Project Report: Audio-Reactive Wave Motion Mechanism

1. Introduction

Our project is an interactive kinetic sculpture where the sculpture's movement is determined by sound. Audio amplitude data is taken from a microphone and then converted into a PWM range to power a DC motor's speed. The DC motor rotates a central origin point of a web of strings, woven in a way where the rotation results in a wave-like movement at the ends of the strings. The intended user could be anyone, and is not just limited to humans, it will react to any sound input. However, the system does assume the following to produce meaningful and visually different output:

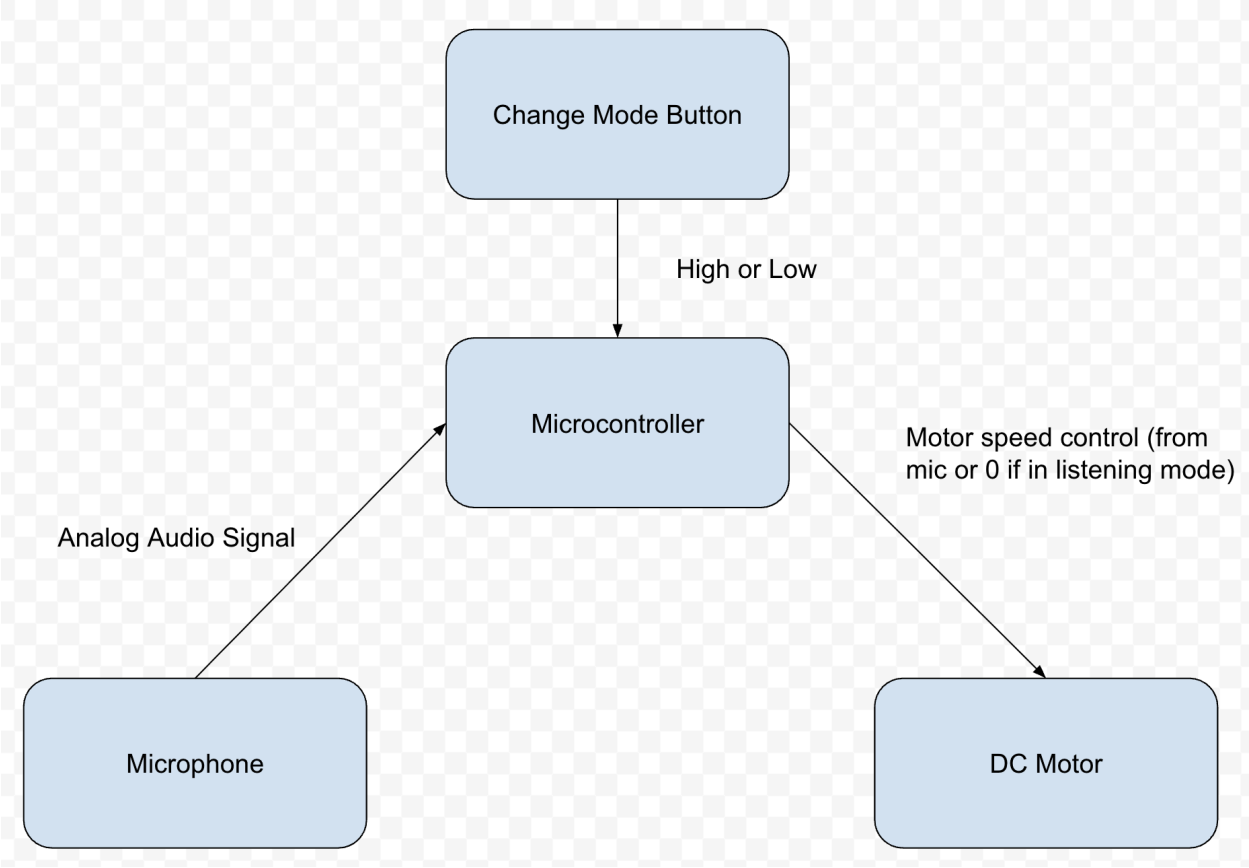
1. The user provides audio input of varying volume.
2. The user providing input is very close to the mic.
3. It is important that the strings don't get tangled, so the sculpture will only be used indoors where there is no wind

2. Requirements

Functional Requirements

1. The system shall be able to switch between listening and playback mode via an interrupt triggered by a button press
2. The motor shall spin continuously in playback mode, with speed proportional to audio volume read into the buffer during listening mode.
3. The system shall loop back to the beginning of the buffer when it reaches the end, ensuring the motor is always spinning when in playback mode.
4. The system shall be configured such that the motor speed range's lower bound should still be able to pull the weight of the sculpture and the upper bound should be at a speed where it doesn't tangle the sculpture.
5. The microphone value range shall be configured such that the lower bound is the value for ambient noise, while the upper bound is a reasonably loud noise level that is achievable by the average user.
6. The system shall be able to correctly map the values in the configured range read from the microphone to the configured range of the motor speed.
7. A watchdog timer shall be configured and pet during regular intervals to ensure no software hangs.

3. Architecture Diagram



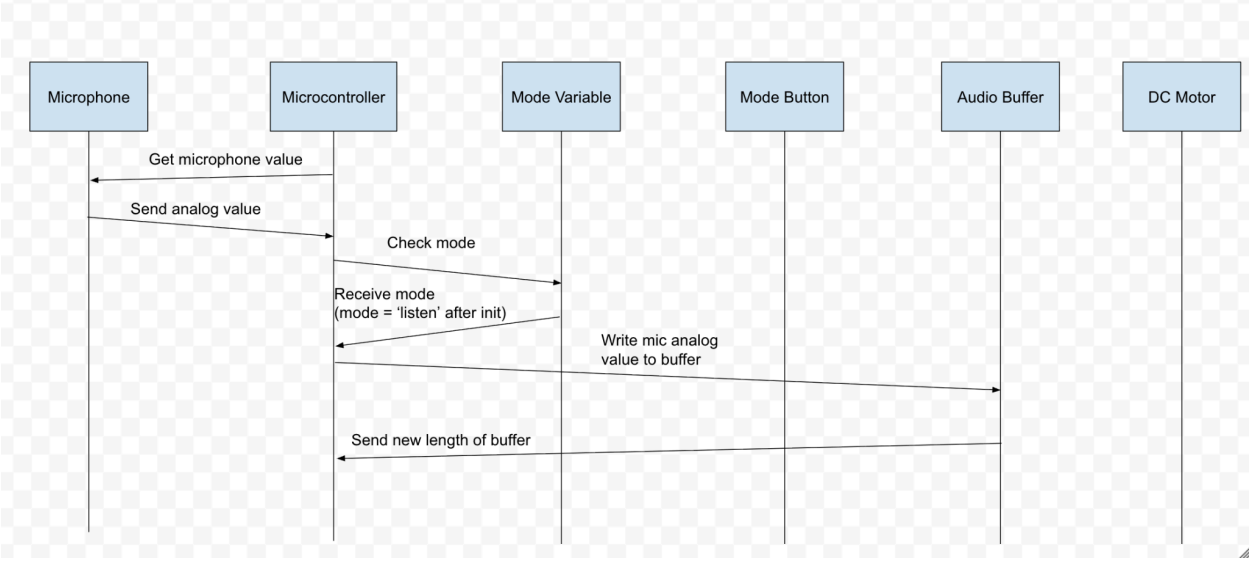
4. Use Case Scenarios + Sequence Diagrams

The mode variable is listed as a component, as it would usually be wired up as an LED but we didn't have the time to add it to our circuit.

Use Case 1: User provides audio after init

The system's default mode after initialization is listening. This case is when the user makes noise (like playing a song over the microphone) after the init state before any mode button presses have been made. It is also generalizable to the sequence whenever the system is in listen mode.

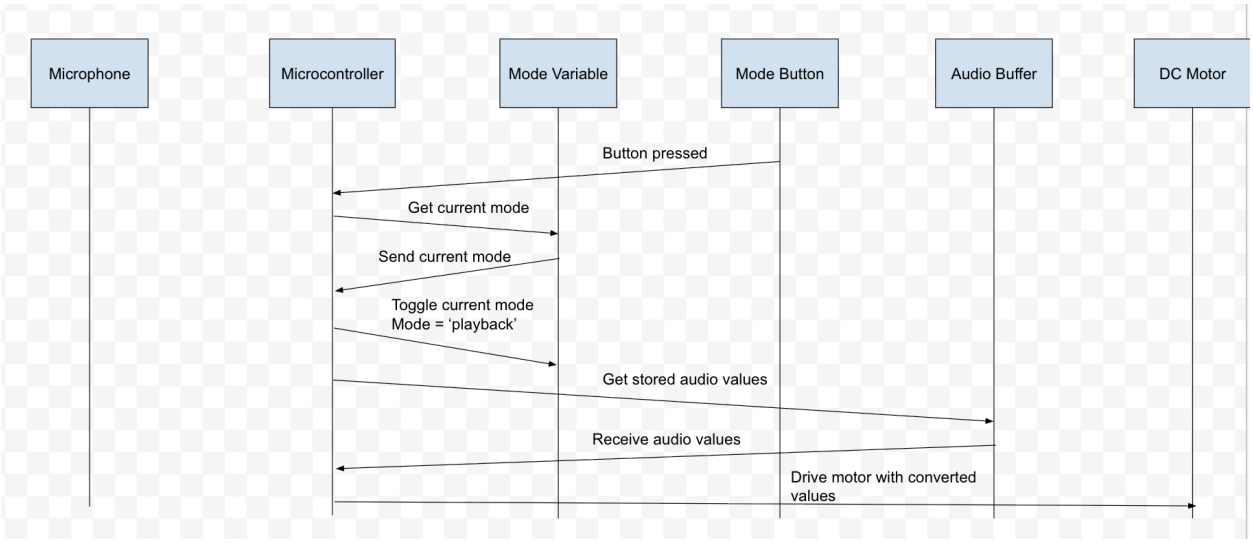
Sequence Diagram: Audio after init



Use Case 2: Switch mode button is pressed after init

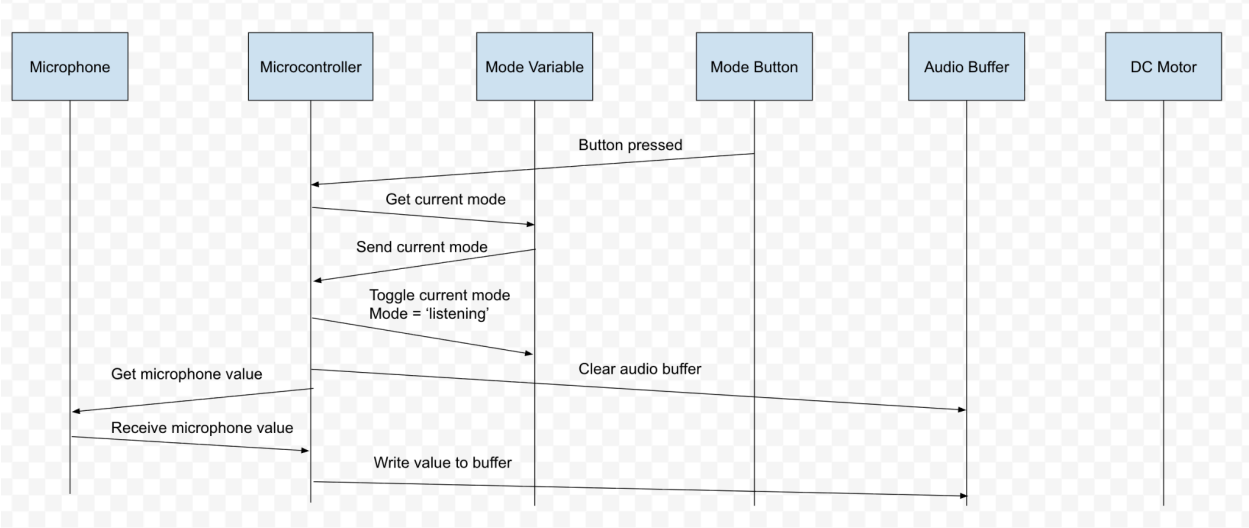
This is the case when the user wants to see the sculpture play back the audio it has sampled during the listening phase. They press the switch mode button, and then the motor should begin rotating to move the sculpture.

Sequence Diagram: Switch mode (listening -> playback)



Use Case 3: Switch mode button is pressed during playback mode

While the sculpture is in motion, the user wants to change the sculpture's movement pattern. They press the button to toggle the mode back to listening mode and begin to provide new input.



5. Revised Finite State Machine (FSM)

States

- **1: INIT** – hardware setup, calibration
- **2: LISTENING** - filling a buffer with audio samples
- **3: MOVEMENT** - use audio samples to drive motor with varying speeds using PWM

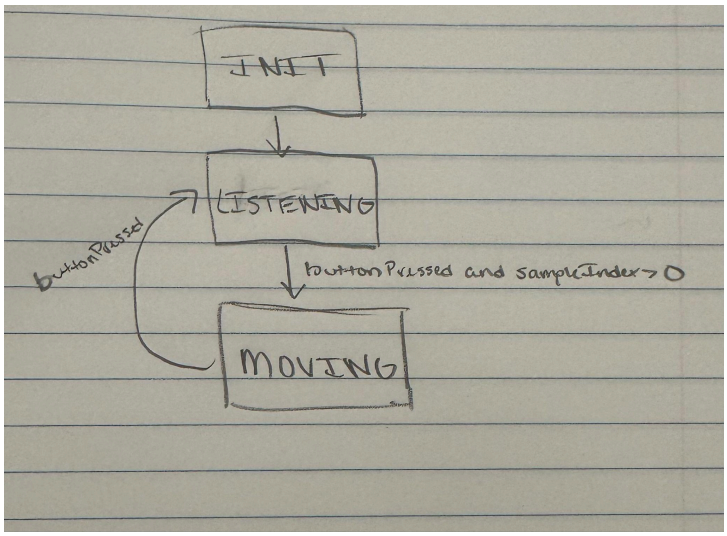
FSM Table / Variables

Variables:

1. currentState tracks the current state of the FSM
2. buttonPressed is a flag set when the button is pressed
3. sampleIndex is the index of the buffer to fill during listening
4. playbackIndex is the index of the buffer to use while moving
5. lastPlaybackTime is the last time the motor speed was updated

Transition	Guard	Output	FSM Variables
1-2	currentState == INIT	Serial.println("Entering LISTENING state");	currentState = LISTENING
2-3	buttonPressed && sampleIndex > 0	Serial.println("LISTENING -> MOVING");	currentState = MOVING playbackIndex = 0 lastPlaybackTime = millis()
3-2	buttonPressed	Serial.println("MOVING -> LISTENING");	currentState = LISTENING sampleIndex = 0 playbackIndex = 0 lastSampleTime = millis()

FSM Diagram



6. Traceability Matrix (FSM ↔ Requirements)

FSM State / Transition	Req 1	Req 2	Req 3	Req 4	Req 5	Req 6	Req 7	Notes
INIT → LISTENING							X	Initialization validates subsystems
LISTENING → MOVING	X							Audio sampling logic to movement
MOVING → LISTENING	X							Movement back to listening state
INIT							X	Init state
LISTENING					X		X	Listening state
MOVING		X	X	X		X	X	Moving state

7. Testing Overview

For our testing strategy, we started by focusing on the fundamentals of our FSM. We wanted to ensure that we had unit tests that tested all of the transitions of our FSM as well as all of the non-transitions.

The overall setup is that there is a helper function in test.ino to reset all of the system variables to allow us to set them ourselves for mocking purposes. Then, each transition is tested in its own helper function by setting the variables appropriately to bypass the guard. A key non-transition that we made sure to test was ensuring that the system stays in the listening state even if the button is pressed if no samples exist. We also ensured that the timing intervals were functioning properly. For example, we made sure that sampling happens only after the 10ms

interval passed. In generalized terms, we unit test key functionalities as well as all the valid and invalid FSM transitions.

In terms of integration testing, we relied pretty heavily on our printed outputs to ensure that the different pieces were communicating and interacting properly with each other. For instance, I made sure to print the mic values that were being read from the buffer in the moving state to ensure that these values were properly mapped to PWM values for the motor input. We can also tie this back to our different sequence diagrams:

Sequence 1 - We print that we are entering the listening state as well as all of the following samples from that state. This ensures both a smooth transition from the init state to the listening state as well as that audio input is working correctly.

Sequence 2 - When the button is pressed, we print both that the button was pressed and the associated state transition. So, if the user presses the button in the listening state, we would see a button pressed message and a state transition from listening to moving in the serial monitor. Furthermore, we can see that the motor begins moving upon pressing the button.

Sequence 3 - This is the same concept as above. The button press triggers a print statement and the state transitions from moving to listening, also outputted to the serial monitor. Furthermore, we can see that the motor stops moving when the button is pressed.

Finally, for system testing, we did this visually using our sculpture. We repeatedly varied audio inputs, pressed the button at different times, and ensured that the motion of our sculpture reflected that update. This also involved varying the string length accordingly to try to make a wav- like shape, and ensuring that the rotation of the motor preserved this shape as best as possible.

8. Safety & Liveness Requirements (LTL)

Variable Definitions:

- I = Init state
- L = Listening state
- M = Moving state
- MOTOR_SPIN = DC motor speed > 0
- R{1-7} = Requirement N
- MAP_OK(m, s) — the motor speed s correctly equals the mapped value from microphone m

Safety Properties

1. $\neg(I \wedge L) \wedge \neg(I \wedge M) \wedge \neg(L \wedge M)$
    a. The states of the system must be mutually exclusive. Concretely, the sculpture should not *never* be moving and listening at the same time

2. M → (MOTOR_SPIN ∧ R2 ∧ R4 ∧ R6)
    a. When the system is in the moving state, the motor speed must be greater than zero and requirements 2, 4, and 6 must be met.

Liveness Properties

1. G (BUTTON_PRESSED → ((L → X M) ∧ (M → X L)))
    a. Globally, when the switch mode button is pressed, if we were listening then next we would start moving, and if we were playing, then next we start listening
2. G (MIC_VALUE → MAP_OK(MIC_VALUE, MOTOR_SPEED))
    a. Whenever we read in a mic value in the correct range, it gets converted to the correct corresponding value in the motor speed range
3. G (M → G F BUFFER_AT_START)
    a. In the moving state, the audio buffer will eventually loop back to the beginning. This property ensures that our motor will always be spinning when in this state.

9. Environment Processes to Model

Button Press Environment

Type: Discrete, non-deterministic
Button press is instantaneous, it's either pressed or not pressed, sends the HIGH or LOW signal. It's non-deterministic because there is no set time or trigger that will cause the user to press it, it is completely random and up to the user's discretion.

Audio Input Environment

Type: Discrete, non-deterministic

The audio from the microphone (amplitude) is sampled discretely in 10ms intervals. Since the amplitude can remain constant or fluctuate randomly, it is non-deterministic.

Motor + Jig

Type: Hybrid, deterministic

Given PWM duty cycle, motor speed follows deterministic physics (within tolerances).

10. Code Deliverables & File Descriptions

main.ino - This houses all of the Arduino logic for the system. It includes the logic for the FSM, mic reading and buffer filling, and motor driving. For the requirements:

1. PWM - This is handled on line 160 in handleMovingState.

2. Watchdog - This is initialized on line 62 then pet on 64 and 114
3. ISR - Attached on line 58, with the ISR function on 66
4. Serial Communication - Both capstone aspects (Alex and Ashton) use serial communication. Line 77 reads serial input to drive the motor based on hand tracking data sent over from the python file test_comm.py
5. Timing - We utilize millis() throughout the code to get the current time and define sampling intervals. For example, line 87 gets the current time to pass to the different states which use it for sampling.

test.ino -This file houses the unit tests for main.ino. It defines key tests for testing FSM transitions and non-transitions as well as integral functions as defined above.

test.h - Simple header file which has the function definition for runAllTests

Alex's Capstone Files - All of the html and javascript files are for Alex's capstone, which builds a webpage that allows digital visualization of the mic readings.

- index.html :
    - Imports p5 javascript drawing library and arduino serial capabilities
    - Script tag that allows sketch.js file to refer to other modules of the code
- mockInput.js:
    - File that mocked arduino serial input in a soft, wave-like form to test browser visualizers without having to actually connect to arduino
    - Commented out because submitted code uses actual serial data
- readInput.js:
    - Decodes serial stream from arduino by connecting to serial port via web browser. Also contains a function to clean data. Satisfies serial communication with connectSerial function from lines 21-34.
- sketch.js:
    - Sets up the webpage, creates canvas, "connect to arduino" button, and handles the draw mode switching with keystrokes.
- Modes folder
    - bar.js:
        - Gets 200 (param configurable) audio samples -> equally buckets audio values -> computes average to get bar height, then displays them with a blue to purple to pink gradient depending on height.
    - circle.js:
        - Draws a base circle then draws chaotic circles using noisy vertices and radiuses. Visualizer inspired by vintage oscilloscopes.
    - particles.js:
        - Draws particles, particle amount depends on audio value from mic, louder = more particles.

Ashton's Capstone Files - test_comm.py handles hand tracking using cv2 library functions and computes a speed of movement. It also handles sending these speeds back to the Arduino using serial communication which are mapped to pwm values to drive the motor.

11. How to Run the Unit Tests

To run the unit tests, all you have to do is set the macro RUN_FSM_TESTS to 1 in the main.ino file.

12. Reflection

This project forced a shift from treating hardware as a collection of parts to designing it as a safety-critical, real-time system. Early iterations focused too heavily on making the mechanism "move," but the later revisions emphasized correctness under all conditions. We did have to constrain our threshold. Defining a clear FSM was particularly valuable: it exposed hidden assumptions (for example, what "no audio" actually means) and made safety behavior explicit rather than implicit in code.

Testing was another major learning point. Writing desktop unit tests with mocked Arduino functions initially felt excessive, but it paid off by catching edge cases that would have been difficult to diagnose on-device. Separating audio processing, motor control, and system supervision improved both testability and reasoning about failures.

Overall, the project reinforced the importance of designing for worst-case behavior, not just ideal demos. The final system is not only expressive as a kinetic sculpture, but also robust, predictable, and defensible as an embedded real-time design.

13. Appendix

Peer Review 1:
https://docs.google.com/spreadsheets/d/1dx05eKIIKssd5AwFTRyEb78A7vvUs4axq_dIrJ4J3NM/edit?gid=0#gid=0

Peer Review 2:
https://docs.google.com/spreadsheets/d/1fTAiOE8A687OHoFq-vfqv_WFEAND8TZ6_jPZeeXmf_A/edit?gid=0#gid=0

Peer Review 3:
https://docs.google.com/spreadsheets/d/1H3Ju6hIuVTWQYRMFF6BE890NuG3xlQkxiQ3_l-vnePc/edit?gid=0#gid=0

Peer Review 4:
https://docs.google.com/spreadsheets/d/1nmLDGC0A5s9tidIq7Ba0KML69VKmf5mjj2HvUWMURZo/edit?gid=0#gid=0