GameFrame

Group 16:
Allen Chion - Computer Engineering
Levi Masters - Computer Engineering
Israel Soria - Computer Engineering
Frank Weeks - Electrical Engineering and CS

Contents

1. Executive Summary

Passing time in the age of the smartphone has become somewhat simplistic and monotonous; open an app, tap the screen in a few places to cycle through content, receive exclusively audio and visual feedback, rinse and repeat. We seem to find ourselves in these cycles of content absorption without any real stimulation beyond what is experienced when awarding a “like” to some kind to a digital post. Gaming on these flat screened devices can also lose a lot of its charm, as all one really experiences is the display of a make-believe game layout and whatever sound the speakers play when a piece is moved. In the modern world, not enough attention is given to the smaller details in gaming and handheld entertainment. Game hardware neglects details such as haptic feedback from button pushing and discrete modules for each individual piece in a game rather than everything blending into one big screen.

The GameFrame breaks some of the monotony of mobile entertainment. With physical buttons that actually move and respond as they are pressed and real physical modules corresponding to the movable areas of a game, the GameFrame has a much more interactive feel as a device compared to a plain flat screen. The GameFrame is a device that can host a game such as chess for multiple players or even for player versus computer. The device being portable and with sufficient battery life can be used as the primary source of entertainment during travel, or other periods of downtime. The GameFrame also does not have loose pieces to be lost in transit, and cleanup is as simple as putting away a book! Being so simple, the GameFrame hopes to establish itself as an essential item for those who want more from their mobile gaming experience.

A game that inspired some of the design of the GameFrame is a rhythm game called “jubeat.” This game plays music and displays patterns corresponding to the music that the player must tap in rhythm. This creates an experience with immersive and sensational appeal. When the game was released into a mobile platform, it seemed to lose most of its charm. The game became dull and the major factors of physical touch response disappeared. This is exactly what the GameFrame hopes to add back into the gaming experience - the enjoyment and stimulation of a real physical game

Using the Jetson Nano to run the games, GameFrame has sufficient computational power to run and play games using its gaming engines and game solvers, of which we might refer to as the AI. The primary factor that impacts the GameFrame’s ability to accomplish these things is cost. Durable housings, high density batteries, and lightweight components all tend to cost more as they move up in performance. Ultimately, the development model and prototyping means we spent more overall, as components were bought individually and manufacturing did not take place at large scale. That is, we were unable to acquire many parts at wholesale prices. The end product can have costs cut substantially as the manufacturing process is improved, parts are bought in bulk and supply chain issues resolve.

As far as environmental impacts, the Game Frame contains a battery, as well as computer chips utilizing silicone and electricity. One of the more popular batteries on the market that are rechargeable are made of lithium-ion, which can cause fires under stressful heat conditions. The GameFrame both eliminates the need for a larger battery as well as reduces its overall power consumption by running efficient hardware components and utilizing efficient programming

practices. With good disposal and recycling practices, our portable device's environmental footprint can be accounted for. Environmentally conscious alternatives are also a consideration for any future prototypes.

2. Customers, Sponsors and Contributors

2.1 Customers

2.1.1 UCF Engineering Department

The department of electrical and computer engineering at the University of Central Florida, as directed by the organization ABET are seen as customers that we (hopefully) satisfied with our project.

The department of engineering requires students to partake in a design challenge. This challenge incorporates all the aspects of the engineering degree of the students involved. For our particular group, this included aspects of both computer and electrical engineering. This means circuit design, logic and application programming, and embedded programming were all incorporated into the project.

This project satisfies the department's demands. It is a design project complete with a physical board layout on a PCB, a microcontroller with embedded programming, as well as software applications. Each member of the group was involved in numerous aspects of the project, but also primarily works within the specialized area of expertise they hope to gain skills for.

2.1.2 UCF STEM Day

STEM day is a free event put on by the University of Central Florida in order to showcase science, technology, engineering, and mathematics programs and activities at the university to Kindergarten through twelfth grade students.

UCF annually hosts the event with different projects and activities the university has in relation to the STEM fields. This event generally needs lots of *interactive* STEM related design projects and presentations. The university prefers having those projects and presentations created by UCF students. In the past, robotic games have been used to display class projects focused on design. They show the desire for interactive game-like projects continues to be high.

The GameFrame was designed to be an interactive gaming board that can accommodate one or two people. This means that students participating in the STEM day would be able to play against other students, or against the computer. They are able to learn about the kind of things possible to achieve at the University of Central Florida in these programs. The GameFrame also serves as a physical representation of programming and software, which is often difficult to showcase in person to younger audiences.

2.1.3 Parents

Another audience seen as customers are parents who have children in the age range that requires constant supervision. Additionally, these children are often developing their critical thinking and logic skills.

The need of these parents often involves their children sitting still and quiet for a period of time while they cook dinner, do laundry, or simply want to recover with some peace and quiet. The easy solution can often be to hand them an internet enabled smart device. However, this presents the problem of moderating what the child is able to access. This winds up being just as much supervisory work. On top of this, lots of seemingly good content on the internet can lack interactivity and lead to children losing out on valuable problem solving and critical thinking development. A device that can be stand-alone and interactive would be a much better solution.

The GameFrame happens to be a stand-alone and very interactive gaming device. No worry needs to be given as to what users or children may be exposed to, as the content is complete and understandable at the time of use. No unexpected content is streamed to the device. With haptic feedback and multiple thought-provoking games to choose from, young minds can be engaged, and problem solving skills can be exercised. The GameFrame provides hours of entertainment for parents to provide to their children while they catch up on the day.

2.1.4 Mobile Gamers

People on the go might want something to do to keep themselves busy or have some entertainment. These people, when traveling, often find themselves with lots of downtime, but little to do during it. Many forms of entertainment either require a stationary area of operation or require a stable internet connection which is not always available to travelers.

The GameFrame provides an excellent solution for the people on the go. The gaming platform does not require an internet connection, has no moving pieces, and is completely self powered and freestanding. Additionally, this device is much more engaging and interactive than smart-phone gaming, and will not eat into the battery life of your cell phone over time.

2.1.5 Waiting Lounges

Offices where people would generally be provided with magazines, toy chests, or televisions while people wait for their appointment could benefit from the GameFrame.

Waiting for an appointment can be one of the most mundane and time-wasting experiences a person can go through. Little work can be done, no real insight as to the timeframe of the wait is known, and entertainment is generally limited to outdated magazines or house renovation TV shows. A solution that both engages and entertains someone would be a perfect addition to any waiting room.

The GameFrame provides an all-inclusive ticket to entertainment for bored customers. With games most people know how to play in a novel package most users have not grown tired of, patients in waiting lounges can have something to do other than staring at their phones or reading

old magazines. The GameFrame has little barrier to entry and accommodates users of most ages. It aims to provide a pleasant and interesting wait period for one's next appointment.

2.2 Contributors

2.2.1 Members of Group 16

Allen Chion
Allen Chion is a computer engineering student at the University of Central Florida. He specializes in embedded systems hardware and programming.

Allen is a member of the development team who is the primary embedded programmer. He is responsible for the main bridge between the hardware and the software, as well as integration testing between the two main components. He is also responsible for designing the housing for the project.

Levi Masters
Levi Masters is a computer engineering student at the University of Central Florida. He specializes in logic systems design, artificial intelligence, programming backend applications, and software architecture.

He is a member of the development team who is primarily responsible for the backend application that the GameFrame runs games on. The logic system of the game as well as the AI that plays against the user in single player mode are also delegated to him.

Israel Soria
Israel Soria is a computer engineering student at the University of Central Florida. He specializes in programming frontend applications, machine learning, and web development.

Israel is a member of the development team who is primarily responsible for the frontend development and user interface and user experience of the device. Additionally, he helps with the backend development and testing, as well as prototyping and creating automated tests.

Frank Weeks
Frank Weeks is an electrical engineering and computer science student at the University of Central Florida. He specializes in the hardware for this project, but also has extensive experience with high level software development due to his computer science background.

Frank is a member of the development team who is primarily responsible for the hardware and electrical engineering aspects of the projects. Much of the choice in batteries, voltage regulators, displays, switches, and input/output devices are chosen or designed by Frank. He is largely the designer for the logistics of the hardware and physical challenges of an 8x8 grid with both input buttons and output displays. Additionally, he is largely responsible for the hardware system testing and the development of the PCB and board layout. He had a large part in the manufacturing of the PCB and Switches and added to the embedded software.

2.2.2 Consultants and Advisors

Jon "Box" Klages
Jon Klages is a contact with knowledge and experience of arcade machines, including jubeat - an arcade-style game that inspired some of the GameFrame's design. He has extensive knowledge of the inner workings of arcade-cabinet-style games. Jon served as a consultant for the development team for the groundwork design ideas and understanding of what components in general will help get the job done.

Arthur R. Weeks Jr.
Professor Arthur R. Weeks Jr. teaches at the University of Central Florida in the department of Electrical Engineering. Dr. Weeks provided consulting information for us on how to implement various aspects of embedded and hardware design. Many tools and techniques that were used for device construction were provided by him.

3. Project Narrative

3.1 Project Motivation

Our group consists of three computer engineers, and one electrical engineer and computer science dual major. So, we were looking for a software intensive project which still involved electronics. While engineering is often used to solve practical problems, we wanted to create something less practical and instead something focused on fun. After brainstorming and working through a few ideas, we arrived at a gaming robot that plays games like chess. This gaming robot would allow for enough design in the electrical engineering space that it makes sense for a senior design project rather than for a software development class. As well, there are plenty of opportunities for software development and there is plenty of potential AI to give our three computer engineers the space to code. Being essentially an 8x8 grid, we also had options and ideas to code in a few different games aside from chess. Such games included connect four, tic-tac-toe, but other ideas could include something like checkers. Being this open ended allows us flexibility in the complexity of the project. We had the idea that we may run ahead or behind schedule and could simply choose to add more or less features. Being behind schedule, this came into play as we ended up simplifying features. If we finish up a couple of remaining integration bugs and then finish off iterations with housing, this project could also be something fun to show off at events like STEM day, which UCF often hosts for K-12 students.

Our very first ideas included a trading card sorter organizer, a poker playing robot, a smart blind system, and an eco-friendly liquid product dispenser (like soap, detergent, cleaning chemicals, etc.) for in-store distribution. After discussing our goals for this project, we leaned into the poker playing robot as it had the most amount of software complexity. However, various issues came to light with this idea. The multitude of mechanical components involved in handling cards and poker chips would be beyond the scope of our fields. One of the biggest issues was the dexterity of the mechanical arm. After some more consideration, we figured it would be better to change the game being played entirely, rather than trying to solve the problem of playing a card game through different means. Eventually, the gambling robot evolved into a gaming robot.

Some of the motivation for our extra considerations, while still balancing key things like budget, was to build a project and skills worthy of our resumes. Experience with certain devices and languages, interfacing hardware components while following standards, and integrating AI into our game engine are all achievements and skills we can display at the end of a project like this. Though our project did not entirely deliver as we initially envisioned, we still were able to garner valuable experience from the project.

3.2 Project Description

The GameFrame is a portable gaming device designed for those who want something more than tapping a screen or keyboard in their gaming experience. It provides a more physical way of interacting with a game, which is something usually restricted to arcades. This device is a stand alone gaming board capable of, at minimum, chess. It can be played with two players against each other or one player against an AI. The GameFrame is intended to be an enjoyable on the go experience. This machine utilizes an 8x8 grid of buttons that displays the current status of the game and also allows the user to make moves by pressing them. The GameFrame has the versatility for additional games to enhance the user experience. The additional games will only need to be implemented via software configuration.

3.3 Project Challenges

For this project, each of the group members took on some new roles we had never performed before. Each time we wanted to implement a new feature we had to educate ourselves with research first before we were able to start making or tweaking said feature. This resulted in much slower development than a typical project which would utilize prior knowledge and experience while just compounding a bit of learning and research. We all have personal lives, school, and jobs to be involved with as well, and did not have an uninterrupted eight hours a day to devote solely to this project as we would with one in a career. Unique schedules that do not always line up also created a need for some kind of organizational scheme. The scheme had to allow us to keep working on a schedule that works for each of us with the short time that aligns being allotted for important meetings and briefings.

This project began towards the end, or arguably the middle, of a pandemic. Working in such a way created a list of additional challenges to overcome. Scarcity of resources has hit most sectors of global trade, and has lasted the duration of this project's development. Although distribution of all sorts is of concern, a very relevant specificity to mention would be the known semiconductor shortage. Higher prices are something we had to cope with, as global infrastructure has taken a massive hit in the past year and a half. Also on top of this, demand for consumer electronics has skyrocketed, which worsened the strain on having electronic parts available. In the beginning phases of this project, we were confined to remote contact only, however this challenge slowly resolved some as the world slowly reopened and more people got the vaccine. Most of the discussion of design and implementation of this project had to take place over VoIP, leading to a lack of aspects of interpersonal communication which we would have had otherwise.

3.3.1 Software Challenges

The original consideration of an artificial intelligence design, we would have had the machine run many different instances of the game against itself and reward it based on how well it does. To create different levels of difficulty, we would add randomness into the algorithm itself, or even reward it differently depending on the different difficulty levels we hope to achieve. Three discrete difficulty levels would have been created for the user to play against and each of them would correspond to the different difficulty levels of easy, medium, and hard. Seeing as our hardware supports the python coding language as well as many different libraries and frameworks, we could have also largely imported certain aspects of AI, but we would have kept most of it proprietary both to have control over it and learn more about AI implementation. Unfortunately, due to time constraints, and unexpected complexity of this approach, this design had to be scrapped, and replaced with a simpler minimax backtracking algorithm.

3.3.2 Hardware Challenges

The hardware had design hurdles due to the screen size, supply chain issues with some parts, PCB etching, working with a CNC router, and the overall surprisingly mechanical demand in button design and creation.

The easiest to discuss would be the obvious supply chain issues. Some parts were only available as hard-to-solder surface mount or not available at all. Likewise off of supply issues was our limited choices in screens. Having vertical limitations on our screen required custom buttons, required using very thin and likewise very fragile wire for visibility, and even required extra accommodations on the GUI end of design to help with hardware-software alignment.

Needing these custom buttons required we learn how to use a CNC router. The CNC router took some experimenting to get right, with many variations. Bit size, type of bit, spindle speed, material leveling, material cracking or burning are just a small fraction. PETG is not brittle and did not end up cracking much, but it did melt. Luckily it didn't burn much, but it did wrap around the bits like cotton candy which gave us some struggles.

The other main issue for hardware arose with the PCB. When etching it, the image was not properly mirrored in both directions. Luckily, since we fit everything into one layer in order to etch it ourselves we managed to solve this issue by putting the ICs on the bottom of the PCB. This caused their pins to get the extra flip we needed to orient them correctly with all of the traces. Additionally a certain node was in the wrong spot on the schematic and we managed to fix it through the same method: soldering to the bottom to make the resistors connect to the correct nodes.

A minor mention was the heat that the 5V regulators generated due to drawing lots of current. This was not a huge issue because it was expected and planned for ahead of time accordingly. A simple heatsink for each of the regulators caused them to work as intended.

3.3.3 Integration Challenges

Despite how it may seem, integrating our systems actually went relatively smoothly. Though some bugs existed, once we started to work those out, integrating went well. Integration was completed the morning before the meeting so we weren't able to fully showcase it in our demo.

That all said, there were two main hurdles for integration. First off, with better management of our version control and working with versions more similar to each other integration would have gone smoother. The other hurdle was that there was only one device and we worked together remotely. This meant not everyone could test their software with the proper inputs.

3.4 Market Analysis of Competitive Products

Competitive products to ours include a palm-sized digital chess device which uses an unsophisticated LCD and typically requires using a stylus. Alternatively, there are portable chess sets which use a variety of physical chess pieces. A consideration for comparison is also a device like an iPad or tablet PC as these can allow versatility to add and play more games with the same device.

Our solution alleviates some of the downsides of these competitive products. Using a rhythm game cabinet known as jubeat for inspiration we designed a portable GameFrame with 8 by 8 (or 64) tangible buttons. The LCD is of a better quality and does not require the hassle of a stylus. Additionally, with portability in mind, supplementary pieces are not only undesirable, but are not required by our device. Even with chess as our focus, the device has software modularity that allows other games to be designed for it. Though it does not have the full game library capabilities of a tablet, the tactile nature of the device should prove to be a more enjoyable experience. A sample of what jubeat looks like can be seen in the appendices under “Other Links.”

With the aforementioned comparisons in mind there were several goals that exist. The device was to be:

- Lightweight - No heavier than the approximate weight of a laptop
- Portable - Easy to use with no external pieces, and sustainable on a battery for period of time
- Reasonably priced - No more than the price of a tablet all together

Though the price and weight were not quite where we hoped, we feel that taking what we know, making another iteration, and having access to a more reasonable supply chain can reduce the profile, weight, and cost of our device.

Figure 1 (Chess products): Example pictures of competitive products. On the left is the aforementioned stylus comparison. On the right is the referenced kind with portable chess pieces.

3.5 Alternate Considerations

We were considering three other projects at first. We thought of a smart blind that would integrate a bunch of features. Some of which included features from a previous group. We thought the design was simple and reliable, but we didn't know if this project would be something that we would be proud to create and tell future employers about. So, we decided to keep it as a fail-safe in case we could not think of a better design. Also, one of our team members wanted to gain more experience with artificial intelligence, so we decided this project would not suffice. A different project we considered was a refillable station that would be used by grocery stores for refilling plastic containers to lower the waste of plastic. This project lacked something that really held our interests similar to the smart blind, so we decided to not choose this project as well. It seemed like it would be something that really could not be built upon with additional features.

We then considered a kind of robotic arm that plays poker with another person. We realized this project could be a bit closer to the field of artificial intelligence since we would make the robot learn when it is wise to bet and when to fold. This project also grasped our attention more in comparison to the other two projects, because it seemed more entertaining than the others. However, we realized that it would be very difficult to pull this off since the arm would need to be very dexterous since it would be very demanding for the arm to be able to pick up playing cards. So we tried to stray away from card games to avoid the challenges that arose from them. We then decided on chess using the grid-based buttons featured in jubeat. With this system, we had the ability to add more features, such as additional possible games on top of the base inclusion of chess, entirely based on our progress. We did not have time to add many additional features, but we did manage to fit a couple of software engines to the device to showcase the versatility.

4. Requirement Specifications

4.1 Hard requirements

Requirement Type	*Name*	*Initial Description*	*Requirement Value*	*Implemented value*	*Notes on Implementation*
Power	Battery Life	As it is portable, the design will require a sufficient power supply unit. The device should last a minimum of 2 hours so as to be sufficient for portable play. Hopefully, we will be able to reach closer to 4 hours.	2-4 hours minimum	3-4.5 hours	Our battery managed to achieve slightly over our expected values. This is probably because we assumed max listed amperage draw at all times.
Physical	Dimensions	The dimensions of the device should not exceed 12-inches in either length or width. The depth is not mandatory to be below a specific required length.	1-foot width or length maximum	15-⅛ by 9-⅝ inches	The width went over our initial designation. Discussion arose when the only screen options that had an acceptable height were very wide. We decided 16” was the absolute maximum to still be considered portable.
	Weight	The weight should not exceed 4 lbs so as to not weigh more than a laptop with similar dimensions	4-lbs maximum	4.865 lb	This was somewhat out of our control with part availability. A better dimension screen and appropriate controllers for the job would make this easily realizable
Requirement Type	*Name*	*Description*	*Value*		
Monetary	Unit Cost	The device should not exceed that of	$400 per unit	$426.05	We didn't quite make the budget,

		a $400 tablet. Being priced above the $40-60 of comparable cheap portable board games is justifiable, however, with our intent of a higher quality experience.	maximum		but actually came quite close considering we started ignoring it towards the end just so we could make something functioning. This is definitely reachable now that we know what we're going for.
Software	Chess	The device should display the ability to play through a game of standard chess with 1 or 2 players.	Playable chess Both 1 and 2-player functionality	1v1 and 1vAI implemented	Integration and version control issues prevented us from showcasing this properly, but our engine successfully allows both player vs player and player vs computer.

Table 1 (Hard Requirements): This lists our mandatory requirements influenced by customer and group member desires.

4.2 Soft requirements

Requirement Type	*Name*	*Description*	*Value*	*Implemented value*	*Notes on Implementation*
Physical	Dimension	A goal of keeping the device within 2-inches deep should help for our goal weight.	2-inches	1.5 + 2.25 = 3.75 inches	This was a soft requirement, but this would be easily attainable by populating our device PCB with surface mount parts and using a PCB instead of a wire grid.
	Drop Resistance	The machine should be capable of surviving a drop from up to 5 feet, the standard for smart-phones.	5-feet	0-feet	Though a soft requirement, this was far from met. The wire grid uses too thin of wire (for the sake of visibility)

					to withstand reasonable drop heights. Swapping to a PCB grid, this would improve dramatically.
	Temperature resistance	The device should be able to withstand conditions from 0-100 Fahrenheit - typical weather fluctuation.	0° - 100° Fahrenheit -18° - 38° Celsius	4-5°C above ambient	With our heatsinks on the regulators and the Nano, we manage to have parts that are only a bit above ambient temperature. The only problem that would occur is if the ambient temperature is too high.
Cost	Unit Cost	A soft requirement is to make it within the cost of more expensive portable chess games which can go upwards of $300.	$300 per unit maximum	$426.05	$300 might actually be a stretch even if we optimize some of the costs. Looking back now, $350 would probably be a better stretch goal.
Software	Computer Move Time	The player should not have to sit and wait very long for the computer to make a move; 1 second is the maximum time.	1 Second	Near instantaneous	Computer moves happen within a number of clock cycles of the nano. This is considerably less than one second.
	Difficulty Levels	AI should have 3 (or more) difficulty levels so players can have more personalized experience.	3 Levels	Easy, Medium, and Hard	Three varying AI levels are offered through tweaks in their algorithm.

Table 2 (Soft Requirements): A table of our desired requirements representing stretch goals we had.

4.3 Constraints

Constraint Type	*Name*	*Description*	*Cause*
Power	Battery	The device must be able to operate off of sustained battery life.	The device is portable
	Power Port	There needs to be a port that allows charging via a USB power source.	The device needs to be rechargeable
Physical	Buttons	There should be 64 physical buttons or partitions.	The device must play chess
Software	Game Library	Games solutions that require a demanding GPU cannot be implemented.	Computational power of hardware
Cost	Budget	The project has a budget limited by the expenditure capacity of our group members.	We are not sponsored
Time	Due date	The time we have to work on our project is limited	Senior design has two semesters

Table 3 (Constraints): A collection of constraints imposed upon us by outside factors or design decisions.

4.4 Realistic Design Constraints

4.4.1 Economic

An economical constraint is that our budget requirement is $400. Aside from that, we personally did not wish to spend an exceptional amount for this device. This left constraints on our product because if one area of the device costs too much it may affect other parts of the device; it caused us to go over our intended budget for some things while cutting corners in other areas of our device. As previously mentioned in our project challenges, the availability of certain parts also caused us to change something out if it was not available.

4.4.2 Environmental

The impact our device has is that it could replace the need for people to purchase multiple different kinds of board games and instead have them all in one device. This would reduce the need to create board games and save those resources, since we produce the same game but

digitally. An effect the environment has on our device is also our power usage. We aimed to reduce our environmental footprint by trying to use recyclable materials where possible. Any chemicals used (ie: for etching) were properly disposed of at local waste management facilities. As stated in the "Related Standards" section, there is a required criteria and satisfying optional ones can lead to us having bronze, silver, or gold level conformance.

4.4.3 Social

Our product's primary social constraint to have considered is the fact that it will only support at most two players. There are not many gridded board games in existence that can support more than two players. However, given the nature of the device, it is still possible for more than two people to use the product through taking turns playing or having non players watch the two players instead.

Another social constraint considered is how the bystanders are affected by the device. For example, the game is not very loud so as to not potentially disturb people in the surrounding area. Proper volume control would be implemented if we finished sound implementation (we only partially implemented it) so the user could adjust according to their environment.

4.4.4 Political

For our political constraints, we dived into the various laws and regulations that dictate consumer electronics. There are many government agencies that oversee these laws such as the Federal Trade Commission and the Department of Energy.

To ensure that a product is safe for children, there are a few laws in place that require strict compliance and testing to allow a product to be marked as safe for children. Otherwise, the product must contain a label that states "keep out of the reach of children." While a lot of these laws, such as the Consumer Product Safety Improvement Act that states electronics intended for children must contain less than 100 parts per million of lead content, are important restrictions, many of them did not have a direct constraint on our ability to build this project. Most of this is due to the fact that we are not manufacturing these parts. We were able to safely assume that any component of the build we used was already compliant with these safety laws, since they were available to purchase from reputable sources. However, we also kept in mind that we needed to thoroughly research a material or part that we intended to use in order to make sure they already complied. If a part seemed like it was not safe or was too sketchy, we reconsidered other parts even if it meant we had to spend a bit more money on a more reputable product.

A law that definitely caused a constraint in our project since we wanted it to be safe for children is the Federal Hazardous Substances Act. In the law, there is a section that states the product must not contain any sharp points or edges. This must also hold true where normal use and potential damage, or abuse, must also not expose any sharp point or edges. While the law states that this must be evaluated by a commission, we did not actually have it evaluated by a government agency. Our prototype is not something that we would yet release to the public because we want to satisfy this rule. However, this law is still an extremely important

consideration for if we went forward with iterations. This constraint has us consider important design touches where the corners and edges are rounded off, the wire grid using nails (though cut and rounded) are replaced by a safe PCB, and that we house it with sturdy material that can not be broken through during normal use, should we decide on continuing forward.

Another important constraint, while not directly a law, is the Transportation Security Administration (TSA) policy that prevents a lithium ion battery of 100 watt hours or more onto a plane. Obviously we wanted a product like ours to hold a really big charge and have extended use capabilities, but we also wanted the product to be easily portable and lightweight. A 100 Wh battery would add a bit too much to the weight and break this policy. The battery capacity we have is 66.6 Wh, with a better weight, good battery life, and also squeezes into this policy. Our potential users can easily bring a battery like this into an airplane or wherever they go.

While this constraint did not affect the physical design of the product, copyright laws determined what games we could install on it. To avoid any potential copyright infringement, the main games we considered were part of the public domain such as chess and checkers. There are obviously many more choices than those, but to make things easier, we checked if the game we wanted to implement were in public domain first.

4.4.5 Ethical

From an original idea perspective, we wanted to make sure that our project is our own unique design and not something copied from someone else. While we did not have to worry about creating our own games since we planned on implementing games in the public domain, it was important to do our own implementation using our controls.

While mentioned in laws that are aimed towards the safety of children, we seeked to ensure the product, or future iterations of it, are safe to use for everyone. Ethically speaking, we should not take shortcuts in development in a way that may harm the user. We sadly did make compromises on our prototype. Our investigation suggests that costs might go up to avoid these compromises in a single-unit environment, but perhaps become cheaper in a manufacturing environment. This is due to the main causes of infringement being the wire grid and nail contacts, which would be replaced with a PCB.

4.4.6 Cultural

Currently, the only demographic that this product is targeted towards are English speakers. Due to the cultural limitations of our project members, there are no other languages we supported on the GameFrame. Ideally, we would like to support many languages so that our product has the potential to reach a more international audience, but given our own cultural constraints, it is likely that English will be its only supported language for now, as it is the language all of us share in common.

Another limitation we have is that it was not worth implementing games our main demographic did not understand. Games from other cultures like Shogi and Mahjong are most likely not

popular enough for people to immediately understand. Of course as previously mentioned, it is not out of the picture to consider if we go along with how we wanted to seek a more internationally inclusive audience. Thankfully, the modularity of the software allows for us to specifically target additional demographics, but it would require extra labor costs and time.

4.4.7 Health and Safety

A health restriction that was considered as a constraint for our device includes ensuring the device does not heat up too much in case someone decides to play the GameFrame on their lap or leaves it in the heat of weather. Making sure that the device is not too heavy was also good to avoid carrying and moving excess weight. This meant that we made strides to ensure the GameFrame is not too heavy or heated too much. We were mindful of the components we used. Cables were exposed in the demo, but we had a housing in production to ensure they would not be exposed for the safety of the users and that they could be contained within the product.

4.4.8 Manufacturability

Manufacturability of the product restricted the design that we were able to implement. We are only able to manufacture at the level of college students, even with access to some tools like a CNC router or 3D printer. We do not have access to typical industrial level manufacturing assemblies or a production pipeline. We were somewhat limited by the services and tools that are available to us. All of the manufacturability of hobbyists, as well as some of the perks of being students, such as educational licenses and lab access, are the main things we got to work with. Luckily, our contacts and consultants also had some equipment available to us.

4.4.9 Sustainability

Issues arose from sustainability. Being aware of the components that were used the most - the panels - helped us in identifying sustainability concerns. Parts of a device that are used the most must be resilient in order to ensure continuous usage. We needed to make sure that the constant application of force on our panels, for example, would not damage them or any other part they interact with. Since the issue was not completely avoidable we at least made it so that the panels have good durability and a certain level of feasible repairability. These are both constraints sustainability caused.

Another design factor that put a strain on sustainability was the portable nature of the device. Since the GameFrame was intended to be portable it needed the ability to endure certain drops and being carried around. The constraint here dictated the viable materials we could use. We restricted ourselves on having sufficient drop durability because of the 13 mil diameter wire that we used.

4.5 Standards

Standards are created to provide a cohesive way of designing, testing, and evaluating engineering products. There are many organizations that create standards for a variety of fields for different engineers to follow. Some of these include the American National Standards

Institute (ANSI), International Organization for Standardization, and the Institute of Electrical and Electronics Engineers (IEEE). For our project, we acknowledged certain standards created by IEEE that affected the overall design of the product. Following these standards helped the product to be easily evaluated if we chose, and will ensure safety for consumers on future iterations. While these standards were acknowledged, most, if not all, of these standards were used as more of a general guideline during our design process for our project rather than a strict list to adhere to, as there were many limiting factors to our actual ability to comply with them. Some of these requirements included certain manufacturing and corporate practices that we had no control of or which we could not apply to the scope of this project. However, if applicable in the future, these requirements could easily be transitioned to have actual compliance. On the software end, we created our own personal standards for how we intended to write our code. This helped streamline our development process. In places when we forgot to adhere to this, integration problems arose. Some of the problems were fixed much more rapidly by changing or creating a standard.

4.5.1 Software

Standard Type	*Standard Choice*	*Explanation*
Square Assignment	(row,col)	Call any particular square by its row and column, starting at the top left with (0,0).
Function Casing	CamelCase	Camel casing is used when calling or creating a function.
Object Casing	snake_case	Snake case is used when calling or creating a function.
Variable Casing	snake_case	Snake case is used when calling or creating a variable.
Global Variable Casing	SNAKE_CASE	Capital snake casing is used when calling or creating global variables.

Table 4 (Software Standards): Personal standards used for software, some of which are based on common practices for the language we used (ie: Python).

4.5.2 Environmental and Social Standards

IEEE Std 1680.1™-2018 Standard for Environmental and Social Responsibility Assessment of Computers and Displays is aimed to reduce the environmental impact and improve social responsibility when it comes to electronic product usage. This standard has two types of performance criteria: required and optional. Required criteria must all be met for the product to follow this standard. Meeting optional criteria can earn a product points to achieve different levels of conformance to this standard. There are three tiers of conformance: Bronze, Silver, and Gold. Meeting all of the required criteria without any of the optional criteria will instantly grant

the product Bronze level conformance. To meet Silver level conformance, the product must meet at least 50% of the optional criteria and all the required criteria. To meet Gold level conformance, the product must meet at least 75% of the optional criteria and all the required criteria.

This standard's requirements can be broken down into 10 parts: Substance management, Materials Selection, Design for end-of-life, Product Longevity, Energy Conservation, End-of-life Management, Packaging, Life Cycle Assessment and Product Carbon Footprint, Corporate Environmental Performance, and Corporate Social Responsibility. Each category has their own detailed required criteria and optional criteria to satisfy compliance of that category. Many of the optional criteria are just expansions of the required criteria. For example, a required criteria to satisfy the standard is to use at least a certain amount of recycled plastic, and you can gain extra points towards a higher tier of compliance by satisfying the optional criteria of using even more recycled plastic.

In the substance management and materials selection, some of the required criteria include no mercury in light sources, less than 25g of bromine and chlorine in plastic, at least 25g of plastics are recyclable, and plastic must be separable. Optional criteria that give more product points include even more restricted cadmium, beryllium, and using more recycled material. Some of these required criteria are geared more towards manufacturing, which is something we were not able to comply with since we bought whatever is available. Where we could, we chose parts already made that comply with these standards.

In the design sections, some of the requirements state that certain components such as lithium-ion batteries need to be treated and must be visibly identified. The plastic parts must also be separable. To ensure product longevity: service support, spare parts, and battery replacement must be services provided by the manufacturer. Some optional criteria include longer rechargeable batteries, ease of repairability, and the ability to remove batteries.

While this did not apply to this project specifically, there are many optional criteria that involve the corporate level of the product. It is highly encouraged for the product's greenhouse gas emissions, greenhouse gas emissions from transportation, corporate carbon footprint and social responsibility be taken into account. Most of these requirements are optional.

Design Impact of the Environmental and Social Standards

While this standard provides a good guideline to follow, there is a lot in the standard that did not apply. The main reason was that many of these standards apply to the manufacturing process of the entire product. For this project, we sourced all of our parts from already manufactured components, and we only created a single completed product for our Senior Design demonstration. Required criteria such as having spare parts available, having service support, and ease of recycling were not met due to the project's scope. We also did not have the proper tools to determine the carbon footprint created from our time creating the product. It would definitely be something to consider in an actual, real world product development cycle. As a result of that, complying with this standard did not work from the get go. These things would

most definitely be considered if we had plans to sell the product or had plans to set up our own manufacturing process.

However, there were some criteria in the standard that provided some good guidelines with how we would like to approach this project. For example, high energy efficiency was something we strived to achieve. While we did not try to meet the ENERGY STAR program of efficiency, it was within our best interest to try and make it as energy efficient as we could.

4.5.3 Battery Standards

IEEE Std 1725™-2011 Standard for Rechargeable Batteries for Cellular Telephones is aimed to ensure a reliable user experience and operation of cell phone batteries. While not applicable to this project, the standard also outlines recommendations for the actual design, manufacturing process, and testing procedures of the cells in the lithium-ion battery itself. Even though this project was not about cellphones in particular, our project did use a rechargeable lithium-ion battery to power a mobile platform. This IEEE entry standardizes the criteria for verification of quality and reliability of those batteries.

While the standard aims to minimize possibilities of hazardous outcomes, it is not possible to analyze every potential issue and scenario that might appear in batteries. The design analysis aims to reduce hazards from one or two independent faults during a battery's use. To do this, various design analysis tools include, but are not limited to:

Failure mode and effects analysis (FMEA)
Fault tree analysis
Empirical and/or destructive testing
Reviewing company service records for failure modes and/or trends
Cause-and-effect (Fishbone) analysis
Detailed and extensive design reviews
Reviews of prior design issues to ensure they are not repeated
Reviews of industry standards and test methodologies

Figure 2 (Battery Design Guidelines): Design analysis tools aimed at preventing faults in batteries.

To ensure the safety of the user, the standard outlines user interactions and responsibilities that the user should know regarding the use of the battery. This information should be printed on something for the user to see, such as on the device itself or on a user manual:

- Do not disassemble or open, crush, bend or deform, puncture, or shred.

- Do not modify or remanufacture, attempt to insert foreign objects into the battery, immerse or expose to water or other liquids, or expose to fire, explosion, or other hazard.
- Only use the battery for the system for which it was specified.
- Only use the battery with a charging system that has been qualified with the system per this standard. Use of an unqualified battery or charger may present a risk of fire, explosion, leakage, or other hazard.
- Do not short circuit a battery or allow metallic or conductive objects to contact the battery terminals.
- Replace the battery only with another battery that has been qualified with the system per this standard. Use of an unqualified battery may present a risk of fire, explosion, leakage, or other hazard.
- Promptly dispose of used batteries in accordance with local regulations.
- Battery usage by children should be supervised.
- Provide an explanation of security implementation and battery authentication.
- Avoid dropping the device or battery. If the device or battery is dropped, especially on a hard surface, and the user suspects damage, take it to a service center for inspection.
- Improper battery use may result in a fire, explosion, or other hazard.

Design Impact of the Battery Standard

Selecting the battery was considered one of the more important parts of the project, but it was also something we did not have the most control over, since we chose one that was already produced and self-contained. However, what this standard helped us do is to know what to look out for when we decided what kind of battery to get for the project and how we should have gone about its installation. Preventing physical damage is part of the standard and that was something important to consider when designing our project. We gave more consideration to housing the battery inside the final build to protect it in the event the device is dropped. We also avoided raw cells for that reason. One important part of the standard we would adhere to if produced for consumers is to correctly display to the user the device's battery life. Another part of the standard that was followed was to make sure that the battery could be easily replaced. With this in mind, we did not integrate the battery into the build in such a way that the battery was hard to remove in case of an issue. However, things such as detecting faults in the battery were not kept in mind due to our limitations. Just like our political constraints mentioned, we were not going to choose a battery being sold on the market which was not already extensively tested, but we did still acknowledge the standard in testing and evaluating if the battery was good and safe to use.

4.5.4 System, Software, and Hardware Verification and Validation

IEEE Std 1012™-2012 Standard for System, Software, and Hardware Verification and Validation is a process standard that defines the verification and validation process that is applied to a product's system, software, and hardware development throughout its design. The process determines if a product meets all the requirements and satisfies its intended use. The purpose of this standard is to establish a common framework for analyzing a product's system, software, and hardware life cycle processes. It defines the tasks for a proper verification and validation process.

Verification and validation are their own separate processes. The verification process aims to check whether the product conforms to requirements, satisfy standards, and satisfy criteria for a complete product life cycle. The validation process aims to check whether the product satisfies

system requirements and satisfies the product's original intended use - does it do the thing it was meant to do? The results of this process can facilitate early detection and correction of any problems that may occur and enhance the management of the development process. It assures that the product will meet the performance metric, schedule, and budget that the project initially set out to meet. It can also provide an early assessment of how the product will perform.

The standard is organized into four parts: Common, System, Software, and Hardware. Common tasks are directly related to planning, support, and management of the verification and validation process. These tasks include generating the verification and validation process, system requirements review, and final report generation. Systems, Software, and Hardware tasks all include design evaluation, interface analysis, traceability analysis, component test plans, integration test plans, qualification and acceptance tests, hazard potential, security analysis, and risk analysis of the product.

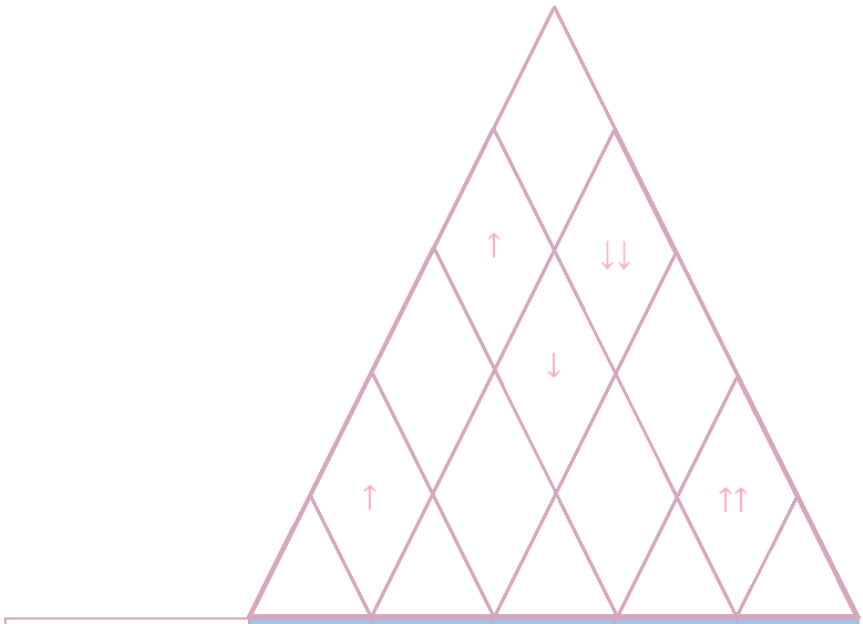
The requirement "perform" means that a verification and validation organization specifies and creates the test, confirms that it has been done correctly, and reviews the results. However, it does not require that the organization does the test itself. As long as the verification and validation organization observes the test, it can be done by the project creators. To obtain the highest integrity level, verification and validation testing is mandatory.

Design Impact of the Verification and Validation Standard

Considerations for this standard shaped the way we tested aspects of our project throughout development. Instead of strict compliance, this standard was used as a guideline for how we verified and validated the goals we laid out.

4.6 House of Quality

The best way to analyze the house of quality would be to explain the relationships on it. Furthermore, it was a good idea to focus on key positive relationships in our design to help meet some goals and requirements. Likewise, avoiding negative relationships or at least balancing them as much as we could was beneficial. Whatever the case for a negative relationship, we needed to lean towards the design decision that would help towards our specifications. In other negative relationships they had no bearing on a requirement, so we were able to lean towards what we saw as most useful or beneficial, or whichever helped us towards a functioning project.



		Unit Cost	Battery Life	Development Time	Dimensions	Weight
		-	+	-	-	-
Low Power*	-		↑↑		↑↑	↑↑
Repair Cost	-	↓				
Development Cost	-	↑↑	↓	↓	↓	
Display Quality	+	↓↓	↓↓	↑		
Lightweight	+		↓		↑	↑↑
Résumé Benefit	+		↑	↓↓		
Target Specs.		<$400	>2 hours	Not specified	<12"x12"x2"	<4-lbs

It should be noted that Low Power is using "-" to denote that less *power consumption is desirable. That is to say, the device would be* more *low power.*

Figure 4 (House of Quality): A QFD of our initial design.

4.6.1 Development Cost and Time

If something takes little time to develop, it likely was something we already knew or was a skill that was not hard. This causes a low development time to have little benefit on our resumes; resume development was a personal goal of ours from the project. Thus, with development time having no impact on requirements and adding nothing towards that goal, resume benefit was what we leaned towards.

The development cost should be noted as being different from cost per unit production. The more we spent on parts for a unit, the more development cost was, but the opposite was not necessarily

true. However, on the topic of development time, in a number of cases, spending more development time saved us on cost to develop. This was most prevalent in our button production, PCB production, and wire grid. Building, crafting, or designing components ourselves was cheaper. Thus our college student wallets pleaded we lean towards devoting more time over the alternative of spending more money.

4.6.2 Display

As the quality of the display increased, the complexity of its architecture and of the design to facilitate it did as well in regards to screens that were not all-in-ones. The quality of the display also cost more and was incredibly detrimental to the battery life of the device in most cases. Too low on visual fidelity would create a low quality product that does not feel good to use. The display quality needed to be sufficient enough so as to not create a negative customer experience while also being as minimal as possible in order to keep the cost low and promote the longevity of the battery. In the end, the screen was the hardest to get just as needed and threw other parts of our design and decision making off kilter.

4.6.3 Size and Weight

Although this is not true across the board, in general, smaller parts tend to cost less. Simply put, this is because smaller parts use less materials. For electronic components, this trend is less applicable, but can sometimes be true depending on the type of component in question. The good thing about this is that we wanted small parts and we wanted less cost. Getting components that have small dimensions and less price was a win-win. It should be noted that the density of varying parts causes cost and weight to have less, if any, correlation.

Density is a varying physical property that makes it hard to reliably say that weight and dimensions correlate. The density is dependent on what the material for a part is. Even if not reliably true, there is an obvious correlation between size and weight. This correlation is especially true when comparing parts of the same material. It is easier to say that lower dimensions have less weight than to say that something that weighs less has lower dimensions. Thus, the conclusion is that using smaller parts resulted in an overall less weight for the most part.

4.6.4 Power

Smaller parts also commonly have lower power requirements. Low power parts also weigh less typically (eg: low power batteries). For obvious reasons as well, using a bunch of low power parts made the battery life goal much more attainable. There were many benefits for low power parts with no huge drawbacks. Any drawbacks that existed were not those that affected our goal requirements or specifications. That is not to say there weren't any unforeseen detriments. For example, supply chain issues were much more prevalent than we anticipated.

A quick note can also be made about large batteries. The larger a battery given a certain energy density, the more power it can supply. This negatively impacts the weight and the dimensions in favor of power. A balance had to be found to meet both of the specifications, since they were deemed as our most important design requirements for the purpose of being considered portable.

To briefly mention, lithium-ion batteries alleviate the troublesome correlation some and provide good energy per weight. This is covered and expanded upon elsewhere in this paper (specifically, under “Research”). As for our weight vs energy tradeoff, we ended up with a battery in the middle at 66.6 Wh compared to the 130 Wh or 30 Wh choices we also considered. This also was much better for our costs than the 130 Wh option, and not much more than the 30 Wh one.

4.6.5 Repair

Repair cost was not an important factor for us, especially considering we preferred to keep unit cost low since the unit cost was an actual goal we aimed for. It probably is selfish of us as developers but repair cost was not on the forefront of our mind for designing this project. On the plus side, repair cost is somewhat alleviated from the fact that the design and parts were inherently modular. Replacing malfunctioning parts is cheaper than buying a new product. Unfortunately, any customers who are not tech savvy would find this to be little consolation.

4.6.6 Resume

Resume benefit was a very influential goal towards some of our decision making. Being able to reach our soft requirement goal of a four hour battery life did not prove to be as much of a challenge as we thought. Deciding on the right battery and developing the right power managing circuitry developed problem solving or experience that was good for a resume. Additionally, though including AI dramatically increased the complexity of the software, it was a much welcome addition to resume experience for those who worked on it.

4.6.7 Post-project Conclusion

Though these tradeoffs remained unchanged, and thus we kept the same house of quality, the biggest thing we had not considered was how much time was actually much less of a commodity as we approached our deadline. Thus the importance of time became much greater than cost than it otherwise would have.

5. Design

5.1 Overview

The “GameFrame” is an integrated board with the intention of full enclosure and consisting of an 8x8 grid of buttons made of PETG on plexiglass with a digital display under them to show the status of each tile. It consists of an NVIDIA Jetson Nano development board as the brains of the software with an MSP430 to handle the hardware.

The current design uses less than the original four switches for each button. The GameFrame uses one contact point per one button. This came to a total of 64 contact points, each of which used two conductive nails (cut and flattened to increase safety of course). The power regulation of a portable device had to be managed well in order to maintain a proper battery life. The hardware kept the power regulation in the forefront of the design to ensure all components received adequate power, did not interfere with each other, and that we could maintain solid

battery life. The design also implemented a separate cheap controller, the MSP430, in order to handle the hardware and to help facilitate concurrent work on the project. The Jetson Nano, the intended software controller, could have sufficed alone, especially to handle the display hardware. It also had more than plenty of pins to spare for decoding. Auxiliary buttons for power or other features were considered for the side of the machine, but the power switch was included in the purchased battery. Auxiliary buttons (especially for volume control for example) are not things that made it into the design, but are prime candidates for features on future iteration.

As for software, it was modular in nature to support easier development of other games for the device. An overview of the subsystems implemented in software can be found under the "Software" part of this "Design" section.

5.2 Technology

The software for this project runs on the NVIDIA Jetson Nano developer kit (series 945-13541-0000-000) - an especially powerful kit specializing in small AI and machine learning for embedded development environments. This kit is specialized for running serial type programs which we utilize in some of our player versus computer modes for the games run on the GameFrame. Though it is probably possible to run the GameFrame on a lower performing chip, we, the developers, wanted to get some experience with this particular variety of chip.

The Jetson Nano uses the "Compute Unified Device Architecture" that we want to get experience with. The CUDA is NVIDIA's own proprietary platform and API model that is used in their devices. So, experience in CUDA allows for future programming on other NVIDIA devices and future projects that may make more full use of the CUDA. CUDA also compiles programs from C, C++, Fortran, Python, and Matlab with some added commands. Picking up the language is more about the method of coding as opposed to learning a whole new language from scratch. The Jetson Nano comes preloaded with Ubuntu linux on it, facilitating an accessible programming environment.

5.2.1 CUDA for game solutions

CUDA allows for high optimization of parallel computation in graphical based processors. This is especially useful in this particular project, as we programmed in games that have up to 64 differentiated sections during play. Lots of parallel computation was useful for either backtracking.

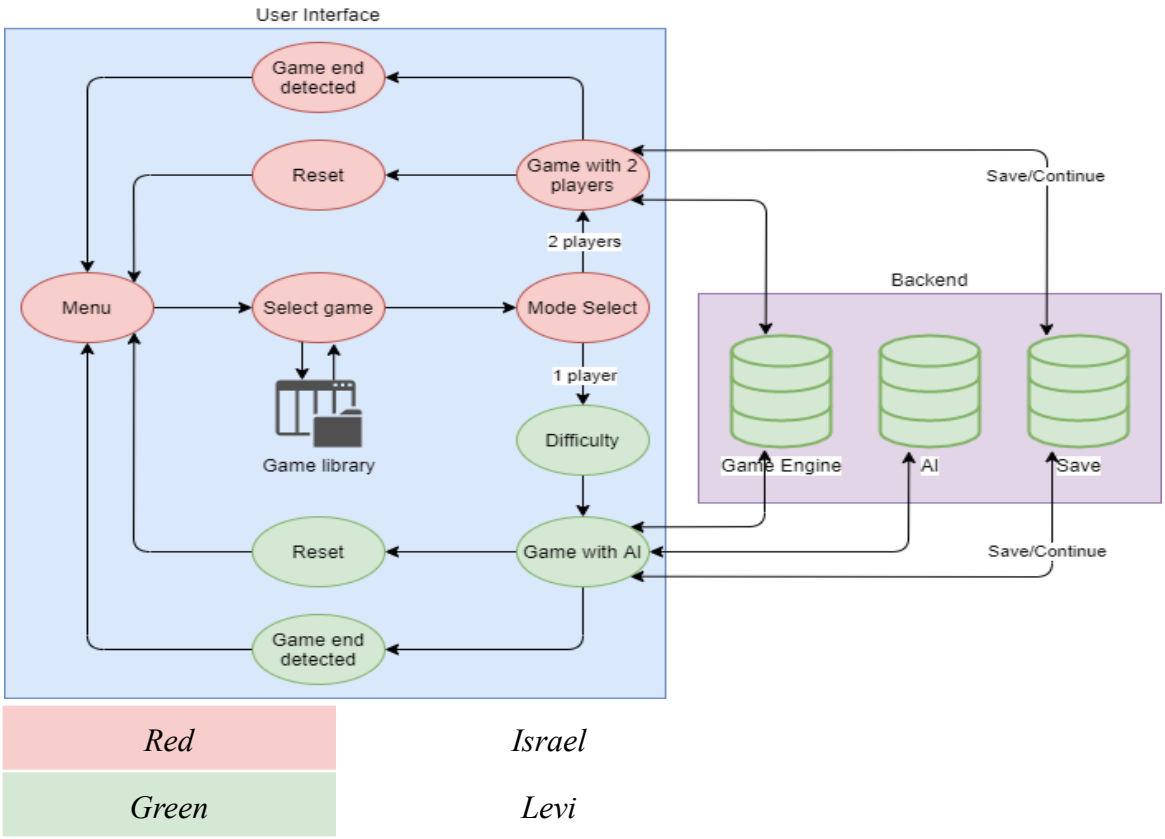
5.2.2 CUDA/Jetson Nano for Embedded programming

CUDA is not especially created to run generic embedded type software/hardware applications like the RaspberryPI or an Arduino board, but it is still more than plenty powerful enough to run these applications. This development kit is absolutely capable of what is needed for the software and hardware interfacing of this project. It may just be somewhat too powerful compared to what is really needed to run the GameFrame hardware components like the buttons, displays and speakers. This device has the necessary 40-pin header with I2C, I2S, SPI, and UART connection types.

5.2.3 Alternatives to Jetson Nano

The Jetson Nano is our choice of development platform primarily for the purpose of solving games when playing against the computer. Other boards would be able to run the games themselves, but may need toned down game-solving algorithms compared to what the Nano is capable of running. RaspberryPI and Arduino boards are two that would be very suitable for running a game and even solving them, though runtimes may be annoying to the player. These boards feature all the necessary components, from embedded programming design to an application capability of running the games we are looking at designing.

5.3 Software



Legend - Despite working collaboratively, this denotes who was primarily responsible for each task.

Figure 5 (Software Block Diagram): The organizational structure of our software design at the highest level of abstraction.

5.3.1 Game Engine Design

The game engine generates a new instance of the game on call of the create game function. This function takes in what kind of game it will create as a parameter, then it initializes a blank state of the game that has chosen to be played. This function is called regardless of if the game is played in single player against the computer or user versus user multiplayer mode. From here,

the status of each piece has a value assigned to it the physical boardis be able to display the game. Game squares have attributes to them such as “is_occupied”, “piece_type = knight,” “color = black/white,” etc to indicate what the current status and availability of each square is. Each square is also assigned a value according to a matrix, so moves can be calculated mathematically. For example, (0,0) is for the top left square with the format of (row,col). The pieces have an attribute as to what square they are on. Both the piece and square are updated upon each move.

The game checks for a win status immediately after every move. Since only one piece is moved at a time for most games, the checking function only checks pieces affected by the most recent move. The checker checks the status of any affected pieces where the piece moved to, as well as the place where the piece moved from. For player versus player games, the game becomes usable again for the next player to make their move after the check is complete and give the user an indication that a move may be made. After the next move from the player, the cycle progresses in the same way until a win, stalemate, or end game condition is found. For player versus machine games, the algorithm to determine the next move is run and executed upon completion of the game win check function. The check win function is run after player input, before the AI picks its move. Likewise, it is run after the AI makes its move, before the player is told to make their move. The game win function also checks to make sure the move is legal and does not result in a gridlock event. In some games, gridlock events are legal to make, but result in a stalemate. The game win function will then end the game and send a signal to send a “stalemate” call to the interface or board.

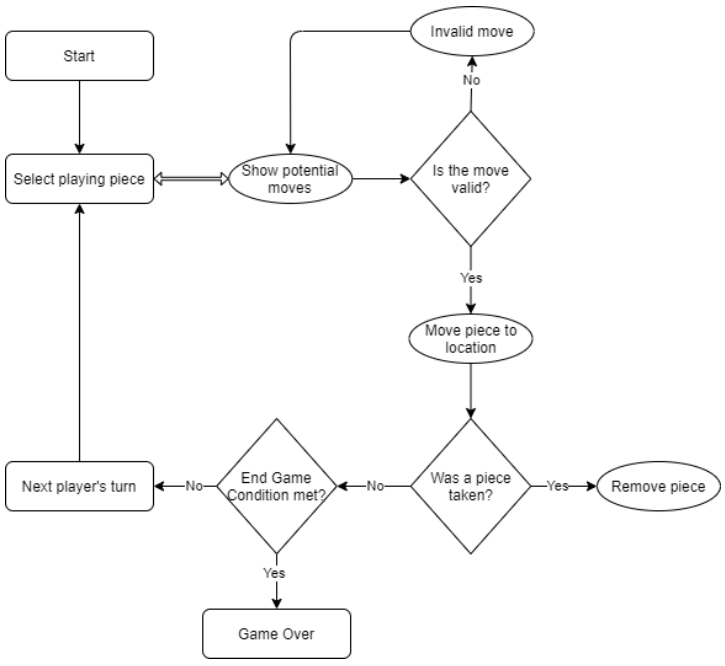


Figure 6 (Activity flow diagram): A diagram depicting a typical gameplay loop.

The storage of squares is in a two-dimensional array. This is consistent with the conventions of calling squares by their column and row. Squares in the array are objects, and declared to have all their individual components assigned initially to zero (or None), but are assigned their true

initial value depending on the create game function. Game squares have attributes related to all games declared, but only those in relation to the game currently being played are modified on call of the create game function. Storage of pieces is within one of multiple different one-dimensional lists or within individually declared variables, depending on what game is being played. Like the squares, pieces are automatically declared zero/None or default values that are modified according to the create game function call. Game pieces are treated as objects with attributes and modifiers.

5.3.2 Player vs. Computer

One key feature of the GameFrame is that one user can play a complex game against the computer. We utilized an algorithm based on a backtracking minimax type algorithm in order to have the computer solve for its optimal move. Once the machine is set to one-player mode, this causes the mode of operation to automatically calculate and execute the next move of a computer player before allowing the user to place their next move. Different difficulties are needed for the computer's move-making algorithm, as not everyone has fun playing at the same difficulty levels. Different games also benefit more from different methods of solving algorithms, so multiple styles may be used to solve games. The minimax backtracking algorithm is the primary one implemented in all games, however certain additional algorithms like random move in tic tac toe, or find possible valid wins were also used.

Since backtracking was determined to be the better solution given our constraints, it was then used instead. In this algorithm, numerous different varieties of solving are implemented in order to determine the difficulty level. In a particularly complex and effective backtracking algorithm, almost every different path that can be taken is explored, then the best one is taken. In order to limit the difficulty of the computer, less paths might be explored, giving the human player more opportunity to “trick” the machine into making a bad move. This also comes with the advantage of resembling more closely how a human player plays chess - analyzing numerous different courses a game may follow and picking the one with the most opportunities of success. By utilizing backtracking, more possible difficulty levels can be chosen from than if using an AI, as a variable can be declared for how much backtracking the algorithm does in the decision making process. Thus, backtracking allows for a slider type of scale instead of a few discrete levels.

5.3.3 User Interface

The part of the software that interacts with the user is the main menu that we constructed and the game software themselves. Once the main menu is displayed, the user interacts with the menu through the use of the buttons on the device. The user selects which games they wish to play and the GameFrame directs the user to that game’s menu screen. Once the user is directed to the game’s menu screen, the user sets the options of how they wish to play that certain game, such as computer difficulty or if they will be playing with another human. The software takes the information given by the user and creates the game that they want to play. Once the game is set up, the user will be communicating with the game trying to follow the game’s rules. The software must be able to make sure that the user is following the rules and the software shows the user how the game is progressing. Once the game is finished, the software will inform the user if the game was a victory, loss, or a draw. The software will then ask the user for it’s next

decision, whether the user wishes to replay with the same settings or if the user wishes to head back to the game's menu.

5.3.4 Class Diagram

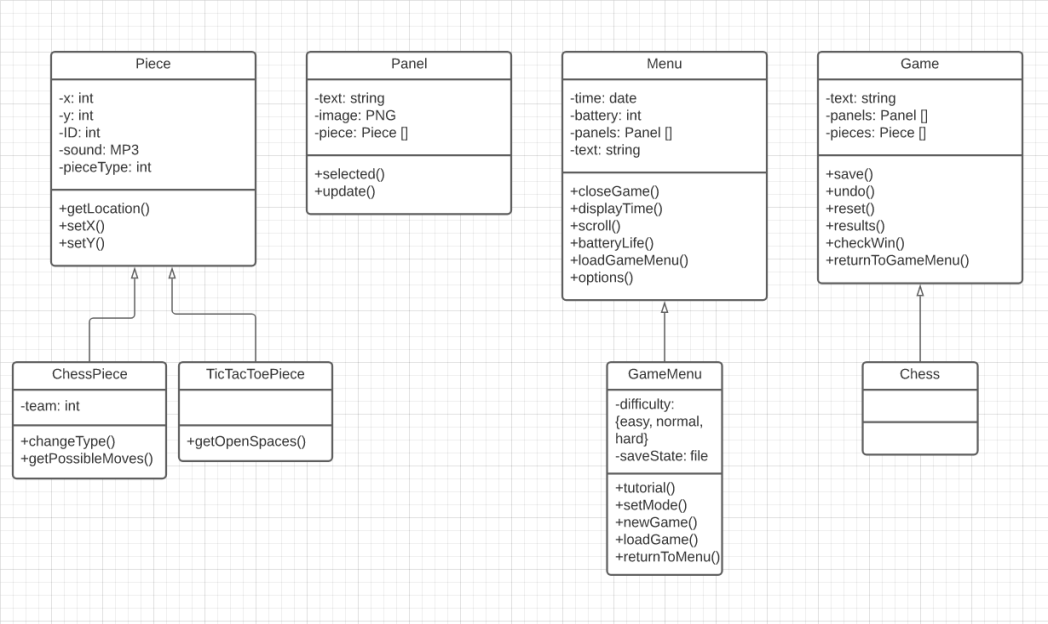


Figure 7 (Class Diagram): The class diagram for our software.

For our class diagram we separated our code into the four separate classes shown above. We will be elaborating on how each class operates individually and how it coincides with the software as a whole.

Menu Class

The Menu Class is essentially the beginning of where the user is taken once they begin to use the Game Frame. Here the user will see a screen similar to the one in Figure 5 in the software prototype. Here the menu must be able to display: the time accurately based on the user's setting, the list of games that are available, the battery percentage remaining in integer form, and text for the game's name and for displaying the product name. The menu also consists of panels which are a derivation of the class Panel. Using these instances of Panel we are able to create certain panels to display text and decide whether they should have functionality or not.

The menu must also be able to perform certain functions such as starting up a game, displaying the time, being able to show the correct battery percent, and load the options for user customization. There is also a scroll() function that we have that allows the user to scroll between the game selection panel. This function was implemented, since we have a game library that exceeds the normal menu page and needs to have more space for other games. Once a game is selected from the menu to play the menu accesses that game with loadGameMenu() and this leads into the game's own menu. This is shown in Figure 6, Game Menu, in the software prototype section.

This menu is known as GameMenu in the class diagram above and is an inheritance of the Menu class. GameMenu performs the same as the main menu but it is tailored to that specific game. It looks similar to the main menu but it also has a tutorial for the specified game and it also keeps the user's last game if they saved it. GameMenu is also where the player specifies whether they are playing with a computer or another person with setMode(). If they play with a computer they will then pick a level of difficulty for the computer, either easy, normal, or hard.

Panel Class

In the Panel class, this is where we focused on the individual panels that will be displayed on the screen. Each panel serves some sort of purpose, they either display something, have a function or they just showcase the background. For example, we already discussed the main menu briefly in the previous class section. The way the panels are used in that scenario is that the panels that lead to important features such as the settings or to games will have the function selected() and will follow the function of being selected. Other panels on the screen however have no functionality to them. Meaning that once they are pressed, nothing will happen. This happens to any unused panels or panels that are simply used to just display text. Panels that are completely unused are simply used to display the background that we decide to place.

Panels that are used in game will also need to be aware as to which game piece should be displayed, for example like displaying a pawn or a knight in chess. This is why Panel has reliance on the Piece class to appropriately display the correct piece. The panels also update themselves whenever the player makes a move that is allowed. The panel updates and then updates another panel that the user has chosen to move. Also we added in features of pressing certain panels for a certain amount of time to perform some feature like an undo turn.

Piece Class

The piece class is responsible for storing the different pieces from different games and their movement. The x and y variables are responsible for the location of the pieces on the board. Since our design is a grid the x variable are the horizontal location and the y variable is the vertical location. Both of the variables range from zero to seven since there are eight panels available for use. Each piece also has an associated sound attached to them to add a nice sense of realism when moving the pieces. We saved the sound files as MP3 files and the images we needed as PNG.

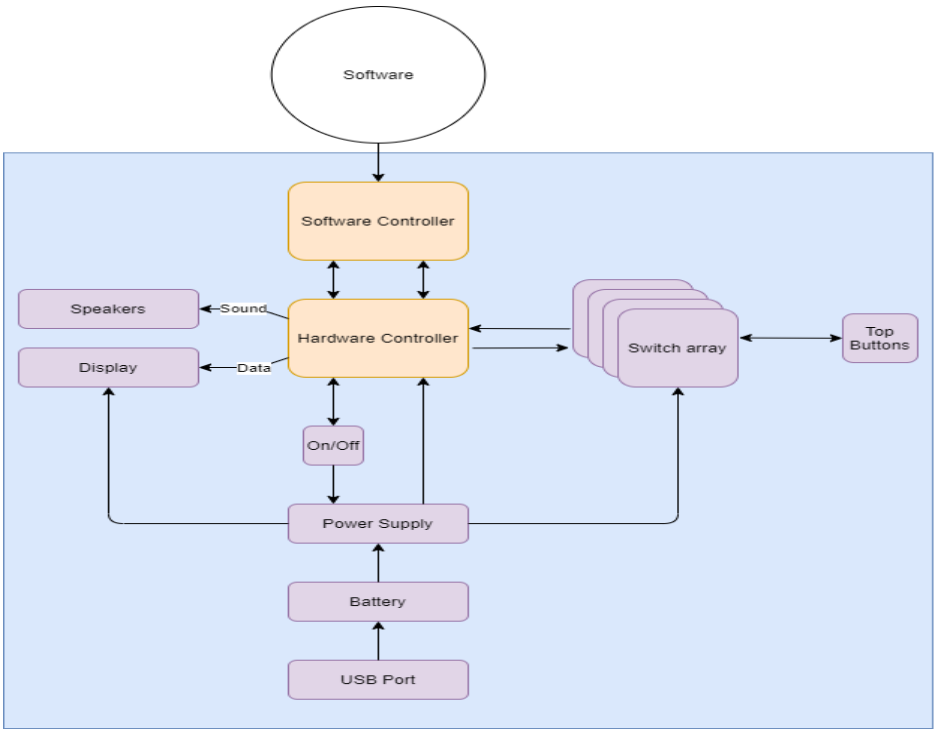
Each piece has a corresponding ID that makes it unique between itself and the other pieces. Every piece also has a type. For example, if the game was chess the piece itself would have it's own ID but the pieceType could be like a pawn or a knight. Based on this typing, the piece is allowed certain movements. We placed example subclasses that would inherit the Piece class, more were added based on the amount of games we implement into the system. The subclass for ChessPiece differs by having a function that allows the piece to change types. This function is only for the pawn types as they can switch to any piece once they reach the opponent's side of the board. Something similar would be implemented for checkers as once the piece reaches the opposite side it can become a king, had we decided to implement checkers.

Game Class

The Game class is the actual game that the user is experiencing. Figure 7 in the software prototype section shows an example of the game being played out. Here the game either begins a new instance of the game or from the game menu the user loads an old save file. The Game class utilizes the panel class and piece class to appropriately set up the game. Once the game is set up the game makes sure if there is a victory or a draw with the checkWin function. While the game is being carried out, the game also has the ability to save the state the game is currently in so that the player may resume at a later time. During the game, the user is able to perform an undo feature in case they committed a move by mistake.

The game also has a feature to return to the game's menu and if they did not save it will ask them if they wish to save before leaving or else they will lose any unsaved progress. If checkWin sees there is a winner or a draw then the results function leads to the result screen where it asks if they wish to reset the game and retry or if they wish to go to the game's menu. The chess subclass is just an example of the games that are inherited from the Game class.

5.4 Hardware



Purple	*Frank*
Orange	*Allen*

Legend - Despite working collaboratively, this denotes who was primarily responsible for each task.

Figure 8 (Hardware Block Diagram): The organizational structure of our hardware design at the highest level of abstraction.

5.4.1 Hardware-Software Interfacing

The main hardware components for interacting with the software were the 64 buttons on top of the screen. However, most microcontrollers did not have enough pins to connect one button to each pin. For example, while the Jetson Nano has 40 pins, only 28 of them are GPIO and even less of those are set to GPIO functionality by default. Some of these pins are used for I2C and UART protocols by default and have GPIO as secondary use if needed. To be able to utilize the limited number of pins to connect all 64 buttons, it was necessary to create a button matrix to connect multiple buttons to a single GPIO pin for input detection. While it was possible to wire all 64 buttons into one pin, it can get immensely complicated trying to uniquely identify each button on a single pin and cause a lot of unnecessary work or possibly use more ICs than really needed. It was a lot easier to divide the buttons up and assign them to separate pins. To strike a balance between pin efficiency and simplicity, we utilized two pins for output and eight pins for input. As a result, only 10 pins were needed for button decoding.

In order to connect the MSP430 to the Jetson Nano serial communication was used. Both devices had UART ports that could be setup for serial communication. The Nano operates on 3.3 V UART communication, while the MSP430 operates with 3.3 V or 5 V. Though both controllers can communicate directly on 3.3 V, we decided to continue using the breakout boards with MAX3232 on them. These ICs and ports added extra error resilience and component protection through voltage regulation and the use of the RS232 standard.

Communication errors still exist every so often on the serial lines, but they are few and far between enough that they could be handled through software. To do this the software only registers a button press once a certain threshold of press transmissions are received. Any transmission that is outside of our range of 0 to 63 is entirely thrown out by the Jetson Nano. The range decided for transmission was the aforementioned 0 to 63 which fits into 7 out of the 8 bits allotted for a single transmission across UART. Other considerations for reducing error were to use the leading bit as a transmit flag bit or to send multiple transmissions for every one registered press. We decided against these because the occasional communication errors have no consequences on our current implementation with the current error handling. Another encoding algorithm considered was to transmit 0-7 in 3 of the bits and 0-7 in the other 3 bits. This might be more optimized since it would use shift operations instead of multiplication, but we went with a single multiplication operation since timing optimization at this level of precision was not necessary for our decoding or transmission.

5.4.2 Hardware Controller

We used a separate microcontroller in order to handle the hardware. This ended up being an MSP430. The MSP430 had a convenient development kit which let us load the program onto the MCU chip before socketing it onto our PCB in a DIP-socket that was soldered to the board. It was easy to use, especially since it was an embedded processor we were familiar with from our course work. More than ease, the MSP430 was on the cheaper end of microcontrollers while also being power efficient. Overall, it is considered a low-power microcontroller.

With the MSP430 having the main function of decoding our buttons, most of our embedded code dealt with this task. The controller outputs a serial and clk to the shift register to strobe the columns. It then has pins setup as inputs to see if the rows have been pressed.

The other task of the MSP430 was to relay the decoded inputs to the Jetson Nano. The details of transmission and communication between the controllers is as described in the previous section (5.4.1 Hardware-Software Interfacing).

The model of choice for the MSP430 was the MSP430G2553. This was exclusively because of our decision on a G2ET LaunchPad. The LaunchPad allowed us the flexibility of using any MSP430 with a 14/20 Pin DIP capabilities and the ease to move it to and from our PCB. It just so happened that it came with an MSP430G2553 as the model included in the experimenter board. The features of the MSP430 did not need to be refined, so we did not look into other DIP-socket compatible MSP430s. We did however do some research on them prior and included details of them in the tables in the research section discussing the MSP430.

5.4.3 PCB

Originally we intended to have two PCBs for our design. We still believe a second PCB would be a better option for implementing our buttons than our current wire grid. As it stands any additional components would best be done as breakout boards or a separate PCB. We only used one main PCB in our design, which actually remained close in function to the original intent.

Main PCB design

Our first idea for the PCB was for it to be home for the hardware microcontroller. In terms of design, we wanted a grid of switches on a large section of the board and thought to relegate the hardware controller and other ICs to one side. An approximation of our idea is depicted in the figure below.

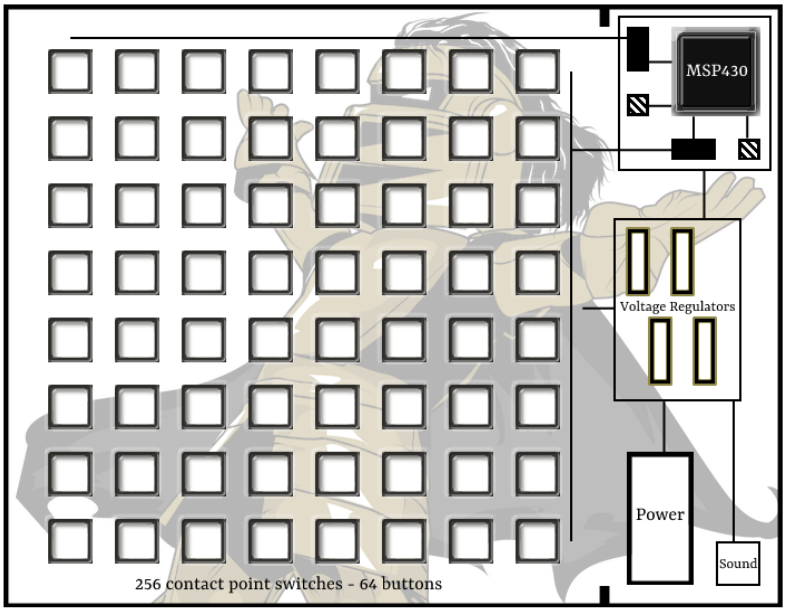


Figure 9 (Main PCB Concept): *Rough diagram of the main PCB. Unlabelled components include clock ICs (striped) and shift registers (solid black).*

The difference in our final design was that we did not use clock ICs to clock the shift registers nor an input shift register since we had 8 pins to spare on the MSP430 to read the rows. As a result of updating the design from the original, the goal of our main PCB ended up dealing with decoding the physical buttons and relaying that information offboard to the Jetson Nano. It also is what manages and distributes the power from our battery. The PCB design itself was built in Autodesk's Eagle.

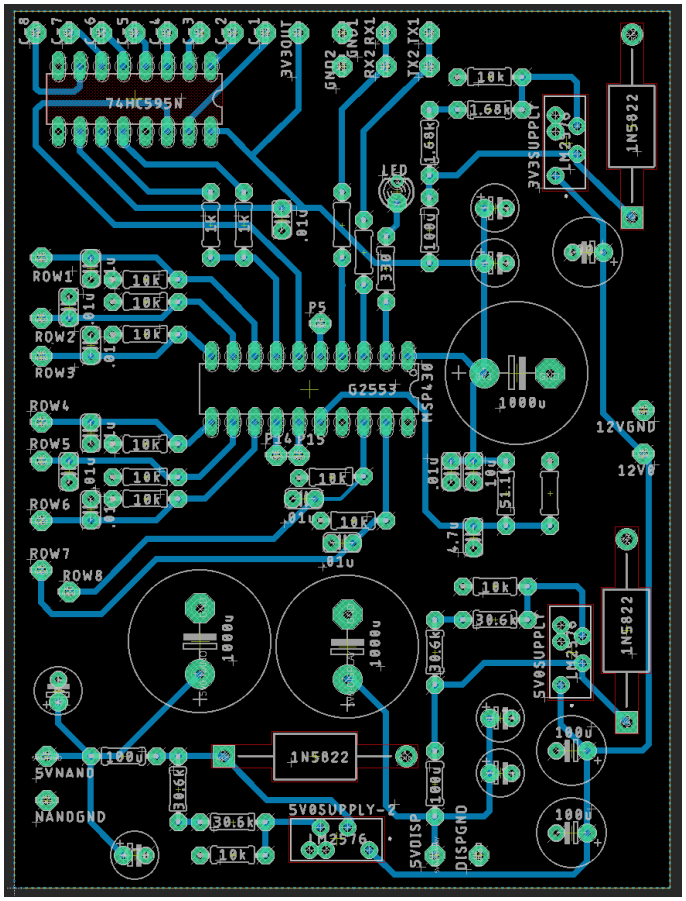


Figure 10 (PCB - Design): *The eagle brd view of our PCB design.*

The design of the PCB is not the cleanest, but we managed to fit it into the 4"x3" area that Eagle provides on its free version. It was hard to fit some of the bypass capacitors close to the component as desired, but they filtered noise just fine anyways. The 1000µF capacitors definitely took a huge amount of space as you can see from the three large circles. We managed to successfully fit everything onto one layer so that we could try etching the PCB ourselves.

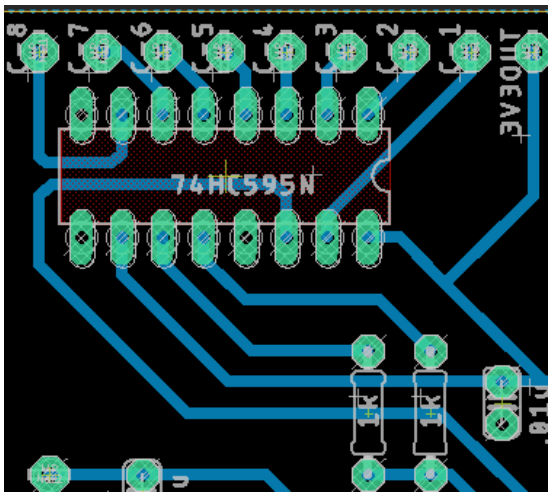


Figure 11 (PCB - Shift Register): Shift register and column lines as seen on the eagle BRD.

The top left section of the PCB houses the shift register and the 8 column output lines. The two 1kΩ resistors are there to smooth out the clock and serial signals coming from the MSP430. They also can function to limit the current into the 74HC595N, but that functionality was not needed.

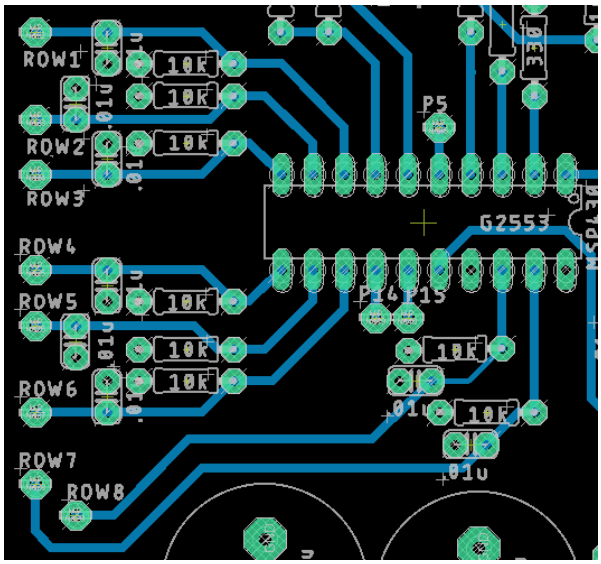


Figure 12 (PCB - MSP430): The MSP430 on our main PCB.

The centerpiece of our PCB is the MSP430. Upwards it feeds a serial and clock line to the shift register as mentioned, and shown in this picture there are 8 rows of input that come into the MCU for decoding. The 10kΩ resistors are pulldown resistors to ensure the voltage we are receiving is read as a logical high. Though 10kΩ was good enough, it might have actually been better to give ourselves more wiggle room with 100kΩ resistors. The capacitors on these lines are not populated and were included in case we wanted to implement hardware debounce.

We mentioned that there was a potential error with switch bounce, and we thought we would need to accommodate this in our design. The initial PCB design also included room to put in the circuitry to handle hardware debouncing if needed. They did not end up being needed.

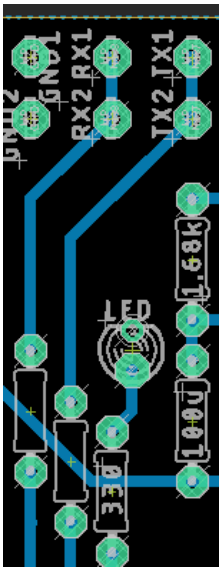


Figure 13 (PCB - TX RX and LED): *Lines from the central MSP430.*

Some more lines from the MSP430 are TX RX for the UART transmission. Additionally we included an LED to let us visually see when the MSP430 was reading button presses. We changed from the 330Ω to a larger value because we found it a bit too bright. The other resistors were placeholders to do the same thing as the signals to the shift register. We ended up using the MAX232 breakout board though and did not need to use them. We could have used 0Ω resistors, but we just used 1Ω instead because that's what we had.

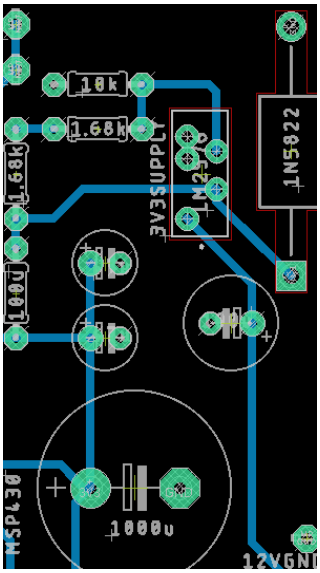


Figure 14 (PCB - Voltage Regulator): *One of the three voltage regulators.*

This is one of the sections for the voltage regulator circuit. Our feedback nod should have actually come after the 100uH inductor (on the left) not before. The 1.68k resistors are actually 2 series resistors that total 16.8kΩ, not 1.68kΩ each. Originally we tried out 1.68kΩ and 1kΩ, but

the values were too small for the regulator. So we used 16.8kΩ with 10kΩ. With the 10kΩ resistor they determine the value of our voltage from the voltage regulator similar to a positive feedback op-amp circuit with an extra factor of 1.23.

The amount of voltage output from the LM2576 is decided by tying the output voltage to the feedback with the resistor orientation mentioned. The additional factor of multiplying by 1.23V gives us the following: 1.23 * 1+(R2/R1). With this in mind, the regulators were set to the needed voltages. The Nano was highly sensitive to the input voltage it was receiving, so the resistors were changed to be even more precise than the others, while we also supplied a tad higher voltage to account for the voltage drop on the way to the device. (An example calculation: (16.8kΩ / 10kΩ + 1) * 1.23 = 3.2964. This is depicting our 3.3 V regulator

PCB-2
We considered another PCB for the sound. We ended up having a way to utilize HDMI to the screen instead if we want sound, but if we wanted anything more sophisticated we'd probably expand on either another PCB or with a breakout board.

PCB-3
We did not use a separate PCB to regulate or manage power like planned and kept it all on the same main PCB. The battery came with its own charging circuitry so we did not have to design it ourselves. If we did, we would probably house all of this on another PCB.

Additional PCBs
Having a PCB to house a display controller and display circuitry might be preferable in the future to expand options of screens. Our screen ended up being the most limiting factor in other facets of design. We also would like to implement the switch array using a PCB like we originally planned since it would improve many aspects of the design towards our original goals. We now also know that we can use a CNC router to cut through copper-fiberglass plates to create the holes needed. For the etching, we could do it chemically like we did our other PCB or try etching with the router itself. The router etching we did not mess with and would have to iterate on first to find a suitable method.

Jetson Nano
As a quick note on the Jetson Nano, it was implemented as it comes in its development kit. This let us utilize the full functionality of the kit rather than us having to try to replicate the ports and features on our own PCB. There was no need to consider another PCB for it.

5.4.4 Switches Layout

There were 256 locations planned for contact switches. These were to correspond to the four corners of each button. We wanted each set of these four contacts to register as the same button press. To simplify the design, especially since we lacked vertical space, we opted for only 64 contact points instead. We still utilized 4 rubber actuators per button so they felt right to press, but only one actually contacted (the bottom right). For a PCB implementation if we want to revert to four contact points, we would also use a different screen size. Either way the contacts can be tied into a four-to-one manner in order to logically OR them, resulting in 64 discrete logical values. The design called for strobing 8 columns with the MSP430 through a 74HC595N

shift register and only 2 output lines to it and then receiving 8 input lines for the rows. This was done in order to reduce the number of inputs and outputs to the MSP430. Initially we chose two 8-bit registers as a way to minimize the 64-buttons into eight wires in and eight wires out, corresponding to the eight rows and eight columns of the device. With spare pins on the MSP430, we decided to forgo the 8 in and just use 1 shift register for the output instead. The shift register is serial and thus requires a clock. Rather than clock ICs we sent a clock signal from the MSP430 to time the shift register.

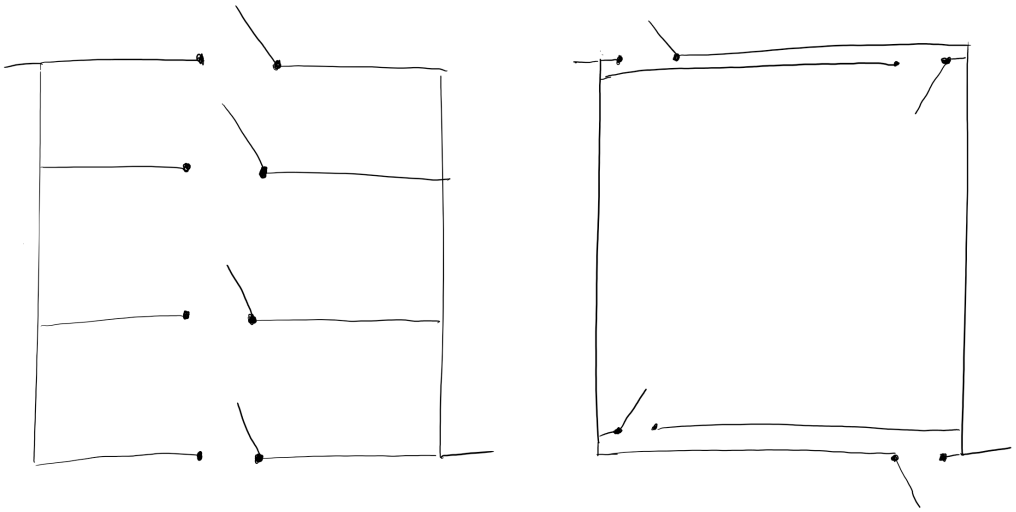
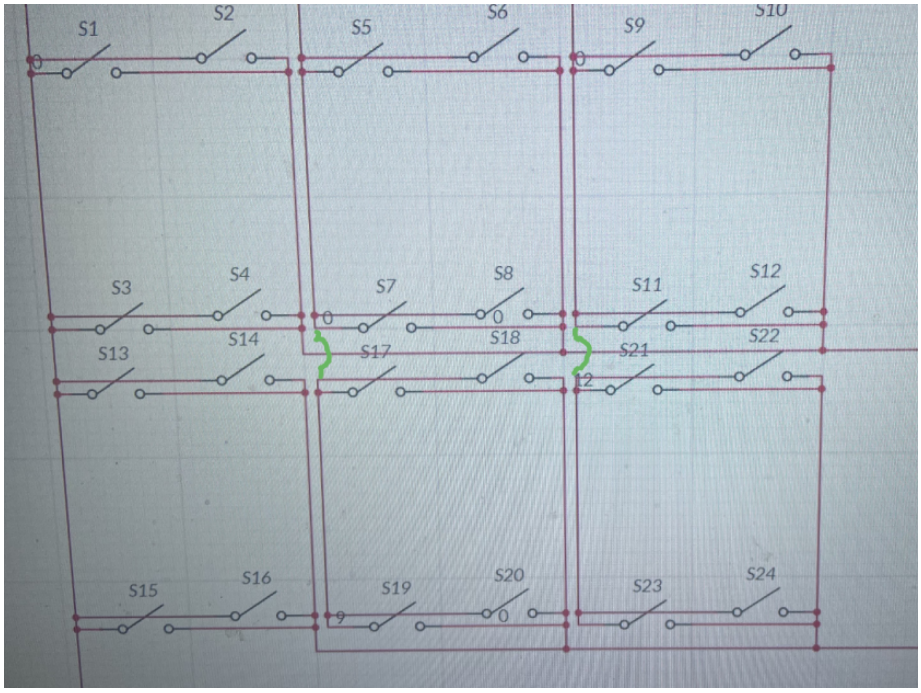


Figure 15 (Hand Drawn Switch Schematic): A basic picture of how each button's switches would be at the circuit diagram level.

In the previous picture a sketch of the original idea of tying the switches together as 4-to-1 is shown. By having four separate switches connecting two nodes we are able to logically OR them as mentioned before without the use of any additional components outside the four switches. The left part of this picture shows an easy to understand implementation of this. Our display has to be visible behind the button panels though. On the right is an alternative visual of the same thing. The alternative drawing represents how we might have implemented this and have a switch in each corner while the middle area is visible for the display behind it. This is the premise for if we were to build a PCB for the switch area. At one point the design even evolved to a schematic showing how the columns and rows would tie together.



<u>Figure 16 (Another iteration of the 4-contact point design):</u> A basic picture of how each button's switches would be at the circuit diagram level.

Pardon the quality of the image, but it was the only remnant of this design path. The green lines represent jumper connections. We designed an early iteration of this as a PCB in Eagle with trace contacts and figured out that the design could fit better if we oriented the rows needing lots of parallel lines vertically (since we had more space horizontally on our screen to hide them). As you can see the horizontal space between two buttons had 5 parallel lines in the concept. Without knowledge of the CNC router being capable of cutting full holes through and not enough time to ship off to a PCB manufacturer at this point, we decided against this idea.

Ideally, with more time then or with the knowledge then that we have now, the PCB solution for the button array would have been more elegant, higher quality, been safer, allowed higher durability, and lowered our device's profile.

5.4.5 Sound

A feature suggested to us was to include a sound subsystem. Sound adds obvious enjoyment quality to games and allows additional games as viable on the device. There was some sound added on the software side, but as a stretch goal we did not go through to implementation and testing. The screen does have built in speakers, but we would have had to figure out how to utilize the hdmi connection from the nano to send it. Adding a dedicated sound system would have added extra complexity and maybe put us further past our design goals. Potential AI applications were on our mind as we contemplated the sound though. A microphone with voice commands, for example, is a consideration related to a sound subsystem that would have tied in with AI. Voice commands could be something we could integrate into the GameFrame with further development by having it select the game the user wishes to play.

Another feature we would have liked to implement if time permitted was to add sound files to pieces while the game is in progress. This is the aforementioned sound that we added in the code but didn't have time to follow through with integrating. We feel like it would have added a nice sense of realism to the game to be able to hear small audio cues of the pieces being moved. We also wanted the game to be more accessible to others so something else we considered as a potential feature was to allow for text-to-speech aiding those who are visually impaired. This would have verbally read aloud the text on the screen and guided the user to be able to select the panel they are trying to reach. The feature would also inform the user where the pieces are located on the board and inform the user of the piece that they currently have selected and where it is allowed to move.

We have concluded that the quality that sound adds to our machine is worth designing for if we ever proceed forward. The plans are to have, at minimum, a simple sound chip or output. The idea is for the software controller, the Jetson Nano, to be where the sound files originate from ideally. HDMI from the Jetson Nano seems the most pragmatic design choice for the sound so as to not add any extra costs or require any additional hardware chips.

5.4.6 Clock and Shift Register

Rather than using a clock IC, output lines from the MSP430 were used to strobe the shift register. This resulted in two output lines and eight input lines - one serial out, one clock out, and eight GPIO pins on the MSP430 used as inputs. The frequency is set at a maximum of 25-29 MHz according to the rating on the 74HC595 8-bit shift register datasheet. The range of frequency is determined by the supply voltage to the register IC. This was plenty as even if we maxed out on the MSP430's high frequency crystal, we'd only peak out at 25 MHz. We definitely didn't send the clock signal out that often though. The clock was utilized by the aforementioned register, but we tied the two required clock lines that the one shift register needed together. By tying them together the SRCLK and RCLK were one cycle delayed from each other. The supply voltage is around 3.3V to keep it consistent with the MSP430, since they both used the same line from the 3.3V regulator.

5.4.7 Power

Part	*Voltage Ratings*	*Current Ratings*
MSP430	3.3V	1-3 mA*
Jetson Nano	5V	1A / 2A
LCD Display	3.3V	110mA
LCD Backlight	5V / 12V	650mA@5V / 240mA@12V
Shift Registers	2-6V	1mA

*The current rating of the MSP430 is dependent on the frequency it is run at. Even with varying current it still only operates at 1-3mA for the supply voltage, quite small.

Table 5 (Power Relevant Ratings): *This tabulates the various estimated voltage and current ratings for our original design's hardware components.*

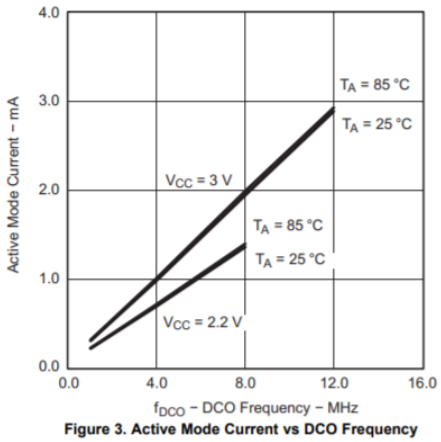


Figure 3. Active Mode Current vs DCO Frequency

Figure 17 (Supply Current Range MSP430): *The datasheet provides an excellent graph to show the current draw based on operating frequency and supply voltage. It also tops it off with convenient information on temperature ranges too.*

Listed for the sake of denoting option variants, the table shows two voltage ratings and their associated current ratings for the LCD Backlight. The 5V is the minimum voltage rating for the device and would not have been selected. It would have cost extra amperage draw and a dimmer backlight. The 5V rating would have allowed us to run every component from the switching regulator. We would have eliminated the need for a battery that goes up to 12V. However, 5V batteries would not have had the amperage supply we would need. The power consumption is a tad more for the 5V rating though.

Voltage Rating	*Total Current Draw*
12V	0.24A
5V	2A*
3.3V	~0.115A
Total Current	2.355A

*Our original total assumed high power operation mode on the Jetson Nano.

Table 6 (Supply Current Range MSP430): *A total of the amperage drawn at each voltage level. This is for ease of calculation and reference for our original switching regulator.*

It is worth mentioning that the 12V comes from the 12V battery directly and not the regulator. So even though our 2.355A total is below the 2.6 maximum that the regulator can output, there is actually an additional 240mA that can be utilized from the switching regulator.

The new ratings contrast to the original since we utilized a different LCD. They are as follows:

Device:	Voltage	Current	Power
MSP430	3.3 V	.350 mA	1.155 mW
Jetson Nano	5 V	1-2 A	10 W
LCD	5 V	1-3 A	15 W
Battery	12 V	3 A	36 W

Table 7 (Voltage-Current-Power ratings): This table contains the ratings for our major components.

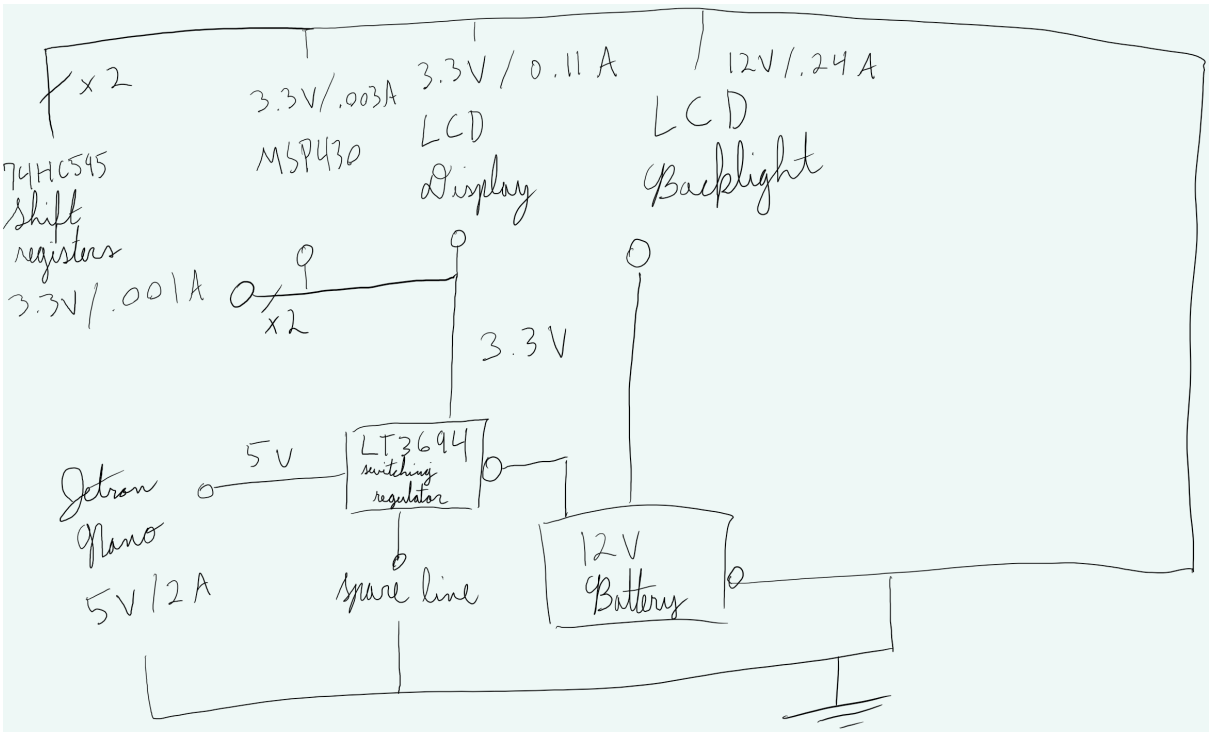
The ratings for power are assumed to be operating at maximum. The ICs were deemed negligible for the purpose of the table (which was to help us decide on regulators). The MSP430 also has next to no power draw, but it was a primary component and at least represents that it and its related ICs need a 3.3 V regulator.

Battery

The battery was intended to have 12V so as to be able to supply the required voltage to the first LCD backlight without needing any boost converters. Instead, we still went with a 12V battery, but that's just so we could have a good amperage draw maximum (compared to a 5V battery) and so that we could choose from a better watt hour. The battery is a 66.6 Wh battery that has its own protection circuit built in, features an on-off switch and even supplies power to a USB port. So one could actually charge their phone off of our device at the cost of its battery life. It came with a generic AC plug power supply and we soldered a connector together to allow the wall supply to charge the battery and power our device at the same time, or for the battery to power the device while the wall plug was not plugged in.

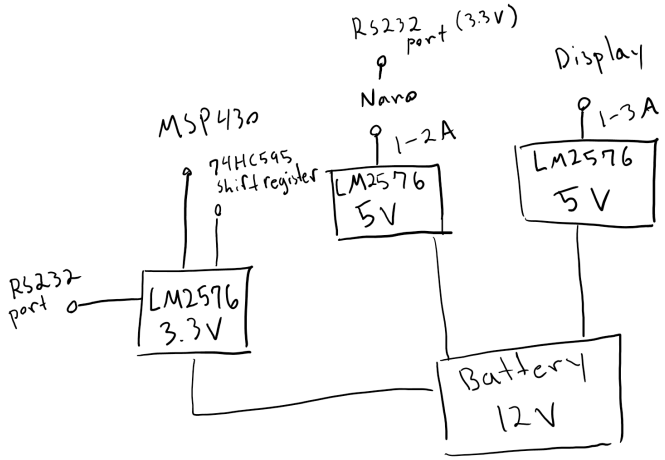
Switching Regulator

The LT3694 allowed us to step down our 12V battery to different voltages for multiple output lines. The maximum current output it can supply to various parts is 2.6A. If we ran the Jetson Nano in the 10W mode this left .6A for the other parts, which was likely feasible. This would have left little room for additional parts or expansion of hardware design though. The 5W power mode for the Jetson Nano was also considered for this regulator and would have left us with more of a cushion of 1.6A from this regulator alone to power the rest of our components.



<u>Figure 18 (High Level Power Schematic Design):</u> An abstracted block-diagram level picture showing voltage and current to our device's components from the LT3694 regulator. Location and layout is not to be considered physically accurate.

After changing our screen and having trouble with obtaining a DIP-socket package of the LT3694 we swapped regulators. A surface mount device version of this was way too small for us to try solder when we got it in. As a result we changed to a regulator we were familiar with from Electronics 2 because we had access to them - the LM2576. We decided on three of them to isolate noise from each other, to ensure there was enough current to the devices, and to give a bit more cushion for expansion (at least compared to trying to fit it all onto one regulator).



<u>Figure 19 (High Level Power Schematic Design):</u> An updated block-diagram level picture showing voltage and current to our device's components from the new LM2576 regulators.

Location and layout is not to be considered physically accurate. Current from the 3.3V is not shown since it is much below the listed 3A current draw maximum of the regulators.

5.5 Decision making

5.5.1 Project

Project Name	*Project Complexity*	*Familiarity with Technology*	*AI Requirements*	*Personal Interest*	*Mechanically Challenging*
Smart Blind	Neutral	Neutral	Low	Low	Low
Poker Hand	High	Low	High	High	High
Refillable Station	High	Neutral	Neutral	Neutral	Moderate
GameFrame	Moderate	High (consultant)	Moderate	High	Low

Table 8 (Project Decision): Project ideas taken into consideration.

The interest for the poker playing robot was the highest, and it was the project with the most prominent AI development and computer vision. The portable game device ended up having good interest (though admittedly slightly less). The benefits of the game device ended up being that it was complex enough without being overly complex, we had an outside source to consult with, it was not very mechanically challenging, and it had some component of AI with which we could implement. This allowed those who wanted to develop those skills to do so while also maintaining a solid level of project interest. Thus, it is what we decided.

5.5.2 Microcontroller

We honed in on the Jetson Nano as our microcontroller of choice for the software. We believed it to be a great learning opportunity that would provide valuable experience to those interested in parallel computing and other technology useful to the AI field. We considered how it has extra cost and power requirements, but ultimately decided that it still was within our acceptable range and might actually help instead. Since we did not exactly know how much processing power we needed when making decisions ahead of time, we figured it was better to be safe and go with a board with more capabilities out of the box. We wanted to avoid any bottleneck issues during development that could be caused by using a cheaper and less powerful microcontroller. The statement that most summed up our feelings was “it sounds like the pros outweigh the cons.” As we delved into more research on it, we felt that it was suited to our interests and more than sufficient for our goals. The only other thing to mention about a decision regarding computation devices or controllers is that we used another controller to manage our hardware. This controller was the MSP430. The division of tasks was accomplished easier with separate devices. Putting

less strain on the Nano also helped to alleviate some of its heat generation and power consumption.

6. Prototyping

6.1 Prototype Overview

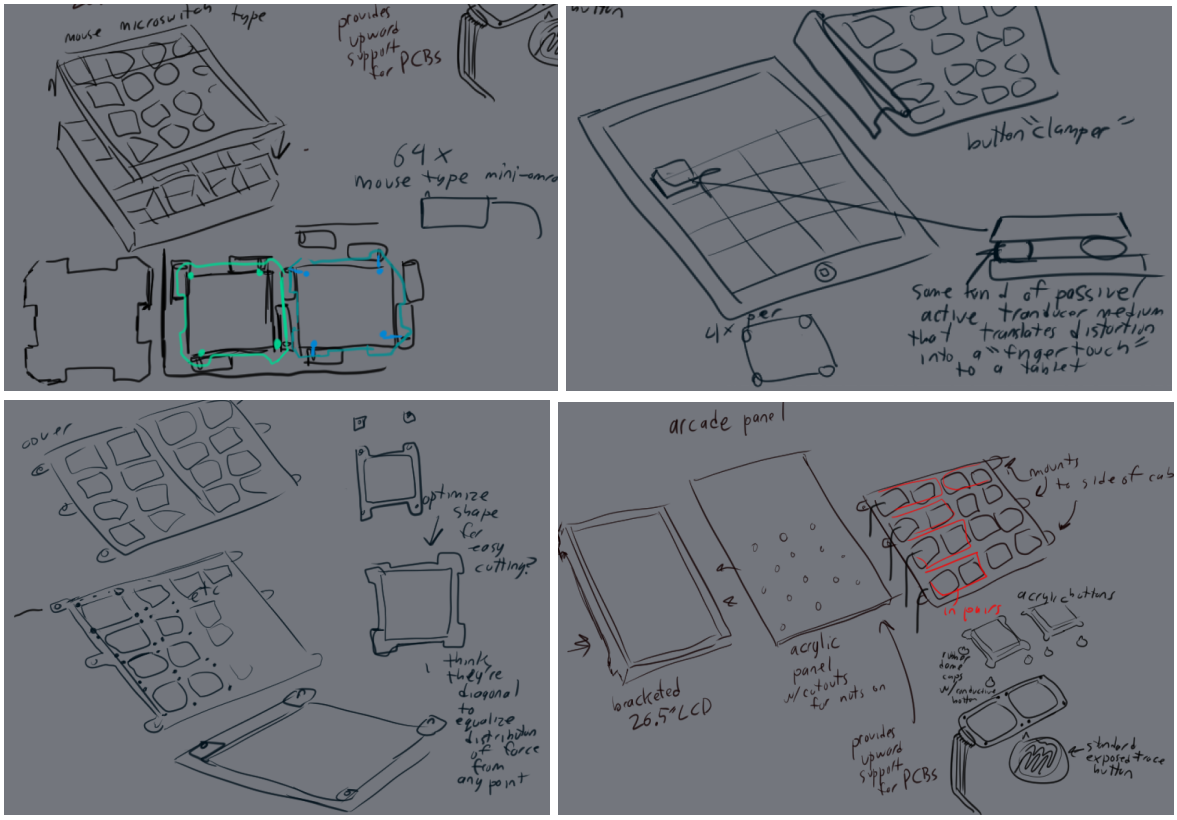


Figure 20 (Prototype Sketches): A collection of rough diagrams we were given

The original visual prototype for the project was based around a video of a DIY jubeat assembly which can be found in the appendix and also based on a rough basic design diagram of the physical components provided to us by our contact (above: Prototype Sketches).

In order to add more clarity to parts of the diagram provided, a description of the components might help. Of particular note is the PCB and the acrylic panel. The bottom right shows an example of jubeat's PCB. The machine uses eight different two-button PCBs. Due to the large holes and thin PCB, a flimsiness exists. The acrylic back panel helps to keep the LCD screen clean of debris of course, but it also functions to add support to the PCB. One other thing that is not clear just from the diagram is that the buttons have a specific diagonal bevel in the corners. This causes a better slide and feel when the button comes back up while also helping maintain alignment of the actuators with minimal material. Throughout the project, we did not have a cost-efficient way to replicate proprietary acrylic buttons, but we did create a custom design focused on making the most of the screen that we had through a CNC router.

Our goals for prototyping focused on cutting costs and creating functional controller systems. With these focus points, it was not possible to reliably tell if we were meeting our weight and power goals. Functionality was our prime goal and so we ended up not paying attention to weight as we developed what we needed to have it work. The power incidentally ended up being tested since we ran off of battery life many times while developing for the system when the hardware was completed.

CAD prototyping was a goal as we developed more specificity, but some dimensions were not realizable until the end of production. A sort of CNC usable CAD called Easel was used for our CNC for the iteration on the buttons. Onshape was also used for early housing development.

In the early stages of prototyping the hardware controller we utilized an experimenter board to easily flash and test code prior to risking pin damage by moving it over and over.

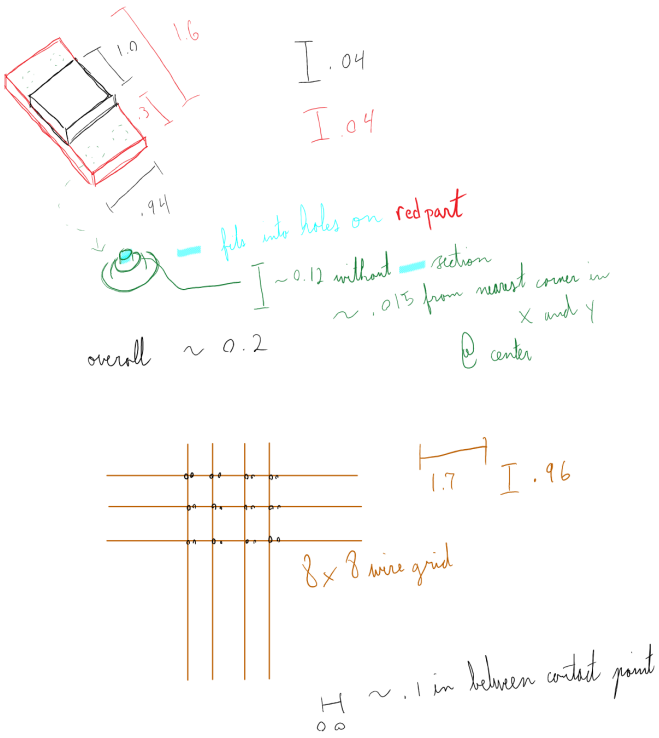


Figure 21 (Prototype Sketches Pt. 2): A later sketch of our grid design after it was built showcasing approximate dimensions.

6.2 Software

In order to test the software, a simple graphical interface was created. This was never realistically something that is interacted with by the end user, and was eventually thrown out completely once the full build was completed. This software prototype app serves as a tool to test the functionality, rule implementation, and capabilities of the computer to play against the user in

a game of player versus computer. Additionally, this application may be used to monitor training of the artificial intelligence or backtracking algorithm.

6.2.1 Sample Menu Prototype

We planned on creating an interface that the user can successfully use to navigate between their desired choice of game to play and settings for the game. Once the device is turned on the software begins and will load the main menu software that we designed. Our initial design of the main menu looks like the figure below and our current operating main menu is displayed on Figure 15. Our final design of the main menu resembles our prototype version with some features that we didn't fully implement. For example, the settings button was created however it was commented out in the code since we didn't finish the additional features. We wanted to have the main menu have other additional features, however, due to time constraints and bug fixes, we decided to focus on functionality rather than cosmetics. The current menu we have now satisfies this goal of Looking forward, these functionalities can be implemented with some assistance from the hardware side. A challenge that was overcome in the GUI creation was creating the first menu and game. Once the first menu and game were created then reproducing this for the other menus and games became much easier.

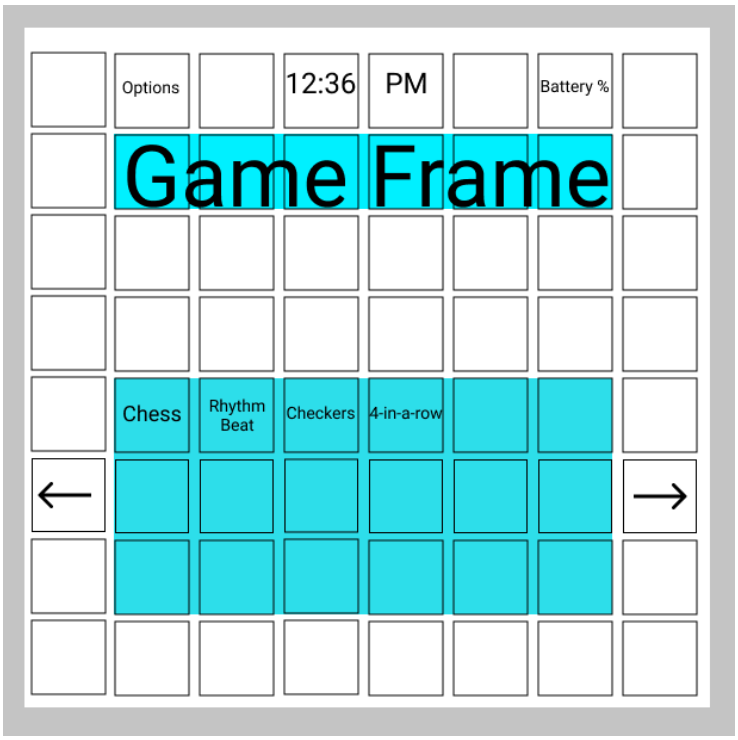


Figure 22 (Prototype Main Menu): Pictured is how we planned on creating the main menu.

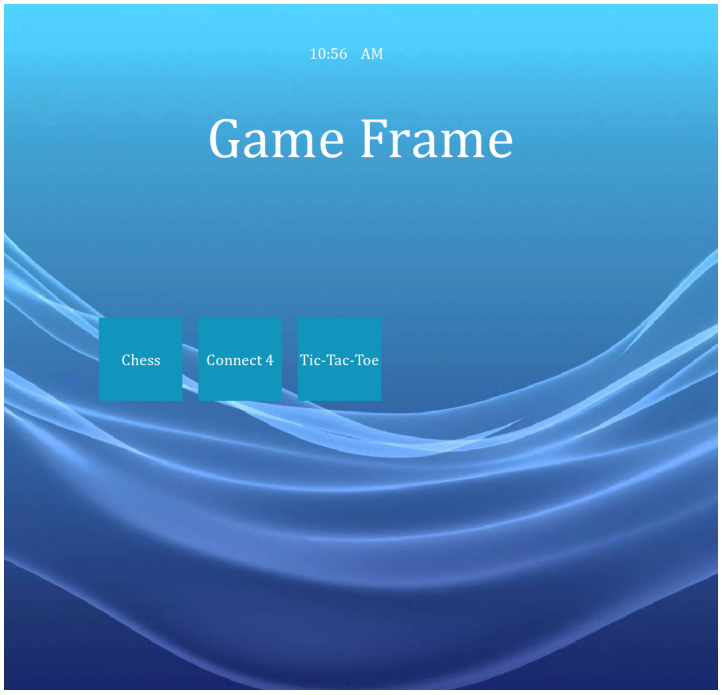


Figure 23 (Main Menu): *Pictured is how we created the main menu.*

The title of the product is displayed at the top and then the library of games is shown on the lower half of the screen. Currently, we have chess, connect four and tic-tac-toe. In the prototype we wanted to have a frame to cover all the game options however the frame of the hardware already blocks this view so we simply made the button be individual rectangles through the PyGame library. Once a game is selected, like chess, then the user will be brought to that game's respective menu. We planned to make this like in Figure 24 and for the most part we replicated this same design in the final product, located in Figure 25.

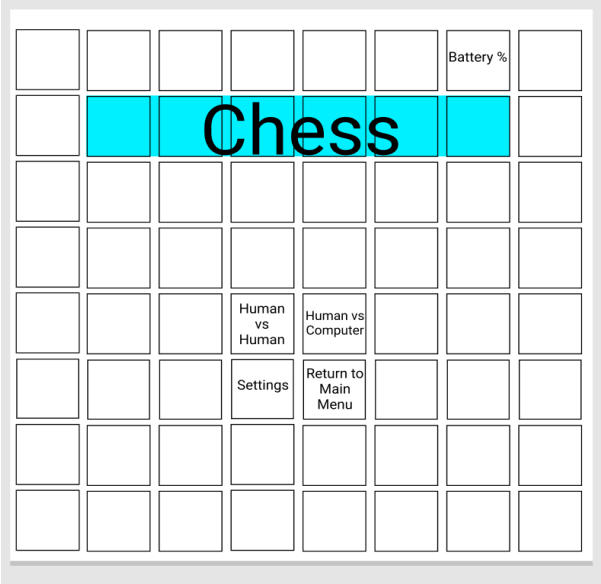


Figure 24 (Prototype Game Menu): *Pictured is the prototype of the game's menu.*

Figure 25 (Game Menu): Pictured is the game's menu.

Using Chess as an example, the menu shows the current game at the top and then the user must select settings such as: *game mode, difficulty, and a team*. Each game menu follows this outline for each game in the library. We wanted to keep the same text as the prototype for better clarification but for simplicity's sake we shortened much of the text. Instead of saying: *Human vs Human* we changed it to *1v1*. If the user selects to play against a computer, the software will make use of the unused buttons below and ask it for a level of difficulty for the computer such as: *Easy, Normal, or Hard*. It will also ask whether the user would like to play on the white side or the black side. When the user selects a difficulty it will remain highlighted to show that is what the computer's difficulty will be. Once the user has selected a difficulty and team, or if they selected *Human vs. Human*, then the software will display the *Confirm* button so that the user can begin to play with their choice of settings.

Another feature we could add in the future is a tutorial panel where the user is informed how to play whichever game they are on. This would also be a good place to inform the user if we add in other features that are not visibly shown such as an undo feature or pausing the game. We did create a feature so it will lead the user to a tutorial page for each game however we ran into a challenge with the button size so we discarded this feature and left it commented out for implementation in the future if we had extra time.

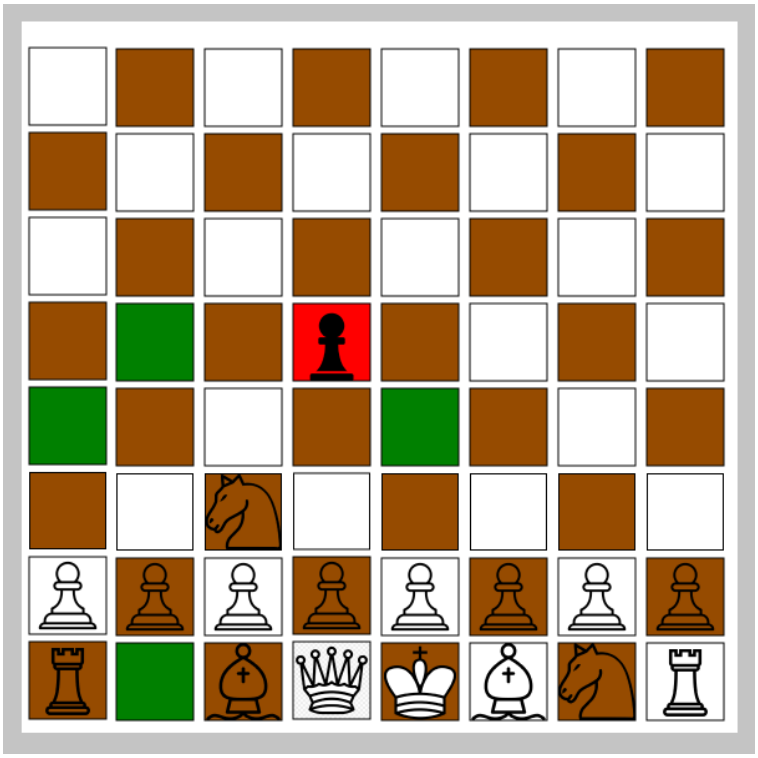


Figure 26 (Chess Gameplay Prototype): Pictured is a prototype of how chess will be played.

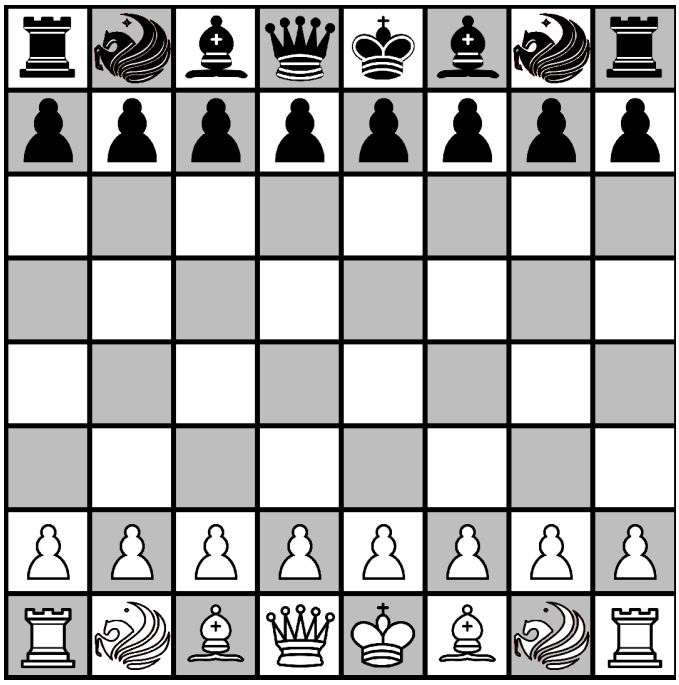


Figure 27 (Chess Gameplay): Pictured is how chess is played.

The highlighting feature that is shown in Figure 26 is something that was created but wasn't tested properly to ensure it was working properly so that feature was also commented out of the code. The way the feature would work is when a piece is selected it would highlight it's path

with green dots. If the piece is blocked by another piece and is not allowed to move to that square then it will not display anything. If the piece is able to remove an enemy piece then it will be red instead of green.

Figure 28 (Prototype End Game Screen): Pictured is a prototype of how the victory screen is displayed.

Figure 29 (End Game Screen): Pictured is how the victory screen is displayed.

Once the game is over, whether it be a victory, loss, or a draw, a small popup appears asking the player if they would like a rematch or be taken back to the game's menu. When this happens, the user has the option to play the game again with the same settings or they can go back to the game's menu where they can change the settings of the game. The final version of the victory screen came out almost identical to the prototype version with some exceptions. The letters and buttons had to be moved more to have it more centered. These could be accomplished in the future and we could also add in a text box that says which team was victorious.

6.2.2 Additional Games

We tried to implement as many games as we could and we succeeded with having chess, connect four, and tic-tac-toe. Unfortunately, we had the design as a constraint to the choices of games since the device is a square with sixty-four panels. However this was also beneficial to us since it works great for certain games like chess and checkers since their boards already have 64 blocks. Working on this with more time, more games could be implemented such as checkers. Checkers would be a game that could be easily created since it is somewhat similar to chess with fewer pieces than chess.

Tic-Tac-Toe

Tic-Tac-Toe is a game where two players alternatively take turns filling out X's or O's in an attempt to get a certain amount consecutively and win. This can be achieved by having symbols appear in a column, row, or diagonally. We gave the players a chance to make the game a bit more interesting and customizable to their likings. For example, the player has the option of choosing the board size whether it be a three by three board, five by five or seven by seven option.

Other features we wished to implement took advantage of the spaces that weren't being used. Any panel that was not being used for the game was going to be fully colored black to indicate that button was not in the playing field. However not all unused spaces will be completely useless as shown in the prototype figure below. We planned on having certain panels show the current score between players and have buttons that may be beneficial such as an undo feature in case a player accidentally makes a choice.

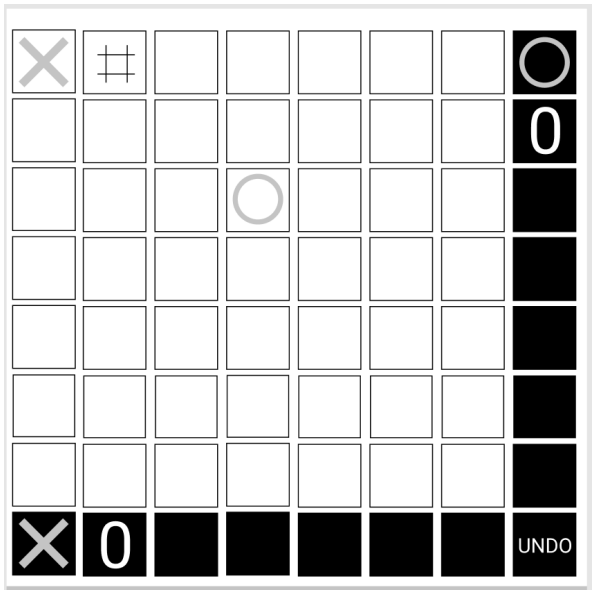


Figure 30 (Prototype Tic-Tac-Toe): A prototype of Tic-Tac-Toe being played out.

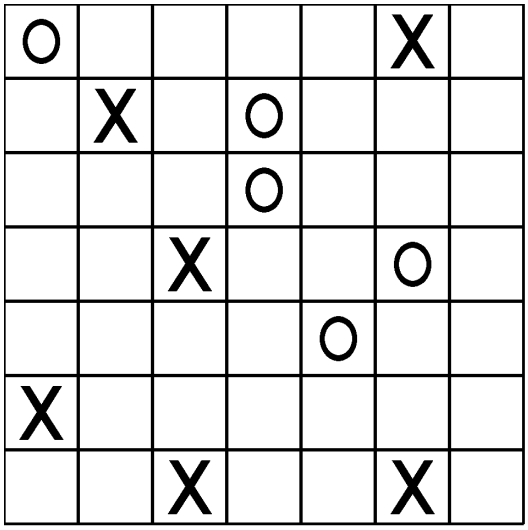


Figure 31(Tic-Tac-Toe): A game of Tic-Tac-Toe being played

In the final design we simply created a black outline to show the user where they can play and did not take advantage of the unused tiles to the right and bottom.

4-in-a-row

This game consists of two players taking turns in placing a ball in the eight columns present. They will take turns trying to get four of their colored balls to be in a row, column, or diagonally. When a user attempts to place a ball with empty spaces below it, the ball will fall down until it lands on top of another ball or the lowest row possible. The software checks after each player has completed their turn in order to see if four of the same color ball is in a row, column, or diagonally. Once a new game has begun, the board is empty and the player who is using the red

ball will go first. The option for teams is presented in the game's menu. If the user plays against the AI, they can choose to be on the blue team and let the AI go first. The final design looks better than the prototype since we added a board tile to each button to make it appear more authentic and the balls fit perfectly inside circles. A feature that we wanted to add in for more authenticity is to have the ball start from the top of the column and drop down to it's requested position. However due to time constraints we took the more simple approach and have the piece simply appear in it's spot.

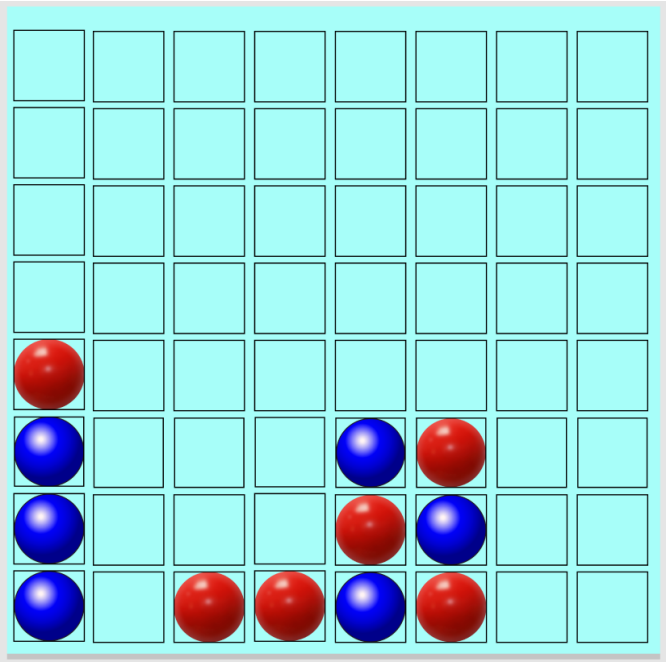


Figure 32 (4-in-a-row): This is a prototype of how 4-in-a-row will be displayed during a game.

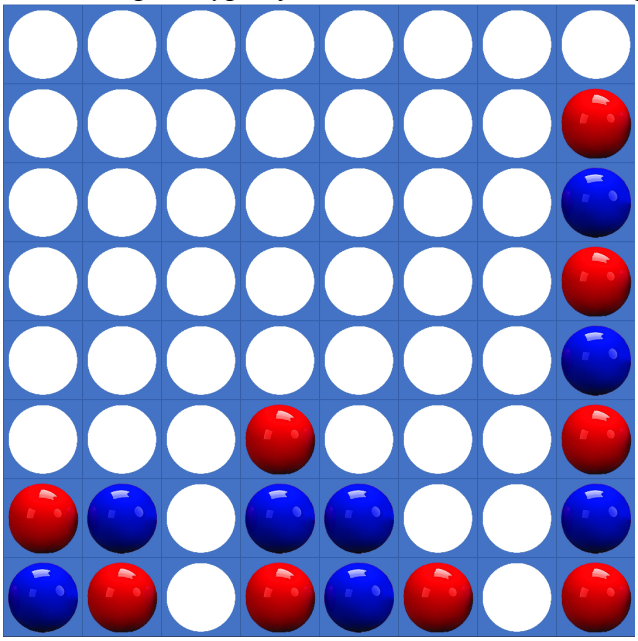


Figure 33 (4-in-a-row): Pictured is a game of 4-in-a-row.

6.3 Hardware

6.3.1 PCB

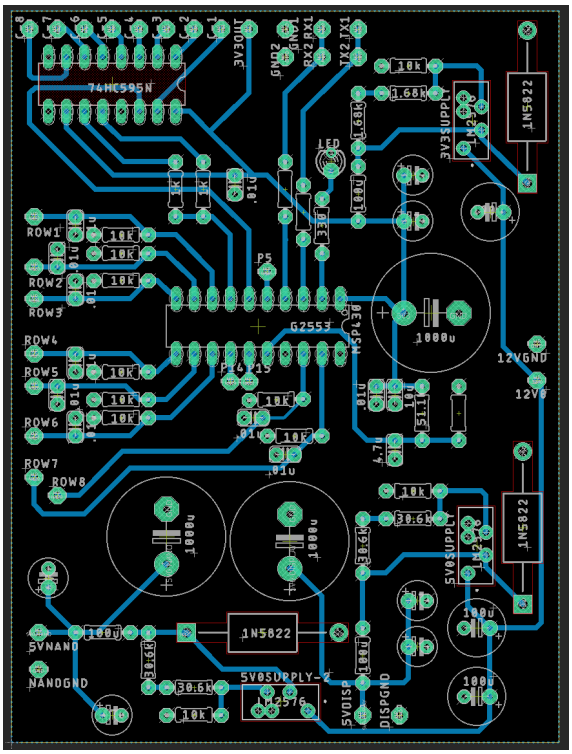


Figure 34 (Eagle PCB BRD): Layout of our PCB in Eagle.

This is the PCB design from eagle. Since we covered this in the design section, we'll look at the eagle schematic view here.

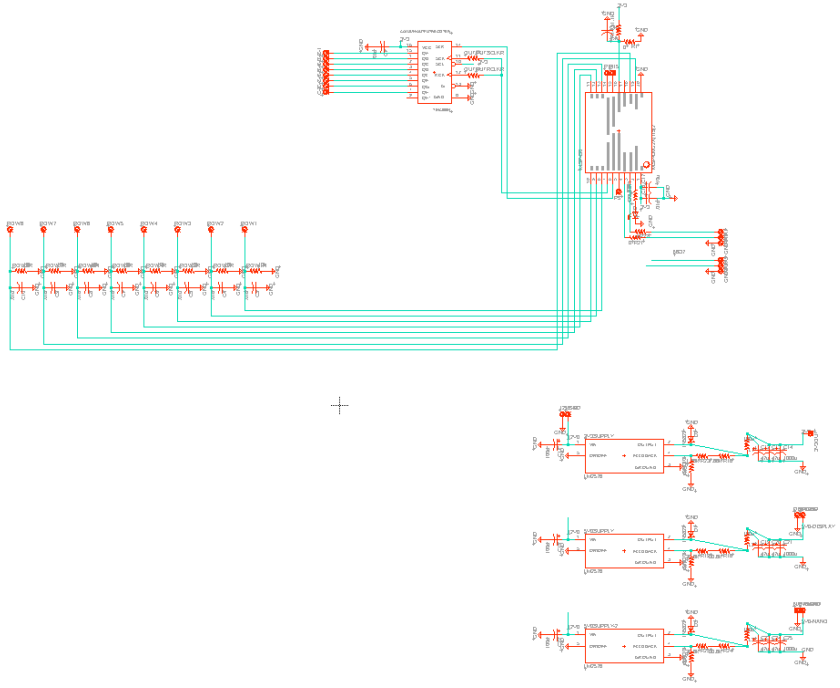


Figure 35 (Eagle Schematic): An overview of all of the schematics for our PCB

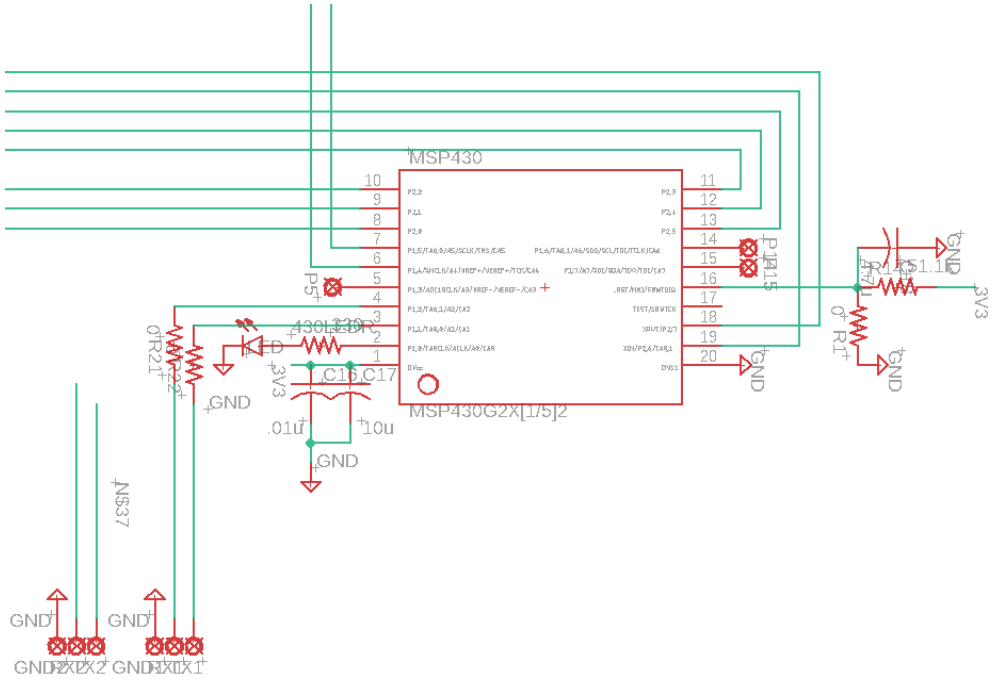


Figure 36 (Eagle Schematic - MSP430): The SCH view of the MSP430 and its surrounding components.

The MSP430 has pins 6 and 7 to control the shift register and pins 8, 9, 10, 11, 12, 13, 18, 19 as input bits from the rows. Additionally pin 5, 14 and 15 were not used, but we left solder pads in case we saw a use for them. We left 17 as DNP however. The reset line on pin 16 has a standard set up for a reset line. The capacitor and resistor values are chosen to ensure a fast startup time when the device is turned on. Additional filtering capacitors have to be on this line if the test pin 17 is used, but we did not. Some bypass filters can be seen on the Vcc line. Pin 2 shows our LED with a resistor to draw current and pin 3 and 4 show a simple resistor to smooth the signal to the RS232 port. Additionally the RS232 ports were duplicated as an additional measure in case we wanted to debug the MSP430 transmission at the same time as sending to the Nano.

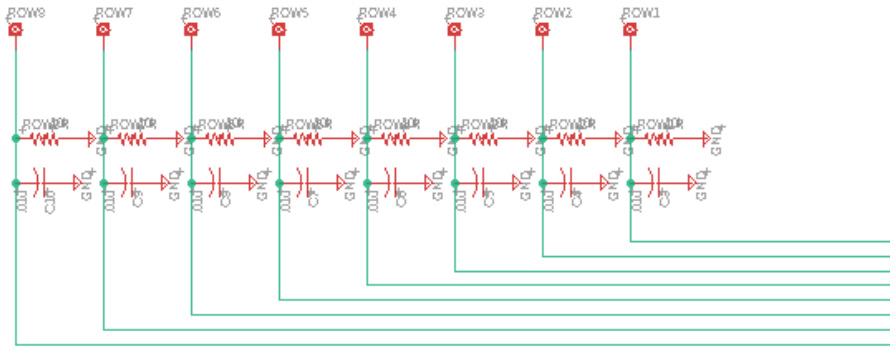


Figure 37 (Eagle Schematic - Rows): The SCH view of the input rows and their surrounding components.

The input rows have pull down resistors to make sure the digital high is read correctly and some DNP capacitors in case we needed debounce filtering.

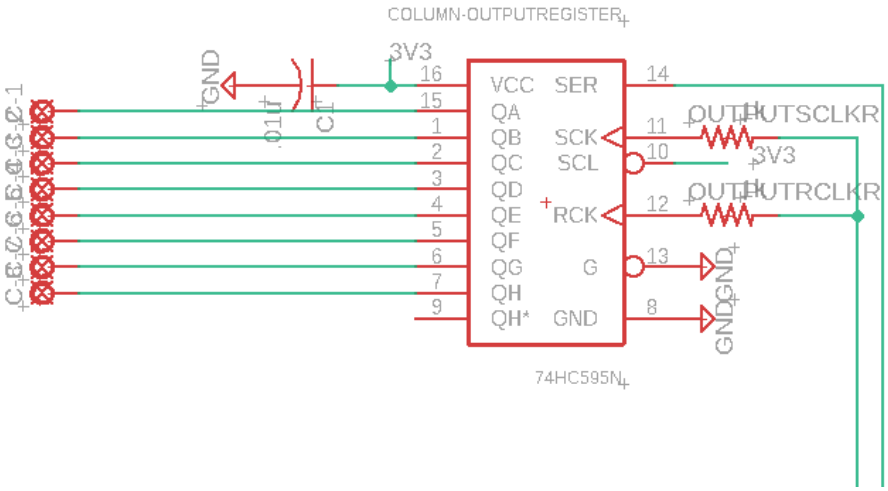


Figure 38 (Eagle Schematic - Shift Register): The SCH view of the 74HC595N and its surrounding components.

There's a bypass capacitor on the input voltage and a resistor to smooth each of the clock signals. Pin 1-7 and pin 15, are the outputs to the columns.

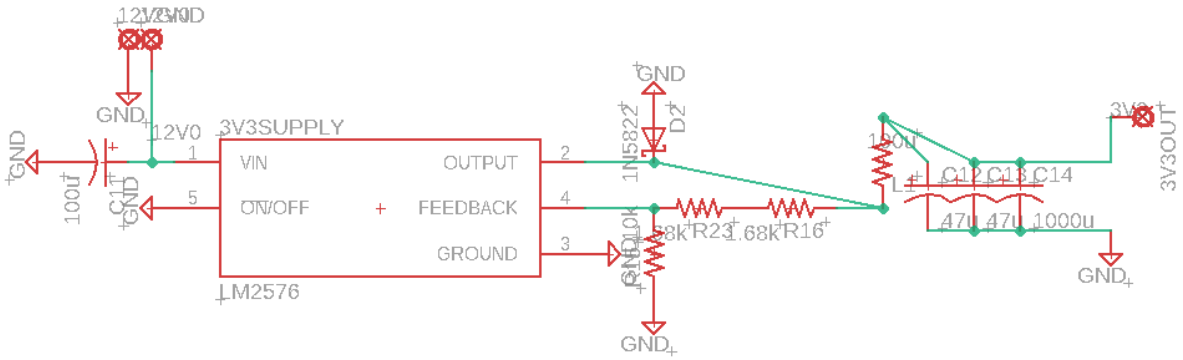


Figure 39 (Eagle Schematic - Voltage Regulator): The SCH view of the LM2576 regulator. Shown is our 3.3 V regulator.

The biggest note on this is the error in tying the feedback resistors from output of the regulator to the feedback of the regulator. The correct node to tie to feedback is the overall output voltage at the top of the capacitors. The way we fixed this was as simple as putting a resistor (denoted R16 in this picture) on the bottom of the PCB when we soldered. We successfully jumped the series resistors to the correct output voltage node. There is a bypass capacitor on the 12V in, a flyback diode for the inductor, and the inductor to ensure steady output current. There are also 3 output capacitors to ensure a steady output voltage, to filter out varying resonance frequencies, and to minimize impedance so as to continue treating the op-amps in the regulator as ideal. Additionally, the resistors on the feedback line control the voltage of the regulator. These resistors are what change between the 3.3V and the 5V regulators.

6.3.2 PCB Etching

Deserving of its own section separate from the PCB is the method in which we made it. The process of etching our own PCB is actually a simple one. The time it takes is much less compared to shipping off to have it made elsewhere as well.

The first step is to print out the PCB as a black an white image. The black part is where we want copper to exist. The reason this is the case is because the toner from the ink will protect the copper plate from the ferric chloride while leaving any exposed copper to be etched away, revealing the green fiberglass layer underneath. The less white there is, the less we have to etch off, and the faster the process is. Since it has to use toner, we had to use an inkjet and not a laserjet.

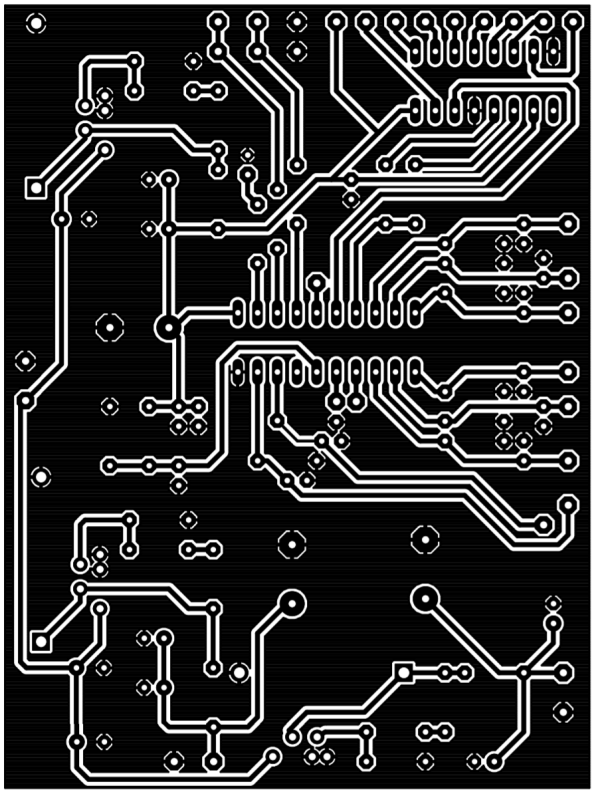


Figure 40 (Negative PCB image): With this printed we can cover the desired copper areas to protect them during etching.

Next, we taped the image to the copper plate with mylar tape (thus why the image needs to be reversed). Afterwards we used an iron to heat press the ink onto the copper plate for about 10-15 minutes. Once the image was pressed we used water to soften and peel away the paper to leave the copper plate and toner.

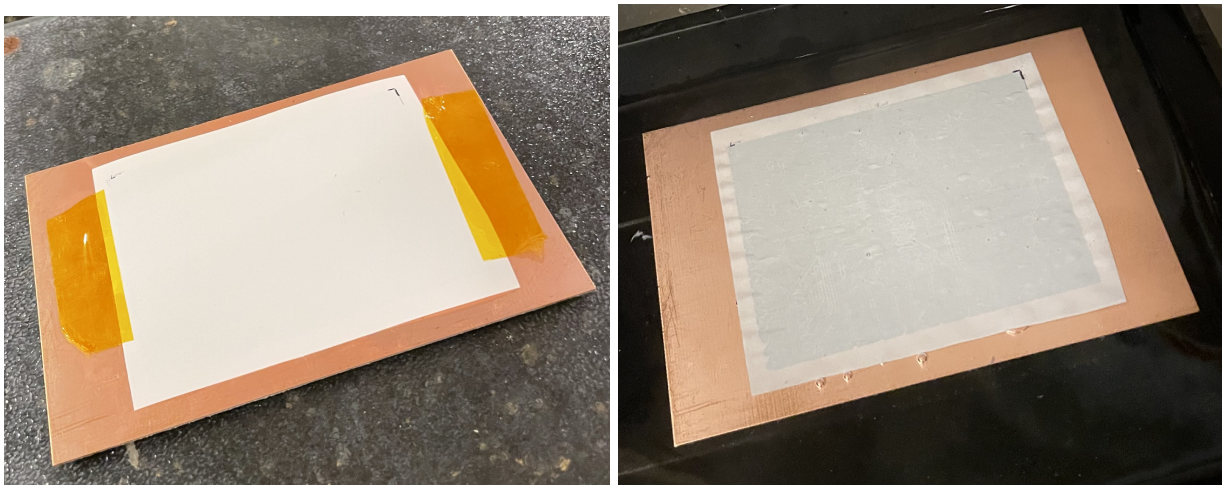


Figure 41 (Toner and Copper Plate Step): On the left shows taping to the plate while the right depicts putting it in water after ironing it on.

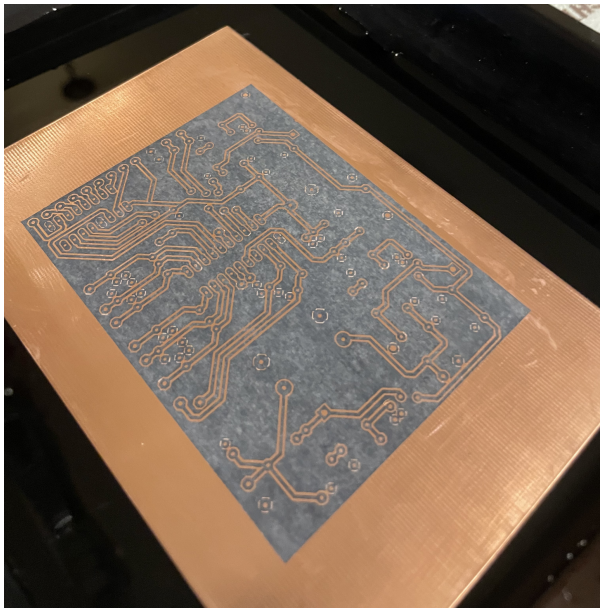


Figure 42 (Toner and Copper Plate Step Pt. 2): After rubbing off most of the paper.

About 10 minutes were spent slowly rubbing the paper away while making sure not too rough. Too much force and not enough heat press time would mean that the toner starts to peel during this step too.

The final step was to replace the water in the bin with ferric chloride. Additionally, we covered the outside copper. Then, in order to gently agitate the exposed copper we rocked it back and forth for about 40 minutes. If we do it too long the toner starts to wash away and will expose the copper underneath, too short and the already exposed copper that we want to etch doesn't get etched.

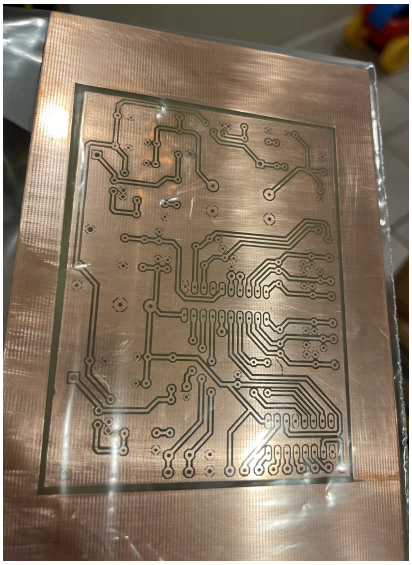


Figure 43 (Finished PCB): Finished with some sand paper and kept in a plastic bag to minimize oxidation while doing other steps.

6.3.3 Wire Grid

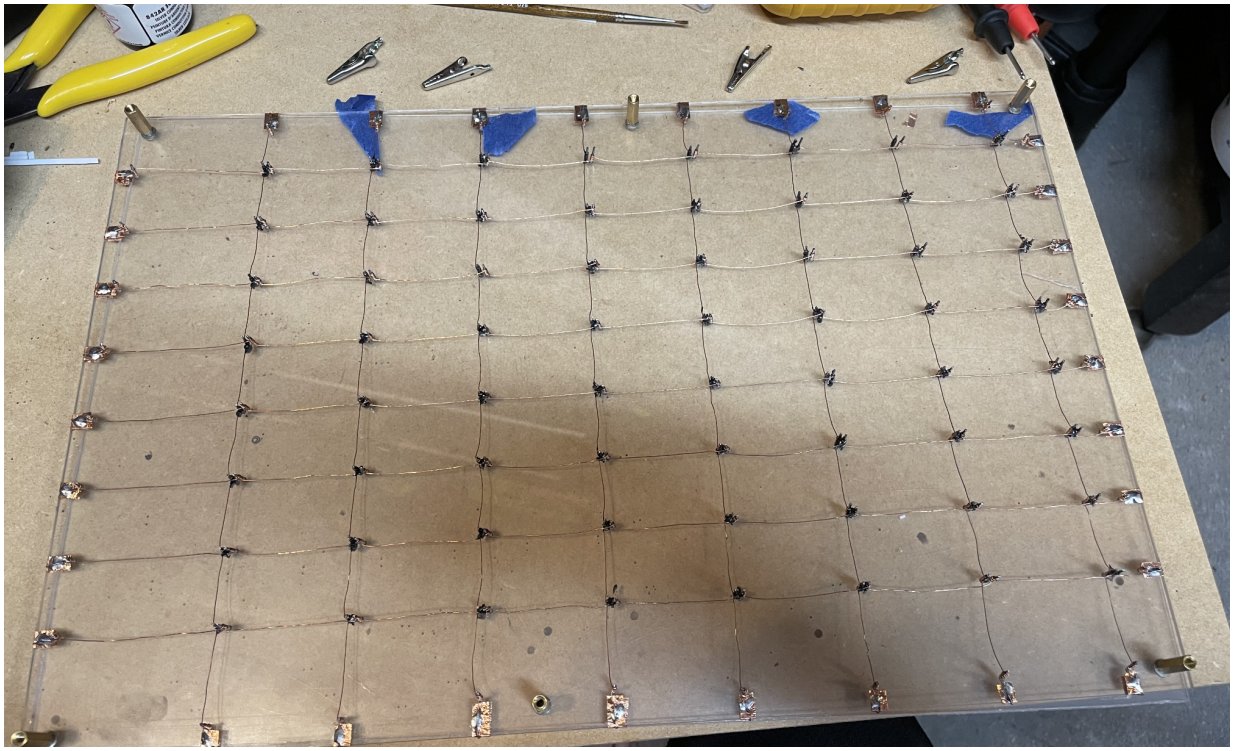


Figure 44 (Wire grid): A picture right after putting the last coat of liquid tape (black) for insulation on some problem shorts to fix them.

There were early plans for the wire grid which are discussed in the design section. Since visibility became a big concern, we weren't sure if a PCB would be viable. Too thin of a PCB would be too brittle. We went with thin wire that is only .013" in diameter to solve this problem. However, it was not any more rigid and ended up being very fragile.

160 nails were placed into a grid shape. 128 were for contact points and the remaining 32 were used as anchoring points on the ends. A set of vector boards were used to align the drill holes. At the junction of the wires, we used liquid tape to insulate the cross wires from each other and to help insulate the contact nails from each other.

This was actually quite a laborious process that was very prone to error. At any point when laying a column or row, snapping a wire (which is quite easy with such a high gauge) meant having to redo that entire row or column. Not wrapping tight enough though would leave unreliable contact with the nails and possibly cause the wires to short with each other from hanging too loose. Overall, probably not how we would implement any future designs if we iterated further on the GameFrame

6.3.4 Power

As our device was intended to be portable it was a DC-circuit that ran off of a rechargeable battery. The original plan called for USB charging, but since the battery we got came with a wall wart, we went with that instead. The entire battery is 12 V with a maximum of 3 A and an estimated Wh of 66.6. We were able to get anywhere from 3-4.5 hours off of the battery

The other aspect of power for our circuit were the regulators. We used three regulators as step down converters to supply power to the three different main components. One 3.3V and two 5V regulators were designed around.

The prototyping just involved soldering the designs to the PCB correctly and plugging the battery in. For further details for the design please see the Hardware Design section of this document (Section 5.4.7 in particular).

Figure 45 (Battery): Our 66.6 Wh battery and its coord/wall adapter. We soldered the split cable and it allows the device to power off of the wall while charging the battery.

6.3.5 Heat Dissipation

Issues of heat were immediately accounted for on the voltage regulators thanks to the datasheet of the LM2576 detailing the thermal characteristics of the regulators. The power dissipation across the 3.3V regulator was extremely low and there were no thermal issues with operating as is. Both of the 5V regulators had a little lower voltage drop across the regulator, but a much higher current draw. Since the LM2576 has a thermal-ambient resistance of about 41℃/W, a power efficiency of about 75%, and a maximum operating temperature of 125 ℃, we needed a heat sink to help with heat dissipation.

Voltage drop across regulator:
(1-0.75) * 12 = 3 V
Power dissipation across regulator (LCD):
3 V * 3 A = 9 W
Power dissipation across regulator (Nano):
3 V * 2 A = 6 W
Temperature increase (LCD):
9 W *41 ℃/W= 369 ℃
Temperature increase (Nano):
6 W * 41℃/W =246 ℃

The heatsinks we used to counteract these large temperature increases dissipate around 16.7 ℃/W. Additionally, in order to ensure a more preferred airflow for the heatsinks a small fan was mounted to pull ambient air away.

6.3.6 Button Prototyping

Originally, we wanted to have buttons that were at least 1 in. by 1 in. However, as we continued designing the hardware, we realized that 1 inch square buttons were not going to be big enough to fit the conductive actuators needed for the hardware to detect button input and provide enough viewing space for users to see the pieces. Since we had almost no vertical space on the screen to expand the buttons, we extended the buttons horizontally on each side by 0.3 in. so that 2 actuators could fit on both sides.

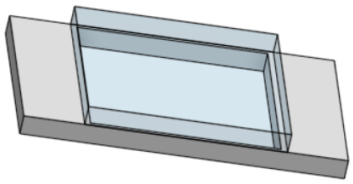


Figure 46: Initial button prototype design with glass tile on top

When deciding how the buttons were going to look, the idea was to 3D print the button rectangular piece button with a hole and simply glue a 1 by 1 inch glass tile on top of it. However, since we had access to a CNC router, we used that for our buttons instead. We used it to cut out two custom pieces for each button from PETG: the 1.6 in. by 0.94 in. base of the button and the 1 in. by 0.94 in. tile that goes on top of the rectangular base. The vertical height needed to be reduced a tiny bit more from 1 in. by 1 in. due to the vertical screen limitations. When assembling, epoxy glue was used to adhere the square tile to the rectangular tile. A clamp locate system was used to ensure proper alignment as the epoxy was applied and during its initial curing phase (about 15 min). Finally, the four conductive actuators were inserted into the four corners of the rectangular base which were custom cut for fitting them as they were milled.

6.3.7 Button Fabrication

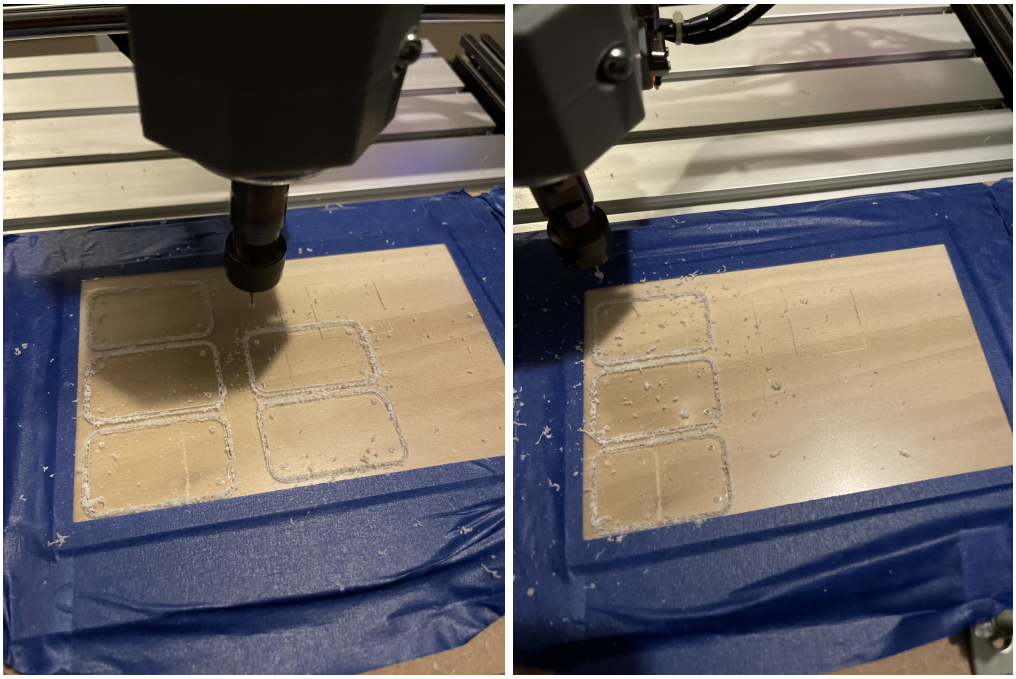


Figure 47 (CNC Router): Printing our rectangular bases on the CNC router. Left was our first test print. The right one is once we had the experimenting done.

The buttons were mainly created using a CNC router. Easel and inkscape were utilized to easily create any designs we wanted for the CNC router and then Easel exported it to gcode for use with Candle - the gcode execution software used by the CNC router. Our first iterations were just one rectangular base at a time and we moved the head manually so that we could get multiple tests on one sheet. This first sheet actually did not cut all the way through due to a warped MDF board underneath. A huge problem, as mentioned in challenges, was that PETG was likely to melt and wrap around the bit like cotton candy during the milling process. In the pictures above a little bit of it can be seen on the bits. After getting past this and finding the right settings, we also optimized to get about 9 bases per sheet as shown in the right picture above.

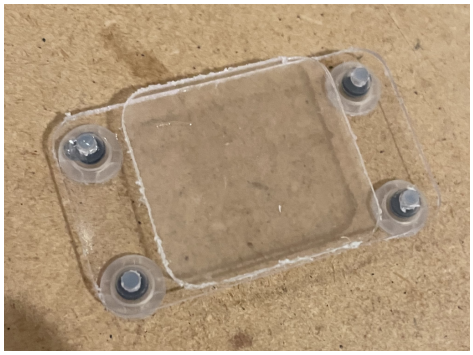


Figure 47 (First Button): After we test printed our first square.

With all of the settings figured out our square tops were quite easy, but we still wanted to mill just one at first. Good thing we did! Turns out we accidentally made the button 1” instead of .94”. This is seeable in the picture above.

After milling everything we smoothed the edges with a razor blade (easier to cut off the rough little tabs on each side with this than, say, a file). During this part we also used a pin vice with a .70 mil bit hand drill each hole to the right size and affixed the rubber actuators into the rectangular bases.

The final step was adhesion. This required some experimenting so we took some of our scrap and extra to try things out. The best way to show that super glue is not the best way to adhere them is the following figure:

Figure 48 (Adhesion disaster): Our first test with gluing.

After trying various methods we actually contacted a materials/mechanical connection of ours to ask how to best adhere two PETG while maintaining visibility. He recommended creating a negative mold of the button from a higher melting point material to use as a clamp-locate system and then melting the PETG tacky with each other while in the mold. We did not have much time for iteration on this to get it right though. In the end we did use a clamp-locate-like system for alignment and were able to get by with just a bit of epoxy in the corners of the square.

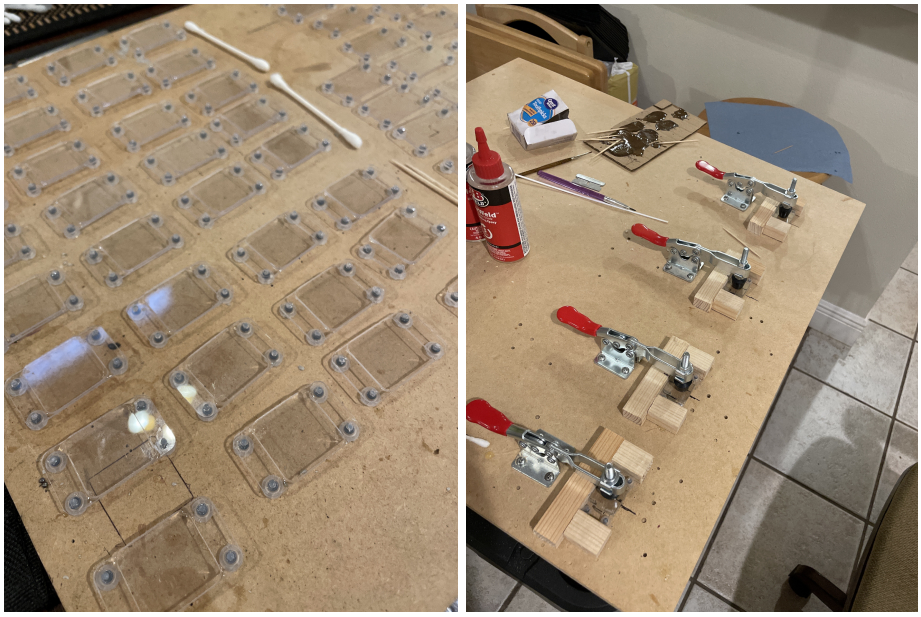


Figure 49 (Button Production): Epoxied buttons drying (left) and alignment system (right)

6.3.8 Housing

The main components that decided the design of the housing were the PCB, power supply, and the screen. However, planning the design could not start until all of those components were finalized. The biggest component that shaped our design was the screen. While the screen was 14 by 8.5 inches, the acrylic panels used to mount the screen and the button hardware were 15.125 by 9.625 inches. With the screen and panel heights combined, the height was about 1.5 inches. Using that as our basis, we knew the housing had to be at least that size to fit the screen. The PCB also needed a height clearance of about 3.5 inches. Due to how wide the acrylic panel and screen already made the housing, the housing automatically had enough space to contain the Jetson Nano and power supply also. For the power supply, a hole was cut out on the side so that there was access to the power switch and charging port. Two shelves were designed to support the screen at the top of the housing so that it was flush with the top opening.Adding about 0.25 inches for the walls, the dimensions of the housing needed to be 15.925 by 10.125 by 5.25 inches.

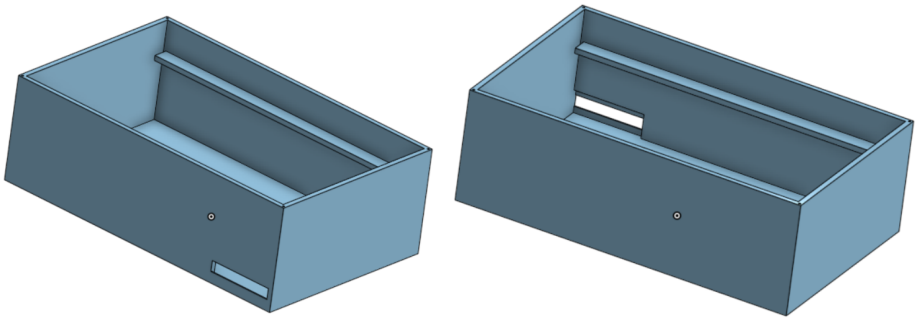


Figure 50 (Housing prototype): Front view(left) and back view(right) of the housing design

6.4 Testing

6.4.1 Game Engine Testing

Overview

The game engine hosts the actual game being played. It should always make sure that the game has been set up. It will call the check win function on each move. The engine interfaces with the board to show the layout of each piece and should only allow the player and computer to make valid moves. Instructions should be given on when moves are allowed. The game engine should essentially allow for player versus player playing of a game, where eventually the AI will function as the second player in two player mode. The game engine must allow for moves to be made, pieces to move from one square to another, remove old pieces, assign the attributes of each move upon modification of each piece, and initialize the check win function after each player (AI included) makes a move.

Passing tests

The game engine passes the tests when the logic of every implemented game function according to the rules of the game, such as pawns move forward one or two on start and attack diagonally. It also must call the check win function after every move, successfully update pieces and squares after each move, and should be resettable by the setup function when it is called. No permanent changes are made to the board layout. The logic of each game must also hold true, such as bishops moving diagonally, or pieces in 4-in-a-row falling to the bottom. The status of pieces should be updated within the game engine, however is a piece is displayed on the board physically when it is not meant to be, or is not displayed when it should be, this may be an issue with the hardware instead of the engine, so it is not always a failed test for the game engine.

Procedure for testing

Testing the game engine includes testing that every piece makes the correct moves and are not able to move to an illegal place. All squares on the board should be verified as reachable. This will be done by manually placing pieces in each square on the board and verifying that the piece exists and can show up on the graphical interface and eventually physical board. Pieces will be sent the signal to move and a technician will verify that the attributes of the piece have updated as well as verify the tile the piece moved to and from was correct. Many of these tests can be automated, allowing for larger scale testing. The automated tests will be run regularly, printing a debug script of the moves made and status of each affected tile and piece involved in the move. Manual testing will also be performed, as sometimes the tests themselves can be flawed. An entire game will be run on the machine from time to time to verify there are no hidden flaws, but the majority of the time, game testing will be automated and reviewed.

Test Cases

Game	*Test Case*	*Details*	*Testing Procedure*	*Passing Test*
Chess	Pawns move	Forward one if no piece in front of it Forward two if it	Tester initializes a game with the piece in the center, then	Once all moves can be made, the test is successful.

		is the pawn's first move Diagonal forward one to attack.	attempts all possible moves from that point.	
Chess	Knights move	One square one direction + two squares 90 degrees rotated to that (in either order), including its attack.	Tester initializes a game with the piece in the center, then attempts all possible moves from that point.	Once all moves can be made, the test is successful.
Game	*Test Case*	*Details*	*Testing Procedure*	*Passing Test*
Chess	Bishops move	Diagonal any place between itself and an occupied square Diagonal to any piece of the opponent's piece if attacking, provided no piece is between them.	Tester initializes a game with the piece in the center, then attempts all possible moves from that point.	Once all moves can be made, the test is successful.
Chess	Rooks move	Horizontal or vertical any place between itself and an occupied square Horizontal or vertical to any piece of the opposite color if attacking, provided no piece is between them During a castle. (see Castle event)	Tester initializes a game with the piece in the center, then attempts all possible moves from that point.	Once all moves can be made, the test is successful.

Chess	Queens move	Diagonal or horizontal or vertical any place between itself and an occupied square Diagonal or horizontal or vertical to any piece of the opposite color if attacking, provided no piece is between them.	Tester initializes a game with the piece in the center, then attempts all possible moves from that point.	Once all moves can be made, the test is successful.
Game	*Test Case*	*Details*	*Testing Procedure*	*Passing Test*
Chess	Kings move	Diagonal or horizontal or vertical one square, including its attack During a castle. (see Castle event)	Tester initializes a game with the piece in the center, then attempts all possible moves from that point.	Once all moves can be made, the test is successful.
Chess	Pawns do not move	Forward if there is a piece directly in front of it.	Tester initializes a game with the scenario that the piece should not be able to move. This is done for all possible configurations of the piece.	The test is successful when all configurations result in the piece not being able to be moved.
Chess	All pieces do not move	If the move results in its own color going into "check" Off the allotted board Looping around the board	Tester initializes a game with the scenario that the piece should not be able to move. This is done for all possible configurations of the	The test is successful when all configurations result in the piece not being able to be moved.

		To an occupied square of its own color.	piece.	
Chess	Piece leaves square	Once the piece is moved to the new square, the piece is no longer displayed on the old square.	Tester initializes a game with a piece on the board. The piece is moved.	The test is successful when the square the piece was previously on no longer displays the piece after the move.
Chess	Piece arrives at new square	Once the piece is moved to the new square, the piece is displayed on the new square.	Tester initializes a game with a piece on the board. The piece is moved.	The test is successful when the square the piece is moved to displays the piece after the move.
Game	*Test Case*	*Details*	*Testing Procedure*	*Passing Test*
Chess	Piece is removed on capture	When a piece is captured by the opponent, the piece should be removed from the board and become unusable for the remainder of the game.	Tester initializes a game where a piece can capture another. The tester then moves the piece to capture the piece.	The test is successful when the piece is removed from the game on capture by the opponent.
Chess	Pawns get knighted	Once a pawn makes it to the other end of the chess board (the opponent's side) the user is prompted as to what piece they want to make the pawn become. The user can choose any piece from the game.	Tester initializes a game where a pawn can be moved to the other side. The tester moves the piece to the knighing position and picks a piece for the pawn to become.	The test is successful when the piece successfully transforms into the piece chosen by the user, and functions as the new piece.
Chess	Castle event is possible	Castling cannot be performed when king or rook have	Tester initializes a game where a castle event is possible.	The test is successful when the castle move results in the

		moved Castling cannot be performed to get out of check Castling cannot be performed when pieces are between king and rook Castling results in the king moving two squares right and rook two squares left. (they cross over each other)	The tester then castles the pieces.	pieces moving to the correct squares after the castle event.
Game	*Test Case*	*Details*	*Testing Procedure*	*Passing Test*
Checkers	Pawns move	Pawns move forward diagonally any place between their start position and the next unoccupied square.	Tester initializes a game with the piece in the center, then attempts all possible moves from that point.	Once all moves can be made, the test is successful.
Checkers	Kings move	Kings move diagonally any place between their start position and the next unoccupied square.	Tester initializes a game with the piece in the center, then attempts all possible moves from that point.	Once all moves can be made, the test is successful.
Checkers	Walls are functional	Pieces treat walls functionally as mirrors.	Tester initializes a game where a piece is against a wall. The tester then moves the piece such that it will reflect off the wall.	The test is successful when a piece can be moved in reflection to a wall.

Checkers	Pieces capture the opponent's piece	Pieces capture the opponent's piece if there is an opponent piece in a forward diagonal direction and there is an available space located in a forward diagonal direction of the opponent's pawn, then the user's pawn is allowed to jump over it and capture it.	Tester initializes a game where a piece is able to capture a piece from the opposite team. The tester then moves the piece such that it captures the opponent's piece.	The test is successful when the piece is captured and removed from the game on being captured from the opponent.
Game	*Test Case*	*Details*	*Testing Procedure*	*Passing Test*
Checkers	Pawns become kings	Pawns become kings if a pawn reaches the first row on your opponent's side of the board, it becomes a king. It will have a symbol of a king's crown once it becomes a king.	Tester initializes a game where a pawn can become a king. The tester then moves the pawn such that it reaches the other side to become a king.	The test is successful when the pawn is turned into a king on reaching the opponent's side, and the piece functions as a king.
Checkers	Piece leaves square	Once the piece is moved to the new square, the piece is no longer displayed on the old square.	Tester initializes a game with a piece on the board. The piece is moved.	The test is successful when the square the piece was previously on no longer displays the piece after the move.
Checkers	Piece arrives at new square	Once the piece is moved to the new square, the piece is displayed on the new square.	Tester initializes a game with a piece on the board. The piece is moved.	The test is successful when the square the piece is moved to displays the piece after the move.

Checkers	Piece is removed on capture	When a piece is captured by the opponent, the piece should be removed from the board and become unusable for the remainder of the game.	Tester initializes a game where a piece can capture another. The tester then moves the piece to capture the piece.	The test is successful when the piece is removed from the game on capture by the opponent.
4-in-a-row	Pieces fall	When a piece is placed on the board, the piece will wall all the way to the bottom of the board.	Tester initializes a game where a piece can be placed in a position such that the piece can fall. The tester then places the piece in such a position that the piece falls.	The test is successful when the piece falls to the bottom most available spot, and the location is set to this position.
Game	*Test Case*	*Details*	*Testing Procedure*	*Passing Test*
4-in-a-row	Pieces do not overlap	When a piece is placed on the board, the piece should fall to sit on top of the highest piece in the row, or the bottom if there are no pieces. Pieces should not occupy the same spot.	Tester initializes a game where a piece can be placed in a position such that the piece can fall on top of a different piece. The tester then places the piece in such a position that the piece falls on top of a different piece.	The test is successful when the piece falls to the bottom most available spot, and the location is set to this position, without overlapping.
4-in-a-row	Pieces cannot be placed above top row	If the column is full, no pieces should be able to be placed in that column.	Tester initializes a game where a column is full. The tester then attempts to place a piece in this column.	The test is successful if the tester is unable to place a piece in this column, and they are still able to place their piece elsewhere, such that they do not lose their turn.
4-in-a-row	Pieces	Pieces cannot be	Tester initializes a	The test is successful

	cannot be placed outside game area	placed outside the area dedicated to the specific game of 4-in-a-row.	game in any state and attempts to place a piece in a position outside the play area.	if the tester is unable to place a piece outside the play area, and they are still able to place their piece elsewhere, such that they do not lose their turn.
Tic-Tac-Toe	Pieces do not overlap	When a piece is placed on the board, the piece should only be able to be placed in an empty spot. Pieces should not occupy the same spot.	Tester initializes a game where a square is taken. The tester then attempts to place a piece in the taken spot.	The test is successful when the piece is unable to be placed in the occupied spot, and the tester is still able to place their piece elsewhere, such that they do not lose their turn.
Game	*Test Case*	*Details*	*Testing Procedure*	*Passing Test*
Tic-Tac-Toe	Pieces cannot be placed outside game area	Pieces cannot be placed outside the area dedicated to the specific game of Tic-Tac-Toe.	Tester initializes a game in any state and attempts to place a piece in a position outside the play area.	The test is successful if the tester is unable to place a piece outside the play area, and they are still able to place their piece elsewhere, such that they do not lose their turn.
All games	Game does not continue after it ends	Once a game is complete by a win/loss, stalemate, quit, or end condition, the game cannot be continued.	Tester initializes a game that is almost over, then completes the game. After the game is deemed over, the tester attempts to continue the game by moving pieces in the game.	The test is successful if the tester is unable to interact with the game after the game is ended.
All games	Game is playable on start	Once the setup function creates a game, the first player should be	The tester runs the setup function to create a new game, then attempts to	The test is successful if the tester is able to make the first move after the game is set

		able to make their first move.	make the first move.	up.
All games	Players only play on their turn	Players should be able to make only one move on their own turn, and unable to make a move when not their turn.	The tester initializes a game and attempts to make a move outside of their own turn.	The test is successful if the tester is unable to make a move outside of their own turn.

Table 9 (Test Cases for Chess): These test cases will help us verify that the software is following all of the rules of chess.

6.4.2 Game Setup Function:

Overview

The setup function is responsible for the creation of new games on startup of a game. The function needs to initialize all pieces in the particular game being played to the default state. To test this, a primitive graphical interface will be created explicitly for the purposes of testing. On initialization of the game, pieces should start at their respective tiles. The graphical interface will then display the location of each piece which will be verified by a technician manually to ensure starting positions are correct.

Passing tests

The graphical display, and eventually board, displays the correct starting condition for the game that is selected to play. The game should also be playable after the setup is complete, meaning that once a move is made the startup function does not reset to the beginning. Games should always be set up to the correct starting position of a given game, and the previous game should not impact the new game. Games should be capable of being quit, reset, lose power, and won by either player including the AI. The correct game should also be set up by the setup function, regardless of what game was played previously.

Procedure for testing

The game setup function will be called with the game type inserted as a parameter. The setup function will execute, then provide a notification that it has completed. The technician will then plug in the intended layout into the graphical interface created for testing to verify that the starting layout has been configured correctly. Numerous different starting conditions will be tested to ensure there are no errors resulting in the startup function requiring the previous game to start in any specific orientation.

Test cases

Game	*Test Case*	*Details*	*Testing Procedure*	*Passing Test*
All games	Computer won	The game before the one being set up by the game setup function was won by the computer against a player.	Tester initializes a game where the computer will win. Once the computer wins, the game setup function is called to reset a new game, and the tester verifies the status of the game that is set up.	The test is successful if the game is set up after the setup function is called.
All games	Player won	The game before the one being set up by the game setup function was won by the player against the computer.	Tester initializes a game where the player playing against the computer will win. Once the player wins, the game setup function is called to reset a new game, and the tester verifies the status of the game that is set up.	The test is successful if the game is set up after the setup function is called.
Game	*Test Case*	*Details*	*Testing Procedure*	*Passing Test*
All games	Player 1 won	The game before the one being set up by the game setup function was won by the user at the "Player 1" position.	Tester initializes a game where player 1 will win. Once player 1 wins, the game setup function is called to reset a new game, and the tester verifies the status of the game that is set up.	The test is successful if the game is set up after the setup function is called.
All games	Player 2 won	The game before the one being set up by the game setup function was won by the user at the "Player 2" position.	Tester initializes a game where player 2 will win. Once player 2 wins, the game setup function is called to reset a new game, and the tester verifies the status of the game that is set up.	The test is successful if the game is set up after the setup function is called.

All games	Draw	The game before the one being set up by the game setup function ended in a draw.	Tester initializes a game where the game will end in a draw. Once the game ends, the game setup function is called to reset a new game, and the tester verifies the status of the game that is set up.	The test is successful if the game is set up after the setup function is called.
All games	Stalemate	The game before the one being set up by the game setup function ended in a stalemate.	Tester initializes a game where the game will end in a stalemate. Once the game ends, the game setup function is called to reset a new game, and the tester verifies the status of the game that is set up.	The test is successful if the game is set up after the setup function is called.
Game	*Test Case*	*Details*	*Testing Procedure*	*Passing Test*
All games	Quit	The game before the one being set up by the game setup function was quit.	Tester initializes a game then quits. Once a new game is created, the game setup function is called to reset a new game, and the tester verifies the status of the game that is set up.	The test is successful if the game is set up after the setup function is called.
All games	Reset	The game before the one being set up by the game setup function was reset during the middle of the game, but not quit.	Tester initializes a game then selects reset. The game setup function is called to reset a new game, and the tester verifies the status of the game that is set up.	The test is successful if the game is set up after the setup function is called.
All games	Power loss	The game before the one being set	Tester initializes a game then disconnects power	The test is successful if the

		up by the game setup function was interrupted by a power loss.	to the device. The game setup function is called to reset a new game, and the tester verifies the status of the game that is set up.	game is set up after the function is called automatically.
All games	Repeated calls	The game reset function is called multiple times consecutively without modifying the game between calls.	The game setup function is called to set up a new game. Immediately, the reset function is hit, then again immediately. The tester verifies that the final game is set up.	The test is successful if the game is set up after the function is called.
All games	Correct setup	The game set up by the setup function is set up in the correct zero state (the first move has not yet been made) of the corresponding game.	The tester calls the setup function and verifies that the game is set up correctly, and in a playable state.	The test is successful if the game is set up correctly after the function is called.
Game	*Test Case*	*Details*	*Testing Procedure*	*Passing Test*
All games	Change game type	The game previously played was different to the game currently being set up by the setup function.	Tester initializes a game, then uses the setup function to set up a different game. The tester verifies the status of the game that is set up.	The test is successful if the game is set up correctly after having a different game played previously.

Table 10 (Test Cases for Setup): These test cases are to verify that a game sets up correctly.

6.4.3 Game Solver Testing:

Overview

The game solver is the "AI" that will be used to play against the user in single-player mode. This AI will need to have 3 (or more) difficulty levels available for the player to play against, and be able to win more games against the same user as the difficulty is increased. The AI needs to be able to function as the second user in a game, interacting with the game engine's input output system. The AI needs to make only valid moves, and should have a failsafe for if somehow it accidentally makes an incorrect move to which the game engine tells it is illegal, so the AI will

make a new, legal move. The AI should be able to make its moves within less than one second of computational time

Passing tests

One second is the maximum allowable time for finding a new move and executing it, however longer times may be allowed for if an "extreme" difficulty level is implemented. If played with a user at the same skill level, the AI should demonstrate a noticeable increase in wins as the difficulty level increases. If a defined solving algorithm is used, multiple games should be sent to the AI to play against and it should win only a few on low difficulty. More should be won on medium. Most should be won on hard. The AI must be able to make legal moves and can move again if the gaming engine tells it that its move was invalid.

Procedure for testing

To test the AI's ability to move, automated tests will be created that feed the AI a given gameboard, then the AI will be told to make its move. A technician will check the graphical interface or board to verify the move was performed legally and within the time constraint. Technicians will also play games against the AI to ensure the AI "feels right" and to see if any patterns emerge that may give the user a clear upperhand. The AI will also play against itself on different difficulty levels and the win loss statistics will be monitored to ensure the higher difficulty AI wins more than the lower level ones.

Test cases

Plays against	*Test Case*	*Details*	*Testing Procedure*	*Passing Test*
All players	Easy win	The computer has a game where it can win in just one move.	Tester initializes a game where the game can be won by making one move. This is done for the easy, medium, and hard difficulty settings. It is expected that all settings should be able to win in this scenario.	All difficulty settings make the moves that win the game.
All players	Medium win	The computer has a game where it can win in one third of the total number of average moves for the given game.	Tester initializes a game where the game can be won by making one third of the total number of average moves for the given game moves. This is done	Medium and hard difficulty settings make the moves that win the game.

			for the easy, medium, and hard difficulty settings. It is expected that the medium and hard settings should be able to win in this scenario.	
All players	Hard win	The game is not in a position that has a well defined end path to it.	Tester initializes a game where the game does not have a known path to win. This is done for the easy, medium, and hard difficulty settings. It is expected that the hard setting should be able to win	Hard difficulty setting makes the moves that win the game.
Plays against	*Test Case*	*Details*	*Testing Procedure*	*Passing Test*
All players	Start board layout	The game is initialized to the status of the start of a normal game.	Tester initializes a game to the default start of the game. The tester then analyzes the computer's moves to ensure it is playing the game as expected.	The computer makes moves on its turn, and plays through to the end of the game.
All players	Given board layout	The game is initialized to the status of a game that has already played out such that the AI does not see the start of the game.	Tester initializes a game to a given random position. The tester then analyzes the computer's moves to ensure it is playing the game as expected.	The computer makes moves on its turn, and plays through to the end of the game.
All players	Blank board layout	The game is completely void of pieces (or filled in entirely)	Tester initializes a game where there are no pieces on the board. The tester	The computer makes no action when no moves are possible, and does not attempt

			analyzes the computer's response to ensure it does not generate bugs. *This test is a failsafe test.*	to generate false moves.
All players	No moves possible	There are no possible moves for the computer to make, from lack of pieces, lack of open spaces, or no moves being legal to make.	Tester initializes a game where the computer is unable to make any moves. The tester analyzes the computer's response to ensure it does not generate bugs. *This test is a failsafe test.*	The computer makes no action when no moves are possible, and does not attempt to generate false moves.
Plays against	*Test Case*	*Details*	*Testing Procedure*	*Passing Test*
All players	No win possible	There is no way for the computer to win the given game.	Tester initializes a game where there is no possible way for the computer to win the game. The tester then analyzes the computer's response to ensure it plays out to the end of the game anyway.	The computer plays the game through to the end, even when it cannot win.
All players	1 second to think	The computer takes no longer than one second to make a decision to move in any state of the game.	Tester initializes a game where the computer needs to make a move and records the amount of time it takes for it to respond and make the move. This is repeated in many scenarios.	The computer does not take longer than 1 second to make its move from the end of the player's move to the end of it's move for any recorded test.

Other computer	Easy v. medium	Two computers play against each other, one being easy, the other medium difficulty setting.	Two different instances of the computer are set up to play against each other, one set to the easy difficulty level, the other set to medium. The tester records the moves, and analyzes the win results of numerous matches played in this way.	The medium difficulty level wins against the easy difficulty level 75% of the time or more.
Plays against	*Test Case*	*Details*	*Testing Procedure*	*Passing Test*
Other computer	Easy v. hard	Two computers play against each other, one being easy, the other hard difficulty setting.	Two different instances of the computer are set up to play against each other, one set to the easy difficulty level, the other set to hard. The tester records the moves, and analyzes the win results of numerous matches played in this way.	The hard difficulty level wins against the easy difficulty level 95% of the time or more.
Other computer	Medium v. hard	Two computers play against each other, one being medium, the other hard difficulty setting.	Two different instances of the computer are set up to play against each other, one set to the medium difficulty level, the other set to hard. The tester records the moves,	The hard difficulty level wins against the medium difficulty level 75% of the time or more.

			and analyzes the win results of numerous matches played in this way.	
Player	Easy difficulty	The computer plays against a player at the beginner, intermediate, and expert skill levels on the easy difficulty setting.	Human players who are self described as beginner, intermediate, and expert skill levels play a few games against the computer on its easy difficulty level. The tester records the number of wins/losses for each player at this level.	+/-20% the player should win against the computer at a rate of Beginner: 50% Intermediate:75% Expert: 95%
Plays against	*Test Case*	*Details*	*Testing Procedure*	*Passing Test*
Player	Medium difficulty	The computer plays against a player at the beginner, intermediate, and expert skill levels on the medium difficulty setting.	Human players who are self described as beginner, intermediate, and expert skill levels play a few games against the computer on its medium difficulty level. The tester records the number of wins/losses for each player at this level.	+/-20% the player should win against the computer at a rate of Beginner: 20% Intermediate:50% Expert: 75%
Player	Hard difficulty	The computer plays against a player at the beginner, intermediate, and expert skill levels on the hard difficulty setting.	Human players who are self described as beginner, intermediate, and expert skill levels play a few games against the computer on its hard difficulty	+/-20% the player should win against the computer at a rate of Beginner: 5% Intermediate:25% Expert: 50%

			level. The tester records the number of wins/losses for each player at this level.	

Table 11 (Test Cases for Solver): These test cases are to verify that the AI is working and works at various difficulties.

6.4.4 Input/Output Testing

Overview

The inputs and outputs should be responsible for correctly transferring data between hardware and software. The hardware should read any input signals properly. In return, the controller should be able to receive the corresponding result from the software and display the desired output in the proper part of the screen. Both the software and hardware should interface correctly so that interaction with the device is seamless. The majority of the communication between the user and the Game Frame will be through the use of buttons. If time permits it, we can also add in a microphone so that the device can also take in voice commands from the user.

Passing tests

The microcontroller should be able to detect when a button is pressed, and specifically which button is pressed. All 64 buttons should be able to send the right signal to the processor and the processor should identify them all separately. Since multiple buttons will be connected to a single pin, the microcontroller should be able to differentiate all the different buttons serviced by the same pin. This should indicate that the button matrix is working properly.

Procedure for testing

Press each button separately and see if they are all registered by the embedded software. The screen should also respond properly to each button being pressed individually.

Test cases

Input/ output	*Test Case*	*Details*	*Testing Procedure*	*Passing Test*
Input	Processor receives input	All of the 64 input buttons are able to send a signal to the processor.	Each button will be tested individually by pressing one button at a time and seeing if the controller correctly reads the 1 or 0 when pressed.	Each button correctly sends a signal to the controller that it has completed a circuit and displays the input. For testing, it is likely that the

				button will be coded to turn on an LED when pressed and off when not pressed for each individual button.
Input	Unique identifiers	Each button has a unique number to identify.	Each button should all be individually recognized by the controller. The controller should know exactly where in the switch array a button is pressed once a signal is detected. Each button will be coded to send a unique identifier back to the computer during the test. For simplicity, each button will be labeled in the code 1 to 64, and when a button is pressed, the correct number should be printed into the screen. It needs to also determine if multiple buttons were pressed at the same time and know which ones.	To pass this test, each number must be individually recognized and also be able to read them in any order they are pressed. It needs to also be able to detect when multiple buttons are pressed at the same time. When multiple buttons are pressed at the same time, the numbers those buttons were assigned to should display at the same time together.
Input/ output	*Test Case*	*Details*	*Testing Procedure*	*Passing Test*
Output	Correct output screens	Screens display the correct square of information.	Divide up the pixels on the LCD evenly and assign each button a part of the LCD. Pressing a button should turn on it's assigned pixels and nothing else. This should work with simultaneous buttons pressed	To pass this test, each button must only turn on the pixels it has been assigned. Every other pixel where a button is not pressed should be turned off.

Table 12 (Test Cases for I/O): Procedures to verify the functionality of I/O.

6.4.5 Game End/Won Testing

Overview

After each move by either the player or the computer, the game end function should be run. Win and lose scenarios, stalemates, draws, quits, cases of over moving, and running out of time are all things that could make the game end. The end game function should check for each of them. Once a game is determined to be over, the end game function will send a signal to display the end game screen and the game engine will be sent a signal to stop allowing players to make moves. The way the game has ended must also be sent to the end game screen, but the score and other details about the game that need to be displayed will be handled by the game engine. The checker should only test the recently affected pieces in order to have a fast runtime and should

never take more than a tenth of a second to run. The game should not be playable until after the checker has completed checking the status.

Passing tests

The end game checker should be able to indicate that the game is over for all possible scenarios and test for a win or stalemate in any situation that the game can be configured for. Even if it is not possible to arrive in the state, like having 10 queens on the board in chess, it should be able to be checked. The checker should be able to indicate whether player 1 or player 2 has won or lost, but the end game screen will be responsible for deciphering if either player is the AI. The checker also keeps a timer running for timed games and must end the game by sending an "out of time" signal to the end game screen for timed games. Upon being told by the input panel that the players choose to quit, the end game checker should treat it similarly to a win, loss, or stalemate by sending the end screen a signal that the game was quit.

Procedure for testing

Since the game engine will have many tests of handling end game scenarios, the end game checker will also be tested a lot with the game engine. The checker will only work when the game engine is working, but they can both be refined at the same time. Games will be set up manually in winning and losing cases, and the checker will be run. Only when the checker has 100% accuracy in determining the win status of a game will it be determined complete. There is no margin of error. Games may also be played out and automated testing can be run in player-player mode. Large quantities of games can be run and can have output logs sent to be analyzed. Games will also be played out for a few moves then quit to verify this function works as well.

Test cases

End type	*Test Case*	*Details*	*Testing Procedure*	*Passing Test*
Win	Win player 1	Player 1 makes the move that wins the game.	Tester initializes a game where player 1 is about to win. The tester then plays through such that player 1 wins. A new game is then initialized, and the tester records the results.	A new game is set up at the correct starting state after the setup function is called, and the game is functional.
Win	Win player 2	Player 2 makes the move that wins the game.	Tester initializes a game where player 2 is about to win. The tester then plays through such that player 2 wins. A new game is then initialized,	A new game is set up at the correct starting state after the setup function is called, and the game is functional.

			and the tester records the results.	
Lose	Lose player 1	Player 1 makes the move that loses the game.	Tester initializes a game where player 1 is about to lose. The tester then plays through such that player 1 loses. A new game is then initialized, and the tester records the results.	A new game is set up at the correct starting state after the setup function is called, and the game is functional.
End type	*Test Case*	*Details*	*Testing Procedure*	*Passing Test*
Lose	Lose player 2	Player 2 makes the move that loses the game.	Tester initializes a game where player 2 is about to lose. The tester then plays through such that player 2 loses. A new game is then initialized, and the tester records the results.	A new game is set up at the correct starting state after the setup function is called, and the game is functional.
Win	Win computer	The computer makes the move that wins the game.	Tester initializes a game where the computer is about to win. The tester then plays through such that the computer wins. A new game is then initialized, and the tester records the results.	A new game is set up at the correct starting state after the setup function is called, and the game is functional.
Lose	Lose computer	The computer makes the move that loses the game.	Tester initializes a game where the computer is about to lose. The tester then plays through such that the computer loses. A new game is then initialized, and the tester records the	A new game is set up at the correct starting state after the setup function is called, and the game is functional.

			results.	
Stalemate	Stalemate player v. computer	A player v. computer game ends in stalemate.	Tester initializes a game against the computer which is about to end in a stalemate. The tester then plays through such that the game ends in a stalemate. A new game is then initialized, and the tester records the results.	A new game is set up at the correct starting state after the setup function is called, and the game is functional.
End type	*Test Case*	*Details*	*Testing Procedure*	*Passing Test*
Stalemate	Stalemate player v. player	A player v. player game ends in stalemate.	Tester initializes a game against a player which is about to end in a stalemate. The tester then plays through such that the game ends in a stalemate. A new game is then initialized, and the tester records the results.	A new game is set up at the correct starting state after the setup function is called, and the game is functional.
Quit	Quit player v. computer	A player v. computer game is quit.	Tester initializes a game against the computer which is then quit by the tester. A new game is then initialized, and the tester records the results.	A new game is set up at the correct starting state after the setup function is called, and the game is functional.
Quit	Quit player v. player	A player v. player game is quit.	Tester initializes a game against a player which is then quit by the tester. A new game is then initialized, and the tester records the results.	A new game is set up at the correct starting state after the setup function is called, and the game is functional.

Out of time	Out of time player v. computer	A player v. computer game ends because the time limit for the game is hit.	Tester initializes a game against the computer which is about to run out of time. The tester then plays through such that the game runs out of time. A new game is then initialized, and the tester records the results.	A new game is set up at the correct starting state after the setup function is called, and the game is functional.
End type	*Test Case*	*Details*	*Testing Procedure*	*Passing Test*
Out of time	Out of time player v. player	A player v. player game ends because the time limit for the game is hit.	Tester initializes a game against a player which is about to run out of time. The tester then plays through such that the game runs out of time. A new game is then initialized, and the tester records the results.	A new game is set up at the correct starting state after the setup function is called, and the game is functional.
Over move count player v. computer	Over move count player v. computer	A player v. computer game ends because the move limit for the game is hit.	Tester initializes a game against the computer which is about to run over the maximum amount of moves. The tester then plays through such that the game runs over the maximum number of moves. A new game is then initialized, and the tester records the results.	A new game is set up at the correct starting state after the setup function is called, and the game is functional.
Over move count player v. player	Over move count player v. player	A player v. player game ends because the move limit for the	Tester initializes a game against a player which is about to run over the maximum	A new game is set up at the correct starting state after the setup function

		game is hit.	amount of moves. The tester then plays through such that the game runs over the maximum number of moves. A new game is then initialized, and the tester records the results.	is called, and the game is functional.

Table 13 (Test Cases for Game End/Won): Procedure to verify if the game is over.

6.4.6 Power Testing

Overview
The power needed to be tested in two main ways: battery life needed to be sufficient and voltage and current to the devices from regulators needed to be correct.

Passing tests
In order to be successful we need at least 2 hours of operation out of our battery. In addition the regulators needed to supply 3.3V, 5V and 5V. We also needed to make sure the amperage drawn by the devices was very low for the 3.3V, and 2A/3A maximum for the 5V regulators.

Procedure for testing
The way to measure the battery was to leave it on with all the components connected and in operation. We accidentally ended up being able to test this very easily since we had to program and develop on the device itself. Once we first got to the end of the battery life we used the stopwatch on an iPhone to time the duration of the battery on the next time we used it (after charging it that is).

The voltage was easily measured at various nodes with a Fluke multimeter and the current drawn with a current clamp. Prior to connecting the actual devices we also tested the regulators using an active load.

6.4.7 Thermal Testing

Overview
The temperature of the device needs to be cool enough to not overheat during use or transport and, though not a concern for us, must not be so cold as to damage the components.

Passing tests
The voltage regulators gave us a good estimate of about 125℃ as a maximum. Being below this was the absolute minimum of passing. Having a 124℃ device is obviously not much better though and still would lessen our device's lifespan

Procedure for testing
To test that the heatsinks were managing the heat correctly we used a thermal camera. We could just use the thermal camera to point at the source of heat and look at the temperature reading. When the room was around 30℃ the thermal camera measured around 34-36℃ for the suspect components. Likewise around 20℃ ambient the thermal camera measured around 24-26℃. We concluded that the hottest components of our device operated at about 4-6℃ hotter than ambient.

6.4.8 Weight Testing

Overview
The device needs to weigh low enough to be transportable.

Passing tests
Our requirement was 4 lbs, so if all components weigh less than 4 lbs when added together we pass.

Procedure for testing
We utilized a mechanical scale to weigh the components in grams and then converted to lbs. The screen and wireframe component did not fit on this scale so we used a digital weight scale to first measure a group member and then again while the group member was holding this part of the device.

6.4.9 Embedded Communication Testing

Overview
The serial communication must be transmitting the right values at both a sufficient frequency and with few enough errors that it can be used.

Passing tests
Correct conveying input values as outputs and fast transmission (many times per second) determine if the device passes.

Procedure for testing
For the transmission testing from the MSP430 an RS232-to-USB is connected to a desktop computer. Then PuTTY terminal is used on the desktop computer to display what is being transmitted. For receive testing on the Jetson Nano, a simple debug script was created that prints to console as it receives values on the UART RX pin.

6.4.10 Electrical Shorts Testing

Overview
The device should not have any electrical shorts so that it works as designed and intended.

Passing tests
Having a resistance between two points that is sufficiently high or adequately low for where it is being measured. That is, there should not be nearly 0 resistance between a trace and the ground plane on the PCB, for example.

Procedure for testing
Testing involves using the ohmmeter setting of a multimeter and exhaustively testing all of the nodes to make sure no shorts exist.

6.5 Evaluation Plan

6.5.1 Evaluation of Hard Requirements

Requirement Type	*Name*	*Test*	*Result Grade*	*Results*
Power	Battery Life	To test the battery life, games against AI should be simulated until the device turns off or stays on past the battery life goal.	Satisfactory: >2 hours Unsatisfactory: < 2 hours	Satisfactory: >2 hours
Physical	Dimensions	While this should not require any extensive test besides measuring, the dimensions should be met as long as it was built around this from the beginning. The only possibility that can cause unforeseen changes to the dimensions is if a different component is physically demanding it.	1-foot width or length maximum	Unsatisfactory: >1-foot width or length maximum
	Weight	Using any weight measuring device	Satisfactory: ≤4-lbs Unsatisfactory: > 4-lbs	Unsatisfactory: > 4-lbs

Monetary	Unit Cost	Evaluate the cost of each material used in a single build and add them all together	Satisfactory: ≤$400 Unsatisfactory: > $400	Unsatisfactory: > $400
Software	Chess	Play through the one game either against AI or against another player without interruption	Satisfactory: No glitches Unsatisfactory: Any bugs or crashes	Unsatisfactory: Any bugs or crashes

Table 14 (Evaluation Plan): A set of tests to evaluate if our hard requirements are met

As a quick note, we definitely missed the mark on a lot of our requirements, but we were not too far off. A little extra time or even one extra iteration of design would fix most if not all of the requirements. Our goals were definitely feasible.

6.6 Facilities and Equipment

Tool	*Description of use*
CNC Router	Sainsmart router to make buttons
Belt Sander	Flatten contact nail tops
Active Load	Hardware testing
Multimeter	Hardware measurements and testing
Tape Measure	Physical dimensions and measurements
Digital Caliper	Physical dimensions and measurements
Power Supply	Hardware testing
Custom clamp-locate system	Aligning buttons during adhesion
Thermal Camera	Thermal measurements and testing
Mechanical Scale	Weighing
Screwdrivers	
3D Printer	Housing
Desktop	Programming, collaboration, and administrative work

Computer(s)	
IDE	Pycharm, Code Composer
Discord	Group communication
Google Docs/Slides	Administrative work and assignments
Linux	Operating system that we developed for and on

Table 15 (Tools): A list of the tools or environments we used throughout the project

Facilities used were our houses or apartments. Most of the work for this was done remotely due to the circumstances.

7. Operation Guide

7.1 Start up

When a user operates the Game Frame for the first time, they must activate the power switch in order to turn on the device. The idea we had, if we had finished implementation, was that Linux would boot the program immediately and the user would be greeted by the main menu. Here the user can select the buttons on the screen to navigate between games that will be located on the lower half of the screen. Ideally we wished for the buttons to be responsible for this job but as of right now the user must navigate with a mouse. In future iterations this will be accomplished with the buttons on the device. Each game is located on it's own corresponding button. Once the user selects which game they are interested in playing, the main menu leads them to that game's menu.

7.2 Menu

Once the game's menu has loaded, the user will be presented with a play option or they can return to the main menu if they selected the incorrect game. Here, on the game menu, the user must specify whether they will be playing with a computer or with another person. If the user chooses a computer to play with, then the menu will ask them what difficulty to select for the computer and which team they wish to play. If the user selects to play with another person then it displays the confirm button so the user can begin once they are ready. Once the game begins, the player will continuously play until someone results in a victory or if there is a draw.

7.2.1 Gameplay

While the game is being played, the software is continuously checking to see if the state of the game results in a victory or draw. As players select and confirm where they wish to move their pieces, the software is double checking to see if that move is allowed or not. The software checks to see if the selected piece is allowed to move and the piece will move only if it is allowed.

7.3 End Game

Once the game inevitably reaches a victory or a draw, the user will be presented by a results screen that shows the result of the game and will ask the user if they wish to reset and play the game again with the same settings or if they wish to return to the game's menu where they can play a new game or the same game but with different settings.

8. Research and Tradeoffs

8.1 Switches

In order for the buttons to even operate we needed switches. There were several options we looked into. The plan was to utilize four switches for each of the 64 buttons - one for each corner. We ended up implementing the device with fewer as the project and its design unfolded.

8.1.1 Tactile Switches

Specification

FUNCTION: Momentary action
CONTACT ARRANGEMENT: SPST, N.O.
TERMINALS: PC pins

Mechanical

ACTUATION FORCE: 130 grams, 160 grams, 200 grams, 260 grams
LIFE EXPECTANCY: 100,000 operations.

Electrical

CONTACT RATING: 50 mA @ 12 V DC.
DIELECTRIC STRENGTH: 250 V AC min.
CONTACT RESISTANCE: 100 mΩ max. initial.
INSULATION RESISTANCE: 10^{11} Ω min.

Environmental

OPERATING TEMPERATURE: -40°C to 85°C
STORAGE TEMPERATURE: -40°C to 85°C

Figure 51 (Tactile Switch Sample Specs): A list of specifications for an example tactile switch. Taken from the PTS645 datasheet.

These small switches were around 18-25¢ and were cheap in cost compared to other options making them good for prototyping. Because they were the cheapest, they also could have led to a device with the least responsive feedback and led to being the cheapest feel amongst the choices. These have a lower voltage rating and minimum current than the next option that we'll go over, making them better for a portable device. The datasheet graciously provided an expected durability of 100,000 uses so that we know its lifespan and could use that to estimate potential repair costs if we went further and made something for market. Repair costs were not an important decision-making factor for our project.

8.1.2 Snap Action Switches

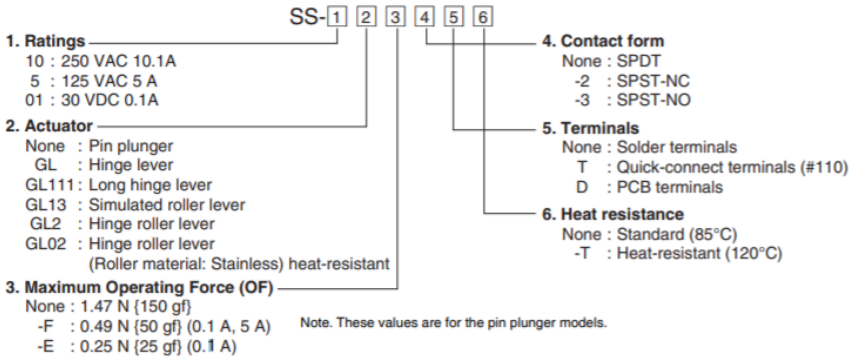


Figure 52 (Snap Action Sample Specs): A list of specifications for an example snap action switch. Taken from the SS-x series datasheet.

Snap action switches were opposite of the previous option in many ways. It was by far more expensive, but had potential for the best tactile response and feel when used concerning purchasable switch components. The DIY video of someone making a jubeat machine used these switches. A huge problem was that it requires much more voltage for operation. The amperage is still manageable, however it was higher than some other options. Higher amperage demand meant that the battery could not last as long. High voltage meant that we needed a boost converter. The datasheet has information pertaining to running the device on lower loads. Even with only 64 instead of the original 256 plan, they were still not what we were looking for. A slight interesting note is that these were used on the CNC router to set up reset/stop points so that that stepper motor wouldn't damage itself by moving the spindle head too far in the X, Y or Z direction.

8.1.3 PCB Trace Switches

One of the best options was to implement electrical contact buttons with PCB trace membranes for the switch. Jubeat uses this method of implementation. PCB prototyping was the most expensive prototyping option leading to an obvious downside. For unit cost this would have been the cheapest to produce since it would be part of the PCB cost.

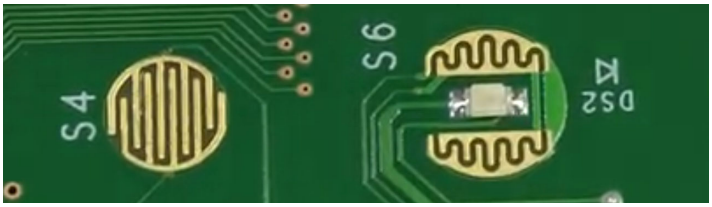


Figure 53 (Folded PCB Traces): A picture provided to visually describe what is being talked about. Origin of image unknown and used under fair use. The picture was acquired from StackExchange.

8.1.4 Arcade buttons

It has been discussed that auxiliary buttons on the side could be used in future design. They would operate not only as UI or power control, but could also be used by various games down the road. They were not necessary and we considered them to be an additional feature. The page for the buttons details an expected 10,000,000 uses on the kind we looked at.

8.2 Controllers

There were a large variety of controllers to choose from. Our build used two. A microcontroller worked for managing the hardware and the array of switches while also interfacing with another controller or computer to manage the software. Another option we considered was to use a microcontroller or other controller with two processors on it. The benefit of the separate controllers was that it would give us the ability to facilitate simultaneous progress and work on the hardware and software side of things at the same time.

8.2.1 Raspberry Pi

Raspberry Pi of course is known for their variety of products and options to use as controllers. The obvious downside of one of these was that they are not as impressive for inclusion on a resume compared to other controllers.

Raspi 4
A Raspberry Pi 4 is monetarily more expensive than other controllers that might have less powerful specs in them. The Raspberry Pi 4, however, has specs powerful enough to implement Python as a default language. The power is not free and could have affected our desired battery life. Though it wouldn't have as much as a Jetson Nano does. We had actually considered this for the software at one time.

Raspi Pico
The Pico was an appealing, incredibly cost efficient option at only $4. The size and lower specs made it desirable as a potential microcontroller, but if it was also sufficient to handle our software, it would be the superior choice. We still came back to the Jetson Nano as the preference because of resume experience though.

RP 2040
It should be noted that the microcontroller chip for Pis can be bought by itself and designed around. This was an even cheaper option.

8.2.2 Teensy

An honorable mention was the Teensy++ 2.0. We stumbled upon this through a DIY jubeat video that our consultant shared with us. In that video, the person utilized this microcontroller for the panels, but we decided that it wasn't for us. The largest benefit was how compact it is. The person in this video utilized a separate computer entirely to run his game. For the software development, it uses a built-in Teensy Loader to load your code and reboots to run the code. To

run code, specifically C or C++, you need to install programs such as the AVR-gcc and the AVR C library.

Touted features directly from the manufacturers include: compatibility with arduino software and libraries, versatile USB connectivity with any type of device, single-push push button programming, easy loader application, free development tools, interoperability on the big three operating systems, a teensy size, and the option to get a model with pins for solderless breadboards.

Specification	Teensy 2.0	Teensy++ 2.0	Teensy 3.0	Teensy 3.1
Processor	ATMEGA32U4 8 bit AVR 16 MHz	AT90USB1286 8 bit AVR 16 MHz	MK20DX128 32 bit ARM Cortex-M4 48 MHz	MK20DX256 32 bit ARM Cortex-M4 72 MHz
Flash Memory	32256	130048	131072	262144
RAM Memory	2560	8192	16384	65536
EEPROM	1024	4096	2048	2048
I/O	25, 5 Volt	46, 5 Volt	34, 3.3 Volt	34, 3.3V, 5V tol
Analog In	12	8	14	21
PWM	7	9	10	12
UART,I2C,SPI	1,1,1	1,1,1	3,1,1	3,2,1
Price	$16.00	$24.00	$19.00	$19.80

Figure 54 (Teensy Model Features): Convenient information table provided by Teensy's website for quick easy comparison between the models.

We originally decided that we wanted a controller we would be sure could handle both the buttons and the game instead. We changed to using two devices, however. The size, lower power requirements, and cost make this a viable contender with the MSP430 if we were still looking to make some changes to the hardware controller.

8.2.3 Pyboard

This is the official board of MicroPython. Python essentially acts like an Operating System for the board. It also contains its own built-in file system. It also has modules for accessing low-level hardware that is specific to the Pyboard. The board was not far in price from a Raspi at about $35, but it had considerably less specs. There were cheaper microcontroller options for handling our hardware, but this could be a consideration for managing software if we were saving on the Nano's cost and power usage and not looking for resume building. MicroPython could have been a good high level language to work with and the power consumption was more manageable than even the Raspberry Pi.

PRICE	
GBP incl. tax	£28.00
approx EUR incl. tax	€39.20
approx USD excl. tax	$35.00
MICROCONTROLLER	
MCU	STM32F405RGT6
CPU	Cortex-M4F
internal flash	1024k
RAM	192k
maximum frequency	168MHz
hardware floating point	single precision

Figure 55 (Pyboard Features): An excerpt of the most essential features of the PYB v1.1 features.

8.2.4 Nvidia Jetson Nano

This device has absolutely massive specifications. This microcontroller touts the highest costs and specs of everything we looked at for our project. This could have handled everything, including the hardware decoding. A positive aspect of this that influenced our decision was that the specific skill of knowing how to code for this looks fantastic to employers. The other most notable factor was how this raised the ceiling on which games we could run on the controller. With a dedicated GPU this also would have potentially allowed us to forgo a microcontroller for a screen panel if a screen without a controller was used in future projects. This really could have been an all-in-one controller for us.

If we did only use this controller we could cut our cost too. This controller has two versions with different amounts of RAM: a 2GB version and a 4GB version. For our use, we went with the 2GB version since it was cheaper and handled all of our software needs. The lower RAM saved us about $40. If we tried to fit everything onto one controller, the 4GB option might have been needed. For the specs that it provides, this controller is actually decently power efficient. We also looked into the 5 watt low power mode, but when not under load it seemed to draw that much anyways. Keeping it in the 10 watt mode helped us retain performance without costing too much extra power. Because of our expenditures on this, we did want to use a cheaper hardware controller, especially since it did not need to do much.

The architecture focuses on parallel processing, which ends up being useful for implementing an AI and definitely would prove useful if we ever need it to handle display control. Ultimately, our desire to implement AI in the game engines led us to get and work with this controller, even if it was not the optimal choice for the job.

8.3 MSP430

We have a separate section for research on the MSP430. Unlike other products like the Jetson Nano, the MSP430 is a line of models, so we needed to select one and go from there (though the Nano does technically have versions). We had one serial out and one clock out for the MSP430 in order to use the shift register. There were 8 input rows as well. We also had a need to communicate with the Jetson Nano. This meant that we needed at least 10 i/o pins for button decoding and an additional 2 for the communication with the nano. Technically however we only used the transmit line and didn't receive from the Jetson. Many of the products in this family

come in series which allow choices of things such as Flash Memory, ROM, RAM, GPIO, ADC choices, and additional peripherals.

8.3.1 DIP Compatible Launchpad

One of the nicest features we looked for in terms of convenience when selecting a chip was one that had a development environment with a dual in-line package socket. Using a chip that can fit into a DIP socket let us develop conveniently with a LaunchPad that also had one. We could then equip our PCB with such a socket (and did) in order to transfer to and from the PCB and LaunchPad. This made flashing our program onto the microcontroller and updating or modifying it a breeze. As for repair considerations, a DIP socket would also make it easier to repair should the chip fail down the road.

Prioritizing such a convenience we already had honed in on a few chips. Namely, we limited it down to anything in the MSP430G2xxx series or the MSP430F20xx series that has either 14 or 20 pin packages. This table from the TI datasheet explicitly lists the models we could have used:

Part Number	Family	Description
MSP430F2001	F2xx	16-bit Ultra-Low-Power Microcontroller, 1KB Flash, 128B RAM, Comparator
MSP430F2002	F2xx	16-bit Ultra-Low-Power Microcontroller, 1KB Flash, 128B RAM, 10-Bit SAR A/D, USI for SPI/I²C
MSP430F2003	F2xx	16-bit Ultra-Low-Power Microcontroller, 1KB Flash, 128B RAM, 16-Bit Sigma-Delta A/D, USI for SPI/I²C
MSP430F2011	F2xx	16-bit Ultra-Low-Power Microcontroller, 2KB Flash, 128B RAM, Comparator
MSP430F2012	F2xx	16-bit Ultra-Low-Power Microcontroller, 2KB Flash, 128B RAM, 10-Bit SAR A/D, USI for SPI/I²C
MSP430F2013	F2xx	16-bit Ultra-Low-Power Microcontroller, 2KB Flash, 128B RAM, 16-Bit Sigma-Delta A/D, USI for SPI/I²C
MSP430G2001	G2xx	16-bit Ultra-Low-Power Microcontroller, 512B Flash, 128B RAM
MSP430G2101	G2xx	16-bit Ultra-Low-Power Microcontroller, 1KB Flash, 128B RAM
MSP430G2111	G2xx	16-bit Ultra-Low-Power Microcontroller, 1KB Flash, 128B RAM, Comparator
MSP430G2121	G2xx	16-bit Ultra-Low-Power Microcontroller, 1KB Flash, 128B RAM, USI for SPI/I²C
MSP430G2131	G2xx	16-bit Ultra-Low-Power Microcontroller, 1KB Flash, 128B RAM, 10-Bit SAR A/D, USI for SPI/I²C
MSP430G2201	G2xx	16-bit Ultra-Low-Power Microcontroller, 2KB Flash, 128B RAM
MSP430G2211	G2xx	16-bit Ultra-Low-Power Microcontroller, 2KB Flash, 128B RAM, Comparator
MSP430G2221	G2xx	16-bit Ultra-Low-Power Microcontroller, 2KB Flash, 128B RAM, USI for SPI/I²C
MSP430G2231	G2xx	16-bit Ultra-Low-Power Microcontroller, 2KB Flash, 128B RAM, 10-Bit SAR A/D, USI for SPI/I²C
MSP430G2102	G2xx	16-bit Ultra-Low-Power Microcontroller, 1KB Flash, 256B RAM, USI for SPI/I²C
MSP430G2202	G2xx	16-bit Ultra-Low-Power Microcontroller, 2KB Flash, 256B RAM, USI for SPI/I²C
MSP430G2302	G2xx	16-bit Ultra-Low-Power Microcontroller, 4KB Flash, 256B RAM, USI for SPI/I²C
MSP430G2402	G2xx	16-bit Ultra-Low-Power Microcontroller, 8KB Flash, 256B RAM, USI for SPI/I²C
MSP430G2112	G2xx	16-bit Ultra-Low-Power Microcontroller, 1KB Flash, 256B RAM, Comparator, USI for SPI/I²C
MSP430G2212	G2xx	16-bit Ultra-Low-Power Microcontroller, 2KB Flash, 256B RAM, Comparator, USI for SPI/I²C
MSP430G2312	G2xx	16-bit Ultra-Low-Power Microcontroller, 4KB Flash, 256B RAM, Comparator, USI for SPI/I²C
MSP430G2412	G2xx	16-bit Ultra-Low-Power Microcontroller, 8KB Flash, 256B RAM, Comparator, USI for SPI/I²C
MSP430G2132	G2xx	16-bit Ultra-Low-Power Microcontroller, 1KB Flash, 256B RAM, 10-Bit SAR A/D, USI for SPI/I²C
MSP430G2232	G2xx	16-bit Ultra-Low-Power Microcontroller, 2KB Flash, 256B RAM, 10-Bit SAR A/D, USI for SPI/I²C
MSP430G2332	G2xx	16-bit Ultra-Low-Power Microcontroller, 4KB Flash, 256B RAM, 10-Bit SAR A/D, USI for SPI/I²C
MSP430G2432	G2xx	16-bit Ultra-Low-Power Microcontroller, 8KB Flash, 256B RAM, 10-Bit SAR A/D, USI for SPI/I²C
MSP430G2152	G2xx	16-bit Ultra-Low-Power Microcontroller, 1KB Flash, 256B RAM, 10-Bit SAR A/D, Comparator, USI for SPI/I²C
MSP430G2252	G2xx	16-bit Ultra-Low-Power Microcontroller, 2KB Flash, 256B RAM, 10-Bit SAR A/D, Comparator, USI for SPI/I²C
MSP430G2352	G2xx	16-bit Ultra-Low-Power Microcontroller, 4KB Flash, 256B RAM, 10-Bit SAR A/D, Comparator, USI for SPI/I²C

Part Number	Family	Description
MSP430G2452	G2xx	16-bit Ultra-Low-Power Microcontroller, 8KB Flash, 256B RAM, 10-Bit SAR A/D, Comparator, USI for SPI/I²C
MSP430G2153	G2xx	16-bit Ultra-Low-Power Microcontroller, 1KB Flash, 256B RAM, 10-Bit SAR A/D, Comparator, USCI for I²C/SPI/UART
MSP430G2203	G2xx	16-bit Ultra-Low-Power Microcontroller, 2KB Flash, 256B RAM, Comparator, USCI for I²C/SPI/UART
MSP430G2313	G2xx	16-bit Ultra-Low-Power Microcontroller, 2KB Flash, 256B RAM, Comparator, USCI for I²C/SPI/UART
MSP430G2333	G2xx	16-bit Ultra-Low-Power Microcontroller, 2KB Flash, 256B RAM, 10-Bit SAR A/D, Comparator, USCI for I²C/SPI/UART
MSP430G2353	G2xx	16-bit Ultra-Low-Power Microcontroller, 2KB Flash, 256B RAM, 10-Bit SAR A/D, Comparator, USCI for I²C/SPI/UART
MSP430G2403	G2xx	16-bit Ultra-Low-Power Microcontroller, 8KB Flash, 512B RAM,, Comparator, USCI for I²C/SPI/UART
MSP430G2413	G2xx	16-bit Ultra-Low-Power Microcontroller, 8KB Flash, 512B RAM, Comparator, USCI for I²C/SPI/UART
MSP430G2433	G2xx	16-bit Ultra-Low-Power Microcontroller, 8KB Flash, 512B RAM, 10-Bit SAR A/D, Comparator, USCI for I²C/SPI/UART
MSP430G2453	G2xx	16-bit Ultra-Low-Power Microcontroller, 8KB Flash, 512B RAM, 10-Bit SAR A/D, Comparator, USCI for I²C/SPI/UART
MSP430G2513	G2xx	16-bit Ultra-Low-Power Microcontroller, 16KB Flash, 512B RAM, Comparator, USCI for I²C/SPI/UART
MSP430G2533	G2xx	16-bit Ultra-Low-Power Microcontroller, 16KB Flash, 512B RAM, 10-Bit SAR A/D, Comparator, USCI for I²C/SPI/UART
MSP430G2553	G2xx	16-bit Ultra-Low-Power Microcontroller, 16KB Flash, 512B RAM, 10-Bit SAR A/D, Comparator, USCI for I²C/SPI/UART

Figure 56 (MSP430 Model Feature Table): Comprehensive list of all models compatible with the MSP430G2ET LaunchPad for easy development. Taken directly from the datasheet for the LaunchPad.

These tables are also useful in the fact that they list the varying features between the processors for easy comparison for the following considerations we had.

8.3.2 Flash Memory, ROM, and RAM

How much we needed was hard to determine without the software being written. In general, more was deemed better with the only meaningful tradeoff being cost. For developing, we decided it was best if we overestimated how much we needed to avoid problems in not having enough memory or processing power. If we want to iterate on the design and decide to still use an MSP430, we could measure how much memory our programs use and decide on a better model accordingly.

8.3.3 GPIO

We already established that we needed at least 12 GPIO. There were some MSP430 that came with at least 10 GPIO. However, luckily for us, all of the DIP socket compatible models in the table had at least 24 pins.

8.3.4 ADC

A minor consideration was when we first considered the MSP430 for handling audio. To have a good quality analog-to-digital converter would be useful in that case. The F20xx series offers some 16-bit Sigma Delta options for better quality, but the 10-bit successive approximation register might have sufficed. It did not matter much since the audio processing would likely have needed to be done on a separate chip since the MSP430 was busy processing our switches. Utilizing the ADC also increases the power usage of the MSP430.

8.4 Programming Languages

The language we used was determined based on the controllers that we utilized in our implementation. There were some considerations regarding a couple languages in the beginning though in terms of hopes and preference. Of which, preference obviously influenced some of our decision making for controllers alongside the tradeoffs of the controllers.

8.4.1 Python

Python is an expensive language requiring more processing resources than others. It also lacks the resource management and speed provided by an alternative like C. Python was desirable for us despite its shortcomings because of its power and ease of use. Python has access to a wide array of open source AI libraries, so for augmenting our game engine with AI, Python already had access to the tools needed to facilitate actualization.

Since the CUDA on the Jetson Nano has a Python wrapper for its architecture, we discussed staying with Python as our primary high level language. This likelihood became the case. It is what we were partial to before and it is what we ended up using. We did have a bit of flexibility with our language choice though since there are several high level languages that are compatible with the CUDA.

8.4.2 Micropython

Another highly favored language during initial discussion was MicroPython - a Python based programming language designed for the limited resources of microcontrollers. MicroPython is a language that includes a small subset of the Python standard library. Using less resources might have allowed saving on unit production costs via cheaper controllers. If we were to go with the Pyboard, Micropython also has libraries specific to the Pyboard that may have proven useful. Some of these included drawing text, primitive shapes, pixel access methods, and touch screen methods. Since we were more familiar with Python and wanted to use the Jetson Nano, Python had the most appeal for our group. In future implementations, our code might be portable to MicroPython.

8.4.3 C

This language is one that everyone on the project is familiarized with and it has unparalleled convenience with access to the low level through memory manipulation. Transferring to and from the low level language also becomes easier with assembly directives. C also allows for especially lightweight and optimized software, allowing for more efficient battery usage, and faster runtimes. C is lower level than most of the high level coding languages, and has very direct interfacing for embedded programming. Many microcontrollers have their own development platform with their own C header files for C integration. The language allows for immediate implementation of I/O and tends to have much more transparent memory usage and hardware interfacing. C/C++ is also heavily supported by the CUDA Toolkit, having a comprehensive development environment for C/C++.

The primary drawback with C was its primitiveness. It had very few features and tools in the language itself, and most of the time involved lots of reinventing of logical systems. C would have been a very time-consuming language to code in, especially for the AI portion of the GameFrame. It ended up as the high-level language that we used on the MSP430.

8.4.4 Assembly

Although we initially thought we would probably have some amount of assembly for any of our choices, we did not end up using it. The embedded code was able to be sufficiently created in C. Assembly is very close to the hardware and could help with further optimizations of the project. The likelihood of using it at any point would be more for reimplementing the hardware rather than for the software.

8.5 Code Composer Studio

Code Composer Studio is an integrated development platform for TI's embedded processors such as the whole family of MSP430. The IDE already includes the C libraries necessary to program their microcontrollers. CCS also contains a bunch of tools for development and debugging purposes, and is optimized to run C/C++. We used this IDE for the MSP430 coding. There were only two IDE options anyways for the controller, so we just went with the proprietary one.

8.6 CUDA Tools

CUDA has a variety of compilers and tools for developing in different languages. Looking at these tools gave us a sense of which language we might want to develop in. The tools we were looking at were those that are officially recognized and suggested. The aforementioned tradeoffs under languages also apply here in our considerations. In the end we did not end up using any of the toolkits or wrappers, but PyCUDA would have been most likely since we used Python to code on our Jetson Nano.

8.7 Displays

Rather than a single LCD panel with the 64 different buttons we considered using 64 partitioned displays for each button instead. This led us to two primary options for the display - partitioned or unpartitioned.

8.7.1 Single LCD Panel

It proved easier to implement the partitioning of the segments on the software side, so a single panel was preferred. Also, due to the cost of the individual panel being considerably lower than buying 64 units of the various segmented options we preferred a single panel. Both of our inspiration sources utilized one LCD. The handheld and portable chess devices on the market also had an individual LCD screen albeit smaller. One screen also allowed for more versatility on the software side, especially when creating additional games. We still believe one screen was the right option, but due to availability of options we did not get one with quite the resolution, or rather physical size, that we wanted.

8.7.2 Separated Display Options

Electronic Specifications

Controller PCB	12VDC
Controller Board Capacity	1 - 16 Buttons
Recommended Power Supply	2VDC 2A
Average Current	12VDC 0.8A
Compliance	RoHS Compliant

Figure 57 (Data Excerpt for SuzoHapp LCD Button): Though it can operate up to 16 buttons, the concern stems from needing 0.8A for only a fraction of our total buttons.

Aside from complicating the software and hardware design, especially at the interfacing level, quality panels for segmentation boasted a significant cost increase due to needing 64 of them. The prices of the particular LCD buttons provided were unavailable unless contacted, so an accurate price estimate of these were not known. SuzoHapp as a manufacturer, while providing quality parts and giving good ideas for components that might fit in our device, tended to have costly products. Using 64 of these drawing .8 A each was quite worrisome for portable device design too even if they were stepped down to 2V DC. A positive would have been that none of the housing or frame space between buttons would have been wasted pixels on screen. The cost of repair and maintenance would be better should one of these fail compared with our 15" LCD screen.

Figure 58 (Sample Segmented Options): Two pictures to help depict how the separated displays might look. The left is the aforementioned SuzoHapp product. On the right is one produced by LCD-Keys (I/O Universal Technologies).

8.7.3 OLED

Some sources suggest OLED could have been a cheaper alternative. Investigation on the matter led to seeing roughly the same prices as LCD. OLED had the ability to be more power efficient than LCD, but it came at the cost of brightness, thus reducing the image quality. The LCD we found also ended up being very power efficient. It should also be noted that OLED had the potential to be incredibly thin. Being so thin would have been an advantage towards the weight requirements. The screen we got however managed to be a portable design that was thin and surprisingly very lightweight.

8.8 Housing

Not many options came to us in regards to housing our production. We were fortunate to be in an age of 3D-printing, so it seemed an obvious choice for us. Still, we wanted to leave room for other considerations should they arise.

8.8.1 None

We obviously had no desire to create a device without housing. We ended up having little time to iterate on housing, since other aspects of the project were deemed a priority and dimensions were not finalized until very late. The cost is as minimal as it gets and we did end up using two acrylic panels to keep our screen and grid together.

8.8.2 Crafted housing

Similar to buying acrylic for mounting the grid we had an idea to fashion our own housing out of some plastic (pre-cut or cut ourselves) and screw them together ourselves. This was cheaper, but also ended up looking less professional than a 3D-printed alternative. We ended up using this approach to some extent as a sort of way to layer our device in an acrylic shelf/sheet and metal standoff combo.

8.8.3 Plastic Enclosures

Some companies, like Polycase, sell premade plastic housing. These can be quite affordable, but they are less customizable. Instead of using these, it was thought to be better to just 3D print a housing. With our lack of time to develop housing at the end, these might have been a good solution depending on their shipping time.

Figure 59 (Sample Polycase Enclosure): The plastic enclosures feel best described by a picture.

8.8.4 3D-printed housing

3D-printers are able to create good prototype housing while also being reasonably accessible. The monetary cost was reasonable, but prototyping and mistakes definitely cost more time. CAD knowledge was required to create something with this method, but this was seen as a boon for the project, since this kind of experience is nice on a resume. The program that we will be using to develop our models for 3D printing is called Onshape. Since one of our members already has

experience using this software, it was easier to use this instead of trying to learn new software. What is good about Onshape is that all of its CAD services are done through cloud-based computing, which means you do not need an exceptionally powerful computer to run Onshape. All you need is a web browser. Onshape also has collaboration features built in such as link sharing with others so they can look and adjust the model.

We only had access to a 200x200x250mm (~7.87x7.87x9.84inch) on short notice. So we ended up needing to design our housing to fit in this area and subsequently melded together.

One alternative we considered was to use a CNC router to develop a housing for our device out of acrylic or other solid plastic. This approach would have required some extra sanding for the rough edges, but also would have required some trial and error since we did not quite have time to figure out acrylic. That is, we kept burning acrylic when trying it for another part of the project, so the spindle speed was probably too fast.

3D Printer Filaments

3D printers come with a variety of usable materials. The advantages and disadvantages are accordingly diverse. The materials focused on for the beginning of the project are those that are more suitable for prototyping. Some materials that might be considered higher end or more industrial are not that much more expensive. Overall, we used the filament that we had access to: PLA, ABS, and PETG.

PLA

PLA is considered one of the easiest to print with due to its low printing temperature requirement and its low flexibility. PLA is susceptible to distortion if left in heat. For example, leaving a portable game device in the backseat of a car, especially in Florida, might cause warping if the device were made of PLA. It is also a bit more brittle as a material compared to other options due to being inflexible. There are a variety of colors and even some options of filled materials for certain textures as a superfluous benefit. The filled materials include wood-filled or metal-filled filaments to name a few. One can also acquire PLA with material that glows-in-the-dark as another option. Most of the properties of these diluted materials were not beneficial for us and just a tad worse than the non-filled PLA alternative. PLA seemed rather appealing as a prototyping filament though, especially as it was one of the materials we already had access to a 3D printer for.

ABS

ABS demands a tad more difficulty when it comes to printing. The benefit is that it is more durable and more heat resistant. ABS was seen as a tad superior to PLA. It has a tendency towards shrinkage or warping while printing. There are also toxic fumes to be wary of while printing. These are downsides during production however, and would not have affected our device negatively. With a similar cost, ABS might have been better than PLA for us, but not for prototyping. While first building, a material that was prone to errors and might need reprints would be much worse and would have cost more while prototyping.

PETG
PET or its related materials PETG and PETT lie somewhere in between the benefits of PLA and ABS. It is a bit harder to print well than PLA, but easier than ABS. It is less brittle than PLA due to its flexibility and durability. The downsides include being susceptible to moisture and having a surface that can scratch easily. PETG has a similar cost to the aforementioned two filament types, but it also is a material that we had access to a 3D printer already set up for.

Additionally, it should be noted that we made our custom buttons from the CNC router with this material. It was less brittle than other options and never cracked on us. However, it did show some trouble at times with melting around the bits as we milled it.

TPE
Not much is necessary to say for TPE in regards to what our plans were. TPE is highly flexible and has properties akin to rubber. For our case, we needed a more rigid material. No components we had planned needed the features this material offers, but it's flexibility and durability are incredibly high and were good points to keep in mind.

Nylon
Nylon beats a lot of other materials in flexibility, durability and strength. The cost of it sits above the rest of the materials as well. This was not a material that we wanted to use because of its flexible nature and cost. Also, it must be kept dry like PETG.

PC
Polycarbonate is as durable if not more durable than Nylon. The main difference comes in the fact that it is less flexible. It requires an incredibly high temperature to print. It resists damage from both impact and heat quite well. It seemed a bit more durable than necessary, but it also was not that much more expensive than other options. Its heat resistance was not needed, but if temperature ever became an issue, we know which material we would have used.

Biodegradable
Though these are worthy of at least a mention because of being environmentally friendly, they tend to be more expensive and have worse qualities. The battery and other electronic components inside our device would need to be properly recycled and disposed of anyways, so having a biodegradable housing was moot.

Conductive
It is quite an interesting option to consider conductive carbon combined with another filament on this list. Though it is limited to low-voltage applications, this was something to consider elsewhere in our design. Obviously, we did not particularly want conductive filament to be used for our housing.

At one point we actually thought of using conductive filament to print our wire grid set up. Remembering it now as we update this document, perhaps it would have been the better solution - a middle ground between the PCB implementation and copper wire grid. We could have potentially reached more of our goals with this.

8.9 Batteries

One of the most important factors for making our device portable was the battery. Because the device is intended to be portable we leaned heavily towards recharging as an option. Disposable batteries also had the benefit of allowing the customer more playtime on the go without the need to recharge. That said, despite disposable batteries being better for monetary design, it was worse for the consumer's pockets and for the environment. That leads us to the most common materials for rechargeable batteries.

8.9.1 Alkaline

Alkaline batteries boast a decent energy density and are relatively cheap. If not charged properly these are more susceptible to exploding. They are not as lightweight as lithium options. Alkaline batteries tend to have higher internal resistance as well.

8.9.2 Lead-Acid, Nickel-Cadmium

Of the battery types both Lead-Acid and Nickel-Cadmium are older technology. They have a cheaper price point, but end up worse in energy density. Lead-Acid is a good consideration for being able to handle high currents. High current supply is good for a car, for example. Our device did not need this, so this material was less desirable.

8.9.3 Nickel-Metal Hydride

Nickel-Metal Hydride is virtually a more modern version of Nickel-Cadmium batteries. Nickel-Metal Hydride have better capacity, retain the ability to keep fully charging, and are more environmentally friendly than Nickel-Cadmium. The tradeoff is that they are more expensive.

8.9.4 Lithium-Ion

Lithium-ion batteries are increasingly popular for portable devices, and for good reason. This material boasts a high energy density as one of its biggest advantages. It has a lower self-discharge rate while not in use, especially when compared to its Nickel-x counterparts. The disadvantages of this technology is that they require some extra protection. They must have protection from being overcharged or exceedingly discharged and also must maintain safe current limits. Some lithium-ion batteries include this protection themselves, but this disadvantage is necessary to consider. Having chosen a battery of this material, we also went with a choice that already had protection built in. The last, and probably obvious, disadvantage that one would assume from its advantages is the cost. Lithium-ion batteries were more costly than other materials, but the cost was worth it to easily attain our desired battery life.

8.9.5 Lithium-Polymer

These types of batteries are more of a subset of Lithium-ion. They have many of the advantages and disadvantages of Lithium-Ion batteries. These end up being a slightly safer technology with even lower self-discharge. They have a lower capacity overall, but they do have slightly better energy density.

8.10 Switch Debounce

There are several options for dealing with the error that comes from switch bouncing. It ended up not being entirely needed, but it is something we designed for and were able to utilize in other ways.

8.10.1 Hardware solutions

NAND
This hardware implementation leads to the best non-IC result. It uses two NAND gates together to implement an S-R flip flop. The self-feedback of the circuit results in preserving the output while the contacts cause bounce. The trade off of this is that the rising edge of the transition is curved. It also requires the most components to create.

IC
There are dedicated integrated circuits for reducing or eliminating switch bounce. The debouncing ICs are a bit pricier of a solution but result in the overall best debounce on hardware. We did not need to use one of these, but it was worth considering.

RC
Using a simple RC circuit can get rid of the error from bouncing. Using a simple resistor and capacitor this is the most heuristic solution, while also being the lowest quality of hardware options. Additionally a diode can be used to speed up how fast the capacitor would charge and gives the option to speed up debouncing. We ended up leaving unpopulated space on our board for these as an option, but did not need it.

8.10.2 Software solution

Rather than using any of the hardware solutions, the cheapest and easiest solution comes at the cost of memory and performance, though slight. It relies on implementing it in software. As we are running our loop on the hardware microcontroller, we are sampling our inputs from the switches. As we sample inputs, we can make sure that a switch input only changes a value if it consistently reads the same over a number of iterations of the controller loop. That is, we can keep a counter and only update the actual value if the counter reaches a threshold. We did end up using this as a way of controlling how long or short we wanted a button press to be rather than as a way to clear bounce error. It was implemented on the Jetson Nano side.

8.11 PCB Development Software

As our project developed, we looked at several softwares to create schematics and layouts to actualize our PCB. Listed here are options we looked at and researched features.We landed on Eagle in the end.

8.11.1 SnapEDA

The first thing to bring up is not software for PCB design. SnapEDA is a library of footprints and CAD symbols that other PCB development programs can use. These are premade and as an article from the official SnapEDA blog says, “this allows them to focus on innovation and design optimization rather than the more mundane and tedious aspects of the design process.” (natasha, 2015).

SnapEDA definitely ended up being very useful. We had to customize some of the footprints anyways, but being able to just add the parts with an accurate silkscreen profile and the right size through holes was invariably convenient.

8.11.2 Eagle

One of the two most popular PCB design softwares used at our university is Eagle. Members of our group had experience with Eagle from junior design. Typically Eagle is not free, but it does have licenses available for educational use for reduced cost. There is robust library support for Eagle (through sites such as SnapEDA). Rich library support was definitely welcome. Though Eagle is what we used and it is full of fantastic features, one gripe is that it is definitely one of the less intuitive designs for a UI that we’ve seen. Swapping between tabs, selecting components in the menus but not the visualization, typing in a script command or two, and going through a long right-click menu to simply resize all traces for example is a bit more arduous than it need be. Also, why is 6 mil the default trace width?

8.11.3 KiCAD

This is the second most popular software at UCF from what we could gather. KiCAD is open source and thus always free. Similar to Eagle it had many options from its libraries. The idea of any and all functionality of KiCAD being free with no limitations definitely sounds more appealing looking back. We had already sunk a bunch of time into Eagle at that point though.

8.11.4 Ultiboard

This program from National Instruments is part of the Multisim suite of products that we were familiar with, and thus it was compatible with them. One of its touted features is its ability to do design rule checking in real time. A big drawback was that it was not supported by SnapEDA. There was an official response from the SnapEDA team which stated this was on their roadmap (Team SnapEDA, 2016). Nothing changed during the duration of our project though.

8.12 Voltage Regulators

Regulating voltages throughout our circuit was a must. The main regulators looked at were the LT3694 and the LM2576. We used the LM2576, but more details on the two can be seen in the Hardware design section of this document (section 5.4). More generally we contrasted linear voltage regulators and DC-DC converters here.

8.12.1 Linear voltage regulators

The most prominent thing to consider for linear voltage regulators is that they can only act as buck converters. They are also known for being power inefficient with most of this inefficiency due to heat.This inefficiency was undesirable due to heat generation and a battery life goal. Linear regulators are cheap, resilient to ripple, have a fast response and are not susceptible to switching noise though, like switching regulators.

8.12.2 Switching voltage regulator

These were the class of regulators we chose from. They have higher efficiency and less heat generation compared to linear regulators. One of the downsides of having increased noise was not as important for our circuit, especially since we had some bypass capacitors on the voltage lines to smooth things out.

8.13 Cooling

Among the many parts researched, one of the biggest problems for any electronics is heat generated by components. One of these heat-producing parts is the microcontroller due to its processor. There were some microcontrollers that came with some form of cooling solution already pre-installed, such as the Jetson Nano with its pre-installed heatsink. It is important to have our components properly cooled for them to run efficiently. It is also necessary to prevent damage to our components and reduce the risk of a system failure as a result of heat. There are however, there are a few main cooling options to consider:

8.13.1 Air cooling

Air cooling utilizes the ambient air to carry heat away from the source and out of the system. It does this simply by using a fan or even multiple fans to circulate air. Ideally, proper air cooling solutions involve pulling in cool, fresh air into the system and passing that air through the heat source, which will transfer some of the heat to the air, and another fan will exhaust the warm air out.

The main benefits to air cooling is that they are relatively inexpensive and do not require some form of liquid solution to carry the heat away. This results in less parts to utilize air cooling, which also leads to less points of failure. If the fan breaks, it is easy to replace instead of having to replace the whole cooler if there is a leakage. It also has greater flexibility due to simplicity in nature air cooling is.

However, air cooling isn’t without its limitations. Air has a low specific heat, which means that it has less capacity to absorb and transfer heat. In typical scenarios, air cooling also requires an extra element such as a heat sink or a heat pipe to initially absorb the heat. Air cooling can also be completely ineffective if air flow is not properly planned ahead of time. If done incorrectly, it’s possible to recirculate heated air inside the system. Without proper circulation, heat is just transferred back into the source.

8.13.2 Heatsinks

A heatsink is a heat transfer device that acts like a middleman to transfer heat from the source to a fluid medium, such as air or water. It utilizes a highly thermal conductive material to absorb heat from a heat-generating source, which can be either passively or actively dispersed. Passive disbursement relies on the physics of air or water to transfer heat away. How this works is when heatsinks absorb heat, the surrounding air touching the heatsink will also have some of that heat transferred to it. This will cause the air to warm up, and air circulation is passively generated by warm air rising. As warm air rises, it carries heat away from the source, which then causes cooler fresh air to take its place to absorb heat from the heatsink again. The same principle can be applied to a water medium where water rises and passively circulates cool water to the source.

However, sometimes the heat source can generate too much heat and passive air circulation is not enough to properly cool a component like a CPU. If passively circulation is not fast enough to transfer heat away, then an active heat disbursement is needed. In a heatsink using air as a fluid medium, a fan would be attached to the heatsink to actively cycle cool air in and warm air out. In a heatsink using water, you would need to have a pump and radiator combo to pump water away from the source which is then cooled by the radiator.

There are many metals with different thermal conductivity, but the two main metals used in heatsinks are aluminum and copper. Aluminum is the cheaper of the two since it is less thermally conductive than copper. Copper is almost three times denser than aluminum, which results in a higher cost for its better performance and higher thermal conductivity. Selecting which material to use will depend on how much heat needs to be transferred.

8.13.3 Heat Pipes

A heat pipe is a heat transfer device that is very similar to a heatsink. Instead, a heat pipe consists of an envelope with a fluid and a wick structure to carry heat away from the source. Heat vaporizes the liquid inside the wick, which then carries the vapor with its heat towards the cooler section of the pipe away from the heat source. The cooling area contains a condenser where the vaptor loses its heat by condensing back to liquid form. The liquid then returns to the heat source through the wick and the process is cycled.

Some of the main benefits include a way higher thermal conductivity than air, can potentially be passive, low cost, and resistant to shock. Since it can be passive, implementing heat pipes as our cooling solution adds the benefit of a quiet system. If we so desire, you may also install a fan to combine air cooling with the heat pipes for an even better cooling solution. However, this does add back noise into the system due to the fan.

One of the main drawbacks is that heat pipes are not suitable devices that demand high power usage. The passive cooling on the heat pipe is not enough to cool most high power demands. However, this drawback is mostly negligible because this project aims to have low power usage. It may be a bigger consideration depending on what parts we pick. For example, the jetson nano might need a different cooling solution since it does generate a lot of heat.

Based on our options, it is likely that we will lean towards a solution that utilizes air cooling. Water cooling, while more efficient, is definitely not necessary for our potential needs. Water cooling will add a lot more unnecessary bulk to our device. Since we are leaning towards using the Jetson nano, it helps that an aluminum heatsink is already pre-installed onto the board, which means copper will not be considered. Depending on how much heat is generated by the Jetson nano, one fan might be installed on top of the pre-installed heatsink. Luckily, there are fans already specifically designed for the Jetson nano with mounting already set up on the board. If air cooling is necessary, the design of the enclosure must be changed to include vents for warm air to leave the device.

8.14 Algorithms for the Computer Player

To play against the computer, a type of AI or bot is used. Throughout the paper, the terms AI, bot, and computer have been used interchangeably to address the algorithm that will play against the human player in 1-player mode, however it was not necessarily determined that an AI is the best solution. The game of chess for instance has more possible games than could ever be played, or stored in any realistic hard drive, let alone the small factor drive that will be not larger than 256GB, likely much less. This being the case, utilizing something like an artificial intelligence would have been the best route to take in order to solve a game of chess. On the other hand, a game of tic-tac-toe has a relatively small number of total games, and is not especially complex in the decision making process. For the computer, a combination of AI and algorithmic solving, such as backtracking, was used to play against the human player. In the end, due to time constraints, backtracking in a minimax type of application wound up being the implemented form of "AI" that the player plays against in 1-Player mode.

8.14.1 Backtracking

A useful strategy to solve problems that have lots of discrete moves is an algorithm called backtracking. This process essentially sets up many instances of simulated moves, where the algorithm "explores" lots of different paths, then once it gets to a point that is undesirable, backs up a bit until it hits a place where it can make a change in decision and keep going. A good example of this is something like a maze solver, where the backtracking algorithm may choose to turn right at every intersection. Once it hits a dead end, it backs up to the last right, then makes a left. It then continues making rights until it hits another dead end, in which case it repeats the same problem. If no additional right exists, it backs up another step. Ultimately, this should result in a solution, but this is not always the case

In backtracking, there exist numerous gridlock states. In the maze solver example from earlier, if the end is surrounded by an island, the backtracking algorithm may never be able to move there, as it could back up beyond the place where turning would get it to the end. With game solving, this is a concern, as sometimes the optimal solution to a game logically may not be accessible, and the algorithm could lose out on many more optimal solutions. Additionally, it may take a substantial length of time to run through every solution to a game, or at least enough to make a good next move. Backtracking needs to have a certain type of solution set for it to be able to run well.

For games like tic-tac-toe, 4-in-a-row, and potentially checkers, backtracking is probably still the better bet. As mentioned earlier, AI is very resource intensive to get up and running, however a backtracking algorithm can be functionally complete in very little time. These algorithms simply need to be told what the rules of the game are, how to play, and a basic strategy that will allow them to test through the game, then they are ready to go. This is good for a game that has few rules and limited potential for cases where the algorithm may not be capable of arriving at a favorable end state. Additionally, the Jetson Nano will make it very possible to optimize these algorithms, as numerous different instances of paths can be explored in parallel, resulting in the desirable 1-second time to move. Though it was not the optimal choice, this wound up being the primary method for all player vs computer solving techniques.

8.14.2 Artificial Intelligence

Artificial intelligence is one of the primary uses of the jetson nano board. This being the case, AI may seem like the all around best option for all solutions of each game. AI however has the downsides of being an incomplete system, and does not always choose the absolute optimal solution. AI is further limited by the fact that lots of useful input needs to be created and input into the training models for the AI to gain any ability to function. This is timely and may require lots of time and resources to train. A system utilizing AI is also prone to errors, as an AI tries to get to its goal given only exactly what the rules are it has to abide by, and nothing more. This means for instance if an AI is incentivised not to lose, it may simply choose not to play at all, making it impossible to lose.

A key upside of artificial intelligence is the modifiability of difficulty that the algorithm plays the game on, and the lack of need to fully program in the solution to the game. The primary way that an AI is trained is essentially it is told how to do something, and what its goal is, then seeded with some randomness to do that, and rewarded once it gets closer to a goal. For instance, if the goal of an AI is to drive from point A to point B as quickly as possible, it may be told how to spin the wheels of its body, use its arms, and change the speeds of each component. Then as it gets closer to point B it may start receiving rewards, or sometimes only once the goal is achieved does it get a reward. The AI then makes a map of what types of things resulted in getting to a reward through a process called convolution. This means that the AI abstracts each thing it does into “ideas” and puts the best ideas at the highest priority and the ones that worked the least at the lowest. The AI will then continue trying to solve the problem by raising and lowering ideas in its priority convolutional network (called a convolutional neural network due to its resemblance to neurons).

For the sake of this project, this is a useful feature to have, as not every aspect of a game of chess for instance will need to be programmed in. It is known to chess players such things as moving a king to a corner can be a good strategy for success, having a castle at the bottom or 2nd to bottom row is beneficial, certain pieces are worth more than others, pawns are typically worth more as just sacrificial pieces, certain opening moves result in more favorable outcomes, etc. but this is quite cumbersome to program into a computer, especially with the fact that many of these cases have numerous caveats and exceptions. AI allows for the programmers to not know much about the strategy of chess and simply reward the AI when it is doing well. Unfortunately, due to time constraints, AI neural networks were not the implemented choice for player vs. computer mode.

8.15 Shift Registers

There was not much to consider when it came to shift register ICs. The important factor was how many bits the register has. The 74HC595 is an 8-bit shift register that worked for us. In case we would like additional bits to work with a 16-bit register in the form of the 74LS674 was researched. The extra bits would have allowed expansion of hardware design without loading more onto our controller's other I/O pins. Additionally we could have daisy-chained two 74HC595 if we wanted. The biggest reason to go above 8-bits would be if we wanted more contact points as discrete values.

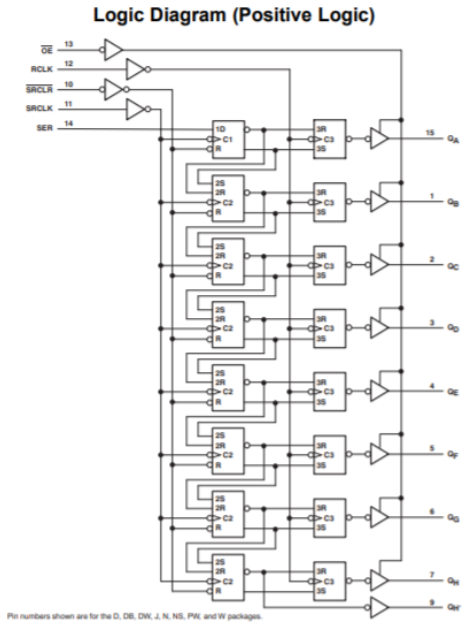


Figure 60 (74HC595 Logic Diagram): Depicts the logic gate architecture of the 8-bit register. Taken from the part's datasheet.

9. Budget and Financing

9.1 Cables and Connectors

Item	*Cost*	*Description*
Breakout to USB-C connector	$8	Supplies power from a voltage regulator to Jetson Nano

2x RS232 breakout connector + Female-female RS232 connector	$16	Connects MSP430 and Jetson Nano for serial communication
Breakout to USB-A connector	$8/2	Supplies power from a voltage regulator to LCD
Dupont Cables	$14	Pin cables for connecting to row/column. Also used when debugging controllers
Battery Connector	$9.50	Charge and power device at the same time
Mini-hdmi to HDMI cable	$9	Used to communicate display information from Nano to LCD
USB-RS232	$11	Used to debug MSP430 serial communication before connecting controllers.

Table 16 (Cable Costs): Various cables and connectors needed for the hardware.

9.2 Button Fabrication

Item	*Cost*	*Description*
Rubber actuators	$60/400 ($3.6/20 non-bulk)	Needed four per button or 256 minimum. Bought spares.
Super glue	$3	Frosted test acrylic, did not use in final product
Epoxy	$18	Adhere buttons and various other gluing during project
Drill bits for cnc milling	$32/20	Broke a lot while experimenting to get the settings right.
4”x6” PETG Sheets	$28/20	Went through about 20 while experimenting. With the right settings, we can produce 64 buttons with 10-12 sheets.

Plexiglass tiles	$50/80	Ended up not using these high quality 1”x1” tiles. Vertical height was just too tall, sadly.

Table 17 (Button Costs): Various materials and components for making custom buttons.

9.3 PCB Etching

Item	*Cost*	*Description*
Ferric Chloride	$13	Used to chemically etch away copper.
4”x6” Copper Plate	$18/10	Only one was needed for etching. A second one was used to test cutting through with the CNC router.

Table 18 (PCB Fab Costs): Material and chemical used to make PCB.

9.4 Misc. Hardware and Electrical

Item	*Cost*	*Description*
Misc. RLC/Diode/heat sink/cheaper ICs/etc	~$20-$30	PCB electrical components
Misc. mechanical hardware	~$30	Screws, washers, etc.
Acrylic Sheets	$13	Three sheets, used two.
“Artistic” Copper Wire	$9	Thinnest copper wire we could find to minimize screen obfuscation.
Fan	$4	To help airflow away from 5V regulators.
Liquid tape	$8	Used to insulate wire grid crossovers or other insulation during production.
Vector boards for alignment	$18/6	Used to have a more accurate grid alignment on contact

		nails.

Table 19 (Misc. Costs): Small, cheap or auxiliary components or tools needed for the project.

9.5 Main components and Housing

Item	*Cost*	*Description*
NVIDIA Jetson Nano 2GB Developer Kit	$60	Software controller
MSP430g2553	$15	Hardware controller
15” LCD (with cables)	$136	Expensive but convenient, low power, and lightweight.
Battery	$38	66.6 Wh battery
DC 5V cooling fans	$18	2x pack. Not utilized
3D printing material	$24	Filament spool for printing housing

Table 20 (Main Costs): Bulk of our costs/budget for main components.

9.6 Unit Cost

Item	*Cost*	*Description*
NVIDIA Jetson Nano 2GB Developer Kit	$60	Software controller
MSP430g2553	$15	Hardware controller
15” LCD (with cables)	$136	Expensive but convenient, low power, and lightweight.
Battery	$38	66.6 Wh battery
Liquid tape	$2	Used to insulate wire grid crossovers or other insulation during production. Entire bottle was not used.
Fan	$4	To help airflow away from 5V regulators.

“Artistic” Copper Wire	$3	Thinnest copper wire we could find to minimize screen obfuscation. Entire spool was not used.
Acrylic Sheets	$8.7	Used two.
Misc. mechanical hardware	$15	Screws, washers, etc.
Misc. RLC/Diode/heat sink/cheaper ICs/etc	$20	PCB electrical components
4”x6” Copper Plate	$1.8	Only one was needed for etching. A second one was used to test cutting through with the CNC router.
Ferric Chloride	$3.25	Used to chemically etch away copper. Entire bottle was not used.
4”x6” PETG Sheets	$16.8	Went through about 20 while experimenting. With the right settings, we can produce 64 buttons with 10-12 sheets.
Drill bits for cnc milling	$1.6	Broke a lot while experimenting to get the settings right.
Epoxy	$9	To adhere buttons and various other gluing needs during the project. Used about half.
Rubber actuators	$38.4	Needed four per button or 256 minimum. Bought spares.
Breakout to USB-C connector	$8	Supplies power from a voltage regulator to Jetson Nano
2x RS232 breakout connector + Female-female RS232 connector	$16	Connects MSP430 and Jetson Nano for serial communication
Breakout to USB-A connector	$4	Supplies power from a voltage regulator to LCD

Dupont Cables	$7	Pin cables for connecting to row/column. Also used when debugging controllers
Battery Connector	$9.50	Charge and power device at the same time
Mini-hdmi to HDMI cable	$9	Used to communicate display information from Nano to LCD
Total	$426.05	

Table 21 (Unit Cost): Cost of producing one GameFrame unit. Anything that could not be exact was conservatively overestimated.

10. Initial Milestones

Milestone	*Milestone Type*	*Due Date*	*Details*
Application can run a game	Software	09/05/21	The software application should have the basic structure of a game (chess) set up, and logic/rules to the game implemented and interactable. Ex: bishops move diagonally, pieces are captured on being attacked, captured king wins the game
Application can play against user	Software	09/12/21	Given a working application, the AI should be able to play a game of chess against the user. It is responsible to both make a decision as to what piece to move, as well as send the game engine a signal to move the piece. It should stop processing moves on capture of a king, draw, or reset.
Functional input/output	Hardware	10/10/21	The microcontroller should be able to read user input from buttons on a 64-panel grid. Input should be sent into the microcontroller for usage in the game engine.
Software to hardware integration	Hardware -Software	10/19/21	The hardware input from the completed player board should fully interact with all components of the application. No computer aid should be required to change game-modes, reset, power on/off, or enable/disable AI or two-player modes.
Fully battery powered	Hardware	10/24/21	The entire game board should fully run off of the battery pack. No external power input should be required, and battery life must last at least 2 hours (as per the requirements).

Physical board layout and encasing	Hardware	11/07/21	The playing board on which the users will interact with the game and AI should be laid out in such a fashion that the corresponding game pieces are in the position they correspond to in the game engine. Power switches, reset buttons, and menu buttons should also be attached in their permanent location.
Fully standalone	Hardware -Software	11/21/21	The game board should not need any aid from wired or wireless computers, internet, or power cords, and it should be able to be transported freely without issues in stability. Ex: If the board is tilted while mid-game, this should not affect performance.
Working Product	Hardware -Software	11/30/21	Game board on startup should boot to an interface which allows the user to start a game, choose game modes, and play a game. Menu should be accessible at the end of a game, and when the menu button is pushed. Board should recover to a usable state after power drain.

Table 22 (Milestones): A general outline of our project development milestones throughout the two coming semesters

Appendices

Works Cited

Not directly referenced

Team Member. (2020, October 28). *How to Castle in Chess?* Chess.com. https://www.chess.com/article/view/how-to-castle-in-chess.

How the Chess Pieces Move: The Definitive Guide to Learning Chess Fast. iChess.net. (2021, July 14). https://www.ichess.net/blog/chess-pieces-moves/.

Hasbro. (2003). *Checkers Instructions*. f.g.bradley's. https://www.fgbradleys.com/rules/Checkers.pdf.

Mott, V. (n.d.). *Introduction to Chemistry*. Lumen. https://courses.lumenlearning.com/introchem/chapter/other-rechargeable-batteries/.

3D Printer Filament – The Ultimate Guide. All3DP. (2021, July 19). https://all3dp.com/1/3d-printer-filament-types-3d-printing-3d-filament/.

Language Solutions. NVIDIA Developer. (2021, April 21). https://developer.nvidia.com/language-solutions.

What is Switch Bouncing and How to prevent it using Debounce Circuit. Circuit Digest. (2019, June 7). https://circuitdigest.com/electronic-circuits/what-is-switch-bouncing-and-how-to-prevent-it-using-debounce-circuit.

Arduino Playground - SoftwareDebounce. (n.d.). https://playground.arduino.cc/Learning/SoftwareDebounce/.

Knight, D. (2021, May 19). Introduction to Linear Voltage Regulators. DigiKey Electronics - Electronic Components Distributor. https://www.digikey.com/en/maker/blogs/introduction-to-linear-voltage-regulators.

Person. (n.d.). *Which ADC Architecture Is Right for Your Application?* Which ADC Architecture Is Right for Your Application? | Analog Devices. https://www.analog.com/en/analog-dialogue/articles/the-right-adc-architecture.html.

Air vs Liquid: Advancements in Thermal Management. Boyd Corporation. (n.d.). https://www.boydcorp.com/resources/resource-center/technical-papers/air-vs-liquid-cooling-advancements-in-thermal-management.html.

Everything You Need to Know About Heat Pipes. Advance Cooling Technologies. (n.d.). https://www.1-act.com/innovations/heat-pipes/#benefits.

Encyclopædia Britannica, inc. (n.d.). *Checkers*. Encyclopædia Britannica. https://www.britannica.com/topic/checkers.

Encyclopædia Britannica, inc. (n.d.). *Pong*. Encyclopædia Britannica. https://www.britannica.com/topic/Pong.

LCD-Keys. www.lcd-keys.com. (n.d.). https://www.lcd-keys.com/.

Buttons – lcd pushbuttons. SUZOHAPP OEM. (n.d.). https://oem.suzohapp.com/products/buttons-decks-and-dps/buttons/lcd-pushbuttons/.

Dc-96f. DC-96F Heavy Duty Plastic Electronics Enclosures | DC Series. (n.d.). https://www.polycase.com/dc-96f.

Directly referenced

natasha. (2015, March 24). *What is SnapEDA? A Primer for Non-Engineers*. SnapEDA Blog. https://blog.snapeda.com/2015/03/21/what-is-snapeda-a-primer-for-non-technical-folks/.

Team SnapEDA. *Why no NI MultiSim/Ultiboard ?* SnapEDA. (2016, December). https://www.snapeda.com/questions/question/why-no-ni-multisimultiboard/.

Other Links

Switch Trace Image Source:
https://electronics.stackexchange.com/questions/281921/pcb-touch-button

Example of jubeat from a top-down view:
https://youtu.be/zfmbBTBlP4Q

Video of a DIY jubeat:
https://www.youtube.com/watch?v=bemd8mn8H4E&ab_channel=MaikeeMaous

Permissions

TI Datasheet Legal Disclaimer:

IMPORTANT NOTICE AND DISCLAIMER

TI PROVIDES TECHNICAL AND RELIABILITY DATA (INCLUDING DATASHEETS), DESIGN RESOURCES (INCLUDING REFERENCE DESIGNS), APPLICATION OR OTHER DESIGN ADVICE, WEB TOOLS, SAFETY INFORMATION, AND OTHER RESOURCES "AS IS" AND WITH ALL FAULTS, AND DISCLAIMS ALL WARRANTIES, EXPRESS AND IMPLIED, INCLUDING WITHOUT LIMITATION ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT OF THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

These resources are intended for skilled developers designing with TI products. You are solely responsible for (1) selecting the appropriate TI products for your application, (2) designing, validating and testing your application, and (3) ensuring your application meets applicable standards, and any other safety, security, or other requirements. These resources are subject to change without notice. TI grants you permission to use these resources only for development of an application that uses the TI products described in the resource. Other reproduction and display of these resources is prohibited. No license is granted to any other TI intellectual property right or to any third party intellectual property right. TI disclaims responsibility for, and you will fully indemnify TI and its representatives against, any claims, damages, costs, losses, and liabilities arising out of your use of these resources.

TI's products are provided subject to TI's Terms of Sale (www.ti.com/legal/termsofsale.html) or other applicable terms available either on ti.com or provided in conjunction with such TI products. TI's provision of these resources does not expand or otherwise alter TI's applicable warranties or warranty disclaimers for TI products.

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265

Datasheets and Reference Documentation

ICs or Controllers

https://www.ckswitches.com/media/1471/pts645.pdf
https://omronfs.omron.com/en_US/ecb/products/pdf/en-ss.pdf
https://www.pjrc.com/teensy/
https://www.ti.com/lit/an/slaa405b/slaa405b.pdf?ts=1626850739943
https://store.micropython.org/pyb-features
https://developer.nvidia.com/embedded/jetson-nano
https://oem.suzohapp.com/wp-content/uploads/2020/06/Snapshot-LCD-Button_Flyer_EN.pdf
https://www.ti.com/lit/ds/symlink/sn74hc595.pdf?ts=1627888610317
https://www.ti.com/lit/ds/symlink/sn74ls674.pdf?ts=1627957879522&ref_url=https%253A%252F%252Fwww.google.com%252F
https://www.analog.com/media/en/technical-documentation/data-sheets/36941fb.pdf
https://www.ti.com/lit/ds/symlink/msp430g2553.pdf?ts=1627963592296&ref_url=https%253A%252F%252Fwww.ti.com%252Fproduct%252FMSP430G2553
https://www.ti.com/lit/ds/symlink/lm2576.pdf?ts=1638658789869&ref_url=https%253A%252F%252Fwww.google.com%252F

Standards

https://ieeexplore.ieee.org/document/6204026
"IEEE Standard for System and Software Verification and Validation," in *IEEE Std 1012-2012 (Revision of IEEE Std 1012-2004)* , vol., no., pp.1-223, 25 May 2012, doi: 10.1109/IEEESTD.2012.6204026.
https://ieeexplore.ieee.org/document/5930304
"IEEE Standard for Rechargeable Batteries for Cellular Telephones," in *IEEE Std 1725-2011(Revision to IEEE Std 1725-2006)* , vol., no., pp.1-91, 10 June 2011, doi: 10.1109/IEEESTD.2011.5930304.

https://ieeexplore.ieee.org/document/8320570
"IEEE Standard for Environmental and Social Responsibility Assessment of Computers and Displays," in *IEEE Std 1680.1-2018 (Revision of IEEE Std 1680.1-2009)* , vol., no., pp.1-121, 19 March 2018, doi: 10.1109/IEEESTD.2018.8320570.