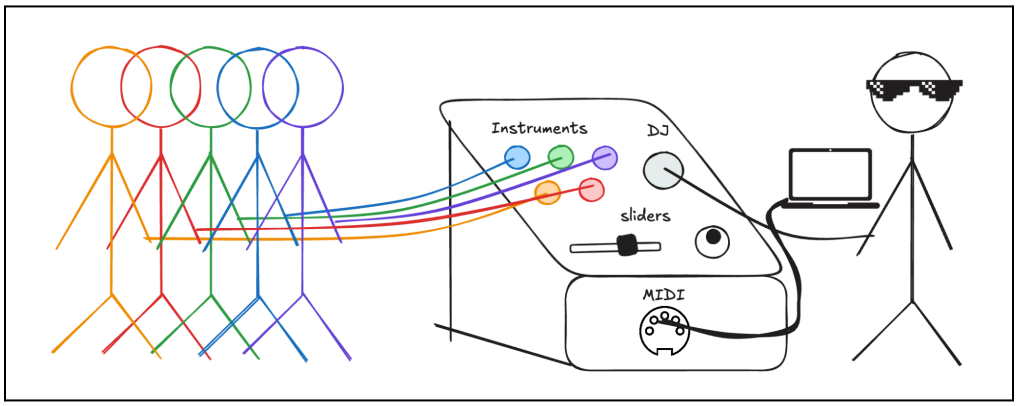
Human DJ: Final Report

Eva Schiller Ishika Tulsian Komron Aripov Rohit Panse



Overview

About: The Human DJ is an interactive musical instrument powered entirely by human touch. Participants, each connected to the device via alligator clips, function as "keys" in a living, collaborative synthesizer. The DJ, who's also wired up, produces music by interacting with people – it could be high fives, taps on the shoulder, fist bumps, you name it. When individuals touch each other, they complete an electrical circuit, allowing the system to detect these interactions using an Arduino through a measured change in voltage. The system acts as a MIDI instrument, interfacing with music software and speakers to allow the DJ to curate the sound. The sounds can be chosen in-software based on a selection of notes on the octave scale, music samples, instruments, pre-recorded audio loops or more. The experience transforms human connection into music, creating a dynamic and fun environment for performance or play.

Assumptions: participants don't require technical expertise to interact with the device, however, the DJ needs to have some background in interfacing with music software (Logic Pro / Garageband). This is supposed to be an audio-sensory experience - users should be comfortable with physical interaction, and wearing provided bracelets to maintain electrical contact with the device. Capstone only: when using the onboard player, no experience with music software is required from the DJ.

Capstone: Onboard Player Video Demo! **humandj_capstone.mov**

Requirements

Inputs:
The `touch_states` binary vector with the states of which participant is touching the DJ
The `fader_states` vector holds the MIDI volume and pitch states (0 to 1023).

Parameters:
The `touchThresholds` integer vector with calibrated threshold voltage values (0 to 1023).

RO-1: At the start of operation: The voltage thresholds shall be calibrated for each person by touching the DJ and stored in the `touchThresholds` vector.

A. The MIDI device shall be initialized and registered with the DAW.

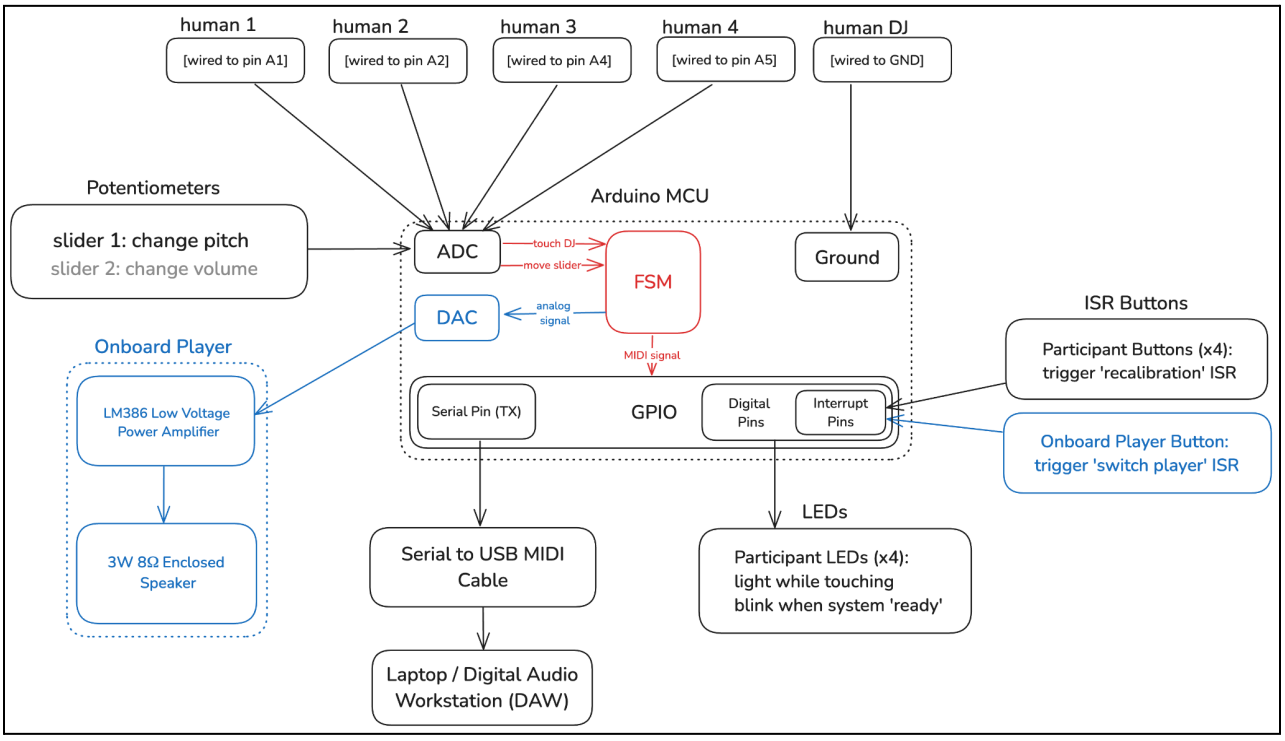
B. The `touch_states` vector shall be initialized with zero values.
C. The pitch variable shall be initialized to the current value of its respective potentiometer.
D. When all components are initialized, the program shall indicate readiness via LED.

RO-2: Any time one of the participants is touching the DJ:
A. The participant's touch state shall be registered in the `touch_states` vector
B. The note associated with the participant shall be sent to the DAW via MIDI.
C. Once the signal has been sent, the program shall continue to check for updates to touch_states or fader_states.

RO-3: Any time one of the participants who was previously touching the DJ stops touching the DJ:
A. The participant's new touch state shall be registered in the `touch_states` vector.
B. The note associated with the participant shall stop being played on the DAW.
C. Once the signal has been sent, the program shall continue to check for updates to touch_states or fader_states.

RO-4: Any time any potentiometer value is changed, the respective variable, either pitch or volume, shall be mapped and modified according to the change.

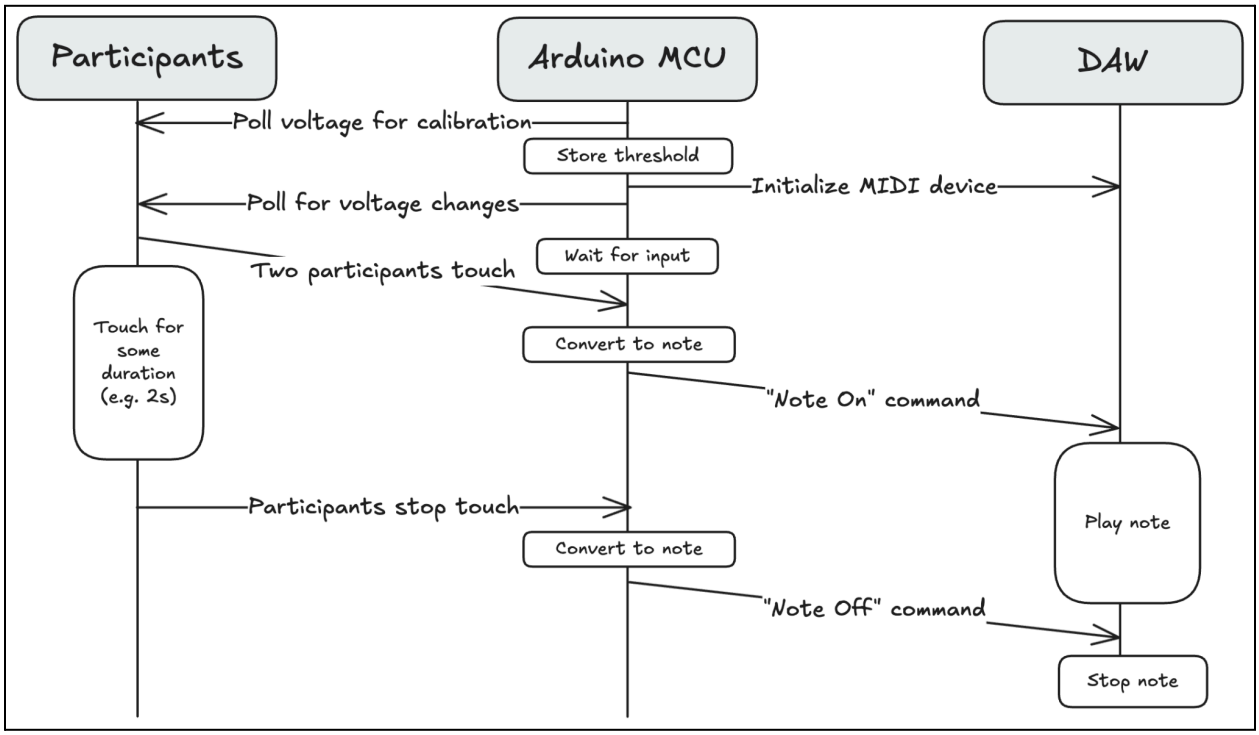
Architecture diagram



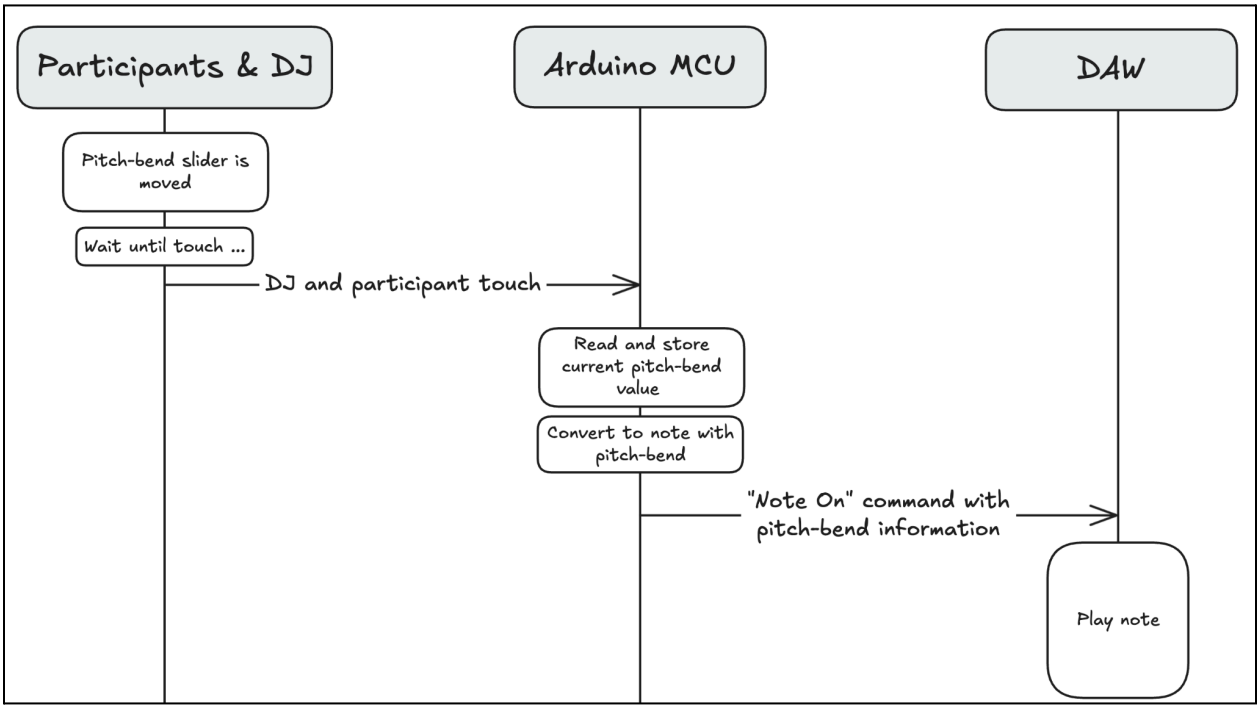
Legend: Red is the software component and blue are the capstone additions by Eva and Komron

Sequence diagrams

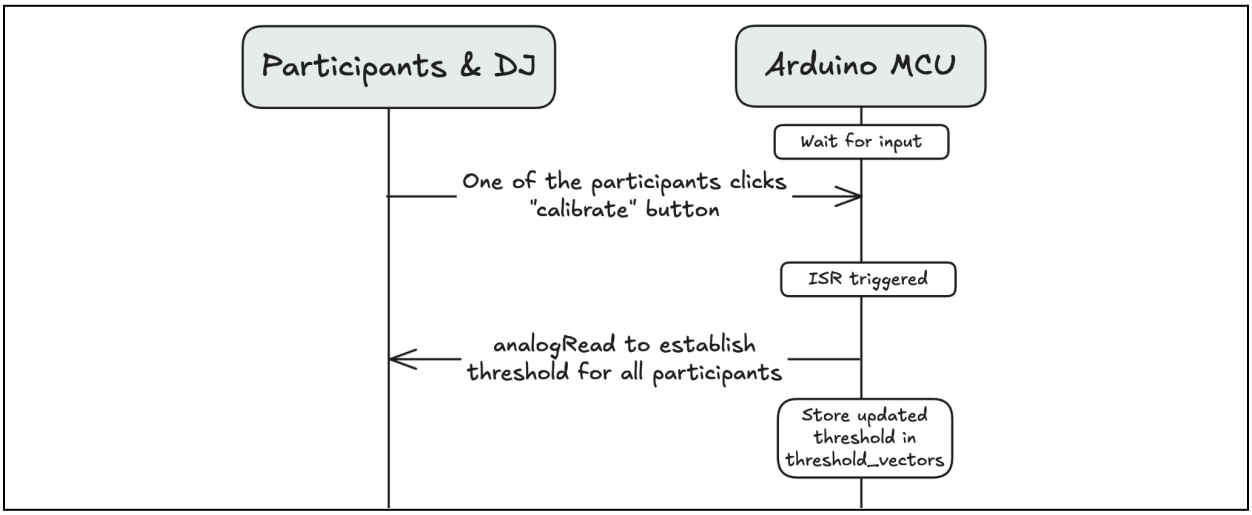
Scenario 1: When one participant touches the DJ, this should cause a note to play on the DAW, until the touch is stopped, and the note should stop playing as well.



Scenario 2: When a participant or DJ moves the pitch-bend potentiometer, the pitch-bend of all the MIDI signals needs to be adjusted.



Scenario 3: When one participant taps the “Calibrate” button, all of the voltage thresholds for all of the participants need to be adjusted.



Finite State Machine

We assume that our configuration has 4 humans plus a DJ, however this code is easily generalizable for any number of humans participating.

Input name	**Description**
touch_states	*Binary vector indicating whether each human participant is currently touched by DJ. The length of the vector is the number of participants. Each index is 1 if that human is being touched, and 0 if not.*
fader_states	*Integer vector holding the values for pitch and volume, 0 to 1023.*

Variable name	**Description**
midi_states	*Binary vector indicating whether MIDI is currently playing a note for each human participant. The length of the vector is the number of participants. Each index is 1 if that human is playing a note, and 0 if not.*
signal_sent	*Boolean value to indicate that the MIDI signal has been sent.*

The **output** to our FSM is the MIDI signals that interface with music software, which is set by calling the helper function `send_signal`. Note that this also sets the LEDs to turn on accordingly.

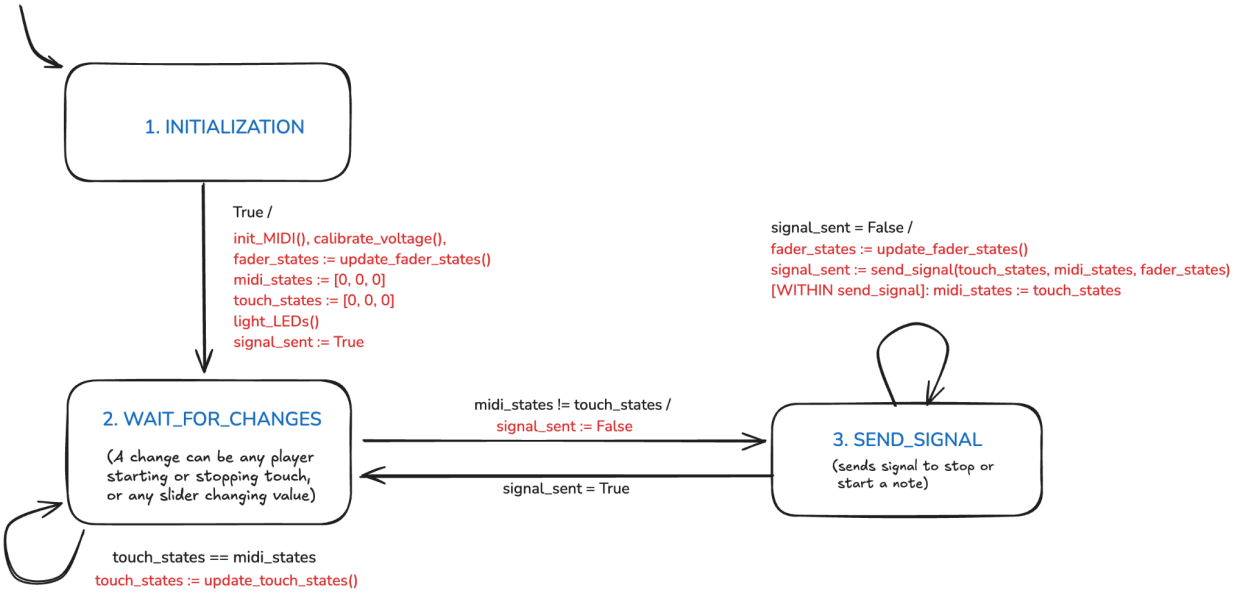
```
bool send_signal(int[] touch_states, int[] midi_states, int[]
fader_states)
```

behavior: *Function to send MIDI signal to send the play / stopPlay note for corresponding participant based on the touch_states and midi_states. Once the corresponding signal is sent, it returns true which signal_sent is then set equal to.*

return: *signal_sent (true)*

Other Helper function descriptions

`void init_MIDI()`
Initializes MIDI software *returns: none*
`void calibrate_voltage()`
Calculates an average voltage read for each human participant. *returns: voltage_thresholds*
`void light_LEDs()`
Indicates that the device is ready for input by lighting LED(s). *returns: none*
`void update_fader_states(int* fader_states)`
Updates the pitch and volume information in the fader states to the values read from the potentiometer sliders. *returns: fader_states*
`void update_touch_states(int* touch_states)`
Updates the touch states based on who the DJ is currently touching / or not touching. *returns: touch_states*
`bool send_signal(int[] touch_states, int[] midi_states, int[] fader_states)`
Function to send MIDI signal to send the play / stopPlay note for corresponding participant based on the touch_states and midi_states. Once the corresponding signal is sent, it returns true which signal_sent is then set equal to. return: *signal_sent (true)*



Transition	Guard	Explanation	Output	Variables
1-2	True	After we initialize all variables, we wait for changes in system to respond	init_MIDI() calibrate_voltage() light_LEDs()	fader_states := update_fader_states() touch_states := [0,0,0] midi_states := [0,0,0] signal_sent := False
2-2	touch_states == midi_states	Continue to wait for a change if no change in variables is detected - idling.	-	touch_states := update_touch_states()
2-3	touch_states != midi_states	New human touch has been detected, moves to sending signal to start or end playing a note.	-	signal_sent := False
3-2	signal_sent	Signal sent to MIDI, starts to continuously play note and goes back to idle waiting state	-	-
3-3	¬signal_sent	Read in new fader state values. Send signal to MIDI to start playing note	send_signal(...)	fader_states := update_fader_states() signal_sent := send_signal(touch_states, midi_states, fader_states) *[within send_signal]: midi_states := touch_states*

Traceability matrix

	R-1 A	R-1 B	R-1 C	R-1 D	R-2 A	R-2 B	R-2 C	R-3 A	R-3 B	R-3 C	R-4 A
STATE											
1	X	X	X	X							
2					X			X			X
3						X	X		X	X	
TRANSITION											
1-2	X	X	X	X							
2-2					X			X			X
2-3						X			X		
3-2							X			X	
3-3						X			X		

Overview of Testing Approach

Our testing suite consists of two major components: unit testing for individual functions and integration testing through FSM transitions. Lastly, we performed overall acceptance / system testing through exploratory use, such as monitoring LED outputs, MIDI outputs on the DAW, and Arduino's serial plotter for thresholding.

For unit testing, we made sure to test all the functions that could be mocked-out in our four primary libraries: music, touch_sense, sliders, and onboard_player. Specifically, we run six unit tests, `test_send_signal`, `test_send_onboard_note`, `test_update_fader_states`, `test_calibrate_voltage`, and `test_update_touch_states`, corresponding to following key functions:

1. In music.ino, we test `send_signal`. Here, we mocked out the MIDI functionality for sending notes, stopping notes, and sending pitch information.
2. In onboard_player.ino, we test `send_onboard_note`. Here, we mocked out the analogWave functionality for playing notes, stopping notes, and modifying the pitch information.
3. In sliders.ino, we test `update_fader_states`. Here, we mocked out analogRead to get deterministic outputs.
4. In touch_sense.ino, we test `calibrate_voltage` and `update_touch_states`. We again mocked out analogRead for deterministic outputs.

For FSM testing, we closely followed Lab 5's testing structure. This involved creating a `testTransition` function that would, given inputs, start state, start variables, end state and end variables, check if the FSM outputs satisfactory values. To ensure that our FSM was well-tested, we wanted to have at least one test per transition. In the end, we run this function over a larger suite of tests that cover all transitions, with many transitions covered multiple times. Furthermore, we wrote a number of functions to help document test failures or regressions, such as `printArray`, and `s2str` to print out states.

We test FSM transitions as follows:

State In	State Out	Inputs, Vars In, Vars Out	Description
sINIT	sWAIT_FOR_CHANGE	[…]	initialization always finishes
sWAIT_FOR_CHANGE	sWAIT_FOR_CHANGE	[…]	if no touch state changes, keeps waiting
sWAIT_FOR_CHANGE	sSEND_SIGNAL	[…]	if participant stops touching, sends signal
sWAIT_FOR_CHANGE	sSEND_SIGNAL	[…]	if participant starts touching, sends signal
…	…	…	…

For acceptance and system testing, we highlight the importance of the Serial plotter to verify our expected outcome. As shown in the figure below, we graph a moving average of each participant's voltage over time, measured through analogRead. We also graph each participant's calibrated threshold. Since the interaction between these two values is the deciding factor for detecting a

touch, we use this plot to visually evaluate whether our sensing process is working properly. We also combine the plot with observation of the LEDs and DAW. This was particularly useful when we tested out the scenarios detailed in our sequence diagrams. For example, for scenario 1 we confirm that (a) a participant's voltage does dip below threshold during touch, (b) the corresponding sound plays through the DAW for the duration of the dip, and (c) the participant's corresponding LED lights up for the duration of the dip. We also confirm that no spurious signals are detected, meaning (a) no participant's measurements go below threshold unexpectedly, (b) no unexpected sound plays through the DAW, and (c) no LED lights up without corresponding touch. For scenario 2, we checked that there was an audible change in pitch when the slider was moved. For scenario 3, we were able to confirm through the Serial Plotter that the threshold does in fact update (recalibrate) when a user presses the button. Having the sequence diagrams helped us test out critical functionality of our project and ensure that the system overall was working as expected.

Our criteria for coverage was assessing whether all individual FSM functions, states, and transitions are covered by each phase of our testing. In unit testing, we have tests for all individual functions named in our FSM. In integration testing, we have multiple cases for every possible transition between states. We have further explored all states of the FSM during system and acceptance testing, while observing Serial Plotter, DAW output, and LEDs. We also have used multiple participants, across multiple days and settings, and developed an idea of how our device responds to changes in participants and the environment. Thus, since our three stages of testing have indeed *covered all functions*, *states, and transitions*, we deem that our criteria has been sufficiently satisfied by our testing protocol.

Safety and Liveness Requirements

Safety Requirements:

Never plays unexpected sounds: If no new touch is detected, the system will never leave the waiting stage; that is, it will never attempt to send a signal without a change to touch_states to trigger it.

- **G**(touch_states == midi_states) → **G**(sWAIT_FOR_CHANGE)

Always returns to waiting once a signal is sent: Once signal_sent is true, the system immediately returns to sWAIT_FOR_CHANGE to continue waiting for new changes to touch_states.

- **G**(signal_sent → **X**(sWAIT_FOR_CHANGE))

Liveness Requirements

Eventually begins touch detection: The system will eventually initialize and reach the stage where it waits for new changes to touch_states.

- **F**(sWAIT_FOR_CHANGE)

System will update midi_states to match touch_states: The system will always modify the midi_states variable to reflect changes to the touch_states input. In other words, it will not 'miss' a change.

- **GF**(touch_states == midi_states)

If there is ever a detected touch, the system will then eventually send a signal: If a mismatch between the midi_states variable and the touch_states input occurs, the system will eventually play a sound in response. In other words, if a touch is detected, the system will send a signal!

- **F**(touch_states != midi_states) → **XF**(sSEND_SIGNAL)

Environment Processes

Environment Process	Description	Type	Determinism	Reasoning for Type and Determinism
Button Presses	Recalibrates a participant or switches to the onboard player (Capstone)	Discrete	Non-Deterministic	Discrete because the button is either being pressed or not being pressed. Non-Deterministic because it can occur at an arbitrary time.
Touch Inputs	Changes touch states based on if participants are touching human DJ	Discrete	Non-Deterministic	Discrete because ADC is sampled at discrete time intervals. Non-Deterministic because touch inputs can occur at an arbitrary time.
Fader Positions	Fader input which is converted with ADC	Hybrid	Non-Deterministic	Hybrid because continuous signals are mapped to discrete values. Non-Deterministic because fader changes can occur at an arbitrary

				time.

File Descriptions

File Name	Description
buttons.ino	Defines a button module that handles the ISR triggered by button presses. **(ISR requirement: line 5, 10)**
humandj.h	Config file which defines constants, variables, types, and functions for the project.
humandj.ino	Implements the FSM, or the main functionality of the project. It manages state transitions, MIDI signals, and onboard playback.
humandj_tests.ino	Provides tests for the FSM. Helper functions are included which help print state transitions. Unit tests are also included to test signal sending, touch state updates, and note playback functionality.
music.ino	Initializes a serial MIDI interface to communicate with a digital audio workstation (DAW). It also maps the states that the slider updates to corresponding MIDI updates, and it updates LED states for visual feedback when a touch state is active. Tests are included to validate starting and stopping of notes and pitch bending functionality. (**Serial communication requirement: lines 19, 59-60, 67-68, 75-76, etc. {anywhere there is 'midi.___'})**
onboard_player.ino	Capstone (Komron and Eva): Converts MIDI notes into frequencies, playing them directly on the device without needing a MIDI connection. It has functions that initialize the onboard player, play and stop notes based on touch states, and apply the pitch bending effects based on the slider values.
sketch.yaml	Specifies the arduino board name used by the Arduino IDE to identify the target board to upload and compile code to.
sliders.ino	Handles slider inputs like pitch and volume by scaling ADC values read from a potentiometer and updating corresponding states. Tests to validate functionality which simulate fixed slider inputs are included. (**ADC requirement: line 40-41)**
touch_sense.ino	Detects touch from multiple participants, updating respective touch states. It also has calibration functionality which establishes a baseline voltage threshold, which is used when code is live and being used to detect touch. Tests are included to validate functionality
watchdog.ino	Configures a watchdog timer that gets triggered when the system is unresponsive. It Includes functions to initialize the watchdog, refresh the watchdog to prevent triggers, and handle interrupts triggered by the watchdog. (**Watchdog timer requirement: function at line 12)**

Procedure to Run Tests

For both unit testing and integration testing, we have everything gated behind the `#TESTING` macro within the `humandj.h` header file. When enabled, the setup function initializes and runs the entire testing suite, outputting to Arduino IDE's serial prompt. The main FSM loop is disabled when in testing mode.

Additionally, ensure that the necessary MIDI interface library, Control_Surface, is installed to ensure the code compiles.

Reflection

Our project satisfies the following requirements:

- **ADC:** a slider module for pitch bending and the touch sense module for voltage readout. All of these analog inputs use a multiplexed ADC to process.
- **Watchdog timer:** a WTD is built-in into the Arduino R4 WiFi and will be enabled to add a layer of additional fault tolerance.
- **Interrupt service routine:** We are using button pushes to trigger an ISR to re-calibrate voltage thresholds for each participant. This will allow us to finetune thresholds without having to reset the code running every time.
- **Serial communication:** some communication to output MIDI, which can be done through Serial (also known as MIDI over Serial)

Our main goal was to be able to demo a *working* live human DJ set up - which we accomplished! For a completely functional demo we met the following goals: accurate touch sensing between the DJ and participants, mapping the correct corresponding MIDI output signals to each participant so that it can interface with Logic Pro software and have the hardware module respond correctly to inputs (LEDs flashing at touch detection, sliders pitch bending etc).

While developing this project we were met with several challenges, specifically:

- **Touch sensing sensitivity to noise and interference:** Since our touch sensing relies on the DAC pin measuring a drop in voltage (below a calibrated threshold), we had to process the signal in order to deal with noise and interference issues coming from the pin readings. Things like using a Magsafe charger would greatly disrupt the signal and were difficult to debug. Ultimately, to deal with this challenge we decided to improve the way we calibrated touch thresholds by using a moving average or variance detection method (for noisier environments).
- **Finding a MIDI library that compiled on arduino, no package manager:** finding a suitable MID library that supported our requirements and could successfully compile on arduino was a challenge. We ultimately found Control_Surface which allowed us to interface with Logic Pro through midi signals. Arduino also lacks a centralized package manager, so each of us had to locally download the correct library manually.
- **Arduino only has 6 ADC pins:** We had to use 4 of these pins for each of the participants. 1 for the pitch bend slider and another for the onboard player. This meant that we couldn't have a volume controller as we originally planned, limiting the number of components we could add on that required ADC.
- **Soldering the electrical components**: assembling the physical circuitry required extensive soldering to ensure stable and reliable connections between components. However, the densely packed circuit introduced challenges with interference between closely spaced wires. For instance, signals from one touch circuit would sometimes couple into another due to capacitive coupling, leading to false triggers or erratic behavior. To mitigate this, we carefully organized the wiring layout to minimize crosstalk and ensured proper grounding across the circuit. We also used shielded wires where possible (like the use of our grounding bracelets as metal contacts). These changes reduced interference and improved the reliability of the overall system.
- **Onboard Player (Capstone):** We (Eva and Komron) also incorporated onboard player functionality with the Human DJ, allowing the device to be a standalone instrument. We

produced a library, onboard_player.ino, to manage the audio output including note-playing and pitch bend, and an ISR to toggle between output devices. On the hardware end, we integrated an amplifier and speaker with our existing design, and added an additional button for toggling between players. Ultimately, the biggest challenge was determining how to translate our music interface from sending MIDI signals to sending analog waves directly to the DAC, including a good mapping from notes to frequencies. One limitation of the onboard player is that it cannot play multiple notes concurrently – instead, it uses a priority system where the most recently touched participant 'wins' the note. However, we are happy with the outcome of the onboard player, as it allows anyone to experiment with the device, whether they have access to a DAW or not!

Overall, we're really happy with what we've achieved in Human DJ. We wanted it to be a fun way to create music through interacting with people, and managed to get a really positive response during demo day! :)

Appendix

Transition or state #	Defect	fix?	Justification if not fix
	Going off of lecture slides 10/7, guards should be predicates on inputs and actions should be settings on outputs. It looks like you are using the same set of states and variables?	☐	We have one variable that is included in the guard, which is essential for simplicity. Otherwise, we would have a redundant input (past touch_states) identical to midi_states.
	Missing definition of inputs, outputs, variables	☑	
1-2	Could simplify by getting rid of the is_ready variable and initialize the other variables in the transition	☑	
3-3	What does send_signal do other than return a boolean? Curious to see when send_signal() updates signal_sent so the self loop ends.	☑	
2-2	Why would fader_state be updating? Wouldn't the state of the fader be consistent when waiting for change?	☑	

Transition or state #	Defect	fix?	Justification if not fix
1-2	Setting is_ready to being true isn't dependent on verification of init_MIDI(), calibrate_voltage(), and update_fader_states() having correct initialization	☑	
3-3	If no signal is ever sent, the system be stuck self-looping at state 3 without being able to go back to state 2	☐	This is intended behavior! However, send_signal can only return true, so the looping will not happen.

2-3	Given that touch_states and midi_states are both arrays of 3 values, is there potential that noise affecting one of these values causes you to jump to state 3 when you actually prefer to stay waiting for a change?	☐	Noise will not be an issue here.
2-2	Is it necessary to call update_fader_states() from 2-2 or is it possible to just make a call to update_fader_states() in state 3 before using it? It seems that there might be some unnecessary calls to it in states 1 and 2	☑	

Transition or state #	**Defect**	**fix?**	**Justification if not fix**
Transition 1 (INITIALIZATION → WAIT_FOR_CHANGES)	No way to return to initialization if a critical error occurs. Single point of failure with just is_ready flag check. Missing handling for is_ready = False case if initialization fails mid-process.	☐	We do not believe error-handling is required in FSM. Re-calibration can be done via ISR.
State 2 (WAIT_FOR_CHANGES)	Could potentially get stuck in an infinite loop if touch_states and midi_states remain equal forever. No handling if update_touch_states() or update_fader_states() fails. This could also create a state where changes are not detected due to stale state data.	☐	If touch_states and midi_states remain equal because no touches are detected, staying in State 2 is desired behavior. If one of the functions hangs, this will be detected by a watchdog timer.
Transition 2 (WAIT_FOR_CHANGES → SEND_SIGNAL)	No validation that the states comparison (midi_states != touch_states) is valid before transition. Missing bounds checking on the state arrays before comparison. Missing verification that SEND_SIGNAL state is ready to receive changes. No handling for states changing during transition.	☐	State array comparison is unit tested, and specific error checking seems unnecessary to include in FSM. SEND_SIGNAL state is always ready. The states will not change unless triggered by transition, per FSM logic.
State 3 (SEND_SIGNAL)	No failure path if send_signal() function fails. Could get stuck if signal_sent never becomes True. Missing timeout mechanism for send operations.	☐	Watchdog will catch software hangs, so a timeout mechanism seems redundant.

Transition 3 (SEND_SIGNAL → WAIT_FOR_CHANGES)	No validation on the midi_states := touch_states assignment. Could create an infinite loop between SEND_SIGNAL and WAIT_FOR_CHANGES states if signal sending repeatedly fails.	☑	

Transition or state #	**Defect**	**fix?**	**Justification if not fix**
1-2	Are there any inputs/outputs?	☑	
2, 3	Maybe mention what the outputs of these are on the user side in addition to how you're changing the variables	☑	
1	Does signal_sent need to be initialized here as well?	☑	
2	Do update_fader_states() and update_touch_states() need any input from hardware? (not sure if this is necessary tho)	☐	Yes, but we abstracted this away in update functions.
	I think if you wanted to extend your FSM further, you could have a little sidebar with descriptions of what each function's inputs/outputs are, like we did in class. Also if you have any hardware that interacts / changes touch_states or fader_states, it might be helpful to explain how that fits in (if you think it does fit into the FSM). It's a really solid FSM, though! Very readable and seems consistent and comprehensive from what I can tell.	☑	