Arduino Hero Manual

Prepared for

Alan Lee, Gilbert Lopez, & Jeff Yamauchi

By

Molly McHenry

In association with:

Patrick Ortiz, Serena Pulopot, Nick Bottone, & David Fryd

May 2, 2024

Contents

Overview

Arduino Hero is an implementation of Guitar Hero using Arduino MCUs. It features a custom LED matrix and LCD screen controlled by one MKR1000, a game controller with another MKR1000, and a music player enabled by an Arduino Uno with a music shield. To play Arduino Hero players strum the controller bar while pressing the colored buttons on the controller in time with the music and the corresponding notes moving down the LED matrix. This prototype is constructed for right-handed users with typical color-sightedness and hearing.

Requirements

R1: On system start, a track select screen shall be displayed on the LCD screen where the user can select the song for the game.

> **R1-A:** When the user presses the UP button, the displayed/selected song shall be changed to the song above the currently selected song in the menu.
>
> **R1-B:** Whichever song is currently highlighted when the START button is pressed shall be selected for the game.

R2: Once the START button is pressed, a three-second countdown shall be displayed on the LCD screen.

R3: After the countdown, the game shall start. The player's combo and score shall both be initialized to 0 and displayed on the LCD screen. The MP3 Shield shall play the selected song. The LED matrix shall display notes by lighting the LEDs corresponding to the approaching notes in the song.

> **R3-A:** The LED board shall have 5 different note lanes (rows of LEDs), each corresponding to a button on the guitar controller.
>
> **R3-B:** The last row on the LED board shall be lit up white to indicate the scoring zone[1].

[1] Note that this scoring zone is interchangeably referred to as the "strike zone."

R4: Notes displayed[2] on the LED matrix shall be displayed starting before their occurrence in the song. As a note occurrence approaches in the song, it shall move down the LED matrix until it is located on the bottom LED row when audible.

> **R4-A:** When the note is upcoming, it shall be displayed higher than the bottom row of the LED matrix.
>
> **R4-B:** When the note is playing (audible) in the song, the note shall be displayed on the button row of the LED matrix.

R5: A note shall be computed as “played” when the button corresponding to the note is held down when the strum bar is strummed on the controller. Each time a note is played, the software shall check for correctness.

> **R5-A:** When a note is displayed on the bottom row of the LED matrix, playing the corresponding notes on the guitar controller counts as a correctly played note.
>
> **R5-B:** Each note shall have a “grace period” during which it can still be playing as correct. The grace period is one beat before or after the note is displayed on the bottom row of the LED matrix.

R6: A player shall have a score that shall be incremented by one when a note is correctly played on the controller. Upon start, this score shall be initialized to zero and displayed on the LCD screen, where it will be displayed throughout gameplay and incremented by one for each correctly played note.

R7: A player’s current combo (the running count of consecutive notes correct) shall be displayed on the LCD screen and incremented during gameplay. Each correctly played note shall increment the current combo by 1. It shall be reset to 0 if a player plays an incorrect note.

---

[2] Each note is represented by a lit LED in the row corresponding to that note on the LED matrix. A “displayed” note is a note that currently has a lit LED on the board corresponding to it.

R7-A: If a correct and incorrect note are played concurrently, the combo counter shall be reset to 0 and no correct notes shall be computed.

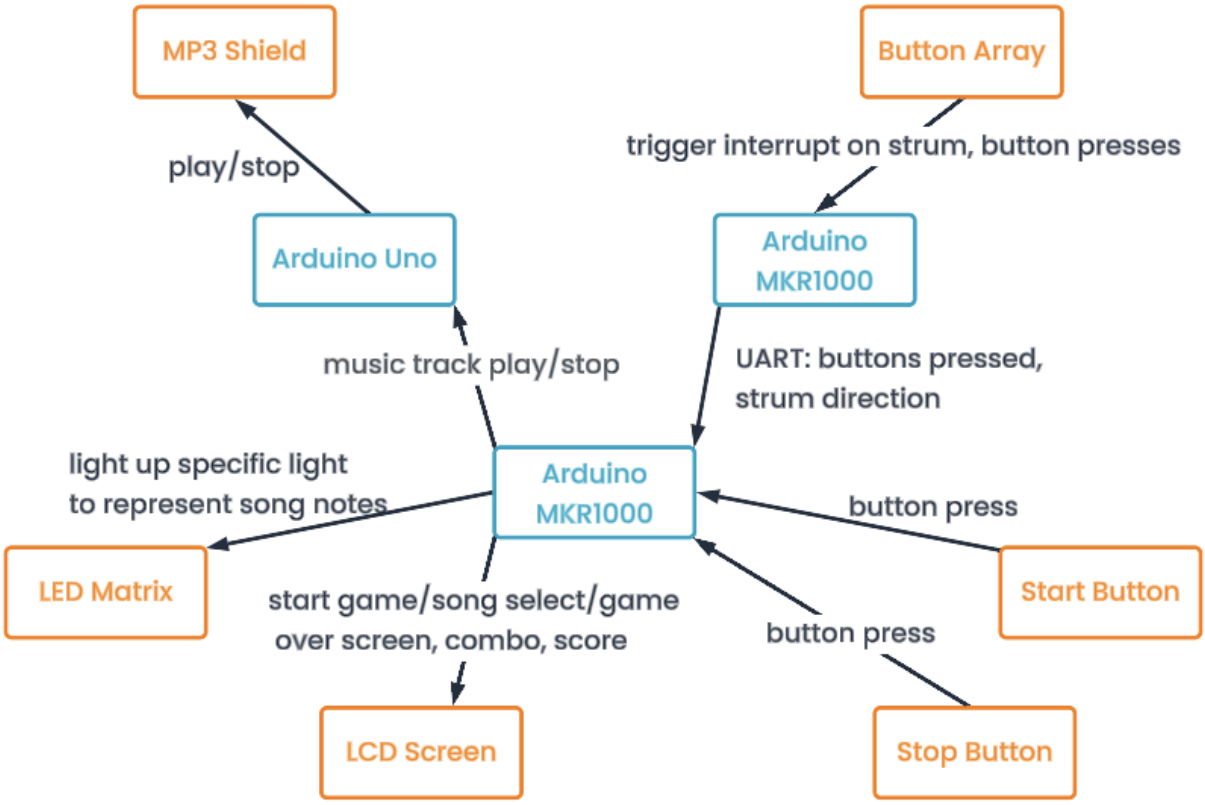
R8: When the track ends, a game-over screen shall be displayed on the LCD with (1) a game-over message, (2) the player score, and (3) the longest combo from the gameplay.

R8-A: The game-over screen shall be displayed for a minimum of three seconds.

R9: When the START button is pressed during the game-over screen (and three seconds have elapsed since the game-over screen was first displayed), the start screen shall be shown on the LCD screen.

R10: A watchdog timer shall bite when the LEDs are not advanced for multiple beats in a row, rebooting the system to the start menu.

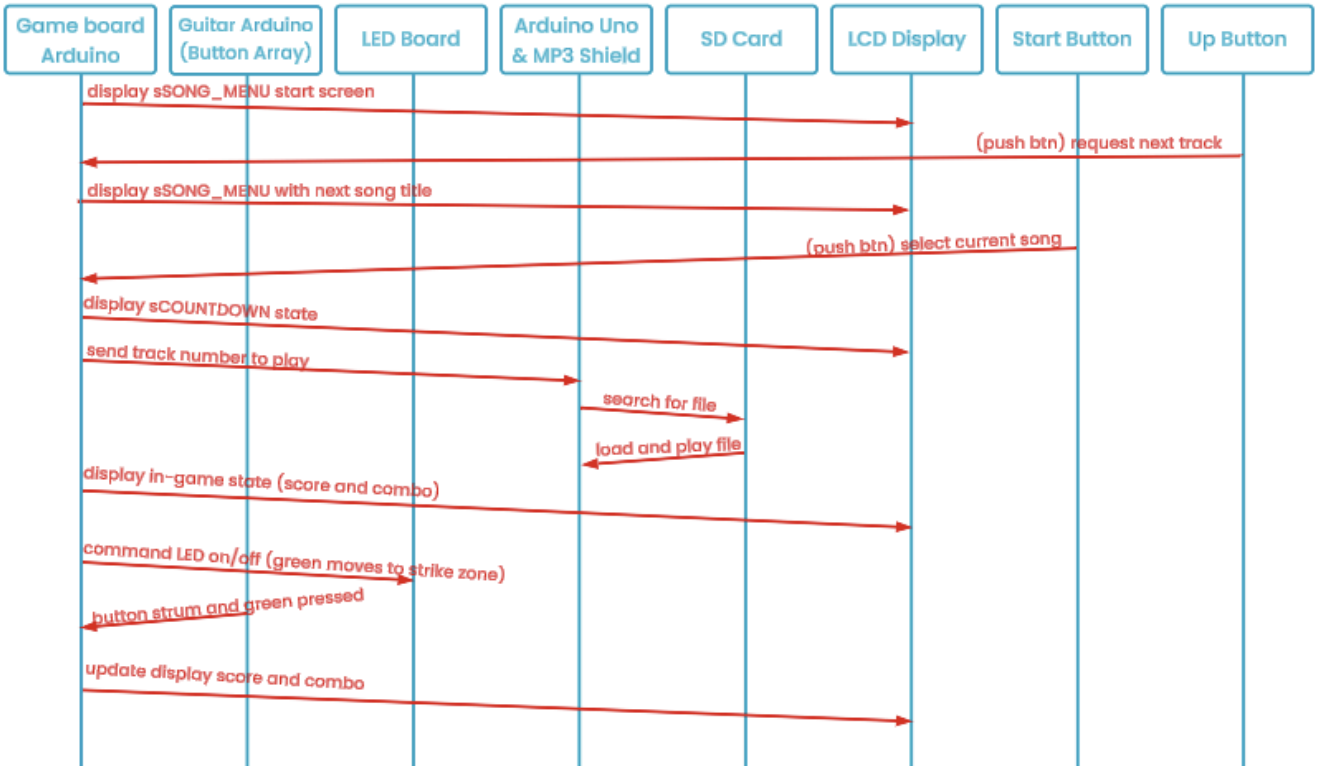
Architecture Diagram



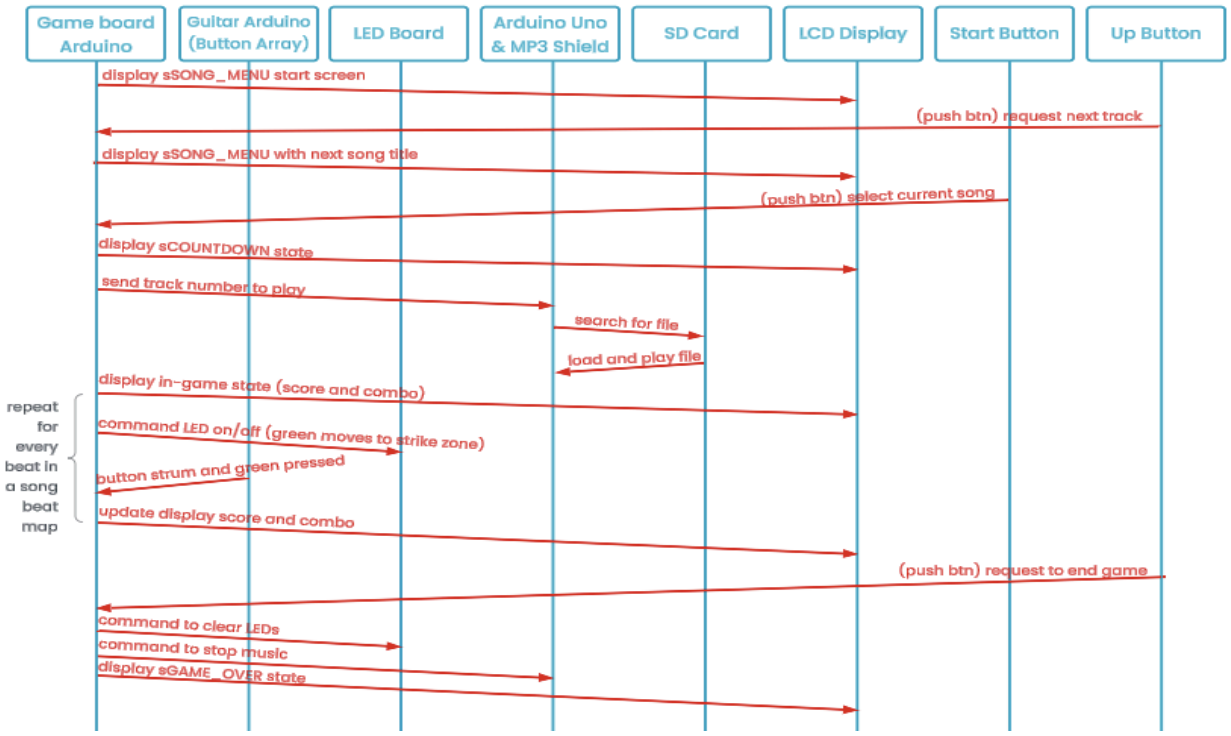
The above architecture diagram summarizes the structure of the system as a whole. Blue boxes indicate microcontroller system components while orange boxes represent components that do not.

Sequence Diagrams for Reasonable Use-Case Scenarios

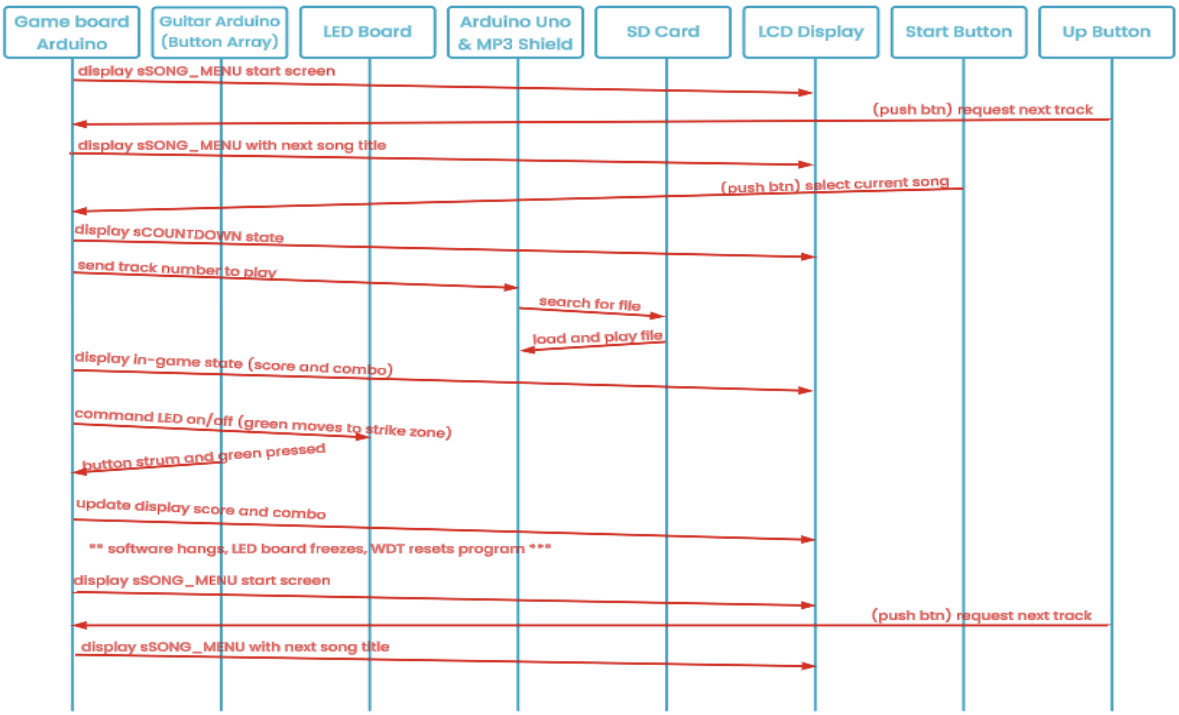
1. A Player strums and presses GREEN as it is the first displayed note.
2. A player ends the game mid-track by pressing on the START button mid-game.
3. An unfortunate player encounters a software hang in the LED game board such that our WDT resets the program and the player selects but does not play the same song.



Scenario 1: A player strums and presses GREEN as it is the first displayed note.

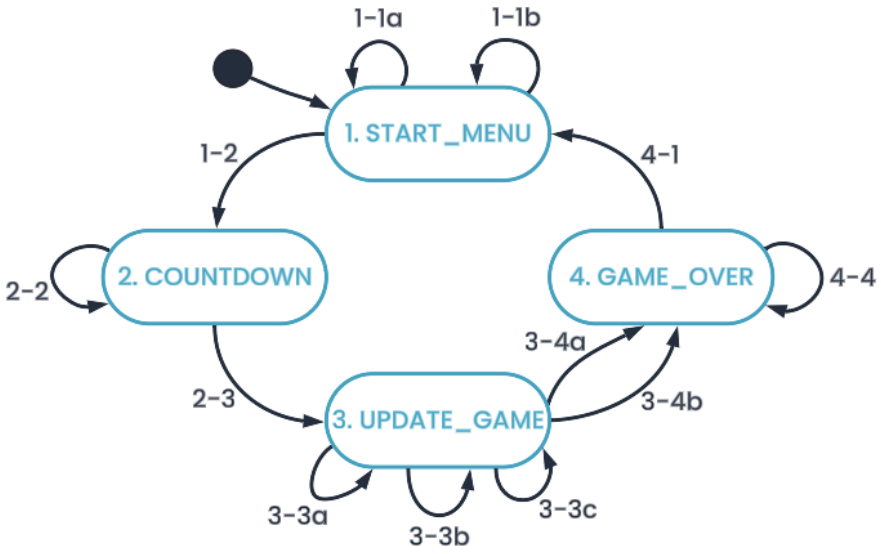


Scenario 2: A player ends the game mid-track by pressing START mid-game.



Scenario 3: An unfortunate player who encounters a software hang in the LED matrix.

Finite State Machine



States

1. START_MENU
2. COUNTDOWN
3. UPDATE_GAME
4. GAME_OVER

Inputs

1. mils: the current clock time in milliseconds
2. start_button: a boolean value indicating if the start button has been pressed since the last call to updateInputs()
3. up_button: a boolean value indicating if the up button has been pressed since the last call to updateInputs()
4. message: the byte received from the controller indicating a player's input. If no message was received, the message byte is set to 0b00000000.

Variables

Variable name	Description
start_beat_millis	Saved value of the clock from the start of a beat being shown
savedClock	Saved value of the clock (to track how long it has been since a specific transition, for example)
countdown	Counter for the countdown before the game starts
nextUpdateTime	this is how we do tempo (we don't use delay, we set the next updateTime)
beat_map	Array of bytes that holds the notes for a loaded song, each byte having bits that correspond to a note
beat_index	Index into the beat map based on the current note
finish_count	Value that counts down the remaining notes to play after the beat map has been completely read
beats_length	Total length of the beat map for the loaded song
isFirstCall	Boolean value indicating the first call to displayStart_LCD and displayEnd_LCD
song_num	Integer value indicating the song number currently selected

Output

1. combo: the count of consecutive correct notes a player scores during gameplay. The combo starts at zero and each time a beat is correctly played, is incremented by one. Once an incorrect beat is played, the combo is reset to zero.[3]
2. combo_max: the maximum combo a player achieved in a game run
3. score: the total score a player achieved in a game run. Each correctly played beat counts as 1 point added to the score. The score counts as zero and is never subtracted from.

[3] This is checked every time a player attempts to play a note, i.e. upon any player input but not otherwise evaluated.

Functions

LCD functions

displayStart_LCD(bool start_button, bool up_button, bool firstCall): displays the start button on the LCD screen with a welcome message followed by a song menu that displays the currently selected song

displayCountdown_LCD(int countdown): displays the countdown on the LCD

displayGame_LCD(score, combo): displays the current combo and score during gameplay

displayEnd_LCD(combo_max, score, start_button, firstCall): displays a game over message and a player's maximum combo and total score

Other functions

updateInputs()**:** updates start_button and up_button based on the sensor readings

moveLEDs(): gets the next beat in the beat map and lights up the corresponding LEDs on the top row of the game board to represent that note, while also moving the other notes down the board

performTimestepDelay(int start_beat_millis): delays between notes in a beat map based on BPM and drift

updateScoring(byte message): updates score, combo, and max_combo based on the player input represented by message

Transitions Table

Trans-ition	**Guard**	**Explanation**	**Output**	**Variables**
1-1 (a)	¬start_button ∧ ¬up_button	If no buttons are pressed, stay on the start menu.	displayStart_LCD(false, false, isFirstCall)	isFirstCall = false
1-1 (b)	¬start_button ∧ up_button	If the up button is pressed, scroll the selected song up.	displayStart_LCD(false, true, false, false)	song_num = song_num + 1
1-2	start_button	When the start button is pressed, begin the countdown.	displayStart_LCD(true, false, false, false)	currentMillis = millis() isFirstCall = true
2-2	((mils - savedClock) >= 1000) ∧ (countdown >= 0)	During the countdown, decrease the count every one second.	displayCoundown_LCD(countdown)	countdown = countdown - 1 savedClock = mils
2-3	countdown < 0	After the countdown, start the game using the currently selected song.	displayGame_LCD(combo_max, combo, score)	savedClock = mils
3-4 (a)	start_button ∨ up_button	If a game board button is pressed during gameplay, stop the game.	clearLEDs()	savedClock = mils finish_count = 6
3-3 (a)	(millis() >= nextUpdateTime) ∧ ¬(beat_map[beat_index] = 0b11111111) ∧ (beat_index < beats_length) ∧ ¬(start_button ∨ up_button)	Continue playing the game if there are still entries in the beat map.	moveLEDs(false) displayGame_LCD(combo_max, combo, score) performTimeStepDelay(start_beat_millis)	beat_index = beat_index + 1 savedClock = mils
3-3 (b)	(millis() >= nextUpdateTime) ∧ ((beat_map[beat_index] = 0b11111111) ∧ beat_index >= beats_length) ∧ (finish_count >= 0)) ∧ ¬(start_button ∨ up_button)	If the beat map has been run through, but there are still notes on the board, continue running until the notes have run off the board.	performTimeStepDelay(start_beat_millis) moveLEDs(true) displayGame_LCD(combo_max, combo, score);	finish_count = finish_count - 1

3-3 (c)	¬(message = 0) ∧ ¬(start_button ∨ up_button)	If a message (played note) is received from the controller, change the scoring variables accordingly.	updateScoring(message)	message = 0b00000000
3-4 (a)	start_button ∨ up_button	If a game board button is pressed during gameplay, stop the game and display the game over screen.	displayEnd_LCD(combo_max, score, false, true)	-
3-4 (b)	(millis() >= nextUpdateTime) ∧ (finish_count < 0)	Once the song has finished, include the residual lights from the beat map.	performTimeStepDelay(start_beat_millis)	savedClock = mils finish_count = 6
4-4	((mils - savedClock) < 3000) ∨ ¬start_button	Stay in the game over state if the start button isn't pressed or 3 seconds from the end of gameplay haven't elapsed.	displayEnd_LCD(combo_max, score start_button_pressed, false)	-
4-1	((mils - savedClock) >= 3000) ∧ start_button	After 3 seconds, if the start button is pressed, return to the start menu.	displayStart_LCD(false, false, true)	savedClock = mils countdown = 3 score, combo = 0 combo_max = 0 start_button = false isFirstCall = true beat_index = 0 bpm_index = 0

Traceability Matrix

The following is a table that maps requirements to the FSM guard in the software design, ensuring that each requirement is properly implemented.

T/S	R1	R1-A	R1-B	R2	R3	R3-A	R3-B	R4	R4-A	R4-B	R5	R5-A	R5-B	R6	R7	R7-A	R8	R8-A	R9	R10
Start	X																			
1-1 (a)	X																			
1-1 (b)		X																		
1-2				X																
2-2				X																
2-3			X		X	X	X													
3-3 (a)								X	X	X										X
3-3 (b)								X	X	X										X
3-3 (c)											X	X	X	X	X	X				
3-4 (a)																	X			
3-4 (b)																	X			
4-4																		X		
4-1																			X	

Overview of Testing Approach

Throughout the development of the system, we typically started with manual testing, as it allowed for rapid prototyping and quicker changes. The first manual testing procedures involved dummy Arduino programs that simply printed the inputs they were receiving. For instance, when first configuring the buttons on the controller, we printed the inputs to verify that hardware soldering and connections were sound. We then advanced to similar dummy applications to test our UART communications. The game board Arduino displayed UART data received from the controller Arduino to the serial monito, which helped us to initially test that these circuits were configured correctly and the interrupts were working.

Since our system is capable of running arbitrary beat maps and songs, we manually developed custom beat maps to test particular functionalities. For example, we created a simple beat map called “test_song,” which simply stepped through each note one-by-one at a slow bpm with no bpm changes. This beat map helped us to manually test many different parts of our system very early on, including our LEDs advancing on the display and our buttons detecting input on the controller. We also created a follow-up beat map called “test_song_2,” which was largely the same, except this mock contains a bpm change. Testing bpm changes early on was very important, since most public online beat maps contained many bpm changes, and the system would inevitably drift offbeat without handling these bpm adjustments.

Once our chart parsing script was written in Python, we used custom-created chart files to manually test our parser and ensure that it was satisfactorily outputting files. We tested both hand-designed chart files as well as community chart files downloaded from the internet.

After our main implementations were completed and we were somewhat confident in their correctness, we proceeded to write unit test suites for the application. We wrote some basic unit tests in Python for the parsing logic using Python’s unit test library. This includes tests for the create_arduinohero_struct function using various valid song_info_dict dictionaries and checking that the returned dictionary has the correct fields and values. We also test various sampling_rate values to ensure that they are processed correctly. We include tests for the process_bpm_changes function with valid song_data dictionaries, including edge cases such as songs with no bpm changes.

Our unit tests for our game board Arduino logic are written in C and primarily verify the software implementation of the state machine. We mock functionalities from the FastLED library to ensure that correct values are being passed to FastLED functions, the LEDs are being set up correctly in the setup functions, and their states are properly toggled as necessary. We test the

FSM and our updateFSM function by setting the current state and the state of the buttons to mock values, calling the function, and checking if the returned state is correct. We also test if the expected functions are called in each state (i.e., updating LEDs, writing to the LCD, etc.) by mocking these functions and their outputs.More specifically, our system tests were written to test all transitions featured on the transition table excluding the ISR “transition” that handles controller input. We made a corresponding testing matrix with test values for our system tests. These tests are located in GameboardTests.ino. See part 11 on how to run them.

Our game functioned well across playtests, so due to time constraints we determined that our testing would be best focused on system testing the code against our FSM and unit testing a particularly tricky component which was our parser. By system testing, we covered the updateFSM function of our main board which then exercised all non-mocked functions within the updateFSM. To facilitate system testing, we mocked out many of the output functions, so end-of-development testing focused on the structural soundness of integration and code rather than on the unit level. That being said, we have reasonable confidence in the unit level of the system components from manual testing that occursed throughout development and the testing infrastructure built into the code and exercised throughout the development process..

Procedure on how to run tests

To run our tests found within Arduino modules, simply uncomment the “#define TESTING” preprocessor directive line near the top of the Gameboard.h file, and then run the code as usual. Instead of launching the game on the Arduino, the tests will run and the results will be printed to the Serial Monitor.

To run our Python unit tests, you can simply navigate to the ArduinoHero/Parser/testing directory and run `python -m unittest parseChartTest.py`, the results will be printed to the terminal.