Arduino Hero Final Report

By: Molly McHenry, David Fryd, Serena Pulopot, Nick Bottone, Patrick Ortiz

Part 1: Overview

Our project is an implementation of Guitar Hero using Arduino. It features a custom LED screen 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. The game has players strum while hitting buttons on the controller in time with the music being played and the notes being shown on the LCD screen.

We assume that our users are interested in video games, music, or Guitar Hero. We assume they can hold a controller, hear, and play games that require hand-eye coordination. Additionally, the game controller is made to fit a right-handed player and the LCD screen assumes typical color-sightedness.

Part 2: Revised Requirements

R1: When the game is uploaded, 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:** To select the song, 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 be initialized to 0 and displayed on the LCD screen. The song selected on the track select screen shall be sent to the MP3 Shield, which retrieves the song from the SD Card and begins playing it. The LED screen shall display notes lit up corresponding to the approaching notes in the song.

> **R3-A:** The LED board corresponding to a guitar controller 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.

R4: Notes displayed[1] on the LED screen shall be displayed starting before their occurrence in the song. As their occurrence approaches, they shall move down the LED screen until they are located on the bottom LED row, at the time that they're audible in the song.

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

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.

R6: A player shall have a score that shall be incremented when a note is correctly played by a player. This score shall be displayed on the LCD screen.

> **R6-A:** When a note is displayed on the bottom row of the LED screen, if the note is played, one point shall be added to the player's score.
>
> **R6-B:** Each note shall have a "grace period" where playing a note one beat before or after it is displayed on the bottom row of the LED screen also counts as correctly playing the note, and shall increment the score by one.

R7: There shall be a combo counter that records a player's current combo (the running count of consecutive notes correct) displayed on the LCD. It shall be reset to 0 if a player plays an incorrect note.

> **R7-A:** Each correct note shall increase the current combo count by 1.
>
> **R7-B:** If a correct note and an incorrect note are played at the same time, the combo counter shall be reset to 0.
>
> **R7-C**: The combo scoring also implements the "grace period" of one beat before and after. If a player plays the correct note in this grace period of a note occurring in the song, the combo is incremented as if the note was played correctly.

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.

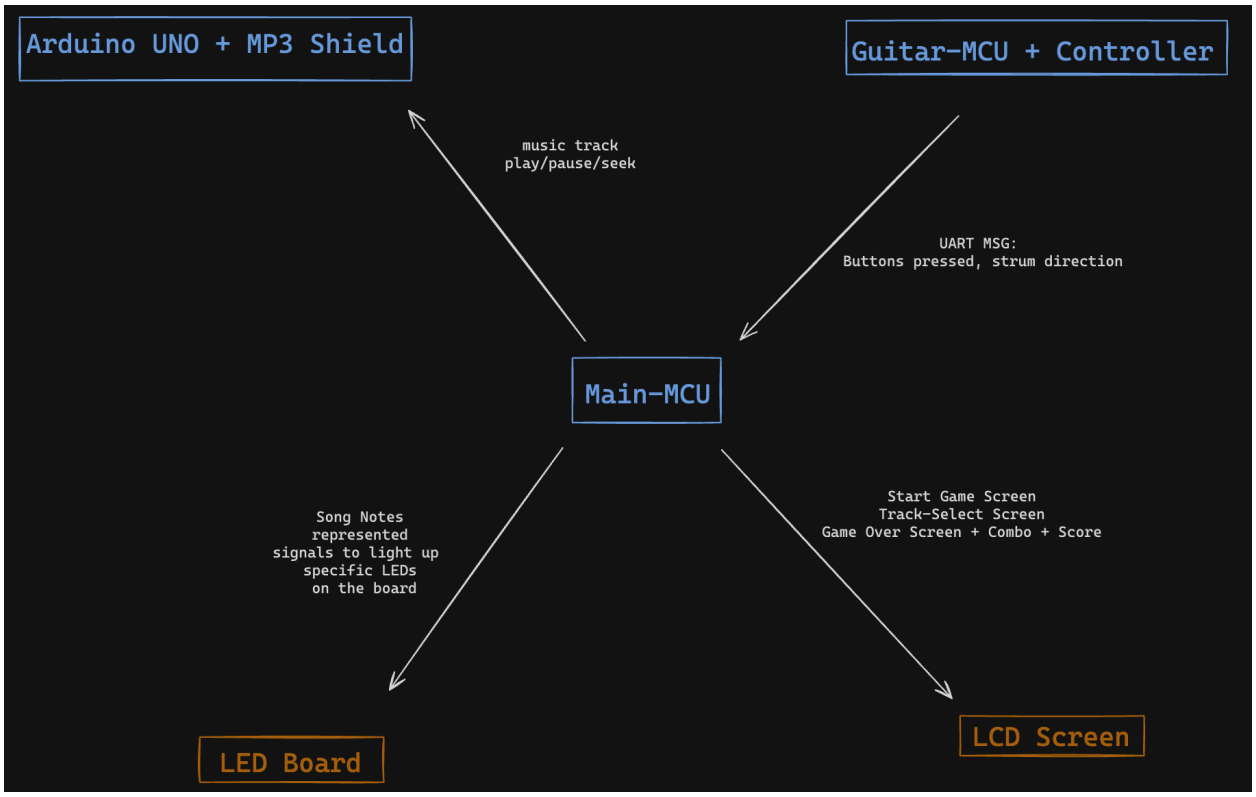
---

[1] 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.

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

R10: A watchdog timer shall bite when the LEDs are not advanced for multiple beats in a row.

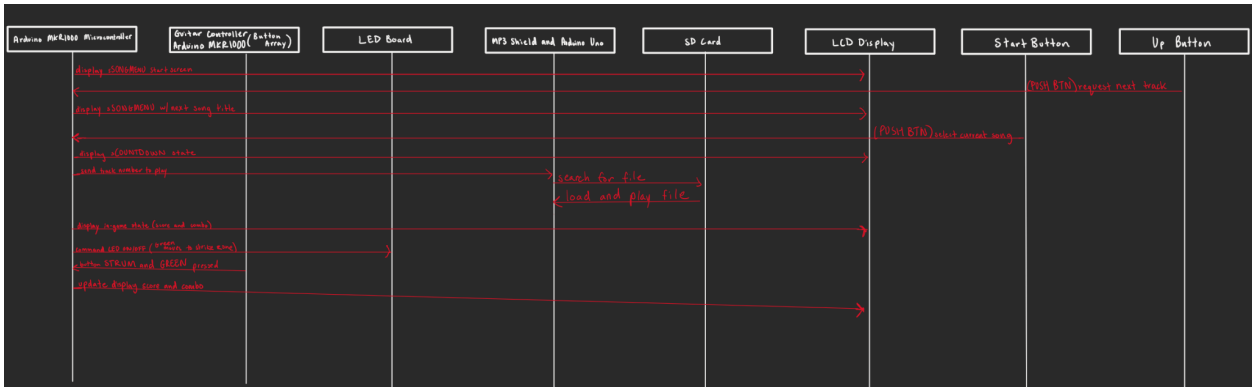
Part 3: Revised Architecture Diagram



The above Architecture Diagram summarizes the structure of the whole system. Blue boxes indicate system components that have a microcontroller onboard while orange boxes represent components that do not.

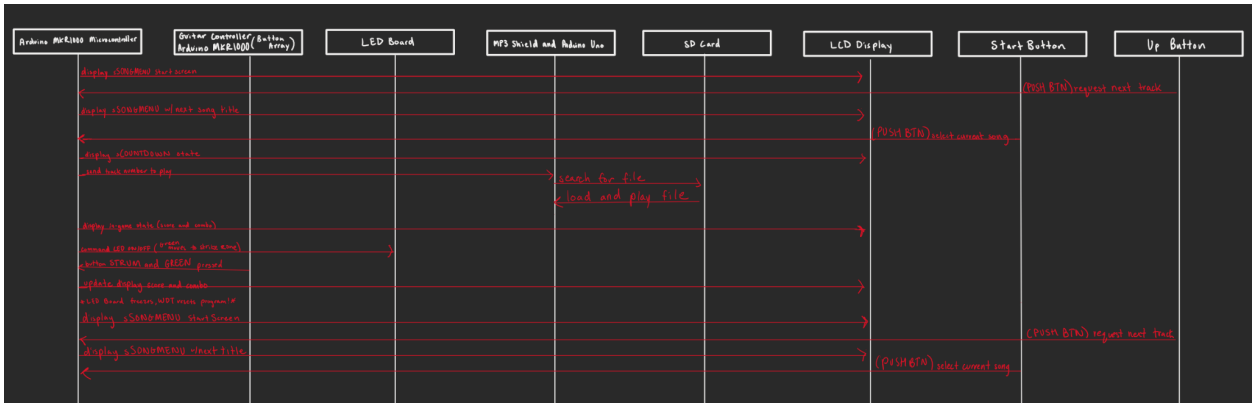
Part 4: Sequence Diagrams for Reasonable Use-Case Scenarios

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



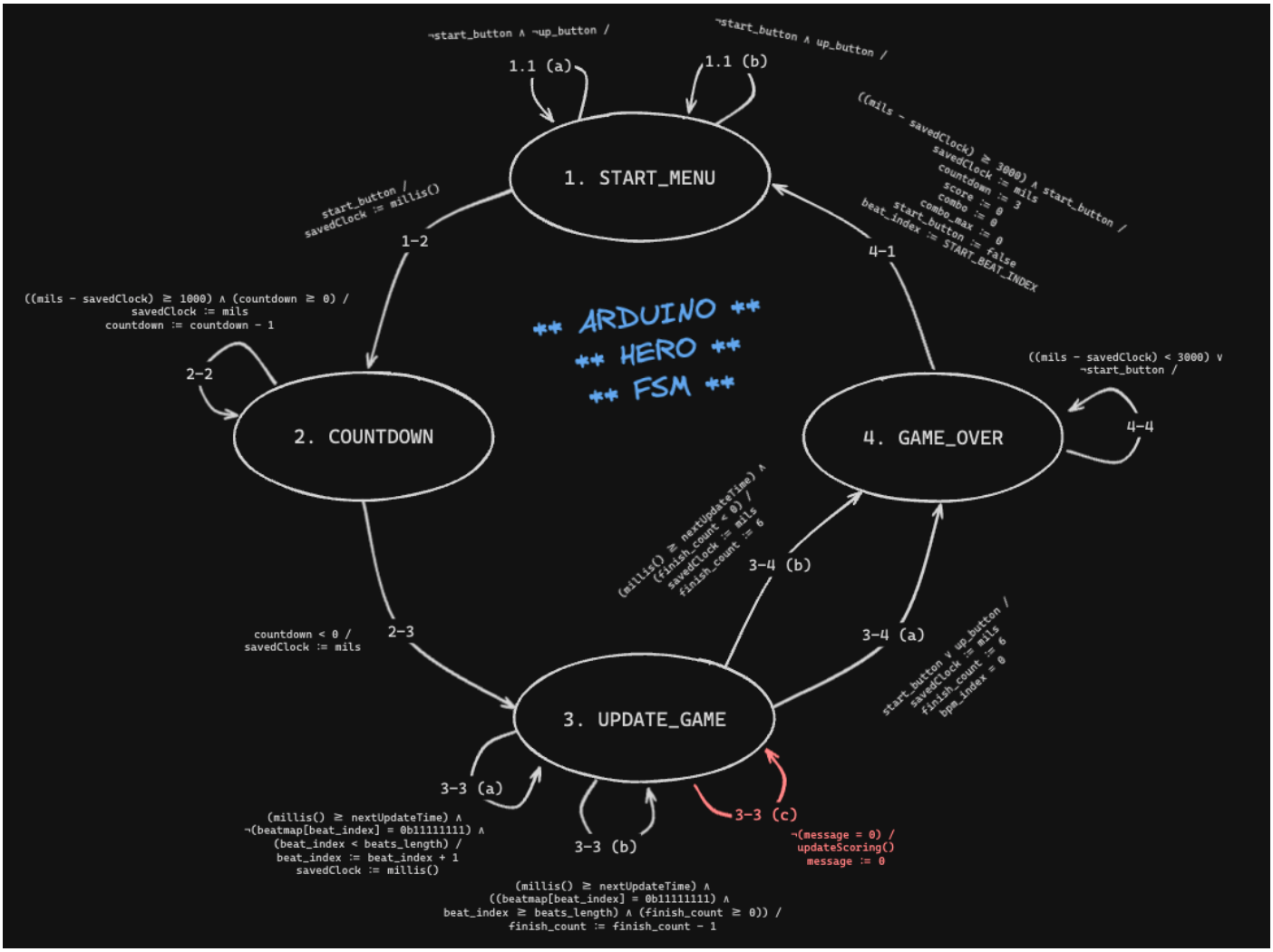
Scenario 1: A Player strums and presses GREEN as it is the first displayed note.

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



Scenario 3: Unfortunate player who encounters a software hang in LED Gameboard.

Part 5. Revised 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	the 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)
beatmap	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 beatmap based on the current note
finish_count	value that counts down the remaining notes to play after the beatmap has been completely read
beats_length	the total length of the beatmap for the loaded song
isFirstCall	boolean value indicating the first call to displayEnd_LCD

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.[2]
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.

Functions

LCD functions

```
displayStart_LCD(bool start_button, bool up_button, bool firstCall)
```

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

behavior: displays the start button on the LCD screen with a welcome message followed by a song menu that displays the currently selected song
returns: nothing

`displayCountdown_LCD(int countdown)`
behavior: displays the countdown on the LCD
returns: nothing

`displayGame_LCD(score, combo)`
behavior: displays the current combo and score during gameplay
returns: nothing

`displayEnd_LCD(combo_max, score, start_button, firstCall)`
behavior: displays a game over message and a player's maximum combo and total score
returns: nothing

Other functions

`updateInputs()`
behavior: updates start_button and up_button based on the sensor readings
returns: nothing

`moveLEDs()`:
behavior: looks at the next beat in the beat_map and lights up the corresponding LEDs on the top row of the gameboard to represent that note, while also moving the other notes down the board
returns: nothing

`performTimestepDelay(int start_beat_millis)`
behavior: delays between notes in a beatmap based on BPM and drift
returns: nothing

`updateScoring(byte message)`
behavior: updates score, combo, and max_combo based on the player input represented by message
returns: nothing

Transitions Table

Transition	Guard	Explanation	Output	Variables
1-1 (a)	¬start_button ∧ ¬up_button	If no buttons pressed, stay in the start screen	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)	-
1-2	start_button	When the start button is pressed, begin the countdown	displayStart_LCD(true, false, false)	currentMillis = millis() isFirstCall = true
2-2	((mils - savedClock) >= 1000) ∧ (countdown >= 0)	During the countdown, decrease the count every second	displayCoundown_LCD (countdown)	countdown = countdown - 1 savedClock = mils
2-3	countdown < 0	When the countdown is over, start the game	displayGame_LCD(co mbo_max, combo)	savedClock = mils
3-4 (a)	start_button ∨ up_button	If a gameboard button is pressed during gameplay, stop the game	clearLEDs()	savedClock = mils finish_count = 6
3-3 (a)	(millis() >= nextUpdateTime) ∧ ¬(beatmap[beat_index] = 0b11111111) ∧ (beat_index < beats_length)	Continue playing the game if there are still entries in the beatmap	moveLEDs(false) displayGame_LCD(co mbo_max, combo) performTimeStepDelay (start_beat_millis)	beat_index = beat_index + 1 savedClock = mils
3-3 (b)	(millis() >= nextUpdateTime) ∧ ((beatmap[beat_index] = 0b11111111) ∧ beat_index >= beats_length) ∧ (finish_count >= 0))	If the beat_map has been run through, but there are still notes to play on the board, continue running until the notes have run off the board	performTimeStepDelay (start_beat_millis) moveLEDs(true) displayGame_LCD(co mbo_max, combo);	finish_count = finish_count - 1
3-3 (c)	¬(message = 0)	If a message is received from the controller, change score/combo/combo_max accordingly	updateScoring(messag e)	message = 0b00000000
3-4 (b)	(millis() >= nextUpdateTime) ∧ (finish_count < 0)	Once the song has finished, including the residual lights from the beatmap	performTimeStepDelay (start_beat_millis)	savedClock = mils finish_count = 6
4-4	((mils - savedClock) < 3000) ∨ ¬start_button	Stay in gameover state if the start button isn't	displayEnd_LCD(comb o_max,	if isFirstCall: isFirstCall = false

		pressed or 3 seconds from end of game haven't passed	start_button_pressed, isFirstCall)	
4-1	((mils - savedClock) >= 3000) ∧ start_button	After 3 seconds, if the start_button is pressed, return to home screen	displayEnd_LCD(combo_max, start_button_pressed, isFirstCall)	savedClock = mils countdown = 3 score = 0 combo = 0 combo_max = 0 start_button = false isFirstCall = true beat_index = 0 bpm_index = 0

Part 6. Revised Traceability Matrix from Part 5 of the milestone report

T/S	R1	R1-A	R1-B	R2	R3	R3-A	R3-B	R4	R4-A	R4-B	R5[3]	R6	R6-A	R6-B	R7	R7-A	R7-B	R7-C	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							X
3-3 (b)								X	X	X					X							X
3-3 (c)												X	X	X	X	X	X	X				
3-4 (a)																			X			
3-4 (b)																			X			
4-4																				X		
4-1																					X	

[3] Controller-related requirement, thus not represented in the Game FSM.

Part 7. Overview of Testing Approach

Throughout the development of our system, we typically started with manual testing, as it allowed for rapid prototyping and quicker changes. Our very first manual testing procedures involved dummy Arduino programs that simply printed the inputs they were receiving. For instance, when we were first configuring the buttons on the controller, we printed the inputs to verify that our soldering and connections were done correctly. Next, we advanced to similar dummy applications to test our UART communications. Our Gameboard Arduino would print when it received UART communications from the Controller Arduino, which helped us to initially test that these circuits were configured correctly. We didn't have any bad connections. While these manual tests may not have been exhaustive and may not have tested all of our application logic, they were critical for increasing the confidence in our initial circuits for each module.

Since our system was capable of running arbitrary beatmaps and songs, we manually developed some beatmaps for testing particular functionalities. For example, we created a simple beatmap called "test_song", which simply stepped through each note one by one at a slow bpm with no bpm changes. This beatmap 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 beatmap called "test_song_2", which was largely the same, except this song contained a bpm change. Testing bpm changes early on was very important for us, since most public online beatmaps contained many bpm changes, and our game would inevitably drift offbeat without handling this.

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 some unit test suites for the application. We wrote some basic unit tests in Python for the parsing logic using Python's unittest 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 and check if the returned BPM values and change indexes are correct, including in edge cases such as songs with no bpm changes.

Our unit tests for our Gameboard Arduino logic are in .INO files in a format similar to that of the testing lab. We mock functionalities from the FastLED library, so we can 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 correct

functions are called in each state (i.e., updating LEDs, writing to the LCD, etc.) by mocking these functions.

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. Here is the link: https://docs.google.com/spreadsheets/d/1HgTLqI4VlkioLNcv2W1mlwIq6yzrlVUMST9VYWkznwc/edit#gid=6631574. These tests are located in GameboardTests.ino. See part 11 on how to run them.

We determined that since our game was able to function well across playtests, 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 make system testing possible, however, we needed to mock out many of the output functions, so these are not exercised. However, we see the numerous successful playtests as sufficient evidence that those functions are working as expected.

Thus, our criteria for testing were successful systems tests in addition to robustness to many playtests. We believe we have met these criteria with successful system testing and playtests.

Part 8. At least 2 safety + 3 liveness requirements

Liveness:

“When the game ends, we enter game over until the user has pressed the start button”

$$\mathbf{G}(\texttt{sUPDATE\_GAME} \land (\texttt{finish\_count}==6) \Rightarrow \mathbf{X}(\texttt{sGAME\_OVER}\ \mathbf{U}\ \texttt{start\_button}))$$

“When start button is pressed, the game will eventually start”

$$\mathbf{G}(\texttt{sSTART\_MENU} \land \texttt{start\_button} \Rightarrow \mathbf{F}\texttt{sUPDATE\_GAME})$$

“When we strum, we enter the ISR to check for points and then return to the game loop”

(Where `strum` represents having received UART message on data line)

$$\mathbf{G}(\texttt{sUPDATE\_GAME} \land \texttt{strum} \Rightarrow \mathbf{X}\texttt{sISR} \land \mathbf{F}\texttt{sUPDATE\_GAME})$$

Safety:

“Score never becomes negative”

$$\mathbf{G}(\texttt{score} > 0)$$

“Combo never exceeds the maximum recorded combo”

$$\mathbf{G}(\texttt{max\_combo} >= \texttt{combo})$$

Part 9. Discussion of Environment Processes to Model to Compose FSM with Models of Environment to Create Closed System

To create a closed system for analysis, let's consider the environment processes that would have to be composed.

1. Button Inputs (up_button, start_button): These inputs represent user action, specifically for selecting songs in the START_MENU state and initiating the game. It would make sense to model these as discrete, non-deterministic signals. They represent events occurring at unpredictable times, so they should fit the discrete model where the state change is event-driven. The user's unpredictable behavior is what makes the button unpredictable.

2. Timing Input: Modeling timing input is important for Guitar Hero to work since the game requires precise timing for scoring and gameplay synchronization. The mils input, tracking elapsed time, can be a hybrid, deterministic system. It is hybrid because it involves continuous time tracking while interacting with discrete events like button presses and game state changes. It is deterministic since it is based on time.

3. Communication Input (message): from the ISR that handles UART communication from a controller Arduino, this input is triggered by user button presses. SImilar to the buttons above, this should be a discrete, non-deterministic signal. The message byte, representing the user input, fits this as it is unpredictable and event-driven.

Part 10. Description of the Code Files

Requirements highlighted in yellow:

Controller

UPLOADS TO GUITAR CONTROLLER ARDUINO MKR1000

Controller.ino

- Contains the logic for reading in the Guitar Controller's button inputs. The guitar has 7 possible ADC inputs: Green, Red, Yellow, Blue, Orange, Strum Up, Strum Down (lines 12-19):

```
pinMode(GREEN_BUTTON, INPUT);
pinMode(RED_BUTTON, INPUT);
pinMode(YELLOW_BUTTON, INPUT);
pinMode(BLUE_BUTTON, INPUT);
pinMode(ORANGE_BUTTON, INPUT);

pinMode(STRUM_BUTTON_UP, INPUT);
pinMode(STRUM_BUTTON_DOWN, INPUT);
```

And read in lines 54-59.

```
for(int i=6; i < 11; i++){
 // Serial.println((int)B, BIN);
 if(digitalRead(i) == 1) {
   B |= (1 << (i-6));
   }
}
```

- Additionally, INTERRUPTS/ISR are attached to the Strum Up and Strum Down inputs such that when either button is pressed, the program will read the combination of color inputs and store it as an 7-bit value, where each bit corresponds to an input's status (lines 21-22):

```
attachInterrupt(STRUM_BUTTON_UP, strumUpInterrupt, RISING);
attachInterrupt(STRUM_BUTTON_DOWN, strumDownInterrupt, RISING);
```

Lines 28-40:

```
void strumUpInterrupt(){
```

```
  byte B = 0;
  B |= (1 << 6);
  handleInterrupt(B);
}
void strumDownInterrupt() {
  byte B = 0;
  B |= (1 << 5);
  handleInterrupt(B);
}
```

- Both interrupts use *uartSend()* in *ControllerUART.ino* to be sent to the main Gameboard using UART COMMUNICATION and our declared output pin, *outPin*. This occurs in *handleInterrupt()*, lines 42-65.

```
void handleInterrupt(byte B){
  unsigned long currentMillis = millis();
  if (currentMillis - lastDebounceTime > debounceDelay) {
    #if defined(CALIBRATE)
      hardwareCheck();
    #else
      noInterrupts();
      /*create byte to represent the user event,
      with each bit corresponding to a button
      press (1 being pressed) */
      // byte B = 0;
      for(int i=6; i < 11; i++){
        // Serial.println((int)B, BIN);
        if(digitalRead(i) == 1) {
          B |= (1 << (i-6));
        }
      }
      uartSend(B);
      lastDebounceTime = currentMillis;
      interrupts();
  }
  #endif
}
```

Controller.h

- The header file contains declarations for all functions and variables in the *Controller.ino* arduino file that could be shared between files.

ControllerUART.ino

- This file preps our Guitar Controller's 7-bit input for 8-bit communication, such that we send an 8-bit value or 1 byte of data to the Gameboard through UART COMMUNICATION. The 8th bit added is a parity bit used to check the validity of the communicated data once in the Gameboard (*uartSend()*, lines 11-43):

```
void uartSend(byte B) {
 Serial.println((int)B, BIN);
 digitalWrite(outPin, LOW);
 unsigned long lastClockTime = micros(); // record when pin was flipped
 int i = 0;
 byte parity = 0;
 while (i < 7) {
  if ((B & 0x1) == 0x1 ) {
    uartDelay(lastClockTime);
    digitalWrite(outPin, HIGH);
    lastClockTime = micros();
    parity = parity ^ 0x01;
  } else {
    uartDelay(lastClockTime);
    digitalWrite(outPin, LOW);
    lastClockTime = micros();
  }
  B = B >> 1;
  i += 1;
 }
 uartDelay(lastClockTime);
 // write parity bit;
 if (parity == 1) {
  digitalWrite(outPin, HIGH);
 } else {
  digitalWrite(outPin, LOW);
 }
 uartDelay(micros());
 digitalWrite(outPin, HIGH);
}
```

- Our design does not require the Guitar Controller to receive from the Gameboard.

Gameboard

UPLOADS TO LED-CONTROLLING ARDUINO MKR1000

Gameboard.h

- A header file that contains function, struct, and variable declarations and Arduino pin Assignments that enable *Gameboard.ino* to update and manage the game/game states.

Gameboard.ino

- This file assigns all pins for communication and LCD control, attaches an ISR to the pin receiving data from the Guitar Controller, sets up a game TIMER/COUNTER, and manages game states via our *updateFSM()* function. Said TIMER/COUNTER is defined on line 26:

```
savedClock = 0;
```

- And used in guards /updated per our *updateFSM()*, for example lines 116 in the *sCOUNTDOWN* state guard:

```
  if((mils - savedClock) >= 1000 && countdown >= 0){ //transition 2-2
```

and soon updated on line 119:

```
    savedClock = mils;
```

*mils is attached to the MCU's *millis()* function call.

GameboardUART.ino

- As we use *Serial1* to communicate with Adafruit's Music Maker Shield, we must manually assign an Arduino MKR1000's pin to transmit information for SERIAL UART COMMUNICATION with the Guitar Controller. In this case, data is only received from the Guitar Controller, processed by *uartReceive()* (lines 20-78):

```
void uartReceive() {
 // delay for 1/3 of the UART period just to get reads towards the middle of bits
 delayMicroseconds(UART_PERIOD_MICROS / 3);
 unsigned long lastClockTime = micros();
 int i = 0;
 byte B = 0;
```

```
byte parity = 0;
while (i < 7) {
  B = B >> 1;
  uartDelay(lastClockTime);
  int inPinVal = digitalRead(UART_IN_PIN);
  lastClockTime = micros();
  if (inPinVal == HIGH) {
    B = B | (0x1 << 6);
    parity = parity ^ 0x01;
  }
  i += 1;
}
// Receive parity bit
uartDelay(lastClockTime);
int inPinVal = digitalRead(UART_IN_PIN);
lastClockTime = micros();
// compare computed and received parity
// if match, use value
if(parity == inPinVal){
  if((beatmap[beat_index-4] & 0b00011111) == (B & 0b00011111) ||
    (beatmap[beat_index-5] & 0b00011111) == (B & 0b00011111) ||
    (beatmap[beat_index-6] & 0b00011111) == (B & 0b00011111)){
    score += 1;
    Serial.print("score: ");
    Serial.println(score);
    combo += 1;
    if(combo > combo_max) combo_max = combo;
    Serial.print("current combo: ");
    Serial.print(combo);
    Serial.print(" (max: ");
    Serial.print(combo_max);
    Serial.println(")");
  } else {
    if(combo > combo_max) combo_max = combo;
    combo = 0;
  }
}
// // get past this last bit so as not to trigger an early interrupt
// uartDelay(lastClockTime);
// Serial.println((int)B, BIN);
}
```

LCD.ino

- Contains the functions that update the state and graphics displayed on a 16x2 LCD screen such as a *Song Menu*, *Countdown*, and *Game Over* display.

LEDS.ino

- Contains the functions that populate an LED array, moving and animating lights down the physical LED Gameboard by reading in from the requested song, stored in the *songs.h* file.
- This file is responsible for driving the LEDs using PWM, with their brightness on each of the RGB channels controlled by the duty cycle. We use the FastLED library functions to drive the display, and you can see we use constants like Blue and Red to refer to colors which are sets of duty cycles for each color channel.

```
if((beat & (1 << 4)) > 0) columnColors[ORANGE][0] = CRGB::OrangeRed;
if((beat & (1 << 3)) > 0) columnColors[BLUE][0] = CRGB::Blue;
if((beat & (1 << 2)) > 0) columnColors[YELLOW][0] = CRGB::Yellow;
if((beat & (1 << 1)) > 0) columnColors[RED][0] = CRGB::Red;
if((beat & (1 << 0)) > 0) columnColors[GREEN][0] = CRGB::Green;
```

WDT.ino

- Contains our WatchDog Timer, implemented such that it is pet in *Gameboard.ino*'s *updateFSM()* in the *sUPDATE_GAME* state, monitoring that the LED board is properly reloaded throughout the song's duration and if not pet on time, will restart the program.It is set up in *Gameboard.ino* line 42:
  ```
  setupWatchdogTimer();
  ```
- In the transition from *sCOUNTDOWN* to *sUPDATE_GAME*, the WDT is enabled, *Gameboard.ino's* line 129:
  ```
  enableWatchdogTimer();
  ```
- The WDT is pet on every beat that occurs when *sUPDATE_GAME* is looping through a song's beat map, *Gameboard.ino's* line 147 and 157:
  ```
  resetWatchdogTimer(); // pet the watchdog timer
  ```
- It is disabled on transitioning from *sUPDATE_GAME* into sGAME_OVER, *Gameboard.ino's* line 142 and 168:
  ```
  disableWatchdogTimer();
  ```

song.h

- A header file containing a *Song* struct definition that *parseChart.py* fills for every *.chart* file to be read by our Gameboard program.

songs.h

- A header file containing multiple songs using the *song.h* struct. This file holds all possible songs we can play on the Gameboard, controlled by *LCD.ino* interactions.

Parser

RUNS ON A PC TO GENERATE FILES FOR THE GAMEBOARD

parseChart.py

- This file takes *.chart* files largely available online (due to the popularity of *Clone Hero*) and converts them to our song structure defined in *song.h.* As an overview, the *Song* struct stores a song's: name, artist, charter, album, year, genre, filename, resolution, sampling_rate, bpm_values (beats per minute; changes during songs), bpm_change_indexes, bpm_values_length, and most importantly, beats, and beat_length. *beats* are stored as a byte pointer, such that every byte represents a beat in a song and can be compared to the prepared input byte sent from the Guitar Controller.

chorus_charts/

- Input folder for *parseChart.py*, containing desired *.chart* files.

parsed_charts/

- Output folder for *parseChart.py* containing converted *.chart* files that can be read by our Gameboard program.

song.h

- A header file containing a *Song* struct definition that *parseChart.py* fills for every *.chart* file to be read by our Gameboard program.

RecieveAndPlaySong

UPLOAD TO MUSIC SHIELD ARDUINO UNO

RecieveAndPlaySong.ino

- This file controls the music of the game through Adafruit's Music Maker Shield and an Arduino UNO. Using the *Adafruit VS1053 Library*, this file receives the track number from the Gameboard Arduino and plays the corresponding track uploaded to a MicroSD card. This file can additionally send back info about the status of the song. This is enabled by following Serial UART COMMUNICATION (lines 59-60):

```
if(mySerial.available()) {

 char inByte = (mySerial.read());
```

Part 11. Procedure on how to run your unit tests

To run our unit 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.

Part 12. Reflection on our Goals

According to our milestone final demo deliverable, we stated:

- We will demo gameplay on our completed system, which will feature 2 guitar controllers and an LED matrix display.
- The LED matrix display will be able to display a beat map while a song is played

We successfully completed our second point, through *parseChart.py* and *Gameboard.ino* functions which update the LED board in the proper timing with a song played in the background. Although we created and completed our single-player game, we did not achieve our multiplayer goal.

CHALLENGES:

There were two main challenges in this process: development time and possible hardware restrictions. We are confident that if given more time, we could implement a second guitar following our first guitar's schema. As we currently use an ISR to read from our single Guitar Controller, adding another Controller would create complications in ISR priority and this suggests an ISR design is insufficient for a two-player system. This would require a little more consideration and a possible redesign of how the controllers communicate with the Gameboard. Beyond our time constraint in figuring out software design, the main Gameboard Arduino MKR1000 has physical hardware constraints that would complicate communication with two controllers. A Guitar Controller only requires one pin to send data to the GameBoard but we would have to research which pins can be multiplexed to do so. These were the main two challenges and reasons why we did not reach our previous goal of two Guitar Controllers since the milestone.

As a final reflection statement, our ultimate goal was to create a functional LED-displayed Guitar Hero-inspired rhythm game that would be fun and addicting to play. With the overwhelming popularity and interest on Demo Day, we felt we achieved this feat!

Part 13. Appendix: Review Spreadsheets

Reviewer names:	Scott Petersen		
Defect record		*For project group use:*	
Transition or state #	**Defect**	**Fixed?**	**Justification**
3-2	ISR termination is not a variable or input transition and thus cannot be a guard	Fixed	ISR is no longer an input or variable.
2-2,2-3,2-4	These transitions are not mutually exclusive	Fixed	New FSM with different states and guards that are mutually exclusive as the game loop progresses
	Where is the documentation for the display Arduino and their communication? Aren't these separate machines?	Will not fix	We redesigned our project such that we no longer have more Arduinos than we need. Additionally, we only have an FSM for our main game Arduino. The peripheral Arduinos *do* communicate and that is displayed in our architecture diagram which was not viewed at Milestone.

Reviewer names:	Abhyudaya Sharma, Andrew Li, Shreyas Raman, Andrew Li		
Defect record		*For project group use:*	
Transition or state #	**Defect**	**Fixed?**	**Comments**
	Each Arduino should have its own FSM	Will not fix	Due to the varying complexities of our peripherals, some Arduinos do not have or require FSMs.
3-2	ISR terminates is not a valid guard	Fixed	ISR is no longer an input or variable.
2-3	There should be a different transition for each of the cases	Will not fix	We believe they mean if either StrumDown or StrumUp is pressed, however, we no longer implement any differing use from these inputs.
2-3 and 2-4	These transitions do not have mutually exclusive guards	Fixed	New FSM with different states and guards that are mutually exclusive as the game loop progresses

Reviewer names:	Coco Kaleel, Angel Xu, Selena Williams, Cali Rivera		
Defect record		*For project group use:*	
Transition or state #	**Defect**	**Fixed?**	**Comments**
2-2, 2-3, & 2-4	The guards don't include the saved clock which prevents mutual exclusivity	Fixed	New FSM with different states and guards that are mutually exclusive as the game loop progresses
2-3	We are not sure if this is a defect, but we are concerned that the ISR handling too much could interfere with system timing	Fixed	This was a valid concern that we kept in mind while we worked on the project.
3-2	Lack of explanation about the ISR finishing. ISR terminating is not a guard	Fixed	ISR is no longer an input or variable.
3-2	There's no inherent output to ISR terminating, which makes it difficult to implement in code	Fixed	ISR is no longer an input or variable.