does with the outputs: the train MCU's inputs are updated by WiFi and not by reading pins, and the controller MCU's speed output is printed to the Serial monitor and isn't attached to a motor.



FSM Model - Transition Table

Transition	Guard	Explanation	Output	Variables
1-1	(0 <= potentiometer < 1024 ^ ¬isPressed(button1)) v (isPressed(button1))	MagLev train is stationary	-	Speed := 0
1-2	0 <= potentiometer < 1024 ^ ¬isPressed(button1)	MagLev train is moving forward	MotorRotate(speed)	brakePressed := false Speed = map(pot, 0, -255, 1023, 255
1-3	0 <= potentiometer < 1024 ^ ¬isPressed(button1)	MagLev train is moving backward	MotorRotate(speed)	brakePressed := false Speed = map(pot, 0, -255, 1023, 255

Transition	Guard	Explanation	Output	Variables
2-1	(0 <= potentiometer < 1024 ^ ¬isPressed(button1)) v (isPressed(button1))	MagLev train is stopping	MotorRotate(0)	brakePressed := true Speed := 0
3-1	(0 <= potentiometer < 1024 ^ ¬isPressed(button1)) v (isPressed(button1))	MagLev train is stopping	MotorRotate(0)	brakePressed := true Speed := 0
2-2	(0 <= potentiometer < 1024 ^ ¬isPressed(button1))	Maglev train is still moving forward	MotorRotate(speed)	brakePressed := false Speed = map(pot, 0, -255, 1023, 255
3-3	(0 <= potentiometer < 1024 ^ ¬isPressed(button1))	Maglev train is still moving backward	MotorRotate(speed)	brakePressed := false Speed = map(pot, 0, -255, 1023, 255

6. Traceability Matrix

	R1	R2	R3	R4
T1-1		X		
T2-1		X		
T3-1		X		
T1-2	X		X	
T1-3	X		X	
T2-2				X
T3-3				X

7. Test Plan

We will run various input values into the FSM and compare the output per cycle to what is expected and reasonable. Some notable points include the potentiometer thresholds mapping to motor drive speed (positive, negative, or stopped). In terms of transition coverage, we cover all valid transitions of our finite state machine, in addition to situations where the user brakes or when they transition from forwards to backwards (broken down from 2 -> 1, and 1 -> 3, and the opposite direction). Non-valid transitions are not tested, as the only invalid transition would be forward / backwards directly, for which we tested the valid, broken down version as intended. Our output results are not mocked, but we verify the parameters passed into the `MotorRotate` function called in our output. We perform integration testing by confirming that the state transitions are as expected by comparing the end state with the resulting state, and that the speed variable changes based on the input as shown in our sequence diagrams. This satisfies our whitebox testing goals.

This step also verifies that our calculations for the potentiometer -> speed are correct within the code compared to the manually calculated versions.

System testing wise, we do end to end testing by turning the potentiometer and see how that changes the motor direction. It can be difficult to detect the direction by eye, so in the early stages what we attempted was instead of using the propellers, we used small foam pads that were marked with markers. We also tested hitting the physical brake to see if it would halt the system. Lastly, we also tested scenarios specified in the sequence diagrams as a testing guide.

Additionally, for our mock wifi system, we also tested using serial input to execute commands and see how well that emulates the intended behavior. We tried a series of input values, and verified that the state transitions and behavior were as intended.

Test	**Start State**	**End State**	**Input**		**Variables**	**Output**
			Potentiometer	**Button1**		
0	1	1	-	false	Speed := 0	MotorRotate(speed)
			-	true	Speed := 0	
1	1	2	512	false	Speed := 0	MotorRotate(speed)

			1023	false	Speed :=255	
2	1	3	512	false	Speed := 0	MotorRotate(speed)
			0	false	Speed :=-255	
3	2	1	1023	false	Speed := 255	MotorRotate(speed)
			512	false	Speed := 0	
4	2	1	-	false	-	MotorRotate(speed)
			-	true	Speed := 0	
5	2	2	1023	false	Speed := 255	MotorRotate(speed)
			1000	false	Speed := 244	
6	3	1	-	false	-	MotorRotate(speed)
			-	true	Speed := kk0	
7	3	3	0	false	Speed := -255	MotorRotate(speed)
			10	false	Speed := -250	MotorRotate(speed)

```
/**
 * Scales potentiometer input from [0,1023] to [-255,255]
 * speed will be modified again later by motor driver func
 */
int speed_map(int pot_input)
     If p-value >= 512: return (p-value - 512) / 2 # forward
     else: return -1 * (255 - (p-value) / 2)       # backward
```

8. Liveness Requirements

Note that the emergency brake is button1 in the latest implementation.

Safety requirements

1. If button1 is pressed, the motor's speed in the next state should be 0:

   $G(button1 \rightarrow X(speed=0))$

2. If button1 is pressed and the potentiometer value is changed, the speed in the next state should not be updated to the potentiometer value:

   $G((button1 \wedge X(pot=pot')) \rightarrow G(\neg(X(speed=\mp map(pot')))))$

Liveness requirements

1. When the potentiometer (speed control) is greater than 0, so long as the button1 (emergency brake) is not pressed, the train should eventually start moving forwards:

   $G(((potentiometer > 0) \wedge \neg button1) \rightarrow F(Speed=value))$

2. When the potentiometer (speed control) is less than 0, so long as the button1 (emergency brake) is not pressed, the train should eventually start moving backwards:

   $G(((potentiometer > 0) \wedge \neg button1) \rightarrow F(Speed=-value))$

3. If the potentiometer (speed control) is 0, the train will eventually come to a stop:

   $G((potentiometer=0) \rightarrow F(trainStopped))$

9. Environment Processes

- Emergency Brake System
    - This is modeled using a push button, which is either pressed/released, which result in discrete values (HIGH/LOW)
    - Best modeled as a discrete system, since the emergency brake is implemented by using TWO values, whose value is either 0 or 1.
    - Best modeled as a non-deterministic signal, since a user can push the button at any time.
- Motor Speed Control System (Potentiometer and Motor Driver)
    - This is implemented using a range of analog values determined by the potentiometer reading, as well as digital values of the motor driver' Enable pins.
    - Thus, this can take discrete values or continuous values, it is therefore labeled as a hybrid system.
    - The user can adjust the motor's speeds at will by turning the potentiometer, so it is best modeled as a non-deterministic signal.

10. Code Deliverable: Descriptions

`arduino_secrets.h`

Defines the SSID in the WIFI connection. “Brown-Guest” in our case.

`fsm.h`

We coded the main part of our FSM in this file, where our FSM takes inputs from the WIFI communication routine (or mocked) and updates its current state, setting the correct speed for the motors. There are also helper functions to read in dummy inputs and print out the current status of the FSM in the serial monitor. In this file, we also defined our ISR handler (line 23 - 34), which, triggered by a button, would stop the train as an interrupt.

`embedded-maglev.ino`

Our main loop is located in this file, where we run our FSM and tests. In `setup()`, we:

- Set up the interrupt button pin and attach interrupt to be triggered (line 35 - 40)
- Set up the pin mode and voltage of the motors to initial stop state (line 41)
- Set up the watchdog timer (line 44)
- Set up and connect both server and client board to the wifi (if not mock)

Inside our main loop, we used if/else to establish some different scenarios when the program runs,

> **Testing:** we run all our previously defined tests, by inputting input variable values, and compare the end state with what we anticipated. Test results are printed to the serial monitor.
>
> **Mock wifi:** we would “mock” wifi communication and read in dummy inputs from the serial monitor. This scenario is efficient for testing the functionality of our FSM and the transitions between states.
>
> **Wifi:** determined by a macro IS_SERVER, we would read in inputs from the client arduino board (which hold our potentiometer and brake buttons), send it to the server as an HTTP request periodically (based on timers), which parses it and updates the state of our FSM. This is the primary scenario we expect our users to be in.

At the end of our main loop (line 119), we also pet our watchdog timer to prevent a reset.

`train_motor.h`

This header file contains the helper functions to set up the pin numbers for the H-bridge (motor controller) and toggle the motors at a certain speed. `SetUpMotors()` set all connected pins to output mode and ensure the motors are stationary as their initial state. `SetMotorSpeed()` toggles both our motors at a certain speed. Since H-bridge uses PWM, we input the clamped potentiometer reading as the function parameter, and make use of the sign to speed value to be the direction of the motors.

`utility.h`

In this file, we wrote out our predefined test cases and inputs into arrays based on the table in 7. Test Plan, as well as the expected end states for comparison when testing. We also defined the state enums to use, and migrated the s2str function from the lab code for use.

`watch_dog.h`

We wrote our watchdog timer setup functions in this file. In `setUpWDT()`, we clear existing or pending watchdog timer interrupts and enable the watchdog following code from our previous lab. We also wrapped petting watchdog into a function called `petWDT()`.

`wifi-status.h`

This file is mostly migrated from lab code that connects to university wifi, and prints out different statistics such as MAC and IP Addresses. Previously, a version of this also took in passwords in order to work with personal hotspot networks instead of Brown Guest. The majority of the server-client code resides in magtrain.ino.

`bluetooth/train/`

This directory contains the bluetooth implementation of wireless communication for the train MCU in train.ino and train.h. Because we had to adapt to bluetooth in the last week and consider the impacts of the protocol on existing implementation, we isolated this functionality in a separate directory. train.ino sets up the MCU to receive datagrams from the HC-05 module using modified serial communication. We configured pins 4 and 5 as RX/TX so we could implement a custom SERCOM_Handler that directly controls the motors without passing data through the Arduino library. Encoding a full set of

instructions in a single UART frame and directly processing the instructions allowed us to achieve extremely low latency. Our message frequency was 1000Hz, which approaches the limitations imposed by a 9600 baud rate. train_test.ino includes hardware testing functions that tests circuit construction.

`bluetooth/controller`

This directory contains the bluetooth implementation of wireless communication for the controller MCU in controller.ino and controller.h. The FSM is almost the same as our wifi implementation but includes a single additional initial CONNECTING state to handle differences with bluetooth. It also outputs a visual indicator of speed and state to an 8 segment display which was used for final testing purposes when Serial communication over USB would have disrupted low-latency communication with the HC-05 bluetooth module. controller_test.ino includes hardware testing functions that tests circuit construction.

11. Instructions for running unit tests

- Clone the repository at https://github.com/michaelkearney55/embedded-maglev.git
- In `embedded-maglev.ino`, make sure that the `#define TESTING true` flag is set to true.
- Upload `embedded-maglev.ino` to the MKR1000 and open the serial monitor. You should see “All tests passed”.
- The tests are defined under `utility.h`
    - If `testEndSpeedVar` is edited so that the last element is `-249` instead of `-250`, the tests should break at `Running test 7`, and notify the user about expected vs calculated speed, as well as how the resulting state is (3) BACKWARD instead of (1) STOP
    - This is an example of checking if the internal calculations for potentiometer -> speed value mapping are correct.

12. Reflection of goals and challenges

Previously, our goals were to have an offboard system that controls the train through WiFi, making use of two Arduino boards. The offboard system would contain a potentiometer and brake that changes the behavior of the on board propeller system.

Some challenges we encountered included WiFi communication between devices not working consistently. We used Arduino MKR1000's that have been set up on Brown's Guest network with their MAC Addresses registered. Both were able to connect to the university network, but when the server-client connection establishment was attempted, the runtime would hang at connecting the two Arduinos. Occasionally it would go through, but the consistency was very low, and even then it broke off after sending three or four messages. We shifted back to testing the sample WiFi101 code on the Arduino website, but that had the same issues. At this point, we weren't able to pinpoint the problem being our code, our hardware, or the university network. So we attempted the same setup on personal hotspot instead, but that just failed at the WiFi connection step.

Our conclusion after discussion with the professor was to use a mock WiFi setup. We've defined a macro for the purposes of enabling and disabling the mock WiFi system

The mock WiFi system takes in serial input, comma separated, as values for the potentiometer and brake button.

The non-mock WiFi system establishes connections with the university network, and the client tries to connect to the server Arduino based on IP address. Interestingly enough, because the network connection takes too long, and server connection hangs, our watchdog timer gets triggered, which demonstrates that it works as intended.

To achieve wireless functionality for the demo, we implemented Bluetooth communication using two HC-05 modules. This allowed us to bypass internet networking and achieve wireless modified Serial communication between the train and controller. The different communication protocol had implications for the model we had developed. Based on this experience, we could have done exploratory testing earlier in the modeling process or made our model more general to give us more leeway in implementation.

13. Appendix: Defect Records

Reviewer names: Hannah Julius, Winston Hackett, Elizabeth Wu

Transition or state #	Defect	Fixed?	Comments
1-2	is toggle==1 ANDed with the other boolean condition included?	Fixed	Typo fix
1-3	is toggle==0 ANDed with the other boolean condition included?	Fixed	Typo fix
2-1	typo in isNotPressed	Fixed	Typo fix
3-1	typo in isNotPressed	Fixed	Typo fix
2-2	is toggle==1 ANDed with the other boolean condition included? Perhaps speed should be motor speed?	Fixed	We make a distinction between potentiometer values and (motor) speed in other sections. Given that there's no other speed factor, we're comfortable keeping speed
3-3	is toggle==0 ANDed with the other boolean condition included?	Fixed	Typo fix

Reviewer names: Heidi Schaefer, Aaron Igra, Omar Orozco, Cody Sims

Transition or state #	Defect	Fixed?	Comments
-	Inputs are not initialized in FSM	Fixed	Added default values
1	Union of 1-1, 1-2, 1-3 does not encompass all inputs	Fixed	Combine isPressed vs isNotPressed