Heads Up Display for MIDI Keyboard

ECE 445 Fall 2018

Design Review

Andy Woodruff

Alexa Hirsch

Introduction

The MIDI HUD

There are to main technological concepts that inspired this project. One of them is the Musical Instrument Digital Interface (MIDI). MIDI is a way to connect devices that make and control sound — such as synthesizers, samplers, and computers — so that they can communicate with each other using MIDI messages [1].

Secondly, we intended to implement a Heads-Up Display (HUD). This is a transparent display that presents data without requiring users to look away from their usual viewpoints [2].

Objective

The objective of our project was to combine these two concepts: to create a convenient display that provides real-time music information to MIDI device users without the use of additional resources. Specifically, the music information we intend to display is the individual notes being played, as the user is playing them, as well as interpreting and displaying arpeggios and chords consisting of multiple simultaneous notes. In order to create a standalone device we intended to make it capable of directly connecting to MIDI keyboard and performing this analysis on data being output directly from the instrument.

Aside from the data processing and information supply objective of the device, we wanted to pair this with a display format that functions differently and separately from typical technology. We decided to do this through the use of a Heads Up Display. Bringing the information away from mass-information technology and placing it in the keyboard-user's line of sight when looking at their sheet music prevents focus from being turned away from the present music focus, and increases the user's informational intake without them having to pause to search the internet. Our goal was to intersect an advantage of digital music, the theoretical analysis, with an advantage of analog music, the performance.

Background

There are many MIDI interface software products currently on the market. These include simple recording programs, pitch bending, sustain pedals, and various sound effects. Additionally, there are web services designed to translate key presses, guitar fret positions, or note names, into chord names. These web services, along with many forum posts looking for ways to determine chord names while playing them, are right at the top of any web search for "how to tell what chord I'm playing." Until now, these products have been largely separate. That is, MIDI interface devices and largely limited to purely storing or manipulating MIDI data, rather than analyzing it, and music analysis software, even when given a MIDI

interface, tends to assume the user is displaying the information on a computer [6]. Our product is thus unique in offering a hardware tool to view the chords played on a MIDI instrument in real time.

High-Level Requirements

1. The projector must accurately display the chord name sprites with enough clarity that the average user misreads fewer than one in fifty displayed chords.
2. The total time the system takes to read MIDI input, determine the chord being played, and pipe that information to the display should be less than the average visual reaction time, which is approximately 250ms.
3. The user must be able to record an arbitrary number and length of MIDI files in real time, only limited to the file size and SD card space, simply by pressing a single button to begin and end recording.

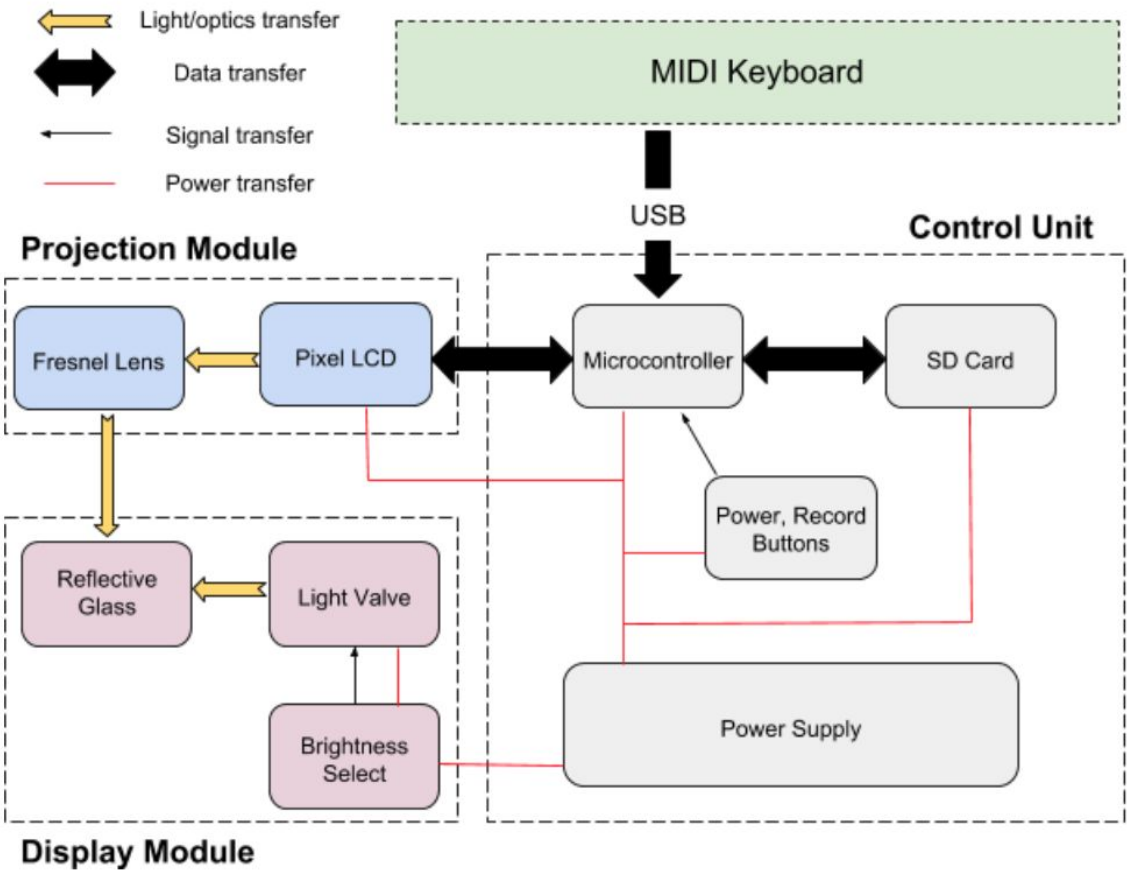

Figure 1. Block Diagram

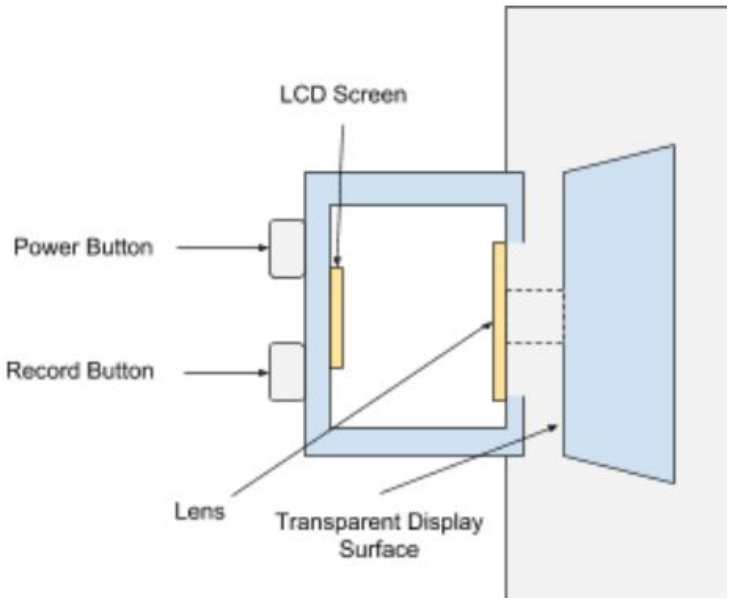

Figure 2. Proposed Physical Design

Design

MIDI Data Input

Design Procedure

Two options, USB or DIN-5. USB is used on more modern devices. Some keyboards don't even have onboard synthesizers, and rely fully on being plugged into a seperate device, usually a computer, via USB. Additionally, USB is two-way, which means if we want to expand the devices functionality in the future, we have the option of sending information to the keyboard as well as receiving it with a software only update. Other hand, DIN-5. This is an old standard. Very low tech which means fewer potential technical problems. It's a one-way protocol which means if we want an input into keyboards we'd need to leave two serial communications ports open on the microcontroller, but the upside is we don't need to do bus priority negotiations. Also, beginner keyboardists (our target audience) aren't necessarily going to own modern keyboards. Second-hand keyboards might be more likely to have DIN-5 outputs, designed when music systems were more about communication between instruments than between computers. Ultimately, we decided to use USB largely for future-proofing, on the assumption that more new keyboards will be made with USB ports than DIN-5 ports. However, we ran into technical issues (specifically the two-way communications negotiation) that forced us to switch over to DIN-5.

Design Details

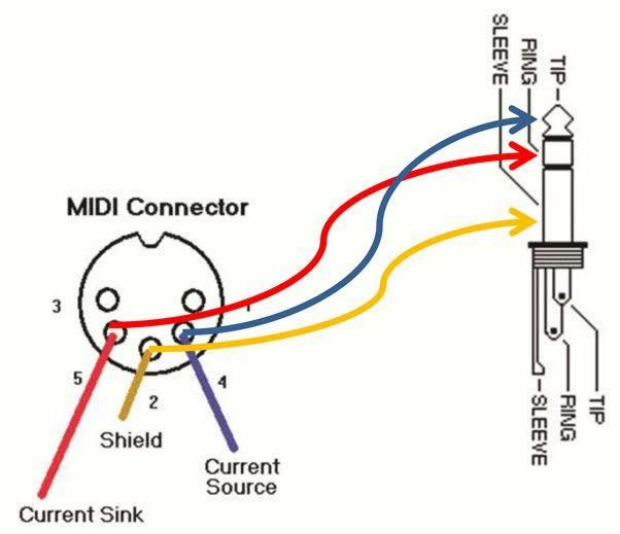

Figure 3. DIN-5 MIDI pinout

Our initial design had us directly plugging in a USB cable into the USB port on the atmega32U4 board. We then discovered that the atmega32U4 was incapable of acting as a USB Host, meaning it could not initiate communication with a keyboard over USB. Without this initiation, no data is sent from the keyboard whatsoever. To fix this, we decided to move over to a DIN-5 communication line rather than incorporating USB hosting into our microcontroller, as that would have taken additional IC's and software we weren't equipped to research at the time. MIDI data is sent in serial packets of three bytes. The data rate is 31250 Baud, and the data is sent via a 5 milliamp passive-high current line (see Figure 3 above).

Our microcontroller does not directly measure current, so we constructed a circuit to convert that 5ma signal into a 3.3 volt signal. The conversion here also required the use of a powered op-amp, as the difference between high and low was initially only around 1.7 volts, and our microcontroller spec requires a 3 volt difference to guarantee reading, see Figure 4 below. We initially attempted to also opto-isolate the circuit, but that wasn't successful, and ultimately it proved unnecessary. It's main purpose would be to prevent accidental over-volting, and to eliminate ground loops that may cause speaker hum. Since our microcontroller isn't going to be running any speakers, the second point isn't important, and the risk of over-volting was deemed sufficiently low.

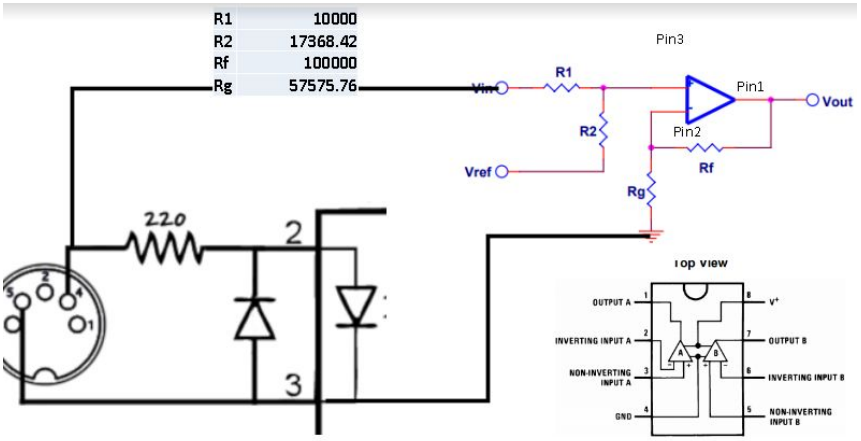

Figure 4. DIN-5 MIDI pinout design

Verifications

To verify the USB setup, we programmed the microcontroller to flash an LED light when it detected any MIDI packet coming over USB. This led to our discovery of the USB hosting issue, when no change of configuration resolved this issue. To verify it wasn't an error deeper in the microcontroller or the keyboard, they were each hooked into PC software. We used a publicly available utility called RTMIDI to test the ability of both the keyboard and the microcontroller to both send and receive MIDI packets. This led us to the realization that the error was in the microcontroller-keyboard communication, which turned out to be the lack of any Host to coordinate said communication. To verify the DIN-5 connection, we analyzed the serial data packets via oscilloscope at various points along the current-to-voltage circuit. Once these were within acceptable ranges, we programmed a microcontroller to flash an LED light when it received a middle C note on packet. Once this was successful, we tested other notes and eventually determined it was reading information correctly.

MIDI Data Storage

Design Procedure

The data storage, or music recording, aspect of the project was defined by a few factors. Firstly, knowing we were working with MIDI data meant that we would be developing a MIDI file format. Secondly, using an SD card for storage and interfacing this with the microcontroller meant adhering to data transfer methods and requirements set by SPI data transfer and the FAT32 data format (which is used by our acquired micro SD).

The most impactful requirement on the software design was adhering to the 512-byte buffer of the FAT32 format. This restricted the write blocks sent over SPI to be written in 512-byte blocks.

The SPI system in this part of the device, and the other SPI controlled part of the device, the LCD display, uses the microcontroller as the master and the connected component as the slave. The sequence and timing of events in writing a block over SPI to an SD can be seen below in Figure 5. Since the incoming MIDI data from the keyboard that would need to be stored would be larger than a single write of 512 bytes, blocks would need to be written multiple times.

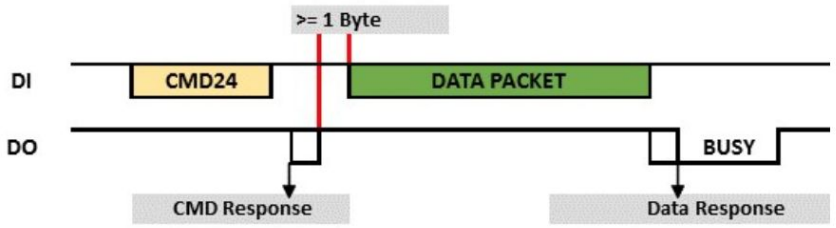

Figure 5. Single Block Write Timing Diagram [9]

Going from these set requirements, we identified 2 possible methods of addressing this issue. One such method was to not access the SD card itself until all desired MIDI recording data had been collected in temporary storage arrays in the microcontroller software. We decided that even though this may simplify our communications with the SD card, the storage size restrictions that would be set by the available microcontroller RAM would not allow for this part of the device to actually be useful. Therefore we turned to our other plan, which would be to write to the SD each time the 512-byte buffer was reached. In terms of data storage restrictions, this would alter our size limitations from that of the microcontroller RAM to that of the actual storage size of the SD being used.

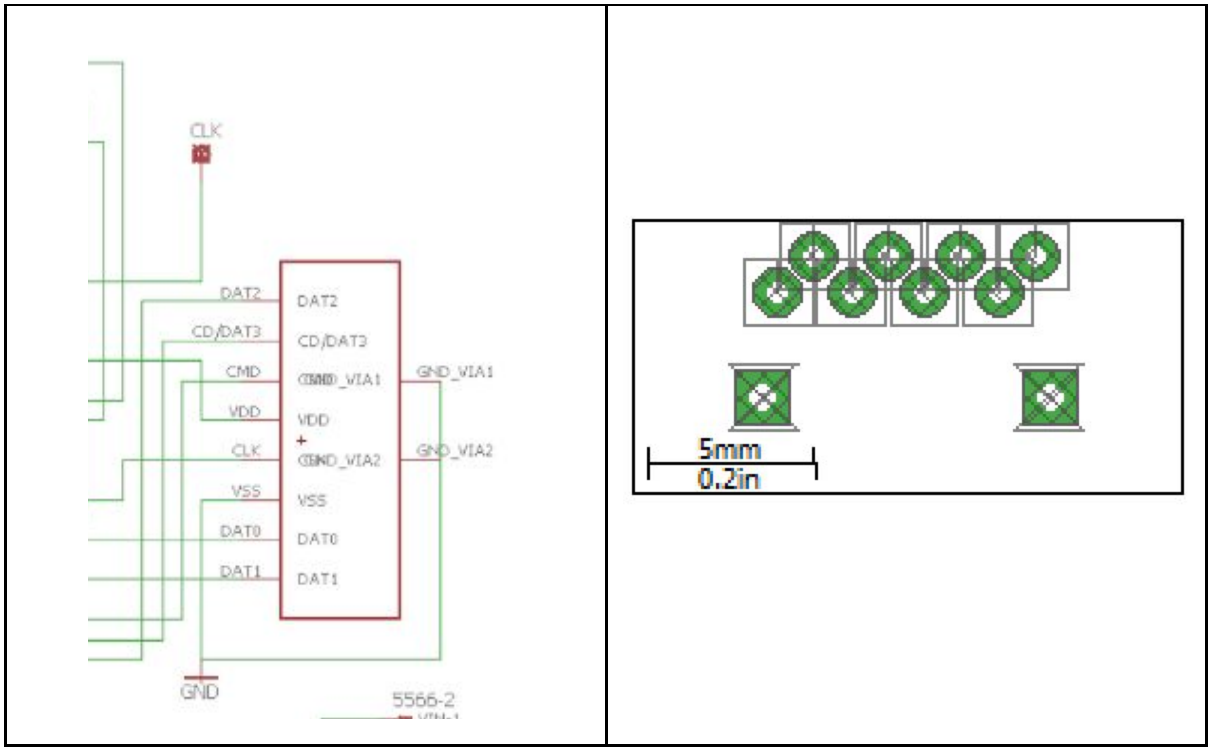

Figure 6. SD connector schematic connections (left) and PCB design (right)

Figure 6 shows the required connections for the SD connector and the corresponding PCB board device design, as developed based off of the datasheet of our selected connector.

The formatting of the MIDI file is summarized in Figure 7. Each of the blocks represents a value that is essentially read as a sequence of bytes. There are two distinct parts that make up the file, the header chunk and the track chunk. The header, which starts with the ASCII value of “MThd”, consists of values for *format*, there are multiple MIDI formats that are possible, *tracks*, or in our case of using MIDI data for music the number of instruments being represented, and *division*. *Division* is set to be the time unit that will be used to calculate the *delta-time* values in the track chunk. *Length* for the header is always set to 6 as this is the set length for a MIDI header.

The track chunk is what the bulk of the file consists of. After stating the presence of track information with the “MTrk” ASCII value, track *length* is the entirety of the instrument track length for the file. The data portion of the track is the list of “events”, or, in our case, individual keyboard notes.

	<---Chunk--->				
	type	**length**	**Data**		
MIDI File :	`MThd`	6	*<format>*	*<tracks>*	*<division>*
	`MTrk`	*<length>*	*<delta_time> <event> ...*		
	:				
	`MTrk`	*<length>*	*<delta_time> <event> ...*		

Figure 7. MIDI File Format

Design Details

All aspects of the MIDI file aside from the bytes communicating the contents of note events can be set to default values in order to shape the file for storage. By sending these defining values initially, we can build the file into which the notes will be saved to. Figure 8 displays the default values we determined for the file in red, with the byte lengths of each of the sections shown in the accompanying grey blocks. The group of bytes that is repeated for every event is colored green, other bytes are not repeated in the file.

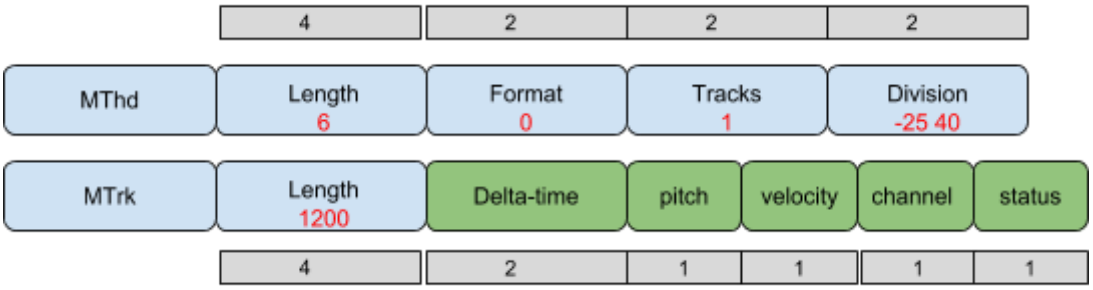

Figure 8. MIDI file default values and byte structure

Format 0 of the MIDI standard states that we are creating a MIDI file entirely consisting of only one track, the MIDI keyboard. This means we also set the number of tracks to 1 in the following portion. Division can be defined either in terms of musical notation by setting a value of ticks to be counted per quarter note (setting the MSB of the 2-byte value to 0), or in terms of frames per second and ticks per frame (setting the MSB to 1). We wanted to set our tick counts equal to a millisecond for ease of use in software. This was calculated by using:

$25\ \frac{frames}{sec}$ with a resolution of $40\frac{units}{frame}$

The length of the track chunk gives the value for the number of bytes in the track after the length value. For our testing and initial development, this value was set to 1200, which gave us a track length of 200 notes of 6 bytes each. This value was chosen in order to test

multiple writes to the SD of 512-byte blocks, and also the requirement to fill the remaining bytes of the array if event packets are no longer being received.
$\frac{1200}{512} = 2.34375$ writes to the SD. $(512)(0.34375) = 176$ null bits to fill 3rd write block.

A state machine described by Figure 9 was developed for the writing of the byte arrays to the SD card. When the record option is selected, the machine moves from IDLE to SETUP, in which the above selected default bytes are loaded into an initial array, and then on to RECORD when this is complete. In RECORD, the note events are loaded into the array as they arrive to the microcontroller as serial input until the 512-byte limit is reached. This is written as a block to the SD, the address for the next write location is returned based on the 512-byte structure size, the temporary array is cleared, and then this process is repeated as event bytes continue to arrive to the system.

SHUTDOWN is entered, in our testing, when the 200-note track limit is reached. In future development this would be altered to occur when the user de-selects the record option. By not having a default value for track length some additional logic would have to be set to find the length of the track after recording has finished, and then this value would need to be updated at its memory location in the SD card.

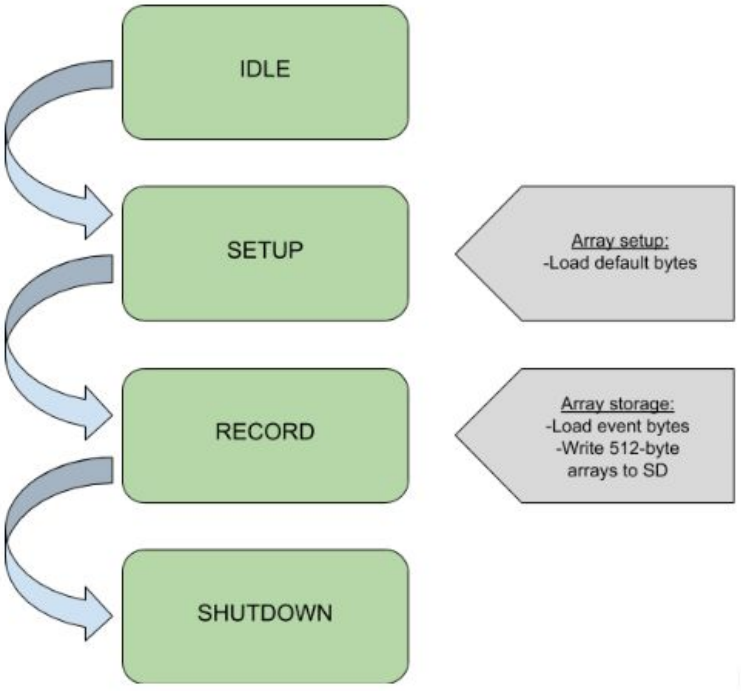

Figure 9. SD write method

Verification

This process could be tested by simulating the serial byte input that would be received by the system from the keyboard. Specifically, we sent a stream of 200 note packets representing the note “C” at different time values in a format described by Figure 10.

dec	0	time
hex	09	status
dec	48	pitch
dec	64	velocity

Figure 10. C note test packets

While we were able to load and write the array blocks, once the SD card was transferred to another device we were not able to view a readable MIDI file. In the future it would be critical for us to spend more time developing tests in order to build a comprehensible file.

Chord Identification

Design Procedure

The decision making process here broke down into two main camps. Which chords we want to identify, and what algorithm we want to use to identify them. Nearly any combination of notes can be assigned at least one valid chord name. Many can be assigned several valid chord names. Which one is most correct is based on musical context that’s subject to individual interpretation. Clearly we need to narrow our scope to situations where we can make a more objective assessment. Additionally, the chords that get played most often need to be our main priority, because even though the user is much less likely to require computer assistance in identifying such chords, a failure to identify them would be a significant failure of the product. Ultimately, we settled on Identifying major, minor, and augmented triads along with their two most common seventh forms. Such chords are fairly easy to algorithmically determine. In algorithm design, the main consideration was how many edge cases we wanted to handle. Specifically, what to do in the cases of partial chord matches or multiple chord matches. Ultimately, we decided that partial matches don’t count for anything, and multiple matches would be resolved by taking the very first match that’s found as the correct one. We made this decision to keep both development time and code execution time down to a minimum. It would take more music theoretic justification to handle those situations than we were willing to research for this stage of development, and every extra comparison takes more computation time.

Design Details

Chord identification is broken down into three parts. Packet parsing, note updating, and chord comparison.

For packet parsing, we simply compare the status byte to the values for “note on” and “note off” commands. If the status byte matches either, we proceed to the next step, otherwise we throw the data out and keep waiting. On the next step, we determine what pitch is being played, and whether it’s the start of a note or the end of a note. Some keyboards don’t use “note off” commands at all, and instead send a “note on” command but give the note velocity as zero. So, if it’s a note off command or a note on with velocity zero, we set a flag to indicate that the note should be turned off. Then we read in the pitch and take its remainder when divided by 12. This gives us its location on a chromatic scale, which is all we need to identify chords. This lets us store the current notes being played in a single twelve bit vector.

Now we’ve finished parsing the packet and can update our note vector. We take that chromatic scale location, go to our twelve bit vector, and if our flag tells us to turn the note on, we set that bit to 1. If it tells us to turn the note off, we set that bit to 0. This brings us to chord identification. Here we have an array of eleven bitmasks that each correspond to a named chord we can identify. If the note vector matches one of these, that’s the chord currently being played. However, that’s only for one key. The default setting assumes the key is C, but circularly shifts the note vector through 12 times, each time checking the key one half-step higher than the last, until we return to the key of C, at which point it brings in the next bitmask for comparison. If we reach all twelve keys for all eleven chords and no matches are found, we return to the main loop without declaring any chord as being played. Otherwise, on the very first match, we exit the comparison loop and return the name and key of the match.

Verification

We verified this process in three stages. In the first, we bypassed the parsing stage and the note update stage, instead feeding full chords into the chord comparison portion of the program. This showed successful matches for every chord in every key, thus ensuring that that portion of the code was working properly. Then we dialed it back one level, and fed the microcontroller individual key presses with a laptop via serial connection. This tested the note update functionality, without touching the packet parsing. Lastly, we hooked the whole thing up to a keyboard and played various chords.

This was largely successful, though at this stage we noticed that the keyboard polling rate was a little slow, causing issues when chords were played too fast. Additionally, we would occasionally get phantom D#/Eb note detections. We attempted to change the keyboard polling to an interrupt, but this did not resolve the issue, and instead made note detection less reliable. The phantom note could have been an issue with either hardware or software. To distinguish between these possibilities, we would need to test the chord and note detection via USB over a laptop, however we did not have the opportunity to perform this test.

Display: LCD Communication and Optics

Design Procedure

For this subset we had the least background knowledge, and as such made our decisions largely based on what would be the easiest to both assemble and verify. Our primary goal was to make some kind of display through which the user could read both the information on the screen and sheet music placed behind the screen. Full projection systems were not considered due to the complexity of their operation and their cost. Instead we found a system that involves simply using a partially mirrored surface to reflect an image of a screen back at the user, with a single magnification stage to allow the reflected image to take up the full size of the reflective surface. We found only a single reference for such a design, in [4], but still felt it would be worth the attempt as the early verification steps could be performed in only a few minutes, and used components that were very easy to source and use.

As for the software, we used a library designed by the retailer for our screen, with only a slight modification to allow for mirroring the display. This was done for expediency and the fact that, as this library was also open source, it was almost guaranteed to be more reliable than anything we could have put together, having already had so many man-hours put into its development. The only decision left to us was what methods to use to display the chord names. We chose to use the built in text printing features, as this required the least processing time. We could have used custom sprites for each chord, which would have given a more aesthetically appealing look, or custom fonts that might have again been less blocky, however both of these would have required external memory access, which would have increased the latency of our display. Decreasing latency also led to another design decision, which was in how we clear the screen before displaying a new chord. Rather than setting the whole screen to the background color, we rewrite the previous chord information to the screen in the background color.

Design Details

Here, the details aren't much more involved than the procedure. We ran connections from each of the control pins on the LCD display into our microcontroller, then glued the display to a foam backing. Then we glued that backing to a foam arm, glued the Fresnel lens and the semi reflective display to the foam arm at our predetermined distances, and finally glued a metal retaining clip to the back of the arm at roughly the center. To make the display mirror properly, we defined a new rotation mode that swapped the control values that tell the display which column of pixels starts the X axis and which ends it. We made that a new rotation mode rather than an entirely new function because the existing four 90 degree

rotation modes already address these control values, and thus anyone reading through the code would have an easier time understanding what was being done given that context.

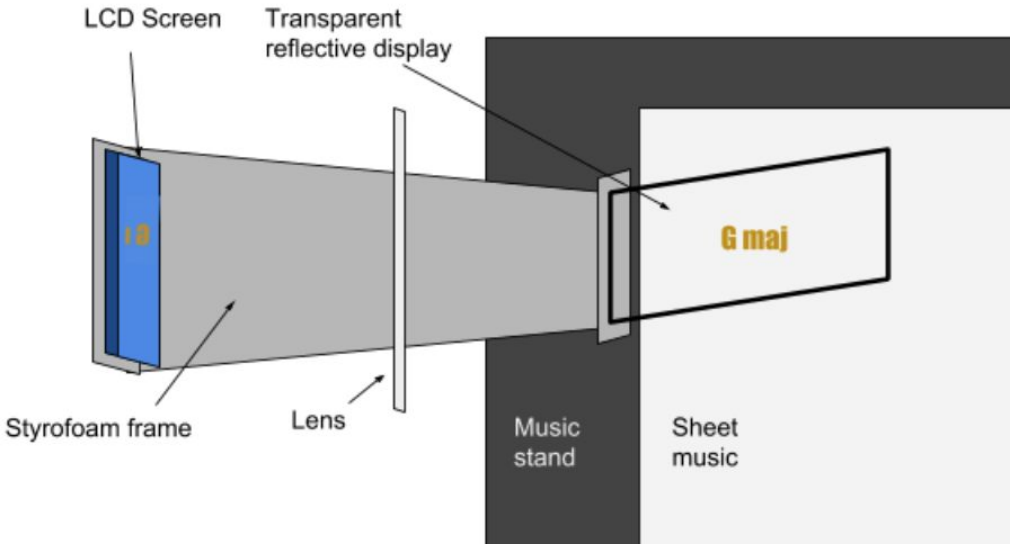

Figure 11. The Demo Physical Design

Verification

To verify this, when our reflective display arrived we set up a cell phone on a vertical stand, playing a video. We placed this stand on top of a sheet of graph paper, and angled the reflective display to where the cell phone video was visible. We then measured the closest and furthest distances at which the reflected image was still easily distinguishable. We then added in a fresnel lens between the phone and reflective display. Following a similar procedure, we marked out a minimum and maximum distance from the phone to the lens, and a second minimum and maximum distance from the lens to the reflective display. These were determined purely subjectively. Based on this success, we moved forward with this projection system. Upon repeating this testing with the final LCD display, we found these values to hold despite the difference in screen resolution.

After that, it was simply a matter of making sure the LCD was functioning properly. Fortunately, the provided libraries came with a series of calibration tests. We ran these and found that the display was functioning within specification for our purposes. The color accuracy wasn't the greatest, however we don't need precise color values, only readable text. Then to test the display speed, we ran our demonstration code with added timers that tested how many milliseconds elapsed between the first display command and the last display command. By moving from a full screen refresh to a more precise text erasing, we reduced that time from 200 milliseconds down to around 20.

The Microcontroller and Board Voltage

Design Procedure

One decision that was made regarding the powering of our device was whether to run all of the components on 5V or 3.3V. The primary adaptation we were forced to make here was that in order to work with the SD card it had to be receiving 3.3V. Here we had two options, either we could run the entire system on 3.3V, or we could run 5V and decrease the voltage only running to the SD card connector. This would require a step-down voltage regulator as well as logic shifters, so that the data pins would not be receiving IO signals out of range as well. Due to this prospect of additional logic we decided to run the entire system on 3.3V.

Since a connection with a USB and thereby interaction with UART data transfer, we knew that we would need a more reliable clock signal to run the microcontroller and external circuit than to use the internal RC clock that the ATmega32u4 was factory-set to run on. We included two 22pF capacitors and a crystal oscillator in our design.

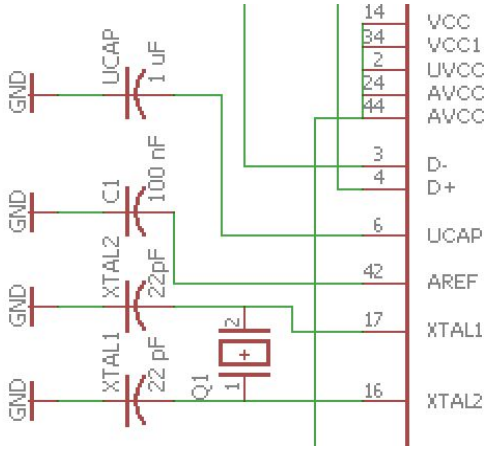

Figure 12. Microcontroller crystal oscillator circuit

Design Details

We initially worked at programming the ATmega32u4 by building an ISP from an Arduino Uno. This entailed creating a collection of temporary voltage divider circuits for the direct connections to the SPI pins on the microcontroller from the 5V logic of the Uno to 3.3V, since during ISP programming we would be altering the fuse bits of the microcontroller to using the 8MHz external crystal (and at 5V the ATmega32u4 has a maximum clock frequency of 16 MHz). After several issues with the ISP configuration, we changed our setup and used a USBtiny ISP, which connected successfully. We then uploaded a bootloader that would have the microcontroller mirror the configuration of the Arduino Micro, which was what we were prototyping our chord detection and LCD display with and also uses an ATmega32u4. Then, we altered the fuse bits to set the microcontroller to using the external crystal oscillator with

slowly rising power, and to output the clock from its designated pin (which we would need for clock signals to the SD connector and LCD).

After disconnecting the ISP, we powered the microcontroller using our lithium ion battery through the 3.3V regulator and observed an uneven clock signal of 8MHz. We were not expecting that running the chip at this maximum frequency would be such an unreliable signal for the board and UART connection. After reconnecting with the ISP, we altered the fuse bits to include a system clock prescaler, which divided the output system clock to 1MHz, and achieved a clean signal after testing.

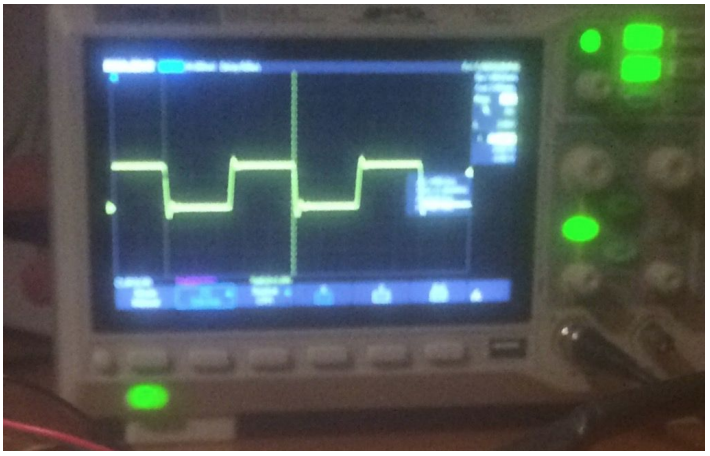

Figure 13. 1MHz ATmega32u4 clock out at 3.3V

Even at using the factory-set RC clock of 1MHz when powered at 3.3V, the microcontroller was calibrated to satisfy USB low speed frequency accuracy [7]. However, once disconnected from the ISP and attempting to load programs directly through the USB we were unable to successfully make a connection and integrate the different modules of our project. In the end, the results of running the whole of the project on 3.3V resulted in more of adapting everything to suit a stepped down voltage, as opposed to just having a stepped down voltage in the SD connection and running the rest on 5V. In future design we think it would be more optimal to adapt voltage to suit the SD individually.

Verification

The voltage running from the output of our voltage regulator met our requirements at all component connection points on the PCB.

Component	Voltage Readings (V)
Microcontroller: Vcc, UVcc, AVcc	3.3
GND	~0.2mV
LCD Vin	3.3
SD Vin	3.3
Power, Record Switch connections	3.3

Figure 14. Voltage value received by PCB components

Additionally, we found that our voltage regulator was extremely successful in outputting a desired voltage for a large range of inputs. Shown below are the results from a range of inputs at the Vout test point on the PCB.

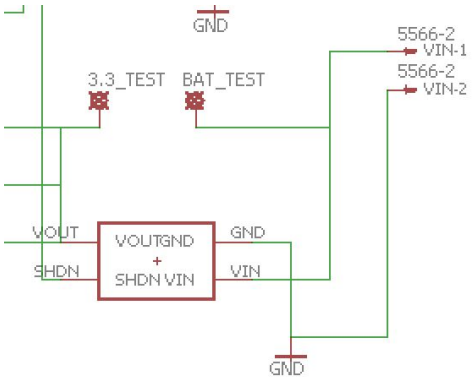

Figure 15. Connected voltage regulator

Vin (V)	$V_{3.3\_test}$ (V)
3	3.352
4	3.355
5	3.356
6	3.356

7	3.360
8	3.361
9	3.362

Figure 16. Tested voltage regulator values

Costs

We ended up each making approximately five purchases around $40, bringing our estimated materials cost to around $400, which we'll round up to an even $500. We then ended up each giving around four 10 hour weeks and two 20 hour weeks, for a total of 160 labour hours. At a rate of $50 an hour, that's $8,000 in labour, bringing our total estimated cost to $20,500.

Conclusions

Throughout the course of this project we learned a lot about where to expect challenges. At the outset, we were very concerned with our optical components, because that's the area in which we had the least experience. We figured the software would be relatively easy, as that's where we were the most experienced. And the hardware we placed somewhere in the middle of our risk assessment. However, risk assessment has very little to do with technical skill, and everything to do with opportunities for failures to arise. Our optics subsystem contained essentially three parts, and thus could only ever have three things go wrong. Our code had hundreds of lines, backed up by dozens of assumptions, any one of which could fail.

With regards to the high-level requirements we had set for this project upon initial design, we were able to display a working device for both points relating to chord display and chord interpretation. However, this was not in conjunction with our developed hardware. We misjudged the critical points that would lead to a success in component integration, especially on a time restriction. Despite these shortcomings, we feel this is a project with a lot of room to move forward, not just in the completion of our initial expectations but in its ability to be added to. Display quality and adjustment, multi-track chord determination, application to acoustic instruments, and a successful recording system are all capable within this system outline.

Ethically, we feel that we have met the guidelines that we initially set relating to the IEEE code of ethics. We were careful to address possible cases of safety risks, particularly in our independent use of equipment such as soldering irons. Regarding point 3, making honest estimates based on good and reliable data, we made steps in our project based on testing and research and feel that we have met these standards. We also accept any honest technical criticism on our project and wish to use this to improve what we can moving forward.

References

[1] Indiana University Jacobs School of Music, “Introduction to MIDI and Computer Music,” [Online]. Available: http://www.indiana.edu/~emusic/361/home.htm

[2] Technopedia, “Heads Up Display (HUD),” [Online]. Available: https://www.techopedia.com/definition/1917/heads-up-display-hud

[3] National Taiwan University Department of Computer Science & Information Engineering , “The MIDI File Format,” [Online]. Available: https://www.csie.ntu.edu.tw/~r92092/ref/midi/?fbclid=IwAR39gPkp0hoLUSSucrctwbpuILMt1LIS_WY5FrU0wOdgAq-B3t5dPVLEDHE#midi_event

[4] Instructables “Smartphone Heads Up Display,” [Online] Available: https://www.instructables.com/id/Smartphone-Heads-Up-Display-for-Car/

[5] Open Music Theory, “Triads and seventh chords,” [Online] Available: http://openmusictheory.com/triads.html

[6] KVR Audio, “Midi Chord Analyzer,” [Online]. Available: https://www.kvraudio.com/product/midichordanalyzer-by-insert-piz-here

[7] Atmel, “8-bit AVR Microcontroller with 16/32K Bytes of ISP Flash and USB Controller,” ATmega32U4 datasheet, 2010. Available: https://www.pjrc.com/teensy/atmega32u4.pdf

[8] IEEE Code of Ethics. Available: https://www.ieee.org/about/corporate/governance/p7-8.html

[9] OpenLab Pro, “Interfacing Micrcontrollers with SD Card,” [Online]. Available: https://openlabpro.com/guide/interfacing-microcontrollers-with-sd-card/