Final Report: Guitar Buddy

ECE 445, Fall 2018
Project Number 15
TA: Channing Philbrick

Austin Born and Chris Horn
December 12, 2018

Contents

1. Introduction

1.1. Objective

The Guitar Buddy system is intended to give learners on a budget the ability to gain real-time feedback about hand position and technique. Traditional instructors can help students with reading music, hand and finger arrangements, speed and quality, and they often have the benefit of prior experiences from when they were still learning to play an instrument. However, paying for private music lessons every week can be prohibitively expensive. Based on a 2014 national study by takelessons.com, music lessons typically range from $30 to $50 per hour [1]. If a student attends just one hour-long lesson each week, this costs between $1,500 and $2,500 per year. The Guitar Buddy system solves this problem by giving new guitarists a way to learn the proper way to play without needing to attend private weekly lessons.

1.2. High-Level Overview

The device consists of a control board connected to any number of printed circuit boards (PCBs) in series, with each board consisting of six LEDs, six steel string contacts, and associated control logic. The LEDs are mapped to the notes on the guitar such that given a basic MIDI file for a song, the device can map them to the array of LEDs to show the student where to depress the strings to play the proper notes. The original high-level requirements proposed for this device were that it could display chords near real-time, it could accurately sense if the correct notes were depressed by the user, and that the device could be powered internally up to 2 hours. As of now, the first and third requirements have been met, with the second requirement verified in individual board tests, though not after full integration. Majority of the requirements have been met and verified, with only a few specific verification failures that prevented the device from reaching full functionality.

1.3. Block Diagram

The high-level block diagram (Figure 1.1) is broken down into six main modules. The power supply module is responsible for the power management of the guitar, including charging and discharging circuits as well as additional safety hardware. The sensing module is responsible for detecting when the user depresses any strings along the neck of the guitar, and reporting that information to the primary microcontroller. The control module houses the primary microcontroller, an ESP32, and is responsible for communication with the software module, storing song information, and sending the control signals for driving the LEDs. The LEDs are driven by the LED output module, which is responsible for managing the 30 LEDs from only a few I/O pins on the primary microcontroller. The final module is the software module, which is responsible for processing music and sending the binary song data to the ESP32 over Bluetooth. Over the course of the project, the only major changes were the use of on-board Analog-to-Digital Conversion in the ESP32 instead of externally, the addition of the ESP32 firmware as software, and the use of the internal memory on the ESP32 instead of external memory. The first change was to reduce the complexity of the sensing module, which saved significant overhead at the expense of some ESP32 processing speed (though ultimately insignificant). The second change was simply an addition of expected software. The third change was the result of expected ease of use of on-board memory, though this proved difficult as the project progressed (explained further in report).

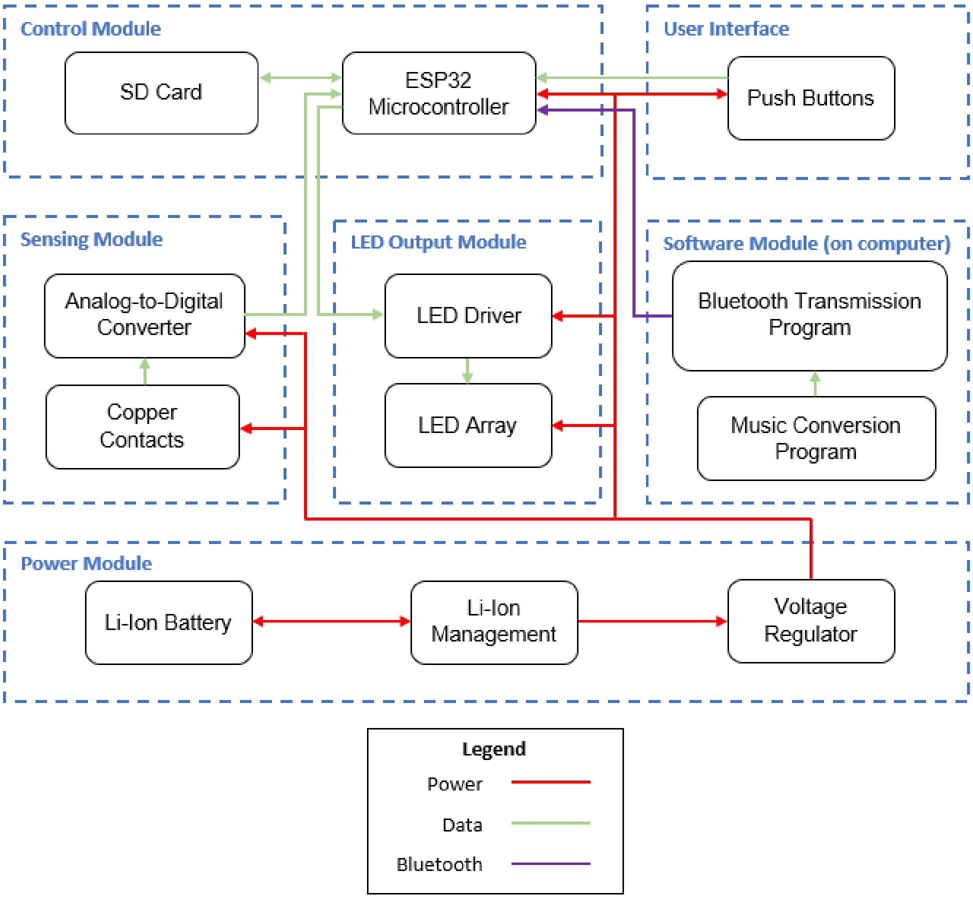


Figure 1.1: Guitar Buddy System Block Diagram from the original design document.

1.4. Overall Design

The physical design consists of five PCBs fixed between the second through seventh frets of the guitar, affixed with non-damaging hot glue. The control PCB is fixed to the body of the guitar along with the power supply, and each PCB is connected serially along the neck of the guitar. The control board and wires are routed so they do not interfere with the player's hand positions. The design of the PCB requires 12 header pins on each PCB to connect data and power lines, with individual wire strips connecting to each board. The control board can function wirelessly due to the on-board power supplied by the battery, and Bluetooth connection to an external laptop allows it to wirelessly send song data as needed. A diagram of the Guitar Buddy system on the guitar is shown in appendix A.1 and a model user of the device is shown in appendix A.2.

2. CONTROL MODULE

2.1. DESIGN PROCEDURE

The control module's role is to manage the external Bluetooth connection with the host laptop, send data to the LED output modules, and manage song data. An ESP32 is used for this purpose, due to its built in Blutooth and WiFi capabilities, built in 4MB of non-volitile flash storage, its high clock rate of up to 240 MHz, and its generous number of GPIO pins [2]. The control flow for the device is managed from the firmware running on the ESP32, and manages populating the frame buffer for upcoming notes, sending and receiving data from the software module, and sending data to the fret PCBs. A picture of the control board can be found in appendix A.6.

Alternative microcontrollers, including the ATMega 328p and MSP430, were considered as potential options for the primary microcontroller. However, lack of RAM, non-volatile storage, and lack of built in wireless capabilities made these inferior choices.

2.2. MICROCONTROLLER

2.2.1. DESIGN DETAILS

The control module is connected to the rest of the device through a few important connections: a serial connection with the fret boards, a power connection to the power module, and a wireless Bluetooth connection with the software module.

The serial connection with the fret boards consistent of three connections: a serial data line (SOUT), a clock line (CLK), and a load effective (LE) line. When a timer interrupt fires on the ESP32, signalling that it is time to update the frame displayed on the frets, the ESP32 immediately starts sending serial data out to the boards. For each rising edge of CLK, all data in the fret's shift registers is shifted, and the value of SOUT is sent to the first board. After all of the data has been serially sent to the fret boards (which requires $N * 8$ CLK cycles for N fret boards present), LE is cycled and the new data is displayed on the LEDs.

The power connection to the power module consists of two power connections (one for V_{BAT} and another for GND), as well as one ADC input to monitor battery temperature. When not sending data to the fret boards or loading song data over Bluetooth, and control module is constantly monitoring the battery temperature to ensure that is stays within the operating range (see Section 5: Power Module for more information).

The Bluetooth connection is composed of a wireless serial interface running at a baudrate of 115,200 bits per second. The ESP32 hardware, along with accompanying libraries, abstracts most of the Bluetooth implementation away, so it can be treated like any other serial port. On boot up, the ESP32 waits for a packet to arrive through the Bluetooth connection, and then parsing the incoming song data and loads it into RAM. From there, it plays the song until a new song is loaded.

Due to time constraints during development, the on-board flash storage is not used to store song data across reboots. The current software implementations allows for up to 100 kB of song data to be loaded into memory. Initial approaches to use the on-board flash storage involved use of the provided ESP32 NVS (non-volitile storage) libraries, but delays caused by blocking portions of code created unacceptable delays and hangs while driving the fret PCBs. As a result, future work involves utilizing off-board storage, likely in the form of EEPROM or an SD card (as originally planned).

2.2.2. Verification

To verify that the ESP32 was fast enough to update the fret PCBs in time with music, an oscilloscope was used to monitor the output signals from the SOUT and CLK lines. The ESP32 was capable of shifting out bits at a rate of approximately 2 MHz (or 2 Mbits/s) at maximum speed, which is well in excess of the minimum bandwidth necessary to drive the display. The minimum bandwidth to drive N boards at F_{target} refresh rate can be calculate with $F_{target} * 8$ bits/boards $* N$. For the prototype with five boards and a targeted 200 Hz refresh rate ($\frac{1}{5\text{ ms}}$), the minimum bandwidth is 200 Hz $*$ 8 bits/board $*$ 5 boards = 8 kb/s.

The power consumption of the device also exceeded initial designed expectations, drawing only approximately 80 mA during regular usage. Peak current can exceed 200 mA during Bluetooth communication, which is also within the range specified in the original design plan.

2.3. Flash Storage

2.3.1. Design Details

The initial design plan to use an external SD card as flash storage was changed part way through the project. Due to the physical construction of the control module board, along with the presence of internal flash memory, the external SD card route was originally found to be clunky and unnecessary. However, limitations in both bandwidth and latency of the built in ESP32 storage resulted in the flash storage component failing to meet some of the original design goals.

While the exact source of the problem is still unknown, some limitations originally encountered with the internal storage have been resolved. On such problem was the inability to allocate more than a few hundred kB of continuous memory within a 4 MB section of flash. This error was ultimately attributed to the development environment's inability to flash custom partition tables onto the ESP32. However, this discovery occurred too late to provide adequate time to address other timing related problems with the internal storage. As a result, song data is only stored in RAM and must be reloaded on every restart.

2.3.2. Verification

While the internal flash storage was not fully functional, certain aspects of the original design requirements were met. The requirement for 3 MB of space was, in theory, partially met due to the 4 MB of storage available on the device. Data could be written to, and read from, portions of flash, but not while in use with the rest of the firmware or in real time. As a result, the original design requirement of R/W speeds of at least 512 kB/s was not met.

Future steps to address this problem include further development and improved implementation of the provided ESP32 NVS library, as well as the use of external storage in the form of EEPROM or alternative flash storage. External storage would also provide the possibility of substantially increased capacity.

3. User Interface Module

3.1. Design Procedure

The user interface module is responsible for interacting with the user through the means of a single button mounted on the guitar. While away from the software module, the interface module will be the user's only way to control the settings of the guitar. The design originally called for three separate buttons for user interactions, but this was limited to a single reset button for the ESP32 to begin reading songs from the start again.

3.2. Push buttons

3.2.1. Design Details

The on-board push button provides a tactile way for the user to navigate through the guitar settings while they are away from a laptop. The button is mounted on the control board mounted on the body of the guitar. This position was chosen because it is close to the user's right hand, and is near the location where the user strums for ease of access. The user input is limited for simplicity of the device, though additional buttons can be added in the future.

3.2.2. Verification

This button was verified with a simple breadboard LED circuit that confirmed the button's continued use after several minutes of rapid, sustained pressing. The button had virtually no visible degradation nor significant change in resistance.

4. LED Output Module

4.1. Design Procedure

The LED output module consists of the LED driver and the LED array itself. The driver is responsible for turning a few control bits from the ESP32 into individually addressable high current outputs. The LED array is distributed across the 30 LEDs located on the five PCBs mounted along the neck of the guitar. The picture in Figure 4.1 shows a single fret circuit board with the LEDs and contacts for each string labeled. Figure 4.2 contains the schematic for the circuits to be mounted within each fret.

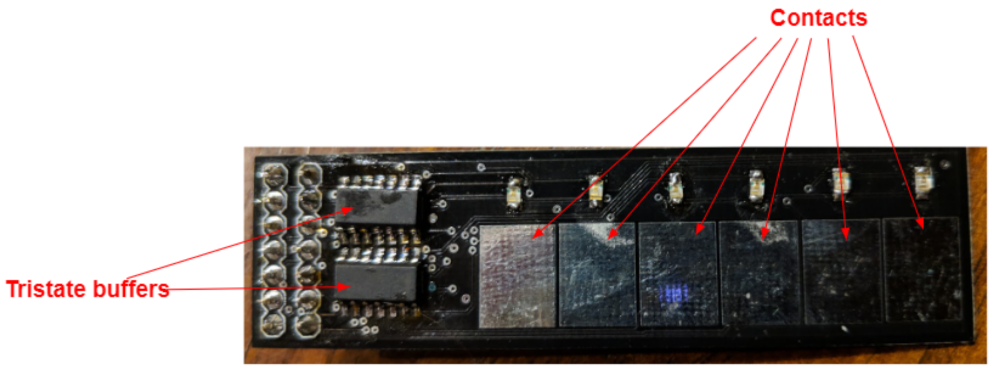


Figure 4.1: Diagram of the PCB for each fret with core components labeled.

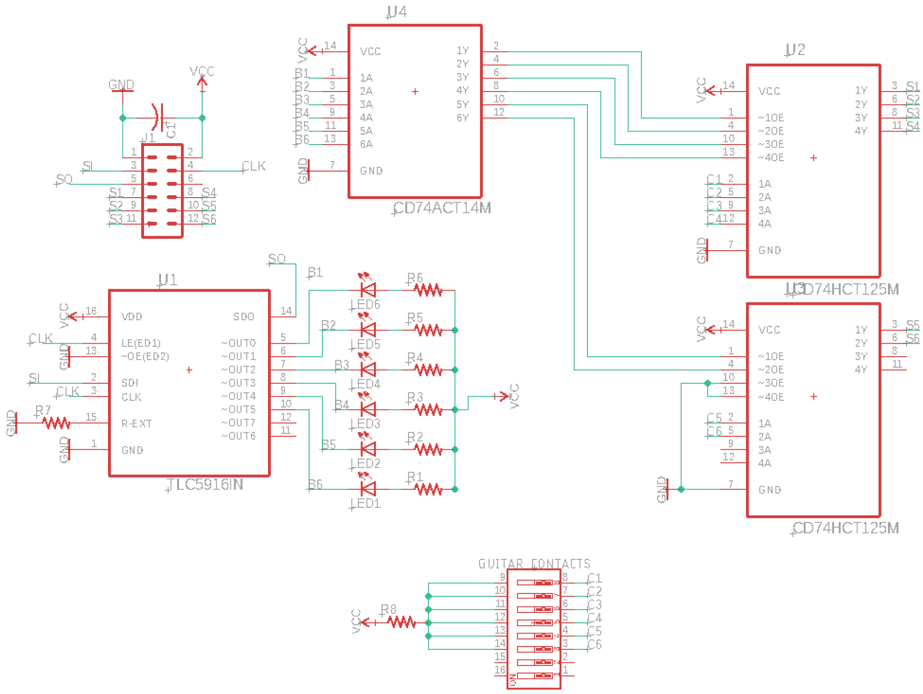


Figure 4.2: Schematic of LED driver and array (one per fret).

4.2. LED Driver

4.2.1. Design Details

The LED driver submodule is composed of five 8-bit constant current shift registers. The registers are distributed across five identical but independent PCBs (see figure 4.2 for schematic). The use of the constant current registers eliminated the need for current limiting resistors for each LED, which reduced the component count for the PCBs. The fret PCBs are daisy chained together, which means that additional boards can be added without requiring the need to expand the data bus or modify the design of any individual board. Each driver has a shift in port for incoming data, and a shift out port to send data further down the line.

For a theorectical board count of 20 boards, the LED drivers would need to shift through 160 bits of data to refresh the entire frame. In order to complete this while meeting our target F_{target} of 200 Hz (see Section 2: Control Module), the maximum clock frequency of the registers needs to be $\frac{160 \text{ bits}}{5 \text{ ms}} = 32$ kHz.

4.2.2. Verification

The LED driver was verified using the basic tests outlined in the requirements and verification table in appendix B. Specifically, the drivers were confirmed to be able to shift in all 30 bits of data to the five PCBs, and did supply the necessary current to clearly see all six

LEDs of a given board turn on, and the entire array of LEDs could light up without significant temperature increase in any of the chips.

4.3. LED Array

4.3.1. Design Details

The LED array is distributed across the fret PCBs. For the initial prototype, this consistents of 30 LEDs spread across five boards. The LEDs are connected to the outputs of the LED drivers, and are controlled through serially shifting data into the LED drivers. The LEDs themselves are compact, bright, and efficient. While the luminosity of the LEDs is substantial enough to be clearly seen from many meters away, the ergonomics of the boards results is some of the LEDs being difficult to see while the guitar is in use. Additionally, the bright, intense light of the LED can cause additional eye strain over long practice sessions. While the former design challenge is difficult to overcome within the tight physical constraints of the guitar, the latter can be addressed through the used of diffused LEDs. Diffused LEDs would provide a softer, more natural light source that is still bright, but causes less eye strain.

4.3.2. Verification

The LED array is clearly visible from as much as 10 m away, and the LEDs are about 1 mm tall, which is half the maximum height necessary as per the requirements and verification table. This satisfies all the necessary requirements laid out for the LED array itself.

5. Power Supply Module

5.1. Design Procedure

A lithium-ion (Li-ion) battery provides high power density for the system, and the Li-ion battery management sub-module ensures the battery stays within safe operating conditions at all times. Specifically, the 18650 Li-ion battery was chosen for its known record of safe functionality, though a thermistor is included to ensure operation within safe temperatures between 0 and 45°C. The voltage regulator steps down the variable Li-ion battery voltage to a constant 3.3 V ± 0.1 V for the rest of the system.

5.2. Li-ion Battery

5.2.1. Design Details

The Li-ion battery serves as the sole energy source for the guitar. Due to the number of LEDs that may be driven at the same time, high power density is also an important factor for battery selection. In addition, due to the convenience of the package size, durability, and safety, protected 18650 li-ion cells are used as the main power source.

5.2.2. Verification

In testing, the Li-ion batteries were confirmed to output under 150 mA during normal use with all LEDs active, and with a nominal capacity of 2500 mAh, even at 15% of the rated capacity, the chosen Li-ion batteries can power the device for the necessary two hours.

5.3. Li-ion Management

5.3.1. Design Details

Li-ion batteries provide large energy and power densities, but require specialized charging and discharging. A Li-ion battery management Integrated Circuit (IC) is used to monitor the voltage of the battery to protect against overvolting and undervolting, as well as monitoring the discharging of current. The battery is not be charged while connected to the guitar, but the battery management circuit is only responsible for ensuring safe discharge while in use. To protect against thermal runoff, a thermistor is embedded in the battery holder such that any excessive heat effectively disconnects the battery nodes.

5.3.2. Verification

The battery management system was tested and verified using the proper verification tests from the requirements and verification table in appendix B, including verification of the thermistor voltages as seen in appendix A.7.

6. Software Module

6.1. Design Procedure

The software module is the only module not located on the guitar (instead it resides on an external Bluetooth-enabled computer). The software module's responsibilities include generating the bytecode for LEDs from a MIDI file, transmitting the data to the ESP32 over Bluetooth, and then processing of the song data on the ESP32 itself (shown in appendix A.3). The decision was to have the device be able to convert MIDI files because they are an accessible way to store song data, and there are thousands of MIDI files available online for many well-known songs. Bluetooth was chosen as the transmission method due to ease of use and the ability for the device to then become wireless.

6.2. Music Conversion Program

6.2.1. Design Details

The music conversion program's role is to convert a given MIDI file into a bytecode that is understandable to the ESP32. As shown in appendix C.1, MIDIToBytes.cpp is compiled with a makefile, and once compiled, ./MIDIToBytes.exe can be called with a song name (equal to the name given in songname.mid for some MIDI file) and the instrument channel number as arguments, and it converts the MIDI file into a CSV with the given notes. The MIDI file format information from McGill University was used when building the MIDI converter portion of the program, which included most of the information used to build the parser [3]. The parser builds the CSV line-by-line with the MIDI event information from the MIDI file, adding the expected frame for an event, and the information about the event (note on, note off, system exclusive message, system resets, etc.) is added to the CSV. After fully parsing a MIDI file, the converter writes a binary file that contains one byte per fret per frame, with the first six bits of each byte representing an LED on a fret. The notes are determined by the fret order on the fret board, with a diagram of the note map example in appendix A.4. The frame number is determined by the LED array's refresh rate, and a calculation for the total size of a binary file in this format is given below:

$$\text{Binary file size (bytes)} = t * r * f$$
$$t = \text{song duration (s)}$$
$$r = \text{refresh rate (Hz)}$$
$$f = \text{number of fret PCBs}$$

For a 5-minute song with five fret PCBs and a refresh rate of 32 Hz, the binary file for this song will be 48 kB long.

6.2.2. Verification

To verify the music conversion program, a CSV is made of every song to ensure that the MIDI file is being properly parsed, and a binary file can be checked to confirm that the chords or individual notes match the expected frame array output. In all songs tested, the first and last 10 frames of the binary were checked for correctness, with code to time the duration of parsing. For Say It Ain't So by Weezer, the 4 minute and 19 second song can be parsed in less than 400 ms, and Hotel California by the Eagles, a six minute and 30 second song can be parsed in under 480 ms. This implies that up to 60, 5-minute songs can be parsed in under 30 seconds, far exceeding the requirements.

6.3. Bluetooth Transmission Program

6.3.1. Design Details

The Bluetooth transmission program is responsible for sending the binary file to the ESP32. This code, given in appendix C.2, takes in a binary file and transmits the data to the ESP32 over a serial port connection. This program is written in Python due to ease of use of serial Bluetooth I/O. From there, the ESP32 firmware ensures proper retrieval of the data.

6.3.2. Verification

Once transmitted to the ESP32, the data was confirmed to have been properly stored in the ESP32 frame buffer to verify correctness. This was tested by at least seven different songs of different lengths and confirmed correct by checking against the original binary files sent.

6.4. ESP32 Firmware

6.4.1. Design Details

The ESP32 must be flashed with a program to retrieve data, store it, and then play it back to the LEDs properly. When connected properly to a laptop, the ESP32 can easily receive data over a serial Bluetooth port. There are delays in the code due to Serial buffer hand-offs with the Bluetooth transmission program on the laptop, but once it has stored all transmitted bytes in the frame buffer, it can replay the songs in real-time. This code must handle properly shifting bytes and loading them onto each fret PCB, but this is done with simple control signals sent by the ESP32 control module to each board serially. A high-level flowchart of this portion of the software is shown in appendix A.5.

6.4.2. Verification

To confirm that the ESP32 firmware worked as expected, we probed the outputs of the ESP32 with an oscilloscope on a regular input stream, confirming that the proper control signals have been sent. This program was not described in the requirements and verification so does not have formal requirements, but the implied requirement of sending proper control bytes and reading binary data accordingly has been verified with proper LED array output after the final integration.

7. Sensing Module

7.1. Design Procedure

The sensing module is responsible for detecting when the user depresses a string using a contact per string per fret. Contacts for each potential finger position was chosen because having contacts next to the LEDs on each board seemed like an elegant solution to determining user accuracy. Another option was to verify that the user played the correct notes with external audio processing, but this would not have been able to tell if the user was playing the correct finger position since different positions on a guitar's fret board can produce the same note (on the same octave).

7.2. Circuit Board Contacts

7.2.1. Design Details

Each fret contains a small board that houses the LEDs and six bare contacts (one under each string). These contacts are connected to 3.3 V through a large pull up resistor. When the user depresses a string, the conductive guitar strings make contact with the pad. The guitar strings are connected to ground through a smaller (but still relatively large) pull down resistor. When the string makes contact with the pad, it pulls the voltage of the pad down to near 0 V. By measuring the voltage of the pads through the use of an Analog-to-Digital converter (ADC), the device can determine which locations the string is being depressed. Since the PCB is lower than the fret bars, they do not interfere with the functionality of the guitar.

The contacts are connected to the string data bus through tristate buffers. The contact is directly connected to the bus if the corresponding LED is turned on; otherwise, the tristate buffer maintains a high impedance output to prevent multiple contacts from attempting to write to the bus with different values.

7.2.2. Verification

The contacts were verified by testing an individual test PCB outside of the full system. This test PCB underwent the verification described in the requirements and verification table in appendix B. The contacts passed all continuity tests with the data bus when the contacts were not connected vs. tied to 0.0 V, confirming that they worked as designed. However, the tristate buffers did not have well-defined expected behavior in the final requirements and verification document, and this may have been a source of error in the final product. The LEDs sometimes exhibited a dim light when they should have been off, and this is likely due to high-voltage signal outputs from the ESP32 that were not a high enough voltage to actually turn off the tristate buffers completely. This, however, was only a problem in some LEDs, and can be fixed by adding a resistor in parallel to the LEDs to tie the voltage down properly.

7.3. ANALOG-TO-DIGITAL CONVERTER

7.3.1. DESIGN DETAILS

The Analog-to-Digital converter (ADC) sub-module is responsible for reading data from the string data bus and determining which pads are in contact with a string. This is done directly through the main ESP32, with six channels of input, as is required to read data from all six bits of the string data bus at the same time. The ESP32 processes the voltage values from the data bus and converts them to digital values that can be used in the control logic of the device. This was originally meant to be processed through another MSP430 microcontroller, but the ESP32 on-board processing was sufficient for the device's functionality, making the MSP430 obsolete for our purposes.

7.3.2. VERIFICATION

The data processing was confirmed on the ESP32 with a basic serial print of the input from the data bus to the ESP32. Due to direct processing in the ESP32, latency and voltage power levels did not need to be verified with the same tests as described in appendix B, but basic verification of timing ensured that the ESP32 could process the data bus input in near-real time.

8. Costs

The hourly development cost is taken from a reference for the average yearly salary of computer engineering graduates as reported by the University of Illinois at Urbana-Champaign, extrapolating to 2018 with a 3% salary raise from 2017 [4]. The University's average reported salary for Computer Engineering graduates was $88,000 in 2017, which is extrapolated to $90,640 in 2018 [5]. With a 40 hour work week and 50 work weeks (assuming two weeks paid vacation), the average hourly salary of a Computer Engineering graduate from UIUC comes out to roughly $46 per hour. Throughout the semester, the average number of work hours per week was 15 hours per person per week. Therefore, the total estimated labor costs for the semester is:

$$\frac{\text{2 people}}{\text{team}} * \frac{\$46}{\text{hour}} * \frac{\text{15 hours}}{\text{week}} * \frac{\text{16 weeks}}{\text{1 semester}} * 2.5 = \$55,200$$

The final estimated cost for our prototype, not including the guitar, is roughly:

Part	Cost
ESP32 development board	$19.50
Push button	$0.25
TLC5919N x5	$ 5.63
CD74HCT125 x 10	$2.40
LED x 30	$3.00
18650 Li-ion battery	$14.95
PCB manufacturing x10	$5.00
Wire (36 ft)	$5.00
Total	**$55.73**

At bulk prices, the cost per unit would roughly be:

Part	Cost
ESP32 development board	$6.80
Push button	$.12
TLC5919N x5	$ 3.50
CD74HCT125 x 10	$1.75
LED x 30	$0.30
18650 Li-ion battery	$7.00
PCB manufacturing x10	$5.50[1]
Wire (36 ft)	$1.00
Total	**$25.97**

[1] The bulk price is more expensive than the prototyping cost due to incentives by manufacturer that don't scale at volume

9. Conclusion

9.1. Accomplishments

The project as a whole encountered a number of setbacks throughout the course of the semester, but most of the functionality for each module has been proven in isolation. Starting with the power module, the battery can successfully power the device with low enough power output to power the device for the required duration for a typical practice session, and the battery management system has been proven to shut off at excessive temperatures. The control module successfully communicates with the off-board laptop through Bluetooth, and can correctly play notes as encoded by the files it receives. The LEDs successfully light up with the control signals sent by the LED driver, and the bits shift through the serially-connected boards as designed. Finally, the software module parses any given MIDI file as needed, and can convert the input MIDI file for a typical 5-minute song into a device-readable byte array within 200 ms. Overall, the device can provide many of the features defined in the design document, including processing input MIDI files, sending the data over Bluetooth to the device, having the device read and record the data, and transmitting the correct bytes to the fret PCBs as needed. However, there are some challenges explained in the next section that prevented Guitar Buddy from performing certain functionalities.

9.2. Challenges

The Guitar Buddy has had many successful outcomes, but a few minor challenges prevented the device from exhibiting every functionality from the original design. First, there were difficulties writing large files to the ESP32. Although the ESP32 purports to have 4 MB of internal memory, this is not easily accessible to the user without advanced implementation of a file system, so the device was unable to hold the full data of the number of songs desired. Without the addition of EEPROM or other external memory, the device could only hold enough information for half of one 5-minute song given the current data organization. In addition, the LED array has had difficulties with the turnoff voltage for the LEDs. Specifically, as described in Section 7: Sensing Module, the LEDs appeared to have a dim light at times when the tristate buffer did not receive the necessary voltage from the input signal. This could be mitigated with resistors in parallel with the LED to adjust the turn-on voltage accordingly.

9.3. Ethics and Safety

As a consumer-oriented device, it is especially important that the Guitar Buddy system does not harm any users, other persons, other devices or other objects. In compliance with code one of the IEEE Code of Ethics, the Guitar Buddy team maintained safe engineering practices to mitigate any potential safety concerns to the user [6]. One such safety concern was the Li-ion battery source for the apparatus. These batteries can be damaged if the temperature is outside the range of 0 - 130 °C, and damage to the battery can result in potentially catastrophic failure and harm to the user [7]. A basic thermistor is incorporated into the battery holder to verify that the battery remains within normal operating range. In addition a voltage regulator ensures that the voltage output does not exceed the rated voltage.

In addition, there is a potential safety hazard in connecting the guitar strings to the sensing circuit. To prevent any harm to the user, the strings are all tied to ground such that there is no case of current passing between strings, potentially injuring the user. In support of expectations enumerated by the National Institute of Standards and Technology, if the device is ever offered to users in the future, there will be warnings of potential hazards from improper use of the device [8].

With the mapping of music to the LEDs, one difficulty is ensuring that the artists for songs are properly attributed for their musical works. In support of the writers of musical works used in this project, and following section 1.5 of the ACM Code of Ethics, credit is given to the writers of any songs used for the Guitar Buddy system, and MIDI files are only taken from properly attributable sources [9].

REFERENCES

[1] TakeLessons.com. What people pay for music lessons, annual report. [Online]. Available: https://support.takelessons.com/hc/en-us/article_attachments/200377329/What-People-Pay-for-Music-Lessons.pdf

[2] *ESP32 Technical Reference Manual*, Espressif Systems, 9 2018, version 3.8.

[3] M. University. Standard midi-file format spec. 1.1, updated. [Online]. Available: http://www.music.mcgill.ca/~ich/classes/mumt306/StandardMIDIfileformat.html#BMA1_

[4] S. Miller. 2018 salary forecast: Smaller real wage increases in the u.s. and globally. [Online]. Available: https://www.shrm.org/resourcesandtools/hr-topics/compensation/pages/2018-salary-forecast-us-global.aspx

[5] U. of Illinois at Champaign-Urbana. Salary averages. [Online]. Available: http://ecs.engineering.illinois.edu/files/2018/03/Engineering_Report_2016-2017_FINAL.pdf

[6] IEEE.org. Ieee code of ethics. [Online]. Available: https://www.ieee.org/about/corporate/governance/p7-8.html?WT.mc_id=lp_ab_ico

[7] BatteryUniversity.com. Lithium ion safety concerns. [Online]. Available: https://batteryuniversity.com/learn/archive/lithium_ion_safety_concerns

[8] MIST.gov. A guide to united states electrical and electronic equipment compliance requirements. [Online]. Available: https://www.nist.gov/sites/default/files/documents/2016/11/15/11-04-2016-8118-guide_to_us_electrical_and_electronic_products.pdf

[9] ACM.org. Acm code of ethics and professional conduct. [Online]. Available: https://www.acm.org/code-of-ethics

A. Additional Diagrams

A.1. Guitar Body Picture

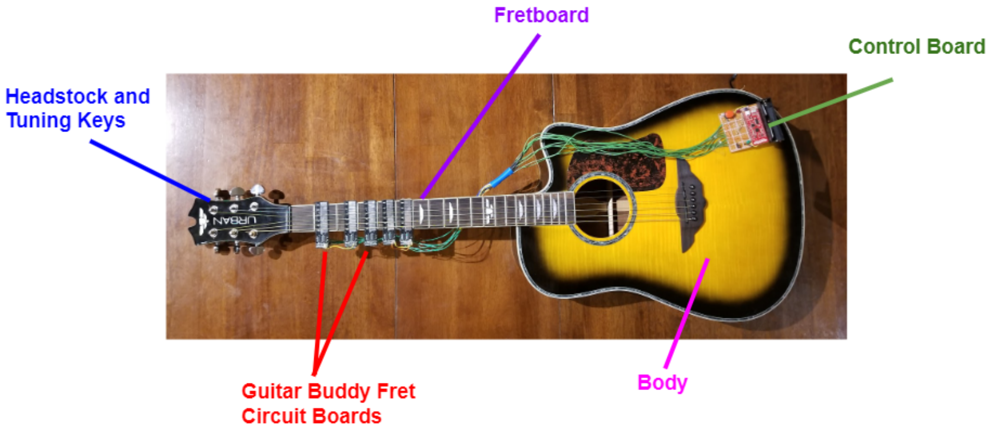


Figure A.1: Diagram of Guitar Buddy with control board and fret PCBs attached.

A.2. Model Picture

Figure A.2: Model of Austin Born holding the guitar and integrated Guitar Buddy system.

A.3. Software Module

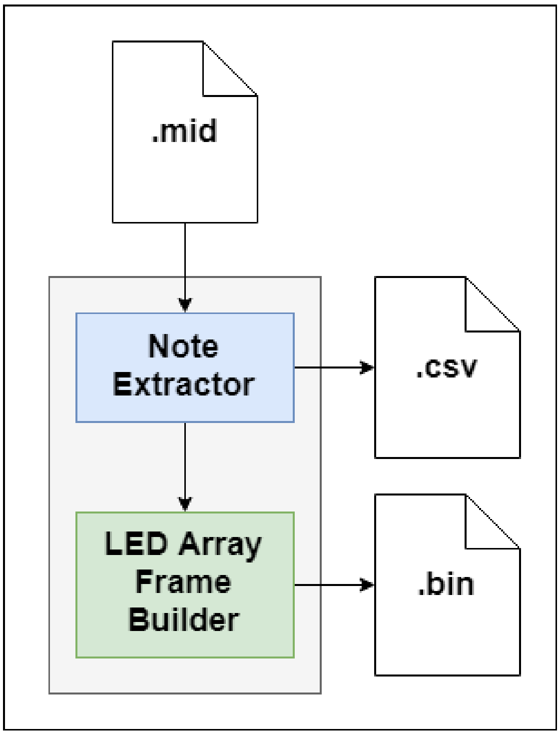


Figure A.3: Diagram illustrating the main mechanism of the software module.

A.4. Guitar Fret Notes map

		Strings					
		E2	A2	D3	G3	B3	E4
Frets	1	F2	B^b2	E^b3	A^b3	C4	F4
	2	G^b2	B2	E3	A3	D^b4	G^b4
	3	G2	C3	F3	B^b3	D4	G4
	4	A^b2	D^b3	G^b3	B3	E^b4	A^b4
	5	A2	D3	G3	C4	E4	A4
	6	B^b2	E^b3	A^b3	D^b4	F4	B^b4
	7	B2	E3	A3	D4	G^b4	B4
	8	C3	F3	B^b3	E^b4	G4	C5
	9	D^b3	G^b3	B3	E4	A^b4	D^b5
	10	D3	G3	C4	F4	A4	D5

Figure A.4: Map of notes on the fret board PCBs.

A.5. ESP32 Firmware Flowchart

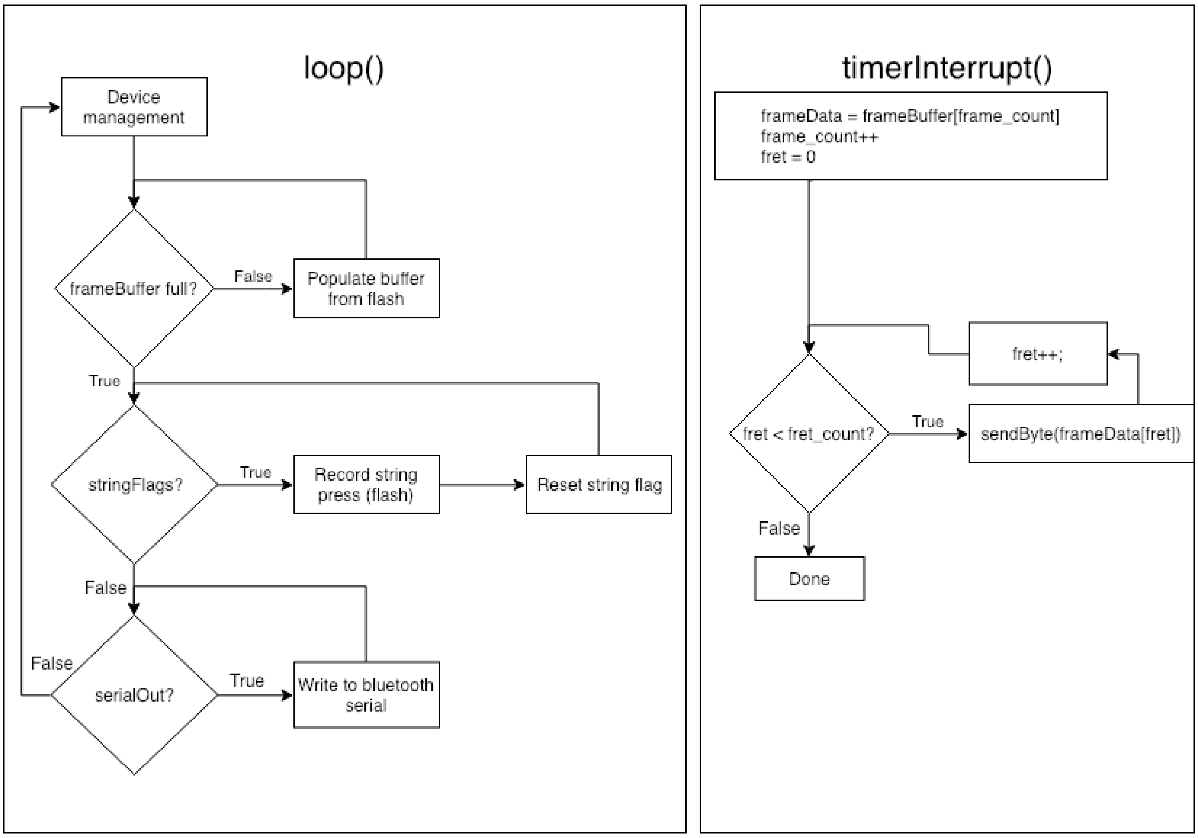


Figure A.5: High-level flowchart for firmware running on the control module (some features excluded).

A.6. Control Board

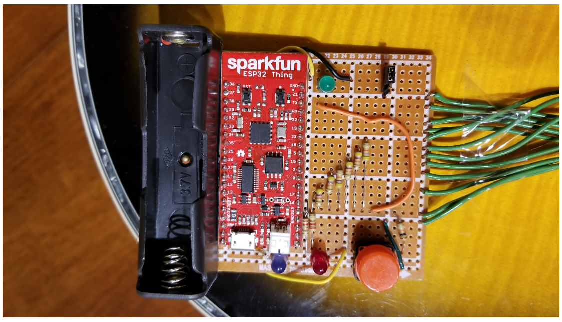
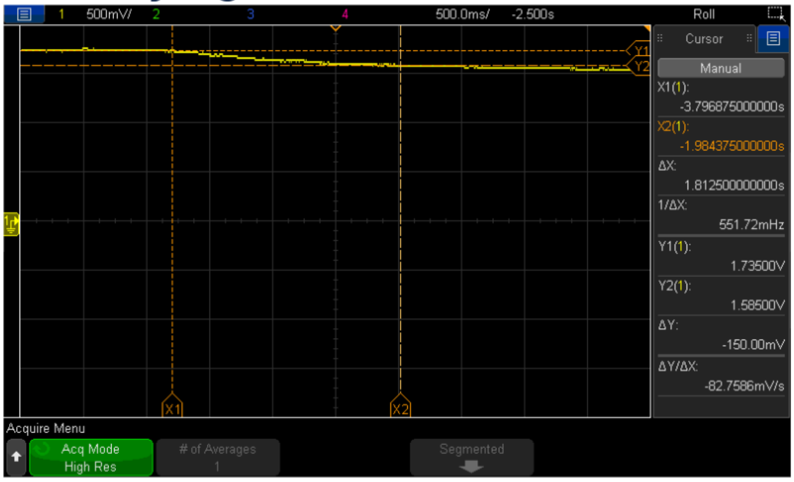


Figure A.6: Picture of the control board used on the guitar.



$$R_{therm_{RT}} = 10512\,\Omega$$
$$R_{cutoff} = 8850\,\Omega$$
$$R_{pullup} = 9754\,\Omega$$
$$V_{cutoff} = \frac{R_{cutoff}}{R_{pullup} + R_{cutoff}} \approx 1.58V$$

Figure A.7: Diagram of the verification of voltage ranges for the thermistor.

B. Requirements and Verification Table

Microcontroller Requirements and Verification	
Requirement	**Verification**
1. Powered by input voltage of 3.3 V ± 0.1 V.	1. a) Power controller with variable voltage source, starting at V_{in} = 3.2 V. b) Upload code setting all output pins to high output. c) Probe each output voltage, ensuring V_{out} = 3.3 V ± 0.1 V. d) Upload song data through Bluetooth connection and ensure the microcontroller doesn't crash. e) Set input voltage to 3.4 V and repeat b) through d).
2. Operating current I_{max} ≤ 500 mA at 3.3 V ± 0.1 V during radio transmission.	2. a) Power controller with 3.3 V ± 0.1 V, attaching an ammeter in series with supply. b) Connect variable voltage source to input pin 1 on controller at 0 V and ground to the GND pin. c) Execute program with constant wireless transmission outputting pin 1 to terminal. d) Alternate variable voltage source between digital low and digital high, and confirm that controller is sending wireless signal. e) Ensure I_{max} ≤ 500 mA.
	Continued on next page

Microcontroller Requirements and Verification (Continued)	
Requirement	**Verification**
3. Operating current $I_{typical} \leq 200$ mA at 3.3 V ± 0.1 V during non-radio operation.	3. a) Power controller with 3.3 V ± 0.1 V, attaching an ammeter in series with supply. b) Execute regular, non-communication program. c) Ensure $I_{max} \leq 200$ mA.
4. R/W compatibility with external SD storage.	4. a) Power controller with 3.3 V ± 0.1 V. b) Insert SD card into the SD card reader. c) Execute SD card test program to write to and read from SD card. d) Output data from SD card to terminal to verify that data is stored on SD card.
5. At least 10 GPIO pins for communication with other modules.	5. a) Power controller with 3.3 V ± 0.1 V. b) Connect variable voltage source to first GPIO pin on controller at 0 V and ground to the GND pin. c) Alternate variable voltage source between 0 V and 3 V, and confirm that controller is receiving signal from pin. d) Repeat previous step for at least 9 other GPIO pins on the controller. e) Disconnect variable voltage source from pins and connect a voltmeter to pin 1 and the ground to GND pin. f) Run pin output program to verify that the pin outputs 0 V when low and 3.3 ± 0.1 V when high. g) Repeat previous step for the other 9(+) GPIO pins.
Continued on next page	

Microcontroller Requirements and Verification (Continued)	
Requirement	**Verification**
6. At least 112.5 kB of SDRAM.	6. a) Power controller with 3.3 V ± 0.1 V. b) Write data to at least 112.5 kB of on chip RAM c) Verify that all data was written and is accessible.

Flash Storage Requirements and Verification	
Requirement	**Verification**
1. 3 MB or larger storage capacity.	1. a) Connect the flash storage to the microcontroller and power the system with 3.3 V ± 0.1 V. b) Write data to at least a 3MB portion of the flash storage. c) Read the written data to ensure that all data is still accessible.
2. R/W speeds of at least 512 kB/s.	2. a) Connect the flash storage to the microcontroller and power the system with 3.3 V ± 0.1 V. b) Write data to at least a 1MB portion of the flash storage. Record the transfer time and ensure that the average write speed is at least 512 kB/s. c) Read the written data. Record the transfer time and ensure that the average read speed is at least 512 kB/s.

Push Button Requirements and Verification	
Requirement	**Verification**
1. Durable and reliable operation; >1000 click life span.	1. a) Mount the switch to a convenient platform (such as a breadboard). b) Time the duration it takes to press the button 100 times. c) Continually press the button for 15 times as long as it took to depress it 100 times. d) Repeat for two other switches. e) Connect all three switches to 5V on one input, and a pulldown resistor to ground on the other. f) Press each switch and measure the voltage of the output of the switch to ensure it is connecting to 5V.

LED Driver Requirements and Verification	
Requirement	**Verification**
1. At least 60 bits of storage	1. a) Connect the LED driver sub module to the microcontroller submodule. Power the system with 3.3 V ± 0.1 V. b) Shift in 60 bits of digital low through the shift registers. Verify that all outputs are digital low. c) Shift in 60 bits of digital high through the shift registers. Verify that all outputs are digital high.
	Continued on next page

LED Driver Requirements and Verification (Continued)	
Requirement	**Verification**
2. Minimum $f_{clock_{max}}$ of 3.2 kHz	2. a) Connect the LED driver sub module to the microcontroller submodule. Power the system with 3.3 V ± 0.1 V. b) Shift in 60 bits of alternating digital high and low signals at a frequency of at least 3.2 kHz. c) Verify that all outputs match the expected value.
3. Supply a minimum I_{max} of 50 mA ± 1 mA per channel while maintaining a temperature below 50°C	3. a) Connect the LED driver sub module to the microcontroller submodule. Connect a LED of the LED array to the first output of the LED driver. Power the system with 3.3 V ± 0.1 V. b) Set the current output of the chip to 50 mA ± 1 mA by using the current trim input on the constant current shift register (specific to IC) c) Verify with ammeter that output current is 50 mA ± 1 mA. d) Allow to run for at 5 minutes. e) Verify the output current is still 50 mA ± 1 mA. f) Verify the temperature of the IC is below 50°C.
Continued on next page	

LED Driver Requirements and Verification (Continued)	
Requirement	**Verification**
4. Minimum I_{max} of 400 mA ± 8 mA per chip	4. a) Connect the LED driver sub module to the microcontroller submodule. Connect one LED of the LED array to each output of the LED driver (for 8 total). Power the system with 3.3 V ± 0.1 V. b) Set the current output of the chip to 50 mA per channel by using the current trim input on the constant current shift register (specific to IC). c) Verify the total output current is at least 400 mA ± 4 mA. d) Allow to run for at 5 minutes. e) Verify the total output current is still 400 mA ± 4 mA. f) Verify the temperature of the IC is below 50°C.

LED Array Requirements and Verification	
Requirement	**Verification**
1. LEDs are easily visible from at least 1 m away without being uncomfortablely bright	1. a) Connect a LED to an output of the LED driver module. Power the system with 3.3 V ± 0.1 V. b) Power the LED with no more than 50 mA. c) Check the LED is visible from 1 m ± 10 cm.
2. LEDs are less than 2 mm tall (surface mount)	2. a) Measure the height of the LED with a caliper. Ensure that it is less than 2mm.

Lithium-Ion Battery Requirements and Verification	
Requirement	**Verification**
1. Peak I_{out} must be at least 1.5 A continuously for 1 minute while maintaining a temperature under 60°C	1. a) Connect the battery submodule to a load that draws at least 1.5 A. b) Verify that output current is at least 1.5 A. c) Allow to run for 1 minute. d) Verify that the output current is still at least 1.5 A, and that the battery temperature is under 60°C.
2. Minimum 1500 mAh capacity	2. a) Connect the module to a 25 Ω load and an ammeter b) Allow the battery to drain until battery module cuts power (due to low voltage). c) Calculate the capacity of the battery and verify that it is at least 1500 mAh
3. Weight must be under 250 g	3. a) Weigh the battery; confirm the battery weights under 250 g
4. Battery must be under 10 cm along its largest axis, and less than 2 cm thick.	4. a) Use a mechanical caliper to measure the longest axis. Ensure that it is less than 10 cm b) Use a mechanical caliper to measure the diameter of the battery. Ensure that it is less than 2 cm.

Lithium-Ion Management Requirements and Verification	
Requirement	**Verification**
1. Protect the battery from over discharging by disconnecting the battery when the battery voltage dips under 3.0 V ± 0.05V.	1. a) Fully charge the battery, and then connect it to the battery management system. b) Connect the battery to a voltmeter. c) Connect the module to a 10 Ω ± 1 Ω load. d) Allow to discharge until the battery management system disconnects the battery and stops outputting power. Ensure the battery voltage does not drop below 2.95 V for any period of time.
2. Battery management must not exceed 80 °C under maximum load (note: this temperature applies to the battery management IC and any components for the management submodule. The battery itself has a lower maximum temperature, specified in the battery submodule.)	2. a) Fully charge the battery, and connect it to the battery management submodule. b) Apply a load to the system that results in a 1.5 A ± 0.1 A current draw. c) Allow to run for 1 minute. d) Measure the temperature of the battery management system and ensure that it is under 80°C. e) Apply a new load to the system that results in a 0.5 A ± 0.05 A current draw. f) Allow to run for 30 minutes. g) Measure the temperature of the battery management system and ensure that it is under 80°C.

Continued on next page

Lithium-Ion Management Requirements and Verification (Continued)	
Requirement	**Verification**
3. Battery management system must cut power if temperature of battery exceeds 60°C.	3. a) Remove the thermistor from the battery housing. b) Connect a charged battery to the battery management system. c) Apply a 100 Ω± 10 Ωload across the output of the battery management system. d) Use an ammeter to confirm that there is current flowing through the load. e) Using a heat gun and a thermometer, heat the thermistor to 60°C to simulate a warming battery. Do not do this while the thermistor is still attached/next to the battery to avoid unnecessarily putting the battery at risk. f) When the thermistor reaches 60°C ± 1°C, use the ammeter to confirm that the battery management system cut power from the battery.
4. Battery management system must cut power if current draw exceeds 2.5 A ± 0.1 A for 0.5 s ± 0.5 s.	4. a) Connect a charged battery to the battery management system. b) Probe the output of the battery management system with a voltmeter. c) Connect with outputs of the battery management system with a high power 1 Ω± 0.1 Ω load. Start a timer. d) Verify the battery management system cuts power to the output within 0.5 s ± 0.5 s.

Music Conversion Program Requirements and Verification	
Requirement	**Verification**
1. Generate the proper bytecode and parity bits	1. a) Using the conversion program, generate the bytecode for a sample song. b) Using a USB connection (or Bluetooth connection if it has been independently confirmed to work) transfer the bytecode the guitar. c) Visual compare the LED indicated chord pattern against tabs for the same sample song. d) Confirm that the displayed chords match.
2. Capable of converting 5 minutes worth of song notes within 30 seconds	2. a) Pick any 5 minute ± 15 s music video for YouTube. b) Use the conversion program to generate the bytecode. Time how long the software takes to run. c) Repeat steps a) and b) for 4 other songs.

Bluetooth Transmission Program Requirements and Verification	
Requirement	**Verification**
1. Verify and transmit bytecode song data to ESP32 with 95% accuracy, and that any failed transmissions are resent.	1. a) Use the conversion software to generate the bytecode for 30 five minute songs. b) Turn on the guitar and place it 2 m ± 20cm away. c) Transmit the bytecode data to the ESP32. d) Log any failed transmissions. e) Clear the flash storage on the ESP32, and repeat step a)-d) four more times. f) The total number of failed transmissions can not exceed 95%. Any failed transmissions must also be reattempted automatically.

Continued on next page

Bluetooth Transmission Program Requirements and Verification (Continued)	
Requirement	**Verification**
2. Transmission rate of at least 500 kbps at 2 m distance	2. a) Turn on the guitar and place it 2 m ± 20 cm away. b) Transmit 2 MB worth of song data. Record the time it takes. c) Ensure the average data transmission rate is at least 500 kbps.
3. Adjust settings on the guitar with ≤ 1.5 s latency while laptop is 5 m away.	3. a) Turn on the guitar and place it 2 m ± 20 cm away. b) Connect the guitar via Bluetooth to the transmission program. c) Change a setting by using the transmission program. d) Use a software timer to measure the time between input the setting and receiving the confirmation packet from the guitar. e) Verify the latency is ≤ 1.5 s.

Copper Contacts Requirements and Verification	
Requirement	**Verification**

Continued on next page

Copper Contacts Requirements and Verification (Continued)	
Requirement	**Verification**
1. The copper contacts must endure the equivalent of at least 1,000 hours of play time (without more than 20% increase in the resistance.	1. a) Connect one of the 6 guitar strings of different gauges and a copper contact to the input and output respectively of an ammeter. b) Sample and record the resistance of the string-contact circuit with at least 10 different locations of contact between string and copper contact. c) Simulate 1,000 hours of play time by rubbing the string against the contact rigorously for at least 15 minutes. d) Once again, sample and record the resistance of the string-contact circuit with at least 10 different locations of contact between string and copper contact. e) Confirm that the maximum resistance after the rubbing procedure is no more than 20% greater than the maximum resistance beforehand. f) Repeat this full procedure for each of the other 5 different gauges of guitar string.
2. Closed circuit current must be below 1 mA.	2. a) Connect a battery to the system and turn it on. b) Temporary insert an ammeter between the pull down/current limiting resistor between the guitar strings and ground. c) Close the circuit by depressing the guitar string onto any copper contact pad. d) Ensure that the current is less than 1 mA. e) Repeat for each of the 6 different guitar string gauges.

Continued on next page

Copper Contacts Requirements and Verification (Continued)	
Requirement	**Verification**
3. Voltage of the copper contact must equalize to <0.2 V within 1 ms of firm contact with the guitar string.	3. a) Connect a battery to the system and turn it on. b) Attach on probe of an oscilloscope to the ground and another to the first bit data bus. c) Shift in a high bit to the shift register on the top fret's top E string. This will connect the top fret's left most copper contact to the data bus. d) Depress the string and measure the time it takes for the data bus voltage to drop to ≤ 0.2 V. Ensure that it is less than 1 ms. e) Repeat for each of the 6 different guitar string gauges.
4. Contact must be disconnected from string data bus through the tristate buffer when the corresponding LED is turned off.	4. a) Connect a battery to the system and turn it on. b) Turn on an LED on the top fret by shifting in a single high bit. c) Verify that the corresponding string bus's voltage is digital high. d) Depress the string corresponding with turned on LED. e) Verify that the corresponding string bus voltage is digital low. f) Repeat for each of the 6 different guitar string gauges.

Continued on next page

Copper Contacts Requirements and Verification (Continued)	
Requirement	**Verification**
5. Resistance between copper contact and guitar strings contact point must contribute less than 10 Ω to resistance.	5. a) Connect a string and copper contact in series to an ammeter. b) Firmly depress the string against the contact. c) Ensure that the net resistance between 1 cm away from the contact point on the string and the opposite end of the copper contact is less than 10 Ω. d) Repeat for each of the 6 different guitar string gauges.

Analog-to-Digital Converter Requirements and Verification	
Requirement	**Verification**
1. Powered by 3.3 V ± 0.1 V	1. a) Connect the V_{cc} of the ADC submodule to a variable voltage source. b) Set the V_{cc} to 3.2 V. c) Connect an input on the ADC to ground, and verify the binary output from the ADC is 0 V ± 0.1 V. d) Connect an input on the ADC to 3.2 V, and verify the binary output from the ADC is 3.2 V ± 0.1 V. e) Set the V_{cc} to 3.4 V. f) Connect an input on the ADC to ground, and verify the binary output from the ADC is 0 V ± 0.1 V. g) Connect an input on the ADC to 3.4 V, and verify the binary output from the ADC is 3.4 V ± 0.1 V.
	Continued on next page

Analog-to-Digital Converter Requirements and Verification (Continued)	
Requirement	**Verification**
2. ADC obtains precision of 0.1 V of the V_{in} within 1 ms.	2. a) Power ADC with a variable voltage source at V_{in} = 3.3 V. b) Connect ADC input to function generator with a square-wave function set to 250 Hz frequency with V_{low} = 0 V and V_{high} = 3 V, and PWM set to 50%. c) Determine the ADC binary output for 0.03 V and 3 V from the ADC. d) If the binary output for 0.03 V is n bits long, find the most significant bit less than the nth-least significant bit of the 3 V binary output which is 1. Determine the ADC pin that corresponds to this bit. e) Connect the function generator and the determined ADC pin to an oscilloscope. f) Confirm that the pin output switches high within 1 ms of the rising edge of the square wave.
3. Minimum 8 GPIO pins (including 6 allocated to ADC)	3. a) Power ADC with a variable voltage source at V_{in} = 3.3 V. b) Connect ADC input to function generator with a saw-tooth function set to 0.5 Hz frequency with V_{low} = 0 V and V_{high} = 3 V. c) Connect the function generator to an oscilloscope. d) For each of at least 8 GPIO pins, connect the pin to the oscilloscope and confirm that the pin switches between low and high at a regular interval. If the pin appears to remain high, steadily increase the function frequency and observe if the pin simply switches too frequently to be seen at 0.5 Hz.

Requirements and Verification Point Assignments		
Module	**High-Level Requirement**	**Points**
Control Module	• Microcontroller must manage wireless communication, read song data from flash storage, and generate control signals. • The microcontroller must be able to play 5-minute songs over the course of at least 2 hours of play time.	15
User Interface Module	• Must provide reliable interface buttons for the user to maneuver the device's settings.	5
LED Output Module	• LED array must be able to hold 60 bits of lighting information and refresh within 25 ms.	10
Power Supply Module	• Power supply must be powerful enough to keep the device running continuously for at least 2 hours. • Battery management system must keep battery within safe operating range.	5
Software Module	• Music conversion program must take online input (for example as MIDI files) and convert it to a format to be transmitted to the device. • Transmission program must send packets of data to device while ensuring data is not lost in transmission.	5

Continued on next page

Requirements and Verification Point Assignments (Continued)		
Module	**High-Level Requirement**	**Points**
Sensing Module	• Copper contacts should provide robust sensing points throughout the duration of the device's lifespan, without significant degradation. • Analog-to-Digital Converter should be precise enough to read the proper voltage changes.	10

C. Core Programs

C.1. MIDIToBytes.cpp for MIDI Conversion

```
/* MIDI to CSV and Streamable Binary format
 *
 * By Austin Born, Fall 2018
 *
 * C++ program to convert notes in MIDI files to a readable CSV and a
 * compressed byte map format to send to the ESP32.
 * The byte map will contain basic header information on song name,
 * tempo, and then a sequence of frames for the entire LED array.
 * Each byte in a frame represents one fret of the guitar, so a
 * single frame will have n bytes of data where n is the number of frets.
 * If there are roughly 32 frames per second, then the byte map for a
 * 5-minute song will be ~100 kB.
 */

/* About the MIDI Format:
 * Format:
 * <Header Chunk> = <MThd><length><format><ntrks><division>
 * <Track Chunk> = <MTrk><length><MTrk event>+...
 * <MTrk event> = <delta-time><event>
 * <event> = <MIDI event> | <sysex event> | <meta-event>
*/

//Include external libraries
#include <unistd.h>
#include <Windows.h>
#include <fstream>
#include <iostream>
#include <queue>
#include <string>
#include <ctime>
#include <cmath>
#include <sstream>
#include <map>
#include <iomanip>
#include "MIDIToBytes.h"

using namespace std;

//Initialize note and octave lists
string notes[12] = {"C", "C#", "D", "D#", "E", "F", "F#", "G", "G#", "A", "A#", "B"};
string octaves[11] = {"0", "1", "2", "3", "4", "5", "6", "7", "8", "9", "10"};

//Initialize note map for binary format
map<string, int> note_map = {//Notes below lowest fret remapped
                              {"C2", 25}, {"C3", 25},
                              {"C#2", 33}, {"C#3", 33},
                              {"D2", 41}, {"D3", 41},
                              {"D#2", 49}, {"D#3", 49},
                              {"E2", 18}, {"E3", 18},
```

```
                            {"F2", 8}, {"F3", 26},
                            {"F#2", 16},
                            {"G2", 24},
                            {"G#2", 32},
                            {"A2", 40},
                            {"A#2", 48},
                            {"B2", 17},

                            //Frets 2-7
                            {"F#3", 16}, {"B3", 17}, {"E4", 18}, {"A4", 19}, {"C#5", 20}, {"F#5", 21},
                            {"G3", 24}, {"C4", 25}, {"F4", 26}, {"A#4", 27}, {"D5", 28}, {"G5", 29},
                            {"G#3", 32}, {"C#4", 33}, {"F#4", 34}, {"B4", 35}, {"D#5", 36}, {"G#5", 37},
                            {"A3", 40}, {"D4", 41}, {"G4", 42}, {"C5", 43}, {"E5", 44}, {"A5", 45},
                            {"A#3", 48}, {"D#4", 49}, {"G#4", 50},/*C#5 above*/{"F5", 52}, {"A#5", 53},

                            //Notes above highest fret remapped
                            {"B5", 35},
                            {"C6", 43},
                            {"C#6", 20}};

//Other constants
const int BUFFER_SIZE = 4;
const int byte_frame_freq = 32;

int main(int argc, char** argv){

    //Input checking
    if(argc != 3){
        cout << "Incorrect number of arguments. Requires 2 arguments: <Song name> <note channel>" << endl;
        return 0;
    }

    int channel_num = strtol(argv[2], NULL, 10);
    if(channel_num < 0 || channel_num > 16){
        cout << "Given note channel is outside total number of channels." << endl;
        return 0;
    }

    //Initializations
    char buf [BUFFER_SIZE];
    int i = 0;

    //Start clock
    clock_t begin = clock();

    //Open MIDI file
    fstream infile;
    string infile_name = "MIDI_Files/" + string(argv[1]) + ".mid";
```

```
    if (FILE *file = fopen(infile_name.c_str(), "r")) {
        fclose(file);
    } else {
        cout << "File does not exist" << endl;
        return false;
    }
    infile.open(infile_name, fstream::in | ios::binary);

    //Create output file
    fstream outfile;
    string outfile_name = "CSV_Files/" + string(argv[1]) + ".csv";
    remove(outfile_name.c_str());
    outfile.open(outfile_name, fstream::in | fstream::out | fstream::trunc);

    //Get MIDI file length
    infile.seekg(0, infile.end);
    long file_length = infile.tellg();
    long bytes_left = file_length;
    infile.seekg(0, infile.beg);

    //Get "MThd", but do nothing with it
    readFromFile(infile, buf, 4, bytes_left);

    //Get length of Header file
    readFromFile(infile, buf, 4, bytes_left);
    long length = buf[0];
    for (i = 0; i < 4; i++)
        length = (length << 8) + (unsigned char)buf[i];

    //Get format of Header file
    readFromFile(infile, buf, 2, bytes_left);
    short format;
    for (i = 0; i < 2; i++)
        format = (format << 8) + (unsigned char)buf[i];
    outfile << "File format - " << (unsigned int)format << endl;

    //Get number of tracks
    readFromFile(infile, buf, 2, bytes_left);
    short ntrks;
    for (i = 0; i < 2; i++)
        ntrks = (ntrks << 8) + (unsigned char)buf[i];
    outfile << "# of Tracks - " << ntrks << endl;


    //Initialize Time Variables
    bool tpq_timing = false;
    long fps, tpf, tpq, tempo;
    double time_multiplier = 0;
    short division;

    //Get time division
    readFromFile(infile, buf, 2, bytes_left);
    for (i = 0; i < 2; i++)
        division = (division << 8) + (unsigned char)buf[i];
```

```
    //If division bit 15 = 1, division[14:8] is negative fps, division[7:0] is ticks per
     frame
    if(division & 0x8000){
        fps = -(int)(division & 0x7F00);
        tpf = division & 0x00FF;
        outfile << "Frames/second - " << fps << endl;
        outfile << "Ticks/frame - " << tpf << endl;
    }
    //Else, division[14:0] is ticks/quarter note
    else{
        tpq_timing = true;
        tpq = division & 0x7FFF;
        outfile << "Ticks/quarter note - " << tpq << endl;
    }

    //Initialize track variables
    long trk_length;
    long trk_bytes_left;
    float total_time;
    int round_time;
    int track_num = 0;
    unsigned char prev_status;

    //Loop through each track
    while(bytes_left > 0){

        //Increment track number
        track_num += 1;
        outfile << endl << "Start of track " <<track_num << " at Byte " << file_length -
     bytes_left << endl;

        //Get "MTrk" but do nothing with it
        readFromFile(infile, buf, 4, bytes_left);

        //Get length of track
        trk_length = 0;
        readFromFile(infile, buf, 4, bytes_left);
        for (int i = 0; i < 4; i++)
            trk_length = (trk_length << 8) + (buf[i] & 0x00FF);
        outfile << "Track length: " << trk_length << endl;

        //Initialize MIDI Event variables
        std::queue<long> delta_time_q;
        bool vlq_left;
        long long delta_time = 0;
        long vlq_byte;

        //Loop through each MIDI Event
        trk_bytes_left = trk_length;
        total_time = 0;
        round_time = 0;
        unsigned char status;
        bool using_previous;
```

```
    while(trk_bytes_left > 0){

        //Get delta time of MIDI event
        vlq_left = true;

        //Push variable length quantity to queue
        while(vlq_left){
            readFromFile(infile, buf, 1, bytes_left);
            trk_bytes_left -= 1;
            delta_time_q.push(buf[0]);
            if (!(buf[0] & 0x80))
                vlq_left = false;
        }

        //Initialize delta_time to 0
        delta_time = 0;

        //For byte in variable length quantity, add to total delta_time value
        while(!delta_time_q.empty()){
            vlq_byte = delta_time_q.front();
            delta_time_q.pop();
            delta_time = (delta_time << 7) | (vlq_byte & 0x7F);
        }
        double delta_float = (double)delta_time;
        delta_float *= time_multiplier;

        total_time += delta_float;
        round_time = round(total_time);

        //Peek status bytes for MIDI event
        status = peekFromFile(infile);

        //Prepare status loop boolean
        using_previous = false;

        do{
            //Parse Status
            if ((status >> 4) == 0x8) { //Note off event
                if(!using_previous){
                    readFromFile(infile, buf, 1, bytes_left);
                    trk_bytes_left -= 1;
                }

                //Get Note number (and skip velocity)
                readFromFile(infile, buf, 2, bytes_left);
                trk_bytes_left -= 2;
                unsigned char note_num = buf[0];

                //Record in CSV
                outfile << round_time << ",Off," << noteFinder(note_num) << "," << (
status & 0xF) + 1 << endl;
                break;
            }
```

```
                else if ((status >> 4) == 0x9) { //Note on event
                    if(!using_previous){
                        readFromFile(infile, buf, 1, bytes_left);
                        trk_bytes_left -= 1;
                    }

                    //Get Note number, use velocity to tell if on or off
                    readFromFile(infile, buf, 2, bytes_left);
                    trk_bytes_left -= 2;
                    unsigned char note_num = buf[0];

                    //Record in CSV
                    if (buf[1] != 0x00)
                        outfile << round_time << ",On," << noteFinder(note_num) << "," << (status & 0xF) + 1 << endl;
                    else
                        outfile << round_time << ",Off," << noteFinder(note_num) << "," << (status & 0xF) + 1 << endl;
                    break;
                }

                //Other unimportant MIDI events
                else if ((status >> 4) == 0xA){ //Polyphonic key pressure
                    if(!using_previous){
                        readFromFile(infile, buf, 1, bytes_left);
                        trk_bytes_left -= 1;
                    }
                    readFromFile(infile, buf, 2, bytes_left);
                    trk_bytes_left -= 2;
                    outfile << round_time << ", Polyphonic key pressure event" << ", Channel: " << (status & 0xF) + 1 << endl;
                    break;
                }
                else if ((status >> 4) == 0xB){ //Control Change
                    if(!using_previous){
                        readFromFile(infile, buf, 1, bytes_left);
                        trk_bytes_left -= 1;
                    }
                    readFromFile(infile, buf, 2, bytes_left);
                    trk_bytes_left -= 2;
                    outfile << round_time << ", Control change event" << ", Channel: " << (status & 0xF) + 1 << endl;
                    break;
                }
                else if ((status >> 4) == 0xC){ //Program Change
                    if(!using_previous){
                        readFromFile(infile, buf, 1, bytes_left);
                        trk_bytes_left -= 1;
                    }
                    readFromFile(infile, buf, 1, bytes_left);
                    trk_bytes_left -= 1;
                    outfile << round_time << ", Program change event" << ", Channel: " << (status & 0xF) + 1 << endl;
                    break;
```

```
            }
            else if ((status >> 4) == 0xD){ //Channel Pressure
                if(!using_previous){
                    readFromFile(infile, buf, 1, bytes_left);
                    trk_bytes_left -= 1;
                }
                readFromFile(infile, buf, 1, bytes_left);
                trk_bytes_left -= 1;
                outfile << round_time << ", Channel Pressure event" << ", Channel: "
 << (status & 0xF) + 1 << endl;
                break;
            }
            else if ((status >> 4) == 0xE){ //Pitch Wheel Change
                if(!using_previous){
                    readFromFile(infile, buf, 1, bytes_left);
                    trk_bytes_left -= 1;
                }
                readFromFile(infile, buf, 2, bytes_left);
                trk_bytes_left -= 2;
                outfile << round_time << ", Pitch wheel event" << ", Channel: " << (
status & 0xF) + 1 << endl;
                //outfile << "status:" << (status & 0xFF) << endl;
                break;
            }
            else if (status == 0xF0){ //System Exclusive
                if(!using_previous){
                    readFromFile(infile, buf, 1, bytes_left);
                    trk_bytes_left -= 1;
                }
                readFromFile(infile, buf, 1, bytes_left);
                trk_bytes_left -= 1;
                outfile << round_time << ", System Exclusive event" << endl;
                break;
            }
            else if (status == 0xF2){ //Song Position Pointer
                if(!using_previous){
                    readFromFile(infile, buf, 1, bytes_left);
                    trk_bytes_left -= 1;
                }
                readFromFile(infile, buf, 2, bytes_left);
                trk_bytes_left -= 2;
                outfile << round_time << ", Song position pointer" << endl;
                break;
            }
            else if (status == 0xF3){ //Song Select
                if(!using_previous){
                    readFromFile(infile, buf, 1, bytes_left);
                    trk_bytes_left -= 1;
                }
                readFromFile(infile, buf, 1, bytes_left);
                trk_bytes_left -= 1;
                outfile << round_time << ", Song select event" << endl;
                break;
            }
```

```
                else if (status == 0xF6){ //Tune Request
                    if(!using_previous){
                        readFromFile(infile, buf, 1, bytes_left);
                        trk_bytes_left -= 1;
                    }
                    outfile << round_time << ", Tune request" << endl;
                    break;
                }
                else if (status == 0xF7){ //End of Exclusive
                    if(!using_previous){
                        readFromFile(infile, buf, 1, bytes_left);
                        trk_bytes_left -= 1;
                    }
                    outfile << round_time << ", End of exclusive" << endl;
                    break;
                }
                else if (status == 0xF8){ //Timing Clock
                    if(!using_previous){
                        readFromFile(infile, buf, 1, bytes_left);
                        trk_bytes_left -= 1;
                    }
                    outfile << round_time << ", Timing clock" << endl;
                    break;
                }
                else if (status == 0xFA){ //Start
                    if(!using_previous){
                        readFromFile(infile, buf, 1, bytes_left);
                        trk_bytes_left -= 1;
                    }
                    outfile << round_time << ", Start" << endl;
                    break;
                }
                else if (status == 0xFB){ //Continue
                    if(!using_previous){
                        readFromFile(infile, buf, 1, bytes_left);
                        trk_bytes_left -= 1;
                    }
                    outfile << round_time << ", Continue" << endl;
                    break;
                }
                else if (status == 0xFC){ //Stop
                    if(!using_previous){
                        readFromFile(infile, buf, 1, bytes_left);
                        trk_bytes_left -= 1;
                    }
                    outfile << round_time << ", Stop" << endl;
                    break;
                }
                else if (status == 0xFE){ //Active sensing
                    if(!using_previous){
                        readFromFile(infile, buf, 1, bytes_left);
                        trk_bytes_left -= 1;
                    }
                    outfile << round_time << ", Active sensing" << endl;
```

```
                break;
            }
            else if (status == 0xFF){ //Reset (escape for meta events)
                if(!using_previous){
                    readFromFile(infile, buf, 1, bytes_left);
                    trk_bytes_left -= 1;
                }

                //Get meta event type
                readFromFile(infile, buf, 1, bytes_left);
                trk_bytes_left -= 1;
                char meta_event_type = buf[0];

                //Get meta event length
                readFromFile(infile, buf, 1, bytes_left);
                trk_bytes_left -= 1;
                char length = buf[0];

                if (meta_event_type == 0x00){ //Sequence Number
                    readFromFile(infile, buf, length, bytes_left);
                    trk_bytes_left -= length;
                    outfile << round_time << ", Sequence number event" << endl;
                }
                else if (meta_event_type == 0x01){ //Text Event
                    readFromFile(infile, buf, length, bytes_left);
                    trk_bytes_left -= length;
                    outfile << round_time << ", Text event" << endl;
                    outfile << "Length of text: " << (int)length << endl;
                }
                else if (meta_event_type == 0x02){ //Copyright Notice
                    readFromFile(infile, buf, length, bytes_left);
                    trk_bytes_left -= length;
                    outfile << round_time << ", Copyright event" << endl;
                }
                else if (meta_event_type == 0x03){ //Sequence/Track Name
                    readFromFile(infile, buf, length, bytes_left);
                    trk_bytes_left -= length;
                    outfile << round_time << ", Sequence/Track Name event" << endl;
                }
                else if (meta_event_type == 0x04){ //Instrument Name
                    readFromFile(infile, buf, length, bytes_left);
                    trk_bytes_left -= length;
                    outfile << round_time << ", Instrument name event" << endl;
                }
                else if (meta_event_type == 0x05){ //Lyric
                    readFromFile(infile, buf, length, bytes_left);
                    trk_bytes_left -= length;
                    outfile << round_time << ", Lyrics event" << endl;
                }
                else if (meta_event_type == 0x06){ //Text  Marker
                    readFromFile(infile, buf, length, bytes_left);
                    trk_bytes_left -= length;
                    outfile << round_time << ", Text Marker" << endl;
                }
```

```
                    else if (meta_event_type == 0x07){ //Cue Point
                        readFromFile(infile, buf, length, bytes_left);
                        trk_bytes_left -= length;
                        outfile << round_time << ", Cue point" << endl;
                    }
                    else if (meta_event_type == 0x20){ //MIDI Channel Prefix
                        readFromFile(infile, buf, length, bytes_left);
                        trk_bytes_left -= length;
                        outfile << round_time << ", MIDI Channel Prefix" << endl;
                    }
                    else if (meta_event_type == 0x21){ //MIDI Channel Prefix
                        readFromFile(infile, buf, length, bytes_left);
                        trk_bytes_left -= length;
                        outfile << round_time << ", MIDI Port" << endl;
                    }
                    else if (meta_event_type == 0x2F){ //End of Track
                        readFromFile(infile, buf, length, bytes_left);
                        trk_bytes_left -= length;
                        outfile << round_time << ",End of track" << endl;
                    }
                    else if (meta_event_type == 0x51){ //Set Tempo
                        readFromFile(infile, buf, length, bytes_left);
                        trk_bytes_left -= length;
                        tempo = 0;
                        for (i = 0; i < length; i++){
                            tempo = (tempo << 8) | (0x00FF & buf[i]);
                        }
                        time_multiplier = tempo*byte_frame_freq*0.000001/tpq;
                        outfile << round_time << ", Tempo to " << (long)tempo << " usec/quarter note" << endl;
                    }
                    else if (meta_event_type == 0x54){ //SMPTE Offset
                        readFromFile(infile, buf, length, bytes_left);
                        trk_bytes_left -= length;
                        outfile << round_time << ", SMPTE event" << endl;
                    }
                    else if (meta_event_type == 0x58){ //Time Signature
                        readFromFile(infile, buf, length, bytes_left);
                        trk_bytes_left -= length;
                        outfile << round_time << ", Time Signature event" << endl;
                    }
                    else if (meta_event_type == 0x59){ //Key Signature
                        readFromFile(infile, buf, length, bytes_left);
                        trk_bytes_left -= length;
                        outfile << round_time << ", Key signature event" << endl;
                    }
                    else if (meta_event_type == 0x7F){ //Sequencer Specific Meta-Event
                        readFromFile(infile, buf, length, bytes_left);
                        trk_bytes_left -= length;
                        outfile << round_time << ", Sequencer-specific event" << endl;
                    }
                    break;
                }
                else{
```

```
                //Use previous status byte as status
                status = prev_status;
                using_previous = true;
                }

            }while(using_previous);

            //Update previous status bytes
            prev_status = status;
        }
    }

    //Open new binary file
    string binfile_name = "BIN_Files/" + string(argv[1]) + ".bin";
    remove(binfile_name.c_str());
    ofstream binfile(binfile_name, ios::binary);

    //C array frame number for Chris' use
    int final_frame_num = 0;

    //Prepare byte_map
    int MAP_BYTES = 5;
    char byte_map[MAP_BYTES];
    for(int i = 0; i < MAP_BYTES; i++)
        byte_map[i] = 0x00;

    //Convert CSV to Binary file
    string str_in;
    char * pch;
    int cur_frame, last_frame = 0;
    bool found_channel = false;
    vector<string> str_vec;

    //Open CSV file and start from beginning
    outfile.clear();
    outfile.seekg(0, ios::beg);

    //Loop through lines in CSV
    while(getline(outfile, str_in)){
        char cstr[str_in.size()+1];
        strcpy(cstr, str_in.c_str());
        pch = strtok(cstr,",");
        while(pch != NULL){
            str_vec.push_back(pch);
            pch = strtok(NULL, ",");
        }

        //If line has 4 comma-separated values, it's a note
        if(str_vec.size() == 4){
            if(str_vec[3] == argv[2]){
                found_channel = true;
                stringstream frame_num(str_vec[0]);
                frame_num >> cur_frame;
                if(cur_frame != last_frame){
```

```

                    //Print byte_map (cur_frame - last_frame) times
                    for(int a = 0; a < (cur_frame - last_frame); a++)
                        for(int i = 0; i < MAP_BYTES; i++)
                            binfile.write(byte_map + i, 1);
                    last_frame = cur_frame;
                }
                //Adjust byte_map based on str_vec[2]
                int byte_map_num = note_map[str_vec[2]];
                unsigned char fret_bits = 0x80 >> (byte_map_num % 8);
                int fret = (byte_map_num / 8) - 2;
                if(str_vec[1] == "On")
                    byte_map[fret] |= fret_bits;
                else
                    byte_map[fret] &= ~fret_bits;
            }
        }
        //If end of track, exit
        else if(found_channel)
            if(str_vec.size() == 2)
                if(str_vec[1] == "End of track")
                    break;
        while(!str_vec.empty())
            str_vec.pop_back();
    }

    //Close files
    infile.close();
    outfile.close();
    binfile.close();

    //C array of file for Chris' use
    char file_bytes[(last_frame)*MAP_BYTES];
    fstream binfile2;
    string binfile2_name = "BIN_Files/" + string(argv[1]) + ".bin";
    binfile2.open(binfile2_name, fstream::in | ios::binary);
    for(int i = 0; i < (last_frame)*MAP_BYTES; i++)
        binfile2.read(&(file_bytes[i]), 1);
    unsigned char file_bytes_2d[MAP_BYTES][last_frame];
    for(int i = 0; i < (last_frame)*MAP_BYTES; i++){
        file_bytes_2d[i%5][i/5] = (const char)file_bytes[i];
        //cout << std::hex << (int)file_bytes_2d[i/5][i%5] << " ";
    }
    binfile2.close();

    //Copy frame array to .txt for debugging
    fstream songfile;
    string songfile_name = string(argv[1]) + ".txt";
    songfile.open(songfile_name, fstream::in | fstream::out | fstream::trunc);
    songfile << "{{" << charToString(file_bytes_2d[0][0]);
    for(int j = 0; j < last_frame; j++)
        songfile << "," << charToString(file_bytes_2d[0][j]);
    songfile << "}";
    for(int i = 0; i < MAP_BYTES; i++){
```

```cpp
        songfile << ",{" << charToString(file_bytes_2d[i][0]);
        for(int j = 0; j < last_frame; j++)
            songfile << "," << charToString(file_bytes_2d[i][j]);
        songfile << "}";
        }
    songfile << "}";
    songfile.close();


    //Stop clock
    clock_t end = clock();
    cout << std::fixed << "Total elapsed time: " <<  int(end - begin) << " ms" << endl;
    return 0;
}

//Helper function to read a number of bytes from MIDI file
void readFromFile(std::fstream& infile, char * bytebuf, int length, long &bytes_left){
    for(int i = 0; i < length; i++)
        infile.read(&(bytebuf[i % BUFFER_SIZE]), 1);
    bytes_left -= length;
}

//Helper function to peek next 2 bytes from MIDI file (only used for peeking status
    bytes)
unsigned char peekFromFile(std::fstream& infile){
        return infile.peek();
}

//Helper function to map proper note and octave
std::string noteFinder(int note_num){
    string this_note = notes[note_num % 12];
    this_note += octaves[(note_num / 12)];
    return this_note;
}

//Used to debugging array copied to .txt
std::string charToString(unsigned char chari){
    int charint = (int)chari;
    std::string hex;
    int big = chari/16;
    if(big < 10)
        hex += "0x" + to_string(big);
    else if(big == 10)
        hex += "0xa";
    else if(big == 11)
        hex += "0xb";
    else if(big == 12)
        hex += "0xc";
    else if(big == 13)
        hex += "0xd";
    else if(big == 14)
        hex += "0xe";
    else if(big == 15)
        hex += "0xf";
```

```

    int little = chari%16;
    if(little < 10)
        hex += to_string(little);
    else if(little == 10)
        hex += "a";
    else if(little == 11)
        hex += "b";
    else if(little == 12)
        hex += "c";
    else if(little == 13)
        hex += "d";
    else if(little == 14)
        hex += "e";
    else if(little == 15)
        hex += "f";
    return hex;
}
```

Listing 1: MIDI to CSV, and CSV to Binary converter.

```python
#Import Modules
import sys
import serial
import os
import time
from array import array

 # Serial Preparations
ser = serial.Serial()
ser.baudrate = 57600
ser.port = 'COM7'#COM10 no work
ser.open()

# Initialize song array
song = array('B')

# Open binary file
file_name = 'BIN_Files/'+str(sys.argv[1])+'.bin'
file_size = os.path.getsize(file_name)

# Send file size over Bluetooth
count = 0
size_1 = file_size & int('0xff00',16)
size_1 >>= 8
size_1 = size_1.to_bytes(1, 'big')
size_2 = file_size & int('0x00ff',16)
size_2 = size_2.to_bytes(1, 'big')
print(size_1)
print(size_2)
ser.write(size_1)
ser.write(size_2)
time.sleep(0.5)

# Send song bytes one at a time
with open(file_name, 'rb') as file:
    for i in range(file_size):
        count += 1
        byte = file.read(1)
        if byte != "":
            ser.write(byte)

# Print bytes sent
print("sent "+str(count)+" bytes")
```

Listing 2: Bluetooth transmission of binary file to ESP32 (laptop end).

```
/*
 * ESP32 Bluetooth Controller Program
 *
 * By Austin Born, Fall 2018
 *
 * Boilerplate code is in the Public Domain, by Evandro Copercini, 2018
 *
 * The code creates a bridge betweeon Serial and Classical Bluetooth (SPP),
 * and shows the functionality of SerialBT.
 */

//Libraries and variable definitions
#include "BluetoothSerial.h"
#include <stdio.h>

#if !defined(CONFIG_BT_ENABLED) || !defined(CONFIG_BLUEDROID_ENABLED)
#error Bluetooth is not enabled! Please run `make menuconfig` to and enable it
#endif

#define BOARDCOUNT 5
#define BUFFERDEPTH 15000

#define CLK 21
#define SDI 22
#define LE 19
#define EN 14

//Variable initializations
int i = 0;
long frame = 0;
int board = 0;
int unavail = 0;
int line = 0;
unsigned char hex[4];
int bytes_left = 0;
long file_size = 0;
int file_size_2 = 0;
char frameBuffer[BUFFERDEPTH][BOARDCOUNT];
long frame_count = 0;
bool song_loaded = false;
BluetoothSerial SerialBT;

// updateFrame() helper function to update current LED frame
void updateFrame(){
      for(int fret = 0; fret < BOARDCOUNT; fret++){
      sendByte(frameBuffer[frame_count][fret]);
    }
    load();
    frame_count++;
}

//Shift-high helper
```

```
void sh() {
  digitalWrite(SDI, HIGH);
  shift();
  digitalWrite(SDI, LOW);

}

//Shift-low helper
void sl() {
  digitalWrite(SDI, LOW);
  shift();

}

// Function to send a byte of data to frets
void sendByte(byte data) {

  //mask last two bits  since bits 7 and 8 are not used
  char mask = 0xFC; // 0011 1111
  data = mask & data;
  //send data serially over SDI
  //reverse order since first bit in is last bit
  for (int i = 0; i < 8; i++) {
    //shift 1000 0000 to right to change bit
    mask = 0x01 << i;

    //send data unmasked bit
    if (data & mask) {
      sh();
    }
    else {
      sl();
    }
  }
}

// Shift helper
void shift() {
  digitalWrite(CLK, HIGH);
  digitalWrite(CLK, LOW);
}

// Load helper
void load() {
  digitalWrite(LE, HIGH);

  digitalWrite(LE, LOW);
}

// Testboards function for debugging
void testBoards(int numOfBoards) {
  for (char i = 0x00; i != 0x40; i++) {
    for (int j = 0; j < numOfBoards; j++) {
      sendByte(i);
```

```
    }
    load();
    delay(50);
  }
  for (int k = 0; k < 4; k++) {
    for (char i = 0x01; i != 0x40; i = i << 1) {
      for (int j = 0; j < numOfBoards; j++) {
        sendByte(i);
      }
      load();
      delay(50);
    }
    for (char i = 0x40; i != 0x00; i = i >> 1) {
      for (int j = 0; j < numOfBoards; j++) {
        sendByte(i);
      }
      load();
      delay(50);
    }
  }
  for (int i = 0; i < 4; i++) {
    for (int j = 0; j < numOfBoards; j++) {
      sendByte(0xFF);
    }
    load();
    delay(300);
    for (int j = 0; j < numOfBoards; j++) {
      sendByte(0x00);
    }
    load();
    delay(300);
  }
}


// Initial setup
void setup() {

  // Prepare serial i/o
  SerialBT.begin("ESP32-GuitarBuddy"); //Bluetooth device name
  Serial.flush();
  Serial.begin(57600);

  // Initialize pins
  pinMode(CLK, OUTPUT);
  pinMode(SDI, OUTPUT);
  pinMode(LE, OUTPUT);
  pinMode(EN, OUTPUT);
  digitalWrite(EN, HIGH);

  // Wait to receive file size bytes over Bluetooth
  while(1){
    if(SerialBT.available()){
      file_size = (int)SerialBT.read();
```

```
      if(file_size == 0){
        delay(100);
        continue;
      }
      file_size <<= 8;
      delay(200);
      file_size_2 = (int)SerialBT.read();
      file_size += file_size_2;
      Serial.print(file_size);
      break;
      }
  }

  // After file size is received, input rest of byte data to frameBuffer
  bytes_left = file_size;
    while(bytes_left > 0){
      if(SerialBT.available()){
        frameBuffer[frame][board] = SerialBT.read();
        if(board == 4){
          frame++;
          board = 0;
        }
        else
          board++;
        bytes_left -= 1;
      }
    }

  // Print frameBuffer for debugging
  for(int f = 0; f < frame; f ++){
    Serial.print("[");
    for(int fr = 0; fr < BOARDCOUNT; fr++){
      Serial.print(frameBuffer[f][fr], BIN);
      Serial.print(", ");
    }
    Serial.println("]");
  }
}

// Loop through frameBuffer, sending each byte with a set delay
void loop() {
//  while(1)
//    testBoards(1);
  board = 0;
  frame = 0;
  bytes_left = file_size;
  frame_count = 0;
  while(bytes_left > 0){
    updateFrame();
    bytes_left -= BOARDCOUNT;
    delay(30);
  }
```

```
}
```

Listing 3: Program flashed to ESP32 for binary receiver, storage, and playback.