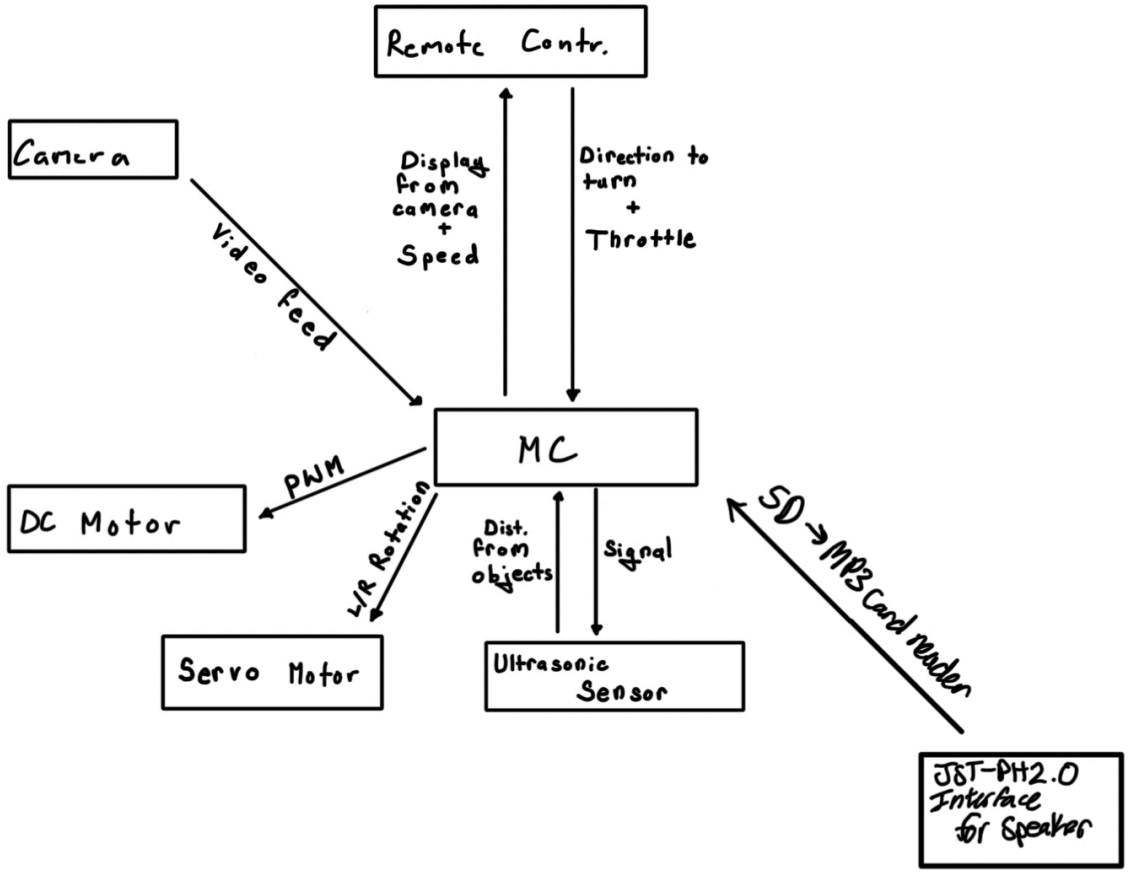
Introduction

Our project is seeking to develop an RC car. This consists of a microcontroller, an ultrasonic sensor, a speaker with configurable music, and our motors, all encased within a 3D-printed chassis. The user will be able to turn and move backwards and forwards using an application on a mobile device. Our assumptions about the user are that they are able to interact comfortably with this mobile device.

Requirements

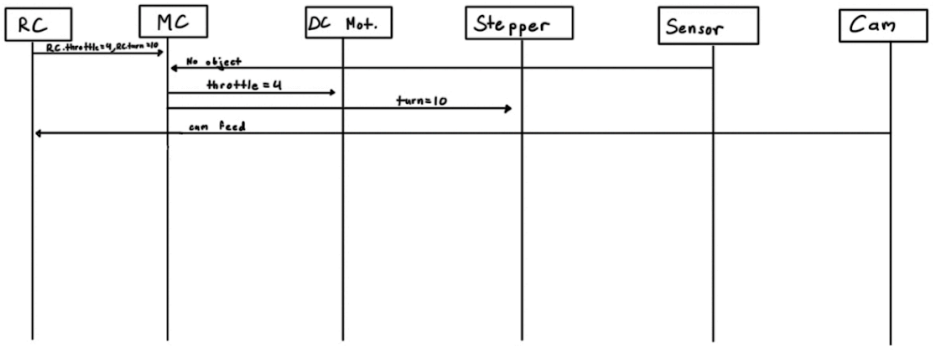
- **RS-1:** The RC car shall drive forwards and backwards via remote control.
- **RS-2:** The RC car shall be able to turn left and right via remote control.
- **RS-3:** Steering shall be controlled via wifi/bluetooth.
- **RS-4:** Shall automatically detect obstacles using an ultrasonic sensor and halt when in range
- **RS-5:** Should have a mobile application to control the RC car remotely from.
- **RS-6:** Should have a camera that gives display feed to a mobile application.
- **RS-7:** Should have a speaker that can play music that the user gives.
- **RS-8:** Should be able to follow a human through a computer vision model

Architecture

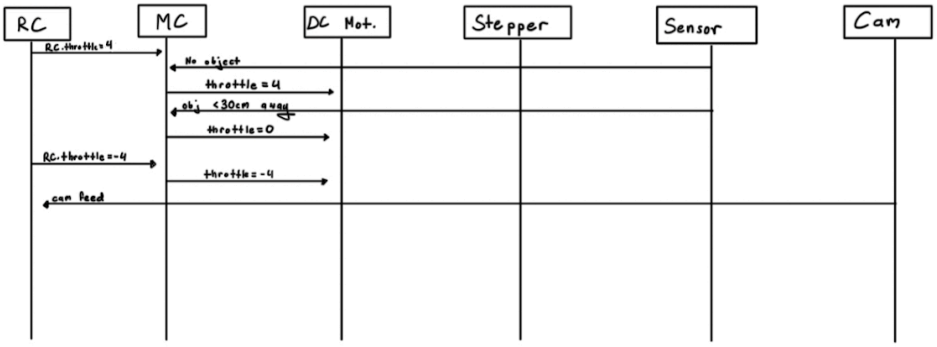


Sequence Diagrams

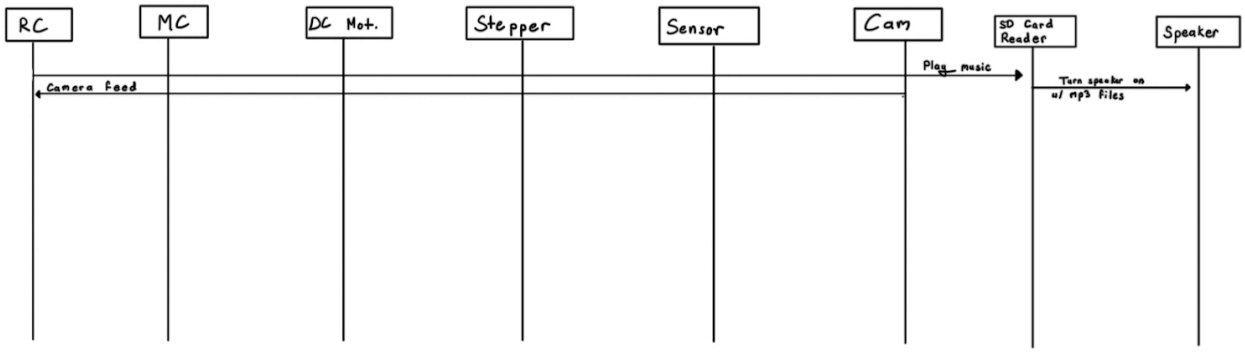
Scenario: The car moves foward and turns



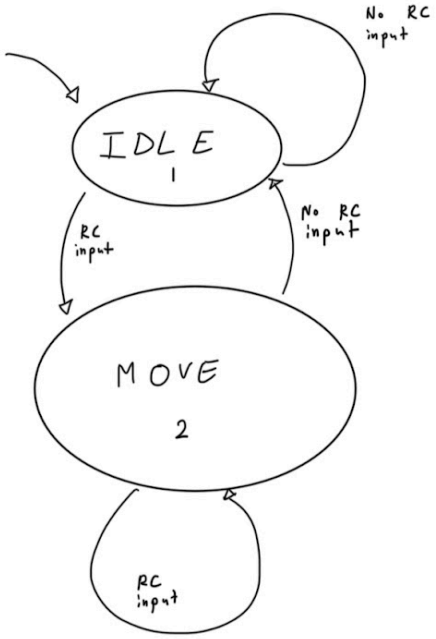
Scenario: The car moves foward detects an obstacle, and moves backwards



Scenario: The user plays music using the controller



FSM



INPUTS

RC.throttle $\in$ {-255,255}

RC.turn $\in$ {-255,255}

OUTPUTS

move (throttle int, turn int)

FSM Variables

throttle = 0

turn = 0

sensed Distance = 0

1-1

GUARD

RC.throttle = 0 $\wedge$ RC.turn = 0

1-2

GUARD

(RC.throttle != 0 $\vee$ RC.turn != 0) $\wedge$!(sensed Distance < 1ft)

FSM Variables

throttle := RC.throttle

turn := RC.turn

FUNCTIONS

move(throttle, turn)

2-1

GUARD

(RC.throttle = 0 $\wedge$ RC.turn = 0) $\vee$ (sensed Distance < 1ft $\wedge$ RC.throttle > 0)

FSM Variables

throttle := 0

turn := 0

FUNCTIONS

2-2

GUARD

(RC.throttle != 0 $\vee$ RC.turn != 0) $\wedge$!(sensed Distance < 1ft $\wedge$ RC.throttle > 0)

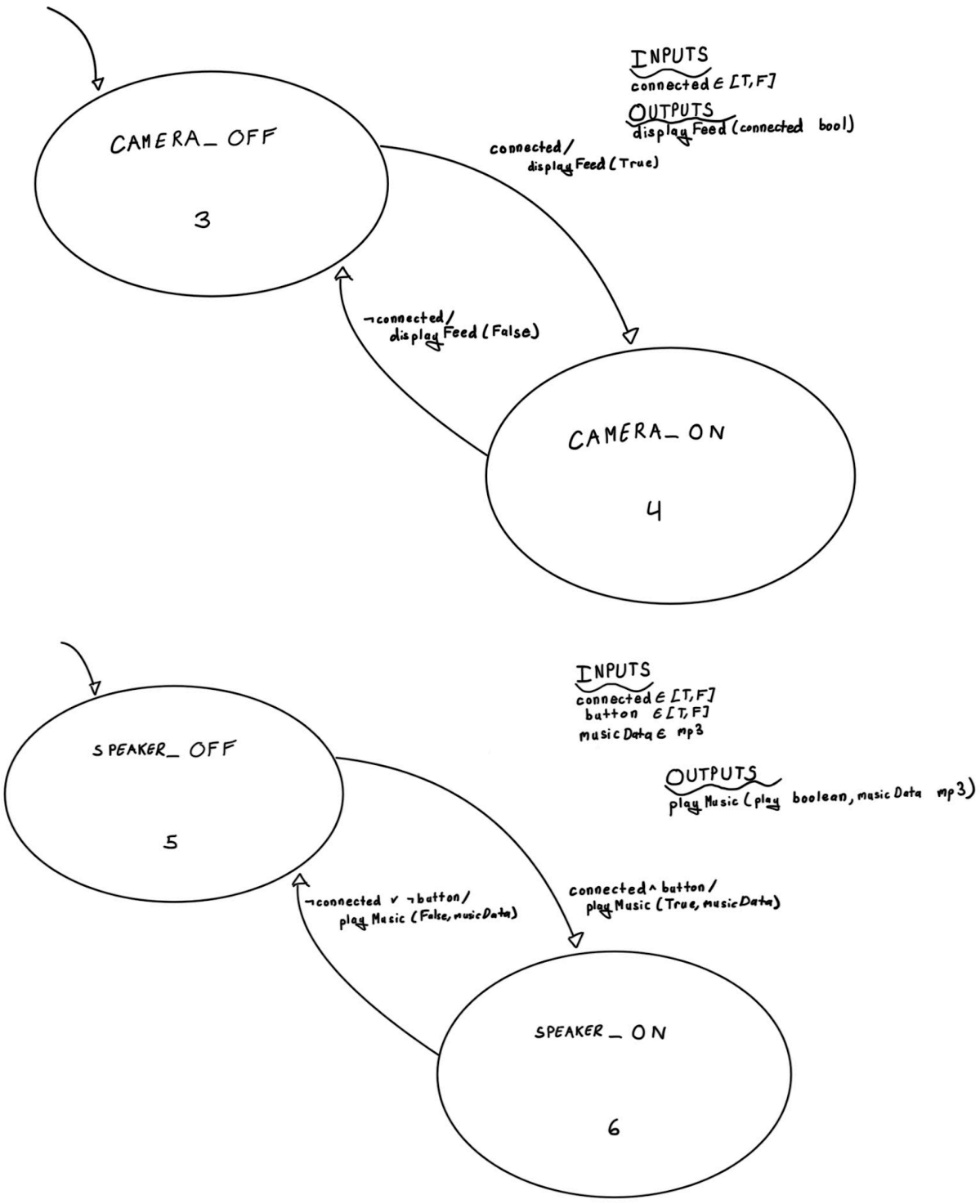
FSM Variables

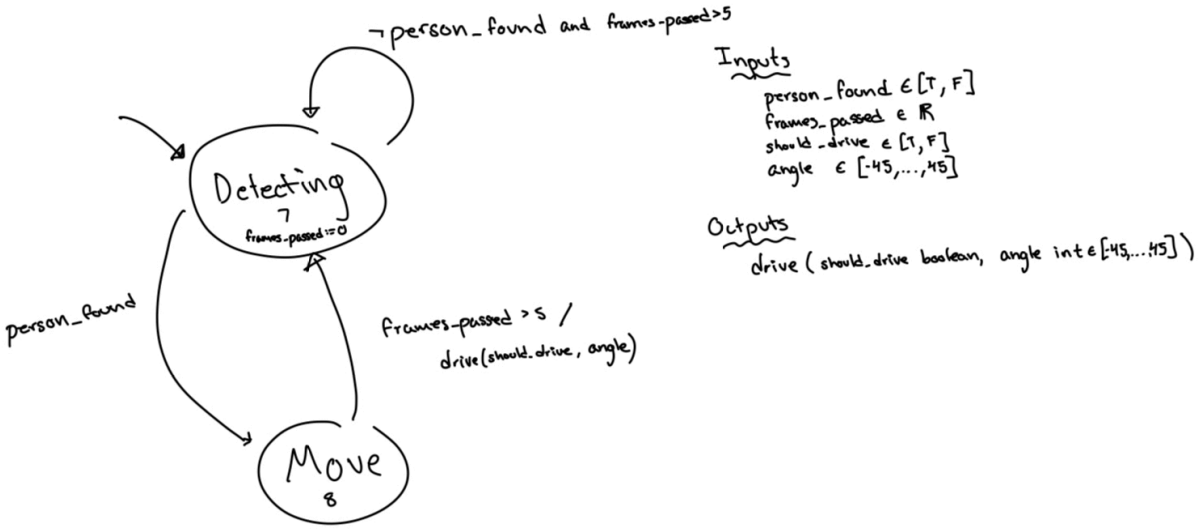
throttle := RC.throttle

turn := RC.turn

FUNCTIONS

move(throttle, turn)


INPUTS
connected ∈ [T, F]
OUTPUTS
displayFeed (connected bool)
CAMERA_OFF
3
connected/
displayFeed (True)
¬connected/
displayFeed (False)
CAMERA_ON
4
INPUTS
connected ∈ [T, F]
button ∈ [T, F]
musicData ∈ mp3
OUTPUTS
playMusic (play boolean, musicData mp3)
SPEAKER_OFF
5
¬connected ∨ ¬button/
playMusic (False, musicData)
connected ∧ button/
playMusic (True, musicData)
SPEAKER_ON
6



Traceability Matrix

	1-1	1-2	2-1	2-2	3-4	4-3	5-6	6-5	7-8	8-7
RS-1		x		x						
RS-2		x		x						
RS-3		x	x	x						
RS-4			x	x						
RS-5	x	x	x	x						
RS-6					x	x				
RS-7							x	x		
RS-8									x	x

Testing

Our testing strategy combined unit testing, integration testing, system testing, and hardware component validation to ensure that the RC car behaved correctly across both the embedded firmware and the mobile application layers. We approached testing in progressively expanding scopes: starting with isolated functional blocks, then validating interactions between modules, and finally confirming end-to-end behavior with real hardware. This structured layering let us confidently verify correctness at each stage without allowing bugs to propagate into later phases of development.

On the embedded side, we began with unit testing of the Arduino firmware. Several core functions—especially the finite state machine (FSM) implemented in updateFSM()—were central to the car's safety and movement logic, so we wrote comprehensive tests executed directly on the Arduino through a Serial-based test harness. These tests exercised every logical branch of the FSM, including transitions between IDLE and MOVE, forward-motion blocking when an obstacle appears, dead-zone behavior for steering and throttle inputs, the "resume distance" threshold, and the reverse-allowed safety logic that permits backing up even when forward motion is blocked. We also unit tested helper functions such as clampInt, clampFloat, and getParamValue, ensuring that the communication pipeline and safety boundaries behaved predictably. Together, these tests provided full logical coverage of the software-only control flow.

Beyond isolated unit tests, we ran integration tests on the embedded firmware to verify interactions between modules before introducing hardware into the loop. These tests connected the HTTP input-parsing stage, the internal command pipeline, the FSM, and the output-control functions. By simulating realistic combinations of user commands and sensor readings, we validated behaviors such as stopping when obstacles appear mid-movement, smoothly resuming forward motion once the path clears, and returning to IDLE when the user releases the joystick. This step ensured every software subsystem cooperated correctly before motor drivers, sensors, or other components became involved.

We also developed integration tests for the mobile application, which serves as the user's primary control interface. Using Jest and the React Native Testing Library, we simulated realistic user interactions—moving the on-screen joystick, pressing the STOP button, and interrupting gestures—and validated that the app correctly translated these gestures into PWM values, updated internal state, and sent accurate HTTP requests to the Arduino over WiFi. We mocked the global fetch function to intercept outbound network requests so tests could run consistently without requiring a real device. These tests confirmed that timing logic, gesture handling, state management, and networking all behaved correctly as a unified system.

Before integrating electronics into the full car assembly, we also individually tested each hardware component using basic Arduino sketches. This included verifying the DC motor's forward and reverse operation, sweeping the servo motor through its full range to confirm smooth movement, measuring known distances with the ultrasonic sensor, testing the JST-PH2.0 speaker by playing tones, and running a standalone capture/stream test on the camera module. This component-level testing step helped identify any defective or miswired hardware early, avoiding misleading failures later during system-level debugging.

Finally, after unit and integration tests were complete, we conducted full system and acceptance tests using the physical RC car, the deployed mobile app, and actual WiFi network communication. These tests confirmed that the car could drive forward and backward, steer left and right, respond to the STOP button, halt automatically when approaching obstacles, stream video, play audio, and accept remote commands from the mobile interface. Multiple team members performed these tests under realistic usage scenarios to verify robustness and user-facing correctness.

To decide when testing was sufficient, we ensured that every functional requirement (RS-1 through RS-7) was validated through at least one test—whether unit, integration, component, or acceptance. We also confirmed that all major branches of the FSM were exercised, since it is the safety-critical core of the system. Every hardware component was

validated both independently and again when integrated. Additionally, the system was tested under realistic, dynamic usage—such as abrupt joystick changes, sudden obstacles, and recovery from edge-case situations—to ensure behaviors aligned with expectations. Although we did not compute formal code-coverage statistics, we achieved near-complete logical coverage of all core software modules and verified correct system behavior through extensive real-world testing. Together, these layers gave us high confidence that the system behaves safely, reliably, and consistently with the project goals.

How to Run Unit Tests:

To run the arduino-side tests: simply uncomment out lines 199-212 in rc_control.cpp. This will run the tests and print the output of the tests to the serial monitor. Then, to run the fsm again and skip the tests, just comment out these same lines.

To run the mobile app-side tests: install npm via running ‘npm i’ in the terminal. Then, from the ‘rc_car/mobile/’ directory run ‘npm test’ and this should run the tests and print the output to the terminal

Safety and Liveness Requirements

Safety Requirements

1. ¬(throttle > 0 ∧ throttle < 0)
    a. The RC car should never move both forward and backward at the same time.
2. ¬(throttle > 0 ∧ sensedDistance < 25)
    a. The RC car should never move forward if a detected object is within 25cm.

Liveness Requirements

1. (throttle = 0) U (RC.throttle ≠ 0)
    a. The car’s throttle is 0 until the remote controller gives a non-zero throttle.
2. (turn = 0) U (RC.turn ≠ 0)
    a. The car’s turn angle is 0 until the remote controller gives a non-zero angle.
3. (throttle > 0 ∧ sensedDistance <= 30) -> F(throttle = 0)

At least 2 safety and 3 liveness requirements, written in propositional or linear temporal logic, for your FSM. Include a description in prose of what the requirement represents. Each requirement should be a single logic statement, made up of propositional and/or linear temporal logic operators, that references only the inputs, outputs, variables, and states defined in part 5 of this report.

Environment process to model for closed system

User Steering and Driving Input: Discrete, Non-deterministic

Each interaction through the joystick represents a discrete event—the user either provides a throttle/steering command or they don't. The timing of these inputs is fundamentally

non-deterministic because we cannot predict when a user will decide to press forward, turn left, or stop. Human reaction times vary based on external stimuli (seeing an obstacle, changing direction), internal decision-making, and unpredictable factors like distraction or hesitation.

While the joystick physically produces continuous analog positions, from the FSM's perspective, we sample these at discrete 100 ms intervals via HTTP requests. Each sample is an independent event with values in the range [-255, 255] for both throttle and steering. The non-deterministic nature means our FSM must be robust to arbitrary input sequences, including worst-case scenarios like sudden full-throttle commands when near obstacles.

Ultrasonic Sensor Distance Measurements: Hybrid, Non-deterministic

The distance to obstacles is fundamentally a continuous variable that changes smoothly as the car moves through space. The physical gap between the car and an obstacle doesn't jump discretely; it shrinks continuously as the car approaches. This makes it a hybrid system component because the distance can take any real value within the sensor's range. However, the system is non-deterministic because we cannot perfectly predict future distance values. The distance is deterministic for the same object, but a person or pet can suddenly walk in front of the car.

WiFi Communication Channel: Discrete, Non-deterministic

Network communication operates on discrete message-passing events. Either a command packet arrives or it doesn't. The timing and reliability of packet delivery are nondeterministic due to latency that can vary or loss due to signal issues.

Physical Car motion: hybrid, Deterministic

Though the FSM itself doesn't model continuous motion, the environment does. When the motors receive PWM outputs, the car's physical movement, like its acceleration, speed, and position, is governed. This makes the process hybrid, as the position, velocity, and acceleration are always continuous with continuous-time physics and physical laws. It's also discrete, as the FSM updates motor outputs at discrete loop intervals. Although PWM commands may arrive unpredictably due to user input, the car's physical response to them is deterministic. Once the motors receive a specific PWM value, the resulting acceleration, speed, and movement follow continuous physical laws. This makes the subsystem hybrid (continuous motion + discrete command updates), but it still follows the same rules: if the same PWM inputs and starting conditions are used, the car's path will always be the same. Any randomness arises from other environmental processes (user input, WiFi delay, moving obstacles), not from the motion model itself. Therefore, the car’s physical motion is appropriately modeled as a hybrid, deterministic process.

External Object or Human Movement: Hybrid, Non-deterministic
Objects in the environment (walls, people, pets) may move independently of the RC car. Their positions change continuously over time, which characterizes this as a hybrid system.

It is non-deterministic because a person could suddenly step in front of the car, or a pet could drift unpredictably in the environment. New obstacles could appear and disappear without warning. These changes affect the ultrasonic sensor readings and must therefore be modeled separately from the car's motion.

Description of the files

Core System Files

main.ino : Entrypoint to the program. Initializes **WiFi** connectivity (lines 5-23), starts the web server on port 8080, and coordinates both the MP3 player and RC car subsystems through a unified request routing system in the main loop.

rc_car.h : Header file defining the finite state machine structure for the RC car, including the `fsm_state` enum (IDLE, MOVE) and `full_state` struct that tracks distance, throttle, turn, and current state.

fsm.h Header declaring the core FSM update function that takes current state, commanded inputs, and sensor data to compute the next state.

mp3.h Header exposing the MP3 player subsystem's initialization, loop, and HTTP request handling functions.

RC Car Control Files

rc_control.cpp : Implements the complete RC car control system including ultrasonic distance sensing, motor control (throttle and steering), **WiFi** request parsing, and hardware interfacing. Lines 250-267 contain car_handleRequest() which parses HTTP query parameters (ud, lr) and updates control commands, aiding to satisfy the **WiFi** communication requirement.

The **interrupts** requirement is satisfied in this file:

- Lines 78-85 - **echoISR()**: Interrupt Service Routine triggered on ultrasonic sensor echo pin state changes
- Line 176 - **attachInterrupt(digitalPinToInterrupt(echoPin), echoISR, CHANGE)**: Attaches ISR to echo pin
- Lines 24-26 - Volatile variables (**echoStart**, **echoEnd**, **echoDone**) used for safe ISR-to-main communication
- Lines 97-101 - **calculateDistance()**: Processes ISR data with interrupt protection via **noInterrupts()/interrupts()**

The **watchdog** timer requirement is also satisfied in rc_control.cpp:

- Line 34 - **const int wdtInterval = 5000**: 5-second watchdog timeout
- Line 215 - **WDT.begin(wdtInterval)**: Initializes watchdog in **car_init()**

- Line 244 - **WDT.refresh()**: Periodic watchdog reset in main control loop to prevent system reset

rc_control.h : Header exposing the RC car subsystem's initialization, loop, and HTTP request handling functions.

fsm.cpp : Implements the finite state machine logic. Uses **analogWrite(SPEED_PIN, speed);** in line 135 to send a **PWM** signal to the L293D H-Bridge. This PWM signal controls how fast the car drives. This file also contains the sensor logic for auto-braking, including deadzones to prevent oscillation when near obstacles (20cm stop threshold, 25cm resume threshold).

The **PWM** requirement is satisfied in this file:

- Lines 113-129 - **setThrottleOutput()**: Implements PWM motor speed control using **analogWrite(SPEED_PIN, speed)** where speed ranges 0-255

MP3/Aux logic files

mp3.cpp : Manages the DFPlayer Mini MP3 module, handling play/pause/next/previous controls via both physical buttons and HTTP requests, track management, and status reporting with synchronized track state. Lines 182-186 contain **mp3_handleRequest()** which handles MP3 control via HTTP endpoints. This helps to satisfy the **WiFi** communication requirement.

Testing Files

carTests.cpp : Unit test suite for the RC car FSM, including 11 test scenarios covering state transitions, obstacle avoidance, hysteresis behavior, and helper function validation.

CarController.test.tsx : Jest/React Native Testing Library test suite for the RC car controller interface. Validates critical safety features including initial state (zero PWM values), STOP button functionality, connection display, and gesture handler termination behavior to ensure the car remains in a safe state during UI interruptions.

Mobile App Files

_layout.tsx : Defines the app's tab-based navigation structure using Expo Router, creating three main screens: Control (RC car joystick), Aux (MP3 player), and Camera (AI vision system), with tab switches.

use-drive-commands.ts : Custom React hook that encapsulates all HTTP communication logic for sending drive commands to the Arduino, including throttle/steering PWM values, auto-follow mode conversions, and request deduplication to prevent network spam.

WiFi requirements satisfied:

- **(Lines 18-43)**: Centralized HTTP command logic

index.tsx : The main RC car control screen featuring a square joystick interface with real-time PWM value display. Uses PanResponder for gesture tracking, converts joystick position to throttle (-255 to 255) and steering values, and sends commands via HTTP every 100ms.

mp3.tsx : A full-featured music player interface that communicates with the DFPlayer Mini via Arduino HTTP endpoints. Implements play/pause, track navigation, volume control, and automatic state synchronization with polling to detect physical button presses on the Arduino.

camera.tsx : AI-powered camera viewer that loads the ESP32-CAM stream in a WebView, injects TensorFlow.js and COCO-SSD for real-time object detection, draws bounding boxes on detected objects, and implements an auto-follow mode that tracks people and sends steering commands.

Camera Board

CameraWebServer.ino : Initializes the ESP32-CAM and connects to **WiFi** (lines 66-89), and starts an HTTP server that streams video.

Reflection

A reflection on whether your goals were met and what challenges you encountered in achieving them. You should only include challenges met or newly addressed since the milestone.

We met most, if not all, of our goals of our project. It was initially hard to find parts that were compatible with one another, and we had some issues in the beginning of building where certain parts we had purchased weren't working together as we expected. For example, we were trying to wire our motor, but the battery we purchased didn't have the right type of wire we could connect to the motor, and so when we tried to use both it created sparks which was scary. And so we decided to buy a different motor and battery that work together. The new motor has a smaller form factor, has much more plug-in play, and the new battery has lower voltage (6 volts) and is compatible with the new motor. We also bought a power supply module, which we hadn't initially planned for, to connect the battery to the breadboard. Using these new parts and new circuit diagrams, we assembled new circuits which ultimately worked!

Another challenge we ran into with parts was 3D printing the chassis, which the BDW printer messed up about four times. Each time, it would break in the middle of printing, print something too flimsy, or misprint it. We had to resort to ordering a 3d printed model online instead, which was unideal.

One of our other goals for the project was creating a mobile application which you can remote control the car from. This was coded in Typescript. Initially, we were having Wifi issues, which we realized was just never going to work on the Brown Wifi network, if we wanted to connect remotely to it. So we decided to work on it on external Wifi networks, in order to connect both the Arduino and the mobile app to the same Wifi network. We wired up a simple circuit and got it to light up an LED from the mobile app, and then transitioned to getting it to move the motor from the mobile app.

Another challenge we then ran into was getting the motor to work with the wheels. We made a model on Tinkercad, which worked, and got the motor to work with one wheel (but not

the other), but then the one wheel stopped working too. But we made it work eventually, and the issue was because of wiring and power supply being spread out. There was a faulty wire as well, which we didn't notice and we had to find out through manually looking at each part.

Another issue we ran into was for the ultrasonic distance sensing, we ran into an edge case that we didn't model properly in our FSM and didn't detect. It was when the car had stopped because it detected an obstacle, it couldn't back up. If you moved the joystick, it wouldn't back up, which we had gotten as a side effect of not wanting it to be able to move forward when it detected an object. But this was a case where we actually do want it to be able to move still. So we had to add a transition to our FSM to allow it to transition from IDLE to MOVE in this case.

After all that, we met all of our goals! We have the ability to drive forwards and backwards via remote control, the ability to turn left and right via remote, steering controlled through wifi, automatically detecting obstacles through an ultrasonic sensor and halting when in range, a mobile app that controls the RC car remotely from, a camera that displays feed to the app, and a speaker that the user can play music from off of the app.