Contents

[1 - Executive Summary 6](#_Toc354998656)

[2 - Project Description 7](#_Toc354998657)

[2.1 - Basic Functions 7](#_Toc354998658)

[2.2 - Motivation 8](#_Toc354998659)

[2.3 - Technical Objectives 9](#_Toc354998660)

[2.4 - Requirements and Specifications 10](#_Toc354998661)

[3 - Research and Investigations 12](#_Toc354998662)

[3.1 - Existing Similar Projects and Products 12](#_Toc354998663)

[3.2 - Microprocessor vs. Microcontroller 14](#_Toc354998664)

[Microcontrollers 14](#_Toc354998665)

[Microprocessor 15](#_Toc354998666)

[Optimization of Boot Performance 17](#_Toc354998667)

[Power On Sequence 17](#_Toc354998668)

[Power Off Sequence 18](#_Toc354998669)

[Pin Multiplexing Control 18](#_Toc354998670)

[3.3 - Digital Signal Processing 20](#_Toc354998671)

[DAC and ADC 20](#_Toc354998672)

[Universal Parallel Port (uPP) 21](#_Toc354998673)

[The Use of a Digital Signal Processor 21](#_Toc354998674)

[3.4 - LED Driver and Lighting 22](#_Toc354998675)

[3.5 - Memory 22](#_Toc354998676)

[Our Choice - Secure Digital 22](#_Toc354998677)

[Alternatives Not Used 22](#_Toc354998678)

[SD/MMC Controller 23](#_Toc354998679)

[RAM 25](#_Toc354998680)

[RAM (DDR) SETUP 25](#_Toc354998681)

[Memory (RAM) Placement 28](#_Toc354998682)

[3.6 - Development Board 29](#_Toc354998683)

[BeagleBoard 31](#_Toc354998684)

[Raspberry Pi 31](#_Toc354998685)

[Arduino Uno 32](#_Toc354998686)

[3.7 - Magnets 32](#_Toc354998687)

[3.8 - Motors 32](#_Toc354998688)

[3.9 - Sensing 36](#_Toc354998689)

[3.10 - Materials 37](#_Toc354998690)

[The Enclosure 37](#_Toc354998691)

[The Board 38](#_Toc354998692)

[The Chess Pieces 38](#_Toc354998693)

[3.11 - Power Supply 38](#_Toc354998694)

[3.12 - Printed Circuit Board 39](#_Toc354998695)

[PCB for BGA 41](#_Toc354998696)

[3.13 - Operating System 42](#_Toc354998697)

[Arch Linux ARM 43](#_Toc354998698)

[Debian ARM 43](#_Toc354998699)

[Ubuntu ARM 43](#_Toc354998700)

[Gentoo 44](#_Toc354998701)

[Android 44](#_Toc354998702)

[3.14 - Shells 45](#_Toc354998703)

[3.15 - Speech Recognition 45](#_Toc354998704)

[How Speech Recognition Works 46](#_Toc354998705)

[Problems with PocketSphinx 47](#_Toc354998706)

[Additional Speech Recognition Problems and Research 48](#_Toc354998707)

[Speech Recognition Alternative - Sphinx4 48](#_Toc354998708)

[Speech Recognition Alternative - eSpeak 48](#_Toc354998709)

[Speech Recognition Alternatives - Android 49](#_Toc354998710)

[Speech Recognition Alternatives - Windows 49](#_Toc354998711)

[3.16 - Board Representation (Logical) 49](#_Toc354998712)

[Language 49](#_Toc354998713)

[Board Data Structure 53](#_Toc354998714)

[Algorithms 54](#_Toc354998715)

[4 – Initial Hardware Design 59](#_Toc354998716)

[4.1 - ARM9 AM1808 Processor 59](#_Toc354998717)

[Accessing GPIO 60](#_Toc354998718)

[PLL 61](#_Toc354998719)

[UPP 61](#_Toc354998720)

[4.2 - OLinuXino iMX233 Development Board 62](#_Toc354998721)

[4.3 - TLV320ADC3101 ADC 63](#_Toc354998722)

[4.4 - PCM1753 DAC 65](#_Toc354998723)

[4.5 - DRV8834 Stepper Motor Driver 67](#_Toc354998724)

[4.8 - TLC5930 LED Driver 70](#_Toc354998725)

[4.9 - COM-08642 Reed Switches 70](#_Toc354998726)

[4.10 - Memory (SD Card | RAM) 71](#_Toc354998727)

[4.11 - ICM7211M LCD Panel Driver 71](#_Toc354998728)

[Powering the Colon 72](#_Toc354998729)

[4.12 - Power Supply/Voltage Regulation 72](#_Toc354998730)

[4.13 - Other Hardware 73](#_Toc354998731)

[M74HC238 Decoder 73](#_Toc354998732)

[4.14 – Design Change 74](#_Toc354998733)

[5 - CyberChess Hardware 75](#_Toc354998734)

[5.2 - Big Easy Drivers 76](#_Toc354998735)

[5.3 - 74HC595 Shift Registers 78](#_Toc354998736)

[5.4 - ROB10846 Stepper Motor 79](#_Toc354998737)

[5.5 - ROB09064 Servo Motor 80](#_Toc354998738)

[5.6 - COM-11120 RGB LEDs 81](#_Toc354998739)

[5.7 – FTDI USB to Serial Cable 82](#_Toc354998740)

[5.8 - AVR Pocket Programmer 82](#_Toc354998741)

[5.9 - Schematic and Printed Circuit Board (PCB) 83](#_Toc354998742)

[5.10 - Arduino Mega 85](#_Toc354998743)

[5.11 – Raspberry Pi 87](#_Toc354998744)

[6 - CyberChess Software 88](#_Toc354998745)

[6.1 - Operating System 88](#_Toc354998746)

[6.2 - Chess Engine 88](#_Toc354998747)

[Data Structures 89](#_Toc354998748)

[Game Saving and Loading 90](#_Toc354998749)

[Timer 91](#_Toc354998750)

[6.3 - Board, Movement and Sound Engine (BMSE) 92](#_Toc354998751)

[6.4 - Speech Recognition Engine 92](#_Toc354998752)

[7 – Hardware Design Summary 94](#_Toc354998753)

[7.1 - Movement System 94](#_Toc354998754)

[7.2 - Sound System 96](#_Toc354998755)

[7.3 - Lighting system 96](#_Toc354998756)

[8 – Software Design Summary 98](#_Toc354998757)

[8.1 - Chess Engine 99](#_Toc354998758)

[8.2 - Speech Recognition Engine 100](#_Toc354998759)

[9 - Project Prototype, Construction, and Coding 101](#_Toc354998760)

[9.1 - Parts Acquisition and Bill of Materials 101](#_Toc354998761)

[9.2 BOM 103](#_Toc354998762)

[9.3 - Plexiglas Chess Board Construction 105](#_Toc354998763)

[9.4 - Encasing Construction 105](#_Toc354998764)

[9.5 - PCB Vendor and Assembly 106](#_Toc354998765)

[9.6 - Chess Engine 106](#_Toc354998766)

[9.7 - Board and Movement and Sound Engine 107](#_Toc354998767)

[10 - Implementing the Design 108](#_Toc354998768)

[10.1 - Prototype Plan 108](#_Toc354998769)

[10.2 - Build Plan 108](#_Toc354998770)

[Hardware Specific Prototyping 108](#_Toc354998771)

[10.3 - Test Plan 108](#_Toc354998772)

[Software Testing 109](#_Toc354998773)

[Chess Engine Tests 109](#_Toc354998774)

[Board and Movement Engine Tests 109](#_Toc354998775)

[Speech Recognition Engine Tests 110](#_Toc354998776)

[Hardware Testing 111](#_Toc354998777)

[Basic Hardware 111](#_Toc354998778)

[Big Easy Drivers 112](#_Toc354998779)

[Shift Registers 112](#_Toc354998780)

[10.4 - Extending the Design 112](#_Toc354998781)

[LCD Display/Video Out 113](#_Toc354998782)

[Wi-Fi 113](#_Toc354998783)

[Camera 113](#_Toc354998784)

[Sensors 113](#_Toc354998785)

[AI 114](#_Toc354998786)

[11 - Facilities and Equipment 115](#_Toc354998787)

[11.1 - IDE and SDK 115](#_Toc354998788)

[11.2 - Development Board 116](#_Toc354998789)

[11.3 - Chess Board 116](#_Toc354998790)

[12 - Abnormal Operation 117](#_Toc354998791)

[12.1 - Motors 117](#_Toc354998792)

[12.2 - Memory Leaks & Segmentation Faults 118](#_Toc354998793)

[13 - Operators Manual 119](#_Toc354998794)

[14 - Administrative Content 125](#_Toc354998795)

[14.1 - Milestones 125](#_Toc354998796)

[14.2 Budget and Finance Information 125](#_Toc354998797)

[14.3 - Summary and Conclusion 127](#_Toc354998798)

# 1 - Executive Summary

CyberChess is an autonomous, voice-controlled chess board designed to allow two players to enjoy an aesthetically appealing game of chess entirely through spoken commands. Each player uses a wired microphone to issue these commands, allowing them to have the system set up a game, execute moves, and display possible moves for any piece on the board. CyberChess makes use of a magnetic XY-plotter to physically move the pieces as needed. An array of LEDs provides visual notifications to the players throughout the game, supplemented by audio to communicate ideas directly to the user. An embedded Arch Linux system runs on the ARM-based Raspberry Pi, utilizing PocketSphinx to accept spoken commands and reacting to them, outputting relevant audio and instructing the connected ATMega 2560 to execute appropriate movement and lighting operations.

CyberChess comes from a love of technology and the desire to expand into new technological terrain. No single element of this project could be considered individually revolutionary, but the combination creates a unique take on the oft-addressed idea of automated chess play. The chance to gain experience with speech recognition, embedded Linux environments, ARM and ATMega based hardware, and with a sizable project all provide opportunities to supplement the knowledge and education of those involved.

The physical, aesthetic design of CyberChess mimics that of actual chess boards, seeking to create a sense of familiarity for the players rather than requiring them to deal with an entirely new environment. All lighting effects have been designed to intuitively provide information and not detract from the user experience - as an obvious example, upon a player’s request to view the possible moves of a piece the relevant spots light up, providing simple access to the knowledge requested. Audio is used similarly to directly communicate with the user and provide a generally pleasing gameplay experience.

The movement system was derived from those employed by similar projects predating CyberChess. With the long history of mechanical engineering, true ingenuity in addressing such a problem would have required extreme time investments and would likely have required a truly unique idea. Lacking these resources, CyberChess does not seek to reinvent the wheel, but instead to make effective use of it in the creation of a larger system. The software driving this system makes use of an open source voice recognition system called PocketSphinx and custom software designed specifically for the project to implement the rules of the game and the corresponding hardware control.

# 2 - Project Description

## 2.1 - Basic Functions

The final CyberChess system allows two users to play through an entire game of chess with no more physical interaction than powering on the system and wearing the headset used to issue spoken commands. At startup, the LED array displays an aesthetically pleasing sequence of light effects, welcoming the players to the game. In the event that a game has been saved to the system from a previous session, the program prompts the users to choose between returning to the saved game, in which case all settings reflect those chosen for that game, and starting a new game.

Throughout the game, the current player dictates their move into a microphone using the standard format of “Move [Piece Name][Current File (NATO Alphabet Pronunciation)][Current Rank] to [Destination File (NATO Alphabet Pronunciation)][Destination Rank]” (e.g. “Move Rook Bravo One to Charlie Three” for Rook at B1 to C3). Each time a player requests a move, the internal chess engine determines the validity and legality of the move. Should the player have requested a move that is either illegal or invalid, the problem is communicated through the speaker system, audibly informing the player that their move cannot be made. This notification is visually accompanied by a simple flashing of the LEDs. The player may ensure that they will request a valid and legal move by asking for all possible moves for a piece using the phrasing “Possible moves [File (NATO Alphabet Pronunciation)][Rank]” (e.g. “Possible moves D3”). In response, the engine generates a list of moves which is used to light up the squares to which the piece can legally move.

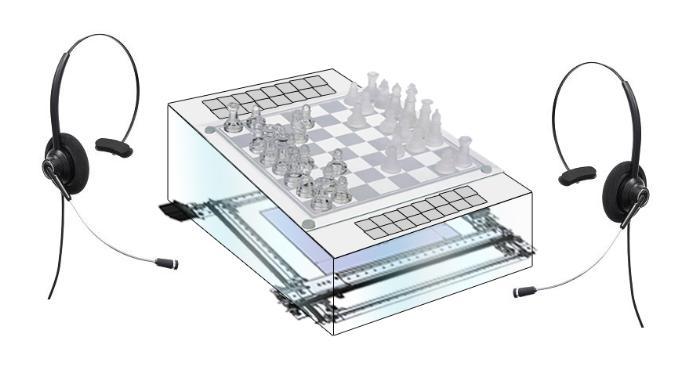
Various aspects of gameplay need not be directly commanded by the players. When a piece is captured, the captured piece is magnetically removed to a spot on the side of the board that is internally tracked by the game engine. When a pawn is promoted, the program audibly prompts the user to choose which piece to replace it with, and henceforth treats the pawn as the chosen replacement piece. Castling is requested by having the king move to its destination location, e.g. “Move King Echo One to Golf One” to have the white king castle kingside. When a player’s king is put in check, the program says “Check!” through the speakers, and the LEDs display an accompanying light pattern.

Game conclusion takes place in the normal means available in any standard chess game. If a checkmate occurs, CyberChess displays an end-of-game lighting sequence and audibly inform the players of the outcome, acknowledging the winner. Similarly, if a draw occurs due to stalemate or the presence of insufficient resources to checkmate, the game indicates the cause of the draw and displays a corresponding lighting sequence. When the opportunity to offer a draw, due to a lack of pawn movement or piece capture in the previous fifty moves, is available but not required, the players are informed and asked if they would like to declare a draw. Additionally, either player may offer a draw at any time, which will end the game if the other player accepts. In any of these instances, the board displays the appropriate lighting sequence and gameplay will terminate. Finally, either player may choose to resign at any time and, after receiving confirmation that the player does really want to resign, the game will end and the winner indicated.

Should players desire to save the game for later, they may at any point say “Save game,” and the current game state will be saved. This can be used to have a sort of checkpoint to go back to after continuing with one line of play, or to allow the system to return the board to the saved state at a later point after other games. Only one game may be saved at any given time, a limit imposed in the interest of simplifying the voice recognition vocabulary. At any point a player may request a new game, and, if their opponent acquiesces, the CyberChess moves all pieces to their starting position and begins a new game. As long as the previous game was saved, it can be returned to at any point.

## 2.2 - Motivation

The development team wanted to create a project that struck a chord with personal interests and combined the aspects of engineering that most interest the members of the development team (programming, electromechanics, critical thinking, etc.). Having discussed various themes in pursuit of catchiness and what best sells the design to others, the team settled on the end product of CyberChess. There have been many senior design projects that go after chess automation, but thus far there has not been another chess board that allows two people to play on the same board with full automation, let alone to do so entirely by way of voice. A scene from one of the popular *Harry Potter* movies sparked the initial idea of including speech recognition in the project. The use of speech recognition in the project allows the game of chess to be played without physically touching the pieces, which could bring about a welcome change for those who are physically disabled.



**Figure 2.2.1 - A Rough** **Digital Sketch of CyberChess**

The LED Lighting Array that has been integrated into the board was originally motivated and inspired by a discarded senior design project idea called LoudLight, which was a box that you could link with your phone via Bluetooth to receive visual notifications. These visual notifications have been worked into CyberChess, providing visual feedback to players on turns, moves, and game actions.

## 2.3 - Technical Objectives

The initial objectives of CyberChess involved creating a system that would be run solely by an internal controller without need for an external laptop, PC, or similar component. The development team intended to do this by way of building from the component level up, doing all of the programming using an embedded Linux environment. Initially the team looked into having a microprocessor run the selected speech recognition program and host the entire suite of engines driving CyberChess’s operations. Eventually, this software was split between a microcontroller and an embedded mainframe. The CyberChess program was first prototyped and tested on a development board. Upon successful testing, it was loaded onto a custom-designed Printed Circuit Board containing all of the necessary memory and input/output. The team intended to and successfully constructed a XY-plotter consisting of a chassis with vex racks and sliders built onto it. The movement of the magnet is controlled by stepper motors (to move in two dimensions) and a servo motor (to raise/lower the magnet to grab/release the chess pieces).

The board, made of a layer of Plexiglas atop a wooden enclosure, itself is large enough to give the magnetic pieces plenty of buffer space between each other. And it will be made of two layers of Plexiglas. The XY-plotter resides below the Plexiglas, moving the magnet beneath the grid of wires and lighting system. More detailed descriptions of the hardware and software are presented later in this document. In the next chapter, we will discuss the research that went into finding the right hardware and software to make CyberChess come to life.

## 2.4 - Requirements and Specifications

CyberChess is an autonomous chess set designed for two human players. The team set out to implement a variety of challenging ideas in CyberChess, but recognized that certain functionality would need to be determined as a minimum level of operation. It was determined that at the very least the board must be speech activated and must have the capability to move chess pieces to the desired locations without the assistance of a human hand. The game was expected to be capable of detecting illegal moves called by the users. An array of LEDs was to make up a lighting system intended to provide feedback and interesting visual effects, to be supplemented by an audio system for additional feedback and enhancement of the user experience. LCD screens were to be present for use as optional chess clocks.

The tables 2.4.1 and 2.4.2 below summarize the requirements and specifications of CyberChess respectively.

|  |  |
| --- | --- |
| 1. | The game shall provide all game feedback via audio and lighting |
| 2. | The game shall allow any move to be completed without touching a game piece |
| 3. | The game shall reset the chess board automatically after game completion |
| 4. | The game shall display possible moves if a user asks for them |
| 5. | The game shall display a visual and audible countdown for modes with timers |
| 6. | The movement system and hardware shall be visible from the top of the game |
| 7. | The game shall be powered by a standard wall outlet |
| 8. | The game shall have a specific startup sequence for lights and audio |
| 9. | The game shall have a specific shutdown sequence for lights and audio |
| 10. | The game shall notify a user of an illegal move through lights and audio |

**Table 2.4.1 - Requirements List**

|  |  |
| --- | --- |
| Be Able to Complete Any Move in : | **10 Seconds** |
| Be Able to Reset Board in: | **5 minutes** |
| Be Able to Save and Load: | **1 Game** |
| Final Product may weigh no more than: | **10 lbs.** |
| The sound system must be audible from at least: | **2 feet** |
| Microphone cords must be at least: | **2 feet** |
| Final Product may be no larger than: | **3x3 feet** |
| Automatically turn off after (prolonged inactivity) | **5 minutes** |

**Table 2.4.2 - Specifications List**

# 3 - Research and Investigations

In order to build a good design plan, research was undertaken to ensure that the budgeted money was spent wisely and that all tasks at hand were thoroughly understood prior to executing them. The following sections discuss some existing projects that inspired the design plans as well as the many options that were reviewed while seeking to optimize the budget and make CyberChess unique. While numerous aspects of CyberChess were derived from observations of the projects listed below, each element was thoroughly investigated to attempt to supplement past successes with any creative thoughts of the team.

While the initial plans made throughout the research process were not entirely without merit, quite a few significant changes to the design of CyberChess were effected throughout the implementation process. As such, large portions of the research contained herein do not directly feature in any aspect of the finished product. These now seemingly irrelevant sections are retained in this document in an effort to document and provide insight into the evolution of the project.

## 3.1 - Existing Similar Projects and Products

The first major topic of research in development of CyberChess was looking into previous instances where similar projects were successfully constructed. These existing projects were seen as good points of reference and sources of general design schemes to build off of. As the core hardware that is most important to the basic functionality of the project is the system that physically moves the pieces, various ideas were sought to solve this problem.

One approach to an automated chess system was found in “Gambit: A Robust Chess-Playing Robotic System”, by C. Matuszek, B. Mayton, R. Aimi, M.P. Deisenroth, L. Bo, R. Chu, M. Kung, L. LeGrand, J.R. Smith, and D. Fox. The Gambit is a robotic chess system that allows the user to play chess against a computer. Gambit is not fully autonomous as it requires the human player to move their own chess pieces. A six degree-of-freedom robotic arm is mounted on the side opposite the user and is responsible for controlling the movement of the computer’s pieces. The sensor system for Gambit consists of a camera installed onto the end of the mechanical arm that can capture images of the entire playing board environment. Similar to Gambit, CyberChess offers the ability to reset the board upon conclusion of a game, but it requires the user to perform the initial setup when it is first powered on. Unlike CyberChess, Gambit is capable of playing against another robot opponent using any arbitrary chess set. (C. Matuszek, B. Mayton, R. Aimi, M.P. Deisenroth, L. Bo, R. Chu, M. Kung, L. LeGrand, J.R. Smith, and D. Fox)

“How to Build an Arduino Powered Chess Playing Robot” is a highly descriptive paper authored by Max Justicz. The document describes all of the parts, assembly, and coding that went into the design of his autonomous chess board. Unlike the human versus human paradigm of CyberChess, his project was designed for a human player to play against a robot. The human player manually makes a move, and the embedded system identifies what action the player has taken and autonomously makes a move in return. The design consisted of an XY-plotter with movement controlled by stepper motors. The grabbing of the chess pieces was done using a neodymium magnet attached to a servo motor. An Arduino Uno MCU was responsible for driving the motors, and an Arduino Mega was responsible for determining the move made by the human, via reed switches. This information was used in determining the system’s next move. (Justicz, Max)

An automated chess set was documented by Brett Rankin, Paul Conboy, Samantha Lickteig, and Stephen Bryant; titled “Interactive Automated Chess Set”. Their design utilizes a mechanical crane to lift the chess pieces from above, move them and place them back on the board. This design is driven in a similar manner as the XY-plotter described by Max Justicz in “How to Build a Chess Playing Robot”. However, the force used to hold the chess pieces is not magnetic. The Interactive Automated Chess Set is built of Plexiglas, and an LED system that lights up individual chess squares to show players potential moves. A player chooses their moves by pushing buttons corresponding to the row/column where they desire to move their piece. (Rankin, Conboy, Lickteig, and Bryant)

Once it was decided that an XY-plotter would be used to control the motion of CyberChess, multiple variations of how to design the XY-plotter were considered. One method researched in designing the XY-plotter involved creation of a pulley-based belt system to move the magnet base of the plotter instead of the linear motion gears the vex rack supplies. Such a construction would be accomplished by attaching a stepper motor at either end of drawer tracks for the X axis and use of a belt on pulleys located at either end. One pulley would be attached to a stepper motor to provide the linear motion.

The main problem identified with this method revolved around attaching the Y axis to the belt. This would have been unstable, inefficient, and would have mainly affected the distance between the magnet and the sensor board. This would result in differing distances between the magnets in the pieces and the one moving them, a distance which should remain constant throughout operation. Additionally, if the board were to be shaken and the belt become loose for any reason, the Y axis would have been able to move up which could cause the larger magnet to attract neighboring pieces that shouldn’t be moved, or the Y axis could have sagged a little and might cause the magnets not to connect at all, leading to a piece being unable to move. Both flaws would compromise the game.

Acknowledging these flaws, the CyberChess project instead makes use of an XY-plotter in order to make use of a solution that has proven to work before and still yields the illusion that the pieces are moving by themselves. While other ideas involving more electronics, more motors, and more complicated designs such as assignment of individual magnets and motors to specific chess pieces were all considered, ultimately the simple design of the XY-plotter came through. These other ideas were far more complex than the XY-plotter with little to no observable performance advantages at steeply greater costs. The final XY-plotter design was inspired by that described in “How to Build an Arduino Powered Chess Playing Robot”, using drawer slides to stand in as linear actuators and providing the linear motion as opposed to using pulleys like the belt system.

The Excalibur Electronic Phantom Force Electronic Chess Set is one of the only products on the market that offers automated chess but only for the computer, essentially creating a real world version of a computer chess game. Players can either play against the computer or watch the computer play itself. The board talks in English, French or Spanish while it is playing you and also sets up itself. One of the main appeals of Excalibur’s electronic chess is the portability due to its compact size.

On the software side of things, a number of open source implementations of chess engines are readily available online. A number of Python chess games are hosted on PyGame, a website dedicated to games written in Python. Investigating the source code of these revealed that there are indeed a wide variety of ways to approach creating a chess engine, with the two most popular PyGame programs varying quite a bit throughout. The majority of available code was aimed at creating Artificial Intelligence systems for gameplay, and thus focus was more heavily placed on move generation and position analysis than was strictly necessary for CyberChess, which needed stronger support for player interaction.

## 3.2 - Microprocessor vs. Microcontroller

### Microcontrollers

Microcontrollers are low-powered embedded computers that are usually dedicated to specific purposes. These single integrated circuits often come with Read Only Memory for the program to reside in, Random Access Memory for variables, multiple purpose Input/Output, and a microprocessor for the CPU. The vast majority of Senior Design projects function entirely using just a microcontroller. Due to the higher processing demands of voice recognition, CyberChess was not such a project - additional processing power was a must. Additionally, most microcontrollers are limited to between 4KB and 512 KB of onboard flash memory, which is insufficient to store the libraries PocketSphinx depends on.

In the initial phases of the research, the development team became fixated with creating an embedded Linux system and research related to microcontrollers rapidly lead to research about running a Linux operating system on one. This lead to the discovery of uClinux, an embedded Linux/microcontroller project that focuses on porting Linux to systems without a Memory Management Unit (MMU). While this seemed like an excellent means of solving the problem of running Linux on a microcontroller, it quickly became evident that running Linux with such limited specifications would come at a cost. uClinux, while having support for multitasking, has a few limitations. uClinux for example does not autogrow stack and has no brk() method. All memory allocations must be done using mmap(), which is actually done by most modern code anyways. uClinux also does not implement fork(), but instead uses vfork(). This means that the parent blocks any other code from running until the child does exec() or exit(). While this means multitasking is still possible, it would have required a great deal of intentionality in the coding process. These complications lead the team to begin investigations into use of a microprocessor with an ARM Linux distribution.

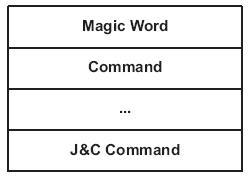
### Microprocessor

Recognizing the higher processing and memory requirements of CyberChess, the development team looked into creating a platform similar to that of a mid-level Smartphone from two years ago. The desired platform was one supporting processing power lying in the range of 200-600 MHz, a range which encompassed a number of ARM-based processors. Due to the availability of samples for testing, the team procured an AM1808 Sitara ARM Microprocessor from Texas Instruments, finding its clock speed and its USB, SD, and LCD support to be in line with the design goals of the CyberChess project.

The AM1808 Sitara ARM Microprocessor makes use of the AM18xx bootloader read-only memory (ROM) image. The technical document provided by Texas Instruments gives a breakdown of the Application Image Script (AIS) boot process, an AISgen tool used to generate boot scripts, protocol for booting from an external master device, a UART Boot Host GUI for booting from a host PC, as well as any limitations, default settings, and assumptions made by the bootloader.

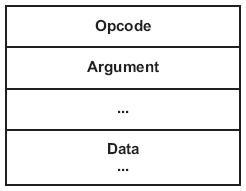
The bootloader supports booting from many embedded memory devices in master mode as well as many external devices using slave mode. A majority of the boot modes, excluding host port interface (HPI) and two of the three NOR-boot modes, use Application Image Script for the boot process. Using AIS for booting provides the developer with a unified interface even when using the AISgen tool.

Application Image Script or AIS is a format type for storing the boot image. AIS is a binary language accessed in 4-byte words in little endian format. A magic word (0x41504954) starts of the AIS, followed by a series of AIS commands executed in sequential manner, and ends with a jump and close command. The structure of AIS is shown below in Figure 3.2.1.



**Fig 3.2.1 - Application Image Script (AIS) Structure**

Between the magic word and the J&C command lies the AIS commands. Each command consists of an opcode, optionally followed by one or more arguments, followed by optional data. This structure is shown below in Figure 3.2.2.



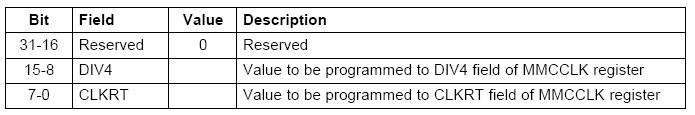
**Fig 3.2.2 - ASI Command Structure**

The opcode and its optional arguments are each one word (4 bytes) wide. Padding may occur when a command is not enough bits, with 0’s being added until the total length is a multiple of 4 bytes.

MMC/SD boot mode is done by transferring the AIS boot image to the user data area of the memory device, which in this case will be an SD card. The bootloader attempts to find the AIS image within the SD card, and once it is found, then looks for the magic word (the same as before). If the magic word is not found, the bootloader increments by 0x200 and searches again. This process repeats until the bootloader has searched the first 2MB of the memory card. The bootloader usually tries to check for an SD card first but this can be avoided if an MMC card is being used. To make sure the bootloader is searching for an SD card, boot pin BOOT[5] must be set low. Setting this pin high skips the SD search and goes straight to MMC. The boot pins are latched by the bootloader when the device exits reset, or the rising edge of reset. The MMC/SD register is shown below in Figure 3.2.3 and described in table 3.2.1.



**Fig 3.2.3 - SD/MMC Register**



**Table 3.2.1 - SD/MMC Register Breakdown**

### Optimization of Boot Performance

During investigation of use of the AM1808, the team sought means of optimizing the boot time of the system. Reduction of boot time associated with the cold boot at initial power on was found to be possible through customization of the bootloader and Linux kernel. Customization would have focused heavily on the size and speed of these elements, stripping the default configurations of the unnecessary code intended to handle hardware components that are not present within the CyberChess system. Additional improvement of boot performance was possible by way of deferring the loading of modular components until later than normal in the boot sequence. This is an improvement that can readily be accomplished in any fully developed Linux operating system and as such was still taken advantage of in the final iteration of the build.

### Power On Sequence

The AM1808 should be powered on in the following order:

1. RTC(RTC\_CVDD) - This may be powered prior to all other supplies being applied or can be powered at the same time as CVDD. In the case where the RTC is not in use, RTC\_CVDD should be connected to CVDD. If CVDD is powered, RTC\_CVDD must not be unpowered.
2. The core logic supplies should be powered on next including both all variable 1.2V - 1.0V core logic supplies (CVDD) and all static core logic supplies (RVDD, PLL0\_VDDA, PLL1\_VDDA, USB\_CVDD, SATA\_VDD).
3. Now all static 1.8V IO supplies including DVDD18, DDR\_DVDD18, USB0\_VDDA18, USB1\_VDDA18 and SATA\_VDDR must be powered on
4. Finally all analog 3.3V PHY supplies including USB0\_VDDA33 and USB1\_VDDA33.

For the above power sequence there is no specific required voltage ramp rate so long as LVCMOS supplies operated at 3.3V never exceed the STATIC 1.8V supplies (step 3) by more than 2 volts.

### Power Off Sequence

As just previously stated, as long as LVCMOS supplies operated at 3.3V

(DVDD3318\_A, DVDD3318\_B, or DVDD3318\_C) never exceed static 1.8V supplies by more than 2 volts then the power supplies can be powered-off in any order.

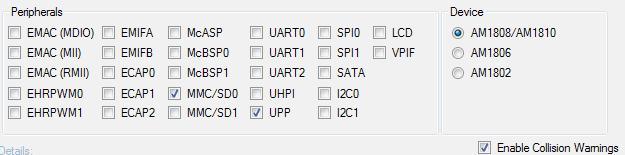
### Pin Multiplexing Control

Pin multiplexing is controlled by registers PINMUX0 - PINMUX19 in the SYSCFG module. For the AM18XX family, multiplexing can be done on a pin-by-pin basis, and is controlled by a 4-bit field in one of the PINMUX registers. The values determine which peripheral pin functions controls the pin’s IO buffer data as well as the output enable values. Default values for nearly every pin is to select none of the functions available, which leaves the pin’s IO buffer held tri-stated. Table 3.2.2 below shows the PINMUX values for our specific application, making use of the MMC/SD0 and UPP controllers. These values were determined using the AM18XX Pin Setup Utility from Texas Instruments.

|  |  |
| --- | --- |
| **PIN** | **Value** |
| PINMUX0-9 | - |
| PINMUX10 | 0x22222222 |
| PINMUX11 | 0x00000022 |
| PINMUX12 | - |
| PINMUX13 | 0x44440000 |
| PINMUX14 | 0x44444400 |
| PINMUX15 | 0x44444444 |
| PINMUX16 | 0x44444444 |
| PINMUX17 | 0x44444444 |
| PINMUX18 | 0x00444444 |

**Table 3.2.2 - Pin Multiplexing Register Values from Pin Setup Utility**

The Pin Setup utility allows us to select which features and hardware controllers we are using. Figure 3.2.4 below shows the selection options and Figure 3.2.5 shows their application to the pins.



**Fig. 3.2.4 - Pin Setup Peripheral Options and Device Selection**



**Fig. 3.2.5 - Pin Setup Active Pins**

## 3.3 - Digital Signal Processing

One of the challenges presented by working with audio (input and output) was being able to put it in a form where our processor can read it as binary bits. Converting analog audio signals into digital bits is done using analog to digital converters, and reversed using digital to analog converters. This section discusses how we planned to set up our hardware and use our processor to manipulate this data.

### DAC and ADC

Our primary reason for needing analog-to-digital and digital-to-analog converters is for input and output audio. We have no need for a stereo system, however we do need two input channels for microphones (one for each player). Since our audio input is strictly voice, we needed an ADC that with vocal friendly sampling capabilities. The human voice can range from 60 to 7000 Hz, so our ADC required sampling at at least 14 kbps, according to the Nyquist criterion. We decided to go with the PCM1753 DAC and the TLV320 ADC from Texas Instruments because they are inexpensive (and TI had free samples) and their sampling capabilities are more than enough to support the audio processes in CyberChess. Other options were available, but they mostly provided higher sampling rates and were thus more expensive. It was determined that these would be connected to our ARM processor via the Universal Parallel Port (uPP). Data conversion and other DSP related chips can be performance optimized using uPP, by allowing the CPU clock to adjust to a more suitable time for the converter. Some converters have a wide range of options for time constraints otherwise.

### Universal Parallel Port (uPP)

This port allows two independent channels at speeds up to 75MHz and is designed to interface with converters with up to 16-bit data width (per channel). uPP also features dedicated data lines, minimal control lines, and can be interconnected with field-programmable gate arrays (FPGAs) or other uPP devices. It can operate in both send and receive mode or in duplex mode, where individual channels operate in opposite directions. Included inside the uPP device is an internal Direct Memory Access (DMA) controller. This DMA controller allows for maximum throughput with minimized CPU overhead during high-speed transmission.

### The Use of a Digital Signal Processor

A Digital Signal Processor (DSP) is a microprocessor specifically designed to process data in real time. Its design means it contains specialized functions for digital signals so it has the ability to process the complex and mathematically intensive calculations much more efficiently than a regular microprocessor. While a regular microprocessor is still able to do digital signal processing, its lack of specialization results fails to make the most efficient use of processing power. As such, using a plain microprocessor for digital signal processing would have required it to work much harder than a DSP would. This would mean more heat output, more energy expenditure, less processing power available for other operations, and possibly a delay in output. These problems are product breaking in cases like a cell phone, which are expected to maintain a small form factor and have usage requirements. However, in the case of CyberChess, none of these factors would have come into play.

The large form factor of our design means heat dissipation is not a problem, even if our processor had had to work extra hard on the digital signals. Unlike a cell phone, our product does not process loads of data beyond the digital audio signals from the microphones, so dedicating a majority of our processor to DSP will not be a problem. In the case of a cell phone, a user may decide to browse the internet while using their phone, which makes the use of a DSP all the more necessary. Delays in a cell phone conversation would be incredibly annoying to users, which makes the use of a DSP necessary. In our case, a few seconds of processing between a command and game action is not product breaking.

The team ultimately determined that a DSP would not be necessary.

## 3.4 - LED Driver and Lighting

To light the board we initially wanted an effect only possible with EL Wire. Upon further investigation we found EL Wire was difficult to work with and hard to provide proper currents/voltages to, in addition to the fact that we would have to wire every square on the board individually if we wanted to light them independently. Because of these problems we decided an LED array would suffice, but we would need a driver. From Texas Instruments we found the TLC5930 12 Channel LED Driver, which supports four RGB’s. The 12 channel chip was chosen because we found it provided a good cost/channel ratio and had free samples available. Upon further investigation, we determined we could easily replicate the functionality of the drivers by way of shift registers.

## 3.5 - Memory

Memory inside CyberChess is used to store all of our code, operating system, and speech recognition libraries. Our memory choices included embedded solutions like NAND/NOR Flash Memory or removal options such as Secure Digital (SD) or a USB Flash Drive.

### Our Choice - Secure Digital

Secure Digital memory cards make use of NAND Flash Memory, but are obviously in card form. SD cards are available in three capacities, each with a variety of speed options marked by a Speed Class symbol. The three capacities, SD, SDHC, and SDXC allow developers to choose the correct amount of memory for their specific use. Since SDXC cards come in capacities of 32GB to 2TB we saw no reason to further pursue them as an option, as they provided memory that far exceeded our needs. We instead focused on standard SD and SDHC cards. The SD Standard allows for cards up to 2GB while SDHC ranges from 2GB to 32GB. For our needs, a 1-2GB SD card would suffice but due to falling prices of SDHC cards, we instead opted for a higher capacity card, using an 8GB SD card in the Raspberry Pi.

### Alternatives Not Used

1. **NAND/NOR Flash Memory**

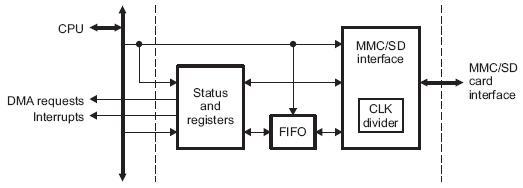
NAND and NOR Flash Memory are an embeddable memory solution for CyberChess. They two specifications, NAND and NOR, refer to the types of logic gates being used internally. NAND Flash Memory has a significantly higher capacity than NOR but NOR Flash Memory is faster, and thus more expensive. A device may choose to use both NAND and NOR memory because of their specific benefits. For example, a netbook may use embedded NOR for the operating system and a removable NAND card for additional memory and storage. While embedded memory would have created a faster environment, modifying our software would have required a wired connection like serial or USB to be added to our PCB. In addition to this, our development board at the time used a Secure Digital Card for all of its memory, so mimicking this setup will allow for easy transition between the development board and our PCB.

1. **Flash Drive**

A flash drive is essentially the third form of NAND Flash Memory that was available to us. This option is stored in an enclosure with a USB connection, and is thus limited by the transfer speeds of USB, around 480 Mbit/s. Use of a USB Flash Drive would also require the addition of a USB connector to our PCB. With other solutions available that do not require such physical modifications and offer sufficient transfer rates, this was not considered practical.

### SD/MMC Controller

The AM1808 processor came with a MMC/SD card controller which was to be used to interface with our secure digital memory card. The controller also featured a programmable frequency of the clock that controls the timing of transfers between the SD controller and memory card and 512-bit read/write FIFO to lower system overhead. The card controller allows use of either the CPU or EDMA (external direct memory access) to write and/or read to and from the card by accessing the registers in the card controller. On one side the controller transfers data between the CPU and the EDMA controller and the MMC/SD card on the other. This separation allows the use of both the CPU or EDMA and is shown below in Figure 3.5.1.

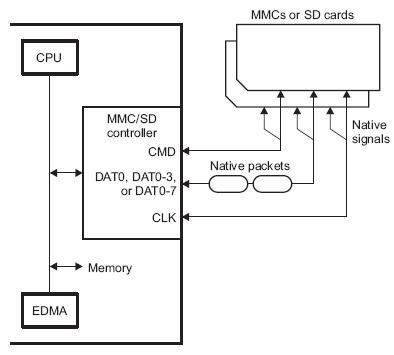


**Fig 3.5.1** - **AM1808 SD/MMC Card Controller**\

In terms of industry standard compliance, the card controller supports the following industry standards:

* Multimedia Card (MMC) Specification v4.0
* Secure Digital (SD) Physical layer Specification v1.1
* Secure Digital Input Output (SDIO) Specification v2.0

The card controller communicates using the MMC/SD protocol and contains support for use of one or more MMC/SD cards, which are selected by using identification broadcast on the data line. The figure and table below summarize the interface.



**Fig 3.5.2 - MMC/SD Card Controller Interface**

|  |  |
| --- | --- |
| **Pin** | **Use** |
| MMCSD\_CMD | This pin is used to provide communication between the connected card and the MMC/SD controller. The commands are transmitted to the card using this pin and the memory card drives response to the commands on this pin. |
| MMCSD\_DAT0, MMCSD\_DAT0-3,  or MMCSD\_DAT0-7 | MMC Cards only use one data line (DAT0), four data lines (DAT0-3) or eight data lines (DAT0-7A). SD Cards use one data line (DAT0), or four data lines (DAT0-3). The number of pins being used is set by the WIDTH bit in the MMC control register (MMCCTL). |
| MMCSD\_CLK | This pin provides the clock to the memory card from the MMC/SD controller. |

**Table 3.5.1 - Pin descriptions for MMC/SD Card Controller Interface**

### RAM

Random Access Memory can be either Dynamic or Static memory. Dynamic Random Access Memory (DRAM) provides a structural advantage over SRAM because it requires only one transistor and a capacitor per bit, as opposed to SRAM which can take four to six. This advantage allows DRAM to reach high densities. Unlike both types of flash memory previously discussed, RAM is volatile, meaning it loses its data when power is lost.

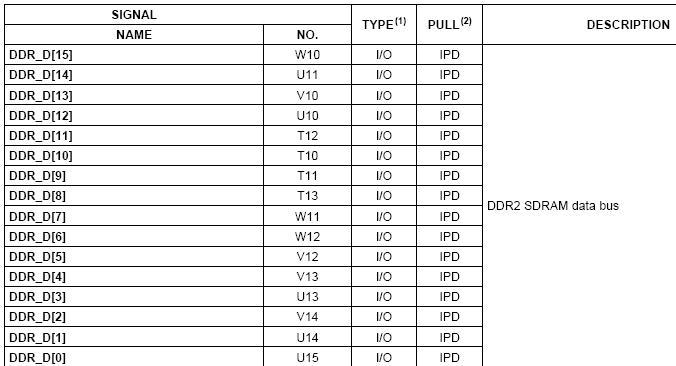
### RAM (DDR) SETUP

While the team has not decided whether we will need random access memory, DDR memory is configurable from within ASIgen when the configure DDR checkbox is selected. This configuration allows a developer to set the DDR controller registers. ASIgen supports both mDDR and DDR2 memory, with certain registers specifically for mDDR. The registers and settings are shown below in Figure 3.2.4.

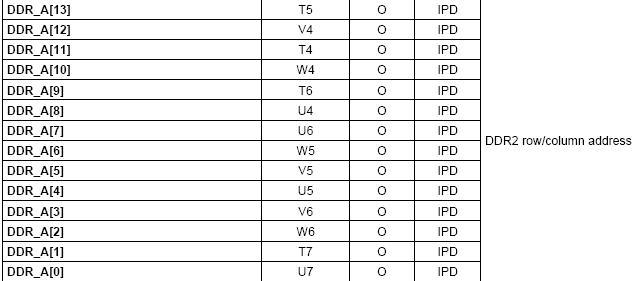


**Fig 3.5.3 - ASIgen DDR Settings**

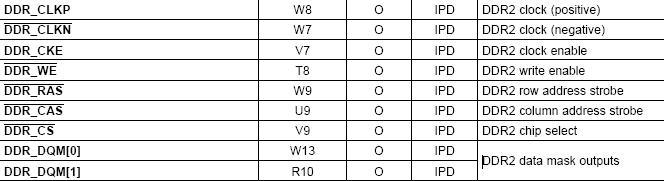
The DDR2 Controller contains a total of 39 pins, 16 of which are used for the data bus, 14 for the row/column address, and the remaining 9 for the clock, write enable, row address and column address strobe, chip select, and data mask outputs. This information is summarized below in tables 3.5.2, 3.5.3, and 3.5.4 respectively (Page 23).



**Table 3.5.2 - DDR2 SDRAM Data Bus Pins**



**Table 3.5.3 - DDR2 Row/Column Address Pins**



**Table 3.5.4 - DDR2 Additional Pins**

Typically speaking, high-speed design flow is a very complex and time-intensive endeavor. Designing the PCB involves many iterative simulations which can cause problems due to inaccurate models, bugs in tools, using incorrect environmental conditions, and errors in processing the large amounts of data involved in the simulation. A solution to this problem is to avoid simulations completely. By making the correct choices in the system specification, these timing specifications can be communicated without simulation models or timing numbers.

Design rules for the DDR/mDDR interface constrain PCB trace length, flight time delay and trace skew, signal integrity and impedance matching, cross-talk, and signal timing. In order to properly create a reliable memory system these rules must be followed.

*Trace, Flight Delay and Skew*

Flight Delay and Skew involve the placement of the components. A value called maximum placement refers to the maximum distance between devices permitted. Controlling this value can limit the maximum trace delay to about the longest Manhattan distance of the signals inside the clock domain. The value is the longest distance because all of the shorter nets are lengthened to skew match the longest one. Flight time delay, flight time delay skew, and the maximum trace lengths are thus factors of the maximum placement.

*Signal Integrity and Impedance Matching*

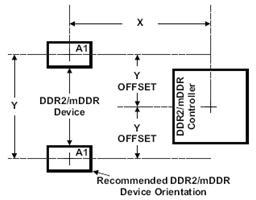
Signal integrity involves overshoot, ring back, and transition edges. With a constrained length it is possible to control signal integrity by controlling the trace topology of the different segments of an interface. The impedance of the PCB traces must be controlled and is governed by the trace width as well as the thickness and dielectric constant of the insulating material. Luckily for us, this is usually left to the PCB fabricator.

*Crosstalk*

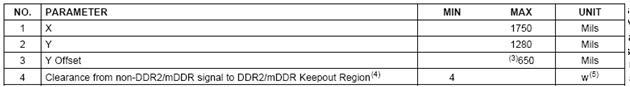
Crosstalk is dependent on the PCB stackup and minimum trace spacing. While good crosstalk simulation can be difficult, a good solution involves high-quality signal return paths and the spreading of the signal traces. Every routing layer should have an adjacent full ground plane to allow for the shortest return current path.

### Memory (RAM) Placement

Proper placement of the RAM memory module limits the maximum trace lengths and allows for proper routing space. For our purposes, we will be using only one memory device so the second DDR/mDDR device may be omitted from the images below. Figure 3.5.4shows the required placement for both the memory controller, or CPU in this case, as well as the memory devices themselves. Table 3.f.2 defines the dimensions.

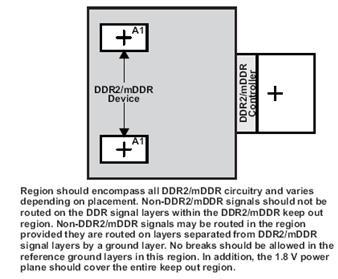


**Fig 3.5.4 - Device and DDR2/mDDR Device Placement**



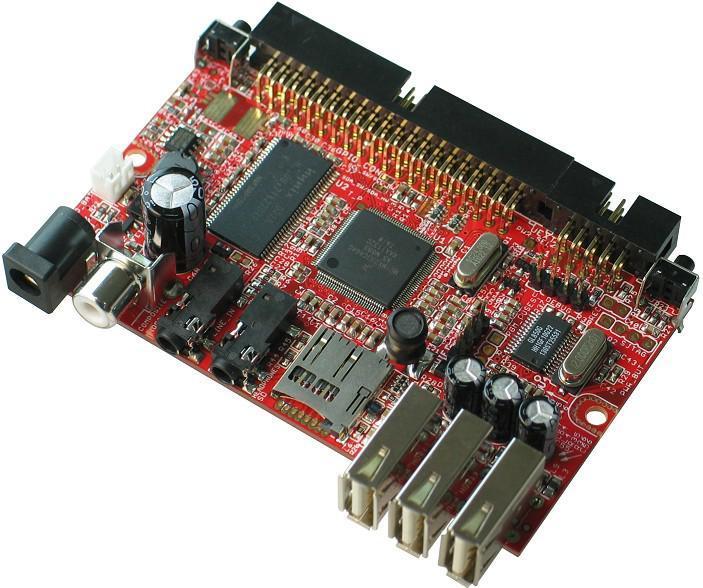
**Table 3.5.5 - Placement Specifications and Dimensions**

Additionally, the DDR/mDDR region on the PCB must be isolated from other signals. A keep out region is thus defined to serve this purpose. The size of the keep out region depends on the placement and DDR routing. The keep out region and its specifications are shown below in Figure 3.5.5.



**Fig 3.5.5 - DDR2/mDDR Keep Out Region**

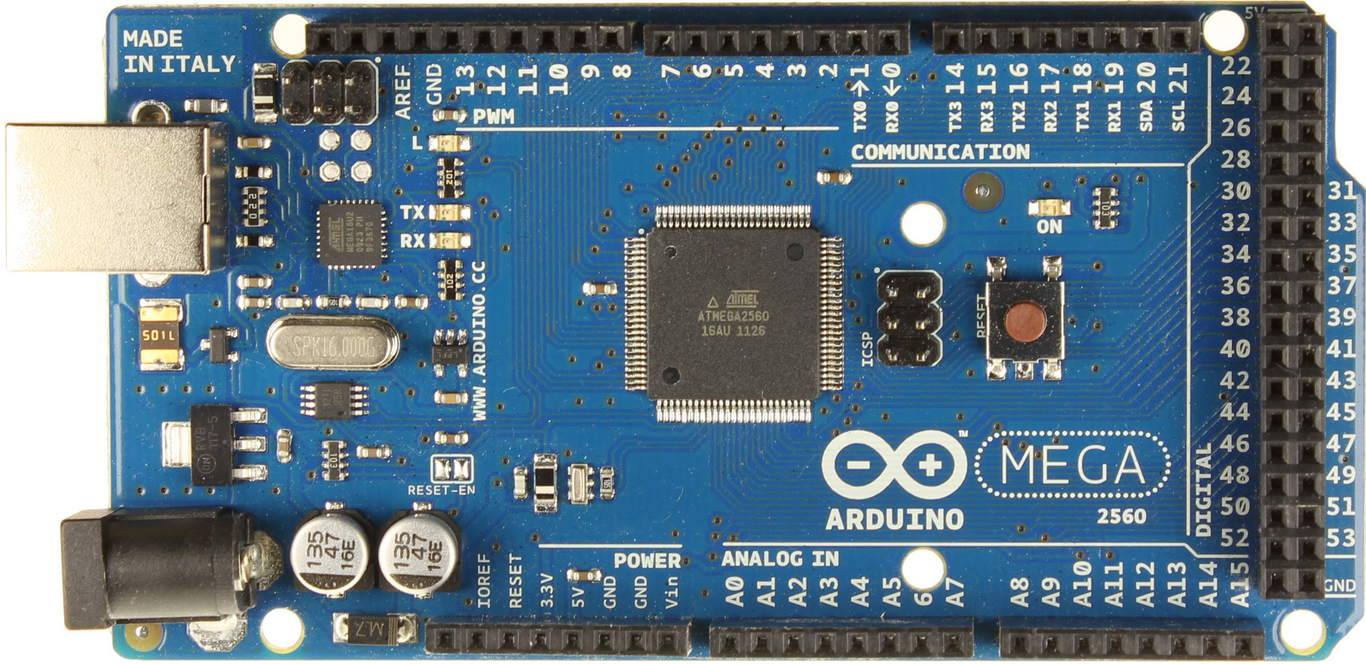
## 3.6 - Development Board



**Fig. 3.6.1 - iMX233-OLinuXino Development Board**

The team deliberated between a few development boards and initially settled on the OLinuXino iMX233 development board because of its Audio I/O, ARM9 processor and SD Card support. The OLinuXino iMX233 runs a ARM926J processor at 454 MHz and carries 64 MB of RAM. This selection was made on the basis of gaining exposure to an ARM9 system with readily available compatibility with various Linux distributions. The board also featured an audio codec that provides DAC with 99dB SNR and ADC with 85dB SNR. The team successfully obtained an OLinuXino iMX233 and installed Arch Linux ARM on it before recognizing the level of skill and work that would be required to replicate its functionality on a Printed Circuit Board designed in house. Successfully migrating any progress made on the iMX233 would have required either a sufficiently complicated PCB or a major rework of the software at the time of switching, neither of which would have been desirable. Acknowledging that this would be too large of gap for the skill levels of the team members at the time, the iMX233 was ultimately scrapped in favor of a development environment closer to what the team was capable of implementing.

The Arduino Mega was the second and final choice for a development environment for the Board Movement and Sound Engine. The rest of the software was developed on a laptop running Arch Linux and eventually transferred to the Raspberry Pi for the final design. The Arduino Mega was good development environment for the motor control because Arduino language is very much like C, a language which all of the group members were familiar with. There is a lot of user friendly documentation published about Arduino as well, including how to run servo and stepper motors, LEDs, and 7 segment displays in the Arduino environment. Arduino has a specialized servo library (servo.h) which allows the programmer to easily send pulse-width modulated signals to the servo motor while running other devices. The Arduino Mega was selected over other Arduino development boards (such as the Uno or the Duemilanove) because of its pin count. The Arduino Mega has 88 I/O pins, including 14 PWM pins. The group needed a high pin count to run all of the systems in CyberChess.



**Figure 3.6.2 - Arduino Mega (2560)**

Originally, the group believed they needed at least three serial ports, one for communication with the chess engine two for sending data to the shift registers. Because the Arduino Mega had 4 serial ports, it seemed like the best option. However, the Arduino language provides a shiftOut function that allows the microcontroller to send serial data, one byte at a time, to the shift registers without using UART communication. Because of this function, the development board actually only needed one serial port after all. Therefore, the Arduino Mega had more features than necessary, but that did not pose any issues with the development of CyberChess.



**Fig. 3.6.3 - iMX233 UEXT Modules**

*Alternatives Not Used As Development Board:*

### BeagleBoard

The BeagleBoard is an open source single-board computer that can be purchased for $125. It is capable of running a variety of Linux operating systems, including the Arch Linux system selected for CyberChess’s development. The BeagleBoard comes with 256MB of flash memory as well as 256MB of RAM. This board uses the 600 MHz Cortex-A8 microprocessor which provides more processing power than what was needed for the purpose of this project. The main reason the BeagleBoard was not pursued as an option was the high price of the platform compared to the hardware it had to offer on its stock board. The Raspberry Pi is another platform that for a quarter of the price provides the same processing power as the BeagleBoard.

### Raspberry Pi

The Raspberry Pi is a credit card sized, microprocessor-based, single-board computer that is capable of desktop-like processing. The Raspberry Pi is a powerful yet basic platform, designed specifically for educational expansion. As such, it does not come with any additional hardware components, what you get out of the box is all you have to work with. This single-board computer drives a 700MHz ARM processor comfortably at 5V and contains up to 512KB of RAM. Despite a relatively recent release in the world of technology, the Raspberry Pi already has a dedicated community supporting its use. As such, a variety of Linux distributions have pre-compiled binaries available which can be simply written to a SD card and used to boot up.

The Raspberry Pi has garnered significant community support due to its low cost and extensibility. The board costs a mere $35 and has built in access to audio, video, USB, and Ethernet. Being unable to obtain a Raspberry Pi until after the switch had been made from microprocessor to microcontroller, it never became the primary development board. However, with the new requirement of a mainframe system to run the chess and voice recognition engines brought about by making this switch, the Raspberry Pi became this integral part of the CyberChess system.

### Arduino Uno

The Arduino Uno is an open source single-board microcontroller that focuses on being user friendly. The Uno contains an 8 bit 16 MHz Atmel AVR microcontroller with 2 KB of RAM and requires a 5V power supply with very few mW. The Arduino platforms also have a wide selection of additional I/O and interface configurations to choose from. The Uno without any additional features goes for $25. With only 14 I/O pins on the Uno, additional I/O would have been required for use in CyberChess leading to the purchase of a $25 mux shield compatible with Arduino. This doubles the cost of purchasing the Uno, a strongly negative aspect of this option. While the Arduino platform combined with the Voice Control Module would have been a sufficient development environment, it did not adequately mimic the desired final environment and design specification.

## 3.7 - Magnets

Magnets come in two forms, electromagnets and permanent magnets. Although both magnets have their advantages and disadvantages for other applications, we decided that a permanent neodymium magnet would be a better choice to use for CyberChess.

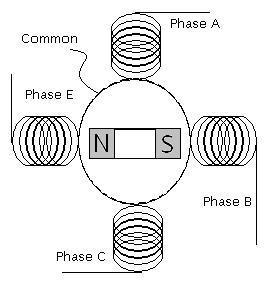
Electromagnets are basically extra big inductors wrapped around a solenoid. They require a power source to drive a current through its wiring and create a magnetic field, the same way a transformer creates a magnetic field to drive a step-down or step-up current in a power line. Electromagnets are very expensive to buy pre-made, and usually end up being too bulky when homemade--because it takes a lot of wrappings to produce a magnetic field for sufficient for the purposes of CyberChess. Because permanent magnets are typically much smaller and stronger, don’t require extra power to operate, and are perfectly capable of holding and carrying chess pieces as flawlessly as possible; permanent magnets were selected to be embedded in the CyberChess chess pieces and used in the XY-plotter to grab the pieces. Small neodymium magnets were embedded in bases of the chess pieces, while a stronger, neodymium magnet was affixed to a servo motor which lifts it to attract the magnetic chess pieces.

## 3.8 - Motors

The source of the mechanical driving forces used to create linear motion in the XY-plotter was another important consideration in the design of CyberChess. Because CyberChess is a relatively small electrical device, motor forces such as hydraulics or gas power were not even considered. Stepper and servo motors were determined to be small and strong enough to operate sufficiently in CyberChess. Every similar existing project previously researched that makes use of a XY-plotter uses the rotational motion of a stepper and/or servo motor to run the show. The physical board of CyberChess was constructed around a home-built XY-plotter which makes use of two stepper motors to control the location of the magnet parallel to the chessboard. The magnet is mounted on a servo motor to control its position in the third dimension. More details about how this was accomplished are presented in the design chapter.

Stepper motors are usually wired in one of two main wiring schemes: bipolar and unipolar. The unipolar stepper motor wiring scheme often results in a lower holding torque at low speeds. However, at higher speeds, the torque of a unipolar stepper motor holds up fairly well. A bipolar stepper motor, on the other hand, may have great torque at low speeds and much lower torque at its higher speeds. CyberChess needed the best of both worlds, so a type of stepper motor called a full coil bipolar stepper motor was used, being the optimal choice. Using this well-rounded stepper motor required a greater amount of power, but this was considered a fair trade for greater flexibility in operation.

Use of stepper motors comes with the additional consideration of how smoothly they will operate. CyberChess requires smooth motion from the stepper motors to ensure that chess pieces are not sent off course by any erratic movement. The smoothness of movement is in part a function of which step mode the stepper motor is operating in - full step, half step, or microstepping.



**Figure 3.8.1 - Basic Stepper Motor Diagram**

***Permission Pending***

Figure 3.8.1 shows the internal diagram for a stepper motor. A magnetic field is created by means of sending a current through the wirings in each phase, effectively controlling the position of the magnet in the middle. In full step mode, a current is sent through one phase at a time, pulling the magnet four times for every revolution. This kind of stepping uses little power and allows the magnet time to slow down before being pulled again. Such behavior ultimately causes the gear that is attached to the motor to move erratically. Half stepping is accomplished by having one phase go on by itself, causing the magnet to rotate, and then following this with the adjacent phase turning on while the first phase is still on. When the first phase is turned off, the adjacent phase is allowed to be on by itself before the process continues with the next adjacent phase. This creates a pull on the magnet eight times during every revolution, resulting in a steadier force during the revolution than that present in full step mode. Tables 3.7.1 and 3.7.2 below show the order of events for these full step and half step modes, respectively.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Event** | **A** | **B** | **C** | **E** | **Magnet Angle** |
| 1 | 1 | 0 | 0 | 0 | 0 deg |
| 2 | 0 | 1 | 0 | 0 | 90 |
| 3 | 0 | 0 | 1 | 0 | 180 |
| 4 | 0 | 0 | 0 | 1 | 270 |

**Table 3.8.1 - Full Step Mode**

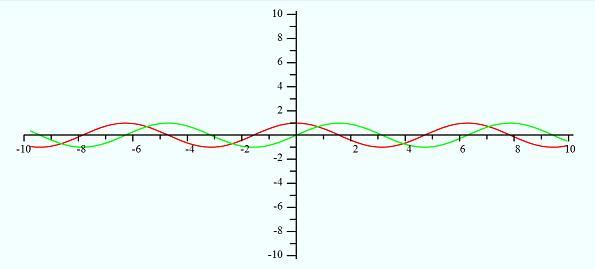
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Event** | **A** | **B** | **C** | **E** | **Angle** |
| 1 | 1 | 0 | 0 | 0 | 0 deg |
| 2 | 1 | 1 | 0 | 0 | 45 |
| 3 | 0 | 1 | 0 | 0 | 90 |
| 4 | 0 | 1 | 1 | 0 | 135 |
| 5 | 0 | 0 | 1 | 0 | 180 |
| 6 | 0 | 0 | 1 | 1 | 225 |
| 7 | 0 | 0 | 0 | 1 | 270 |
| 8 | 1 | 0 | 0 | 1 | 315 |

**Table 3.8.2 - Half Step Mode**

Microstepping is an even smoother form of stepping that breaks the currents fed into each phase into different levels. The current of a phase is incrementally increased to a peak value and then incrementally weakened while the subsequent phase begins to be incremented.

Stepper drivers with indexing capabilities can microstep at 16 to 256 steps per 90 degree angle change in the motor. A 1/32 microstepping index would control a stepper motor with about 238 different levels of current sent through the windings during a revolution. Assuming the steps occur frequently and at a steady pace, the force imposed on the internal magnet should be very constant, and thus the motion of the magnet should be very smooth.

The signals sent to the stepper motor almost trace out a sine and cosine wave (Figure 3.8.2). The out-of-phase, sinusoidal inputs cause the north and south poles of the internal magnet to constantly see the same magnitude and direction of the induced magnetic field, which in turn causes the internal magnet to continually (and steadily) experience the same repulsing force from the field. Creation of an analog sine wave is accomplished by implementing a DAC between the processor and the motor. Due to the complexity of circuitry necessary to convert a processor’s signal into a waveform that would adequately run the stepper motor at different speeds, the team decided to look into making use of stepper motor drivers that put all of this into a simple black box. As stepper motor drivers are inexpensive, it was determined that the time and money involved in constructing a driver would be at best unnecessary.



(Vac = green, Vbe = red)

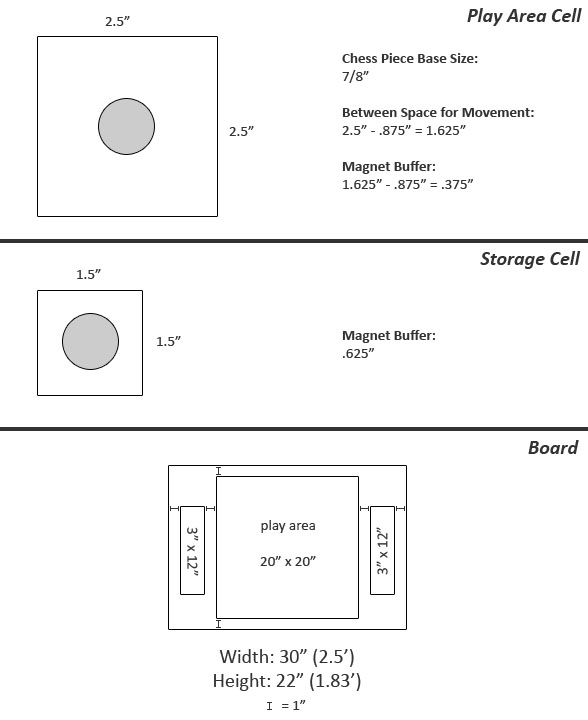
**Figure 3.8.2 - Analog Signals to Operate Stepper**

## 3.9 - Sensing

Hall Effect sensors were investigated to be used as a proximity switch on the sensor board. In recent years the reed switch has been developed to improve its quality to rival the Hall Effect sensor making performance of both sensors not a deciding factor in which would be a viable option. While reed switches were identified as being a superior option due to their comparatively lower cost and simpler integration, the ability of CyberChess to determine the physical presence of pieces was dismissed as a feature that would not be adequately used to its potential, and thus not worth the time investment. Without being able to uniquely identify pieces, either form of sensing only offered knowledge that any abstract piece was present where one was expected, leaving checking the board’s state ineffective. Additionally, the limited presence of the sensing system would not sufficiently assist the movement engine in determining where it had lost a piece if such an event were to occur. Thus, unable to justify the additional cost and work, sensors were stricken from the design.

The chess board play area, or the chess board itself, could have been either manufactured by the development team or purchased from a manufacturer directly. The board was custom built by the team to fit the special sizing specifications necessitated by the unique automation employed by the CyberChess system. The need to be able to navigate movement of any piece around any other piece necessitated sufficiently large spacing to ensure the two pieces would not magnetically interact during passage. This meant each play area cell must be big enough so that two together, with two chess pieces, still had enough room to allow a third piece to pass in between.

Using a chess piece size of ⅞”, areas were determined for initial play area cell size as well as storage cell size. These dimensions, shown in Figure 3.9.1, provide magnet buffers of .375” for the play area and .625” for the storage cell. Besides space for movement, this magnet buffer was the main variable in determining the proper dimensions. Since all of the chess pieces had magnets attached to the bottom, the magnet buffer allows for the pieces to be moved independently, since other pieces will not be attracted to the movement magnet or each other.



**Fig. 3.9.1 - Chess Board, Play Area, and Capture Area Dimensions**

## 3.10 - Materials

### The Enclosure

Once it was decided that the CyberChess playing board would be custom built, we had to decide what materials would be necessary to create the proper environment for the game. At first a glass enclosure was considered to house the whole project, the reason being that it would add to the visual appeal of CyberChess. The user will be able to see the movement system at work under the playing board and watch the mechanics behind the chess pieces moving. This approach would also include more L.E.D’s aligned along the borders of the whole enclosure to keep with the theme of the project. A glass enclosure would add a significant amount to the cost of CyberChess. Glass is not cheap and on top of purchasing the sheet of glass, additional cost will be added to employ someone to cut the glass to the specifications needed.

Wood was the main choice for building the enclosure of CyberChess because it could be handled at home with no need for special equipment or assistance and is readily available at low cost. The wood needed to be treated correctly so it did not warp with time, as this could affect the XY-plotter and full range of motion.

### The Board

Plexiglas was chosen as the main material for the chess board as it provides the clarity for the LEDs. Plexiglas was also chosen because it provides an easy surface for the chess pieces to slide on. The surface was only sufficient when the protective film was kept on the Plexiglas. The team tested removal of the film, but this created an increase in friction and thus a reduction in movement of the pieces.

Using ball bearings on the bottom of the chess pieces was an alternative to keeping the film on, making it easier for the chess pieces to move across the board. This would have required the chess pieces to be hollowed out and have the ball bearings to be secured. The magnet beneath the sensor board could have also attracted the ball bearings in the pieces, interfering with their linear motion when a chess piece is moved.

### The Chess Pieces

The chess pieces used in CyberChess were purchased fromWal-Mart due to ease. There were several choices for which chess piece set could be used for CyberChess, each could have brought a different aspect to the design of the game. It was first thought to use clear chess pieces so they would be illuminated by the L.E.D.s beneath the playing board, and these are inevitably what was used.

Another alternative was to purchase metal chess pieces. The metal chess pieces would be able to interact with the magnet attached to the XY-plotter allowing it to move across the playing board.

## 3.11 - Power Supply

The power supply uses a wall plug with an output voltage in the range of 5V to 15V with a max output current of 1A. It was determined that a bypass capacitor may have been needed between the power source and printed circuit board in order to reduce the amount of noise caused by remaining AC pulses of the current after it passes through the bridge rectifier built into the power supply. Multiple voltage regulators are connected to the power supply to help maintain the proper voltage for each hardware component CyberChess is comprised of.

Linear voltage regulators and switching regulators were each considered for use in CyberChess. Each provide their own advantages and disadvantages for powering the circuit board that is CyberChess. Linear regulators tend to be cheaper and take up less space than switching regulators while still providing very little noise in the output. The main downside to using linear regulators is the heat that is distributed when the pass transistor receives heavy current when the difference between the input voltage and output voltage is large. They are most suitable for low powered projects where a high load current would not be an issue.

A heat sink can be attached to the linear regulator to help solve the issue of heat given off but many heat sinks tend to be quite bulky or expensive. This can lead to a pricy and clustered design if many regulators are needed. The linear regulator is also only capable of stepping down the input voltage, it is not able to produce an output voltage higher than the voltage used as input. The formula below is a general equation used to help determine if a linear regulator is wasting too much power.

Power wasted = ( Vi- Vo) Iload

If the linear regulator is giving off more than a few watts of power it is advised to use a more efficient switching regulator in its place.

Switching regulators work by rapidly alternating the pass transistor between an on and off state sending the input voltage as small pieces and moving them to the output. This separation of energy causes the switching regulator to give off very little heat enabling it to handle much larger load from higher voltages.

When higher power is needed switching regulators have the advantage over linear regulators because additional components are not needed to remove heat. However, placement of switchers on the printed circuit board is important, if the loop gain or phase of the switching regulators is altered by interference the regulator can become unstable and oscillate.

The biggest factors to consider when deciding which regulators will be used in CyberChess are how much space the regulators will take up on the PCB and how much heat will dissipate from the components. To avoid complications with overheating hardware, it is advised that the power supply and regulators be external to the main PCB.

## 3.12 - Printed Circuit Board

When building a printed circuit board, there are many small things that could negatively affect the performance of the design. The unwanted factors in the PCB design are known as parasitic elements. It is imperative that the proper techniques are implemented to avoid these parasitic elements. Many things come into consideration when dealing with electronic parasitics: capacitance, resistance, inductance and temperature are the most common unwanted occurrences.

Parasitic resistance can occur when a trace is too long. The resistivity of a copper trace, for example, is 1.7 μΩ·cm. For a copper trace of a constant area, a longer trace will create a higher overall resistance in the line. For large currents and voltages, this may not be a huge problem, but PCB electrical characteristics are often are small as well. Some MOSFET transistors which utilize leakage currents to run their operations may have currents as low as a few nanoamps. With such small currents running through the traces, a small change in resistance could change the current by a relatively large factor. A system designed to respond to small currents may not react so well to the error. Another unwanted effect of parasitic resistance is the voltage drop. If two neighboring traces experience a large enough voltage drop, the coupling capacitance could cause problems as well.

This unwanted capacitance is called parasitic capacitance. One of the main problems with parasitic capacitance has to do with amplification. If the capacitance it near an amplifier, the error could be largely increased. A large enough parasitic capacitance could create a path of feedback and destabilize the electronic network. At low frequencies, parasitic capacitance can usually be negated. The problems mostly occur at higher frequencies. Inductors are known to create parasitic capacitance within themselves. While an inductor is charging or discharging, the windings have small voltage differences between each turn. At high frequencies, an inductor experiences rapid changes, and therefore the potential between the windings may be greater, causing a larger capacitance. This can slow down the performance of a network relying on the inductor.

PCB stacking is another aspect of the Printed Circuit Board that must be taken into consideration. Supplying the board its ground and voltage source via traces is not recommended for designs that require good performance. A few problems can arise due to varying lengths of trace between different components and the ground or supply. As mentioned earlier, resistance in a trace gets larger as the trace becomes longer. So if two pins should be grounded, but one is connected to ground by a significantly longer trace than the other, there may actually be a voltage difference between the two. In effect, one of the pins won’t actually be grounded. And the operation of the system may suffer because of this. Uniformly charged layers can be created by means of stacking the PCB. The vias extending from these layers to the component pins on the top layer of the board will be about the same distance, minimizing the potential difference between two pins stemming from the same source.

A few common stacking configurations are the two-layered, four-layered, and six-layered PCBs. The two-layered board is usually set up with a signal layer above a ground layer, which are both considered conductive layers. The conductive layers are separated by a dielectric layer made of material called prepreg. Prepreg is basically fiberglass mixed with epoxy resin. Below the ground layer is the core, which is made of fiberglass epoxy resin also. In many cases, another dielectric layer, made of soldermask, will be added above the signal layer. All in all, a 2 layer PCB has two conductive layers as well as 2 or 3 non-conductive layers. In a 2 layer board, the parasitic resistances due to bad grounding can be pretty much eliminated. However, the supply voltage still has to be distributed at the signal layer, through copper traces. This leaves some improvement to be desired.

The four layered PCB is set up with the first conductive layer being the signal, the second layer as the ground, the third layer is the supply voltage, and the fourth layer is another signal layer. The supply and ground can reach the signal layers through vias, just like in the two-layered boards. The signal layers can reach each other via through-holes. Through-hole technology allows for connections to be soldered on the bottom side of the board, between components that are sitting on the top layer. Similar to the two-layered PCB, the four-layered PCB has dielectric layers separating the conductive layers, as well as soldermask on the outermost layers. This PCB design resolves a lot more issues than the two-layered design.

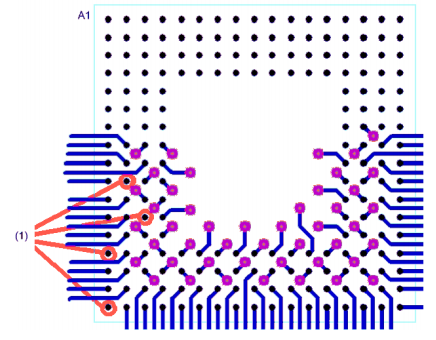
The six-layered PCB adds two signal layers between the voltage supply plane and the ground plane. More layers can be used in a PCB to reduce parasitic problems produced within the signal layers.

### PCB for BGA

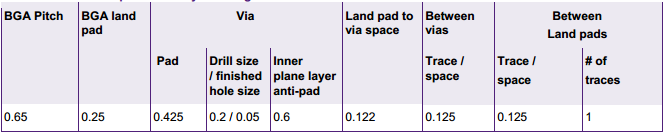
An early iteration of CyberChess would have required the use of a fan out technique to accommodate a pin-heavy Ball Grid Array (BGA) microprocessor. The use of BGAs has become quite common for applications requiring medium to high pin-count IC packaging. In researching fan out techniques it was also discovered that we must pay close attention to additional areas such as solder paste chemistry, reflow solder profile and solder paste stencil etching, all of which can affect the outcome of our printed circuit board.

Since the BGA balls are arranged in matrix fashion, routing all of them on a single layer would be nearly impossible. Through the use of fan-out vias, or escape vias, this problem can be avoided by allowing the routing of the balls on other layers. Currently we have samples of the ZCE AM1808 package which contains a BGA with a ball pitch of .65mm while the alternative package, the ZWT, has a ball pitch of .80mm. These two packages create different specifications for our vias, land pad to via space, trace space between vias as well as trace space between land pads.

For our purposes, the fan-out pattern for our package centers each via within the space between four adjacent BGA land pads. The vias are placed 1.3 mm apart, with every other location being skipped, and alternating them between adjacent rows. A partial example is shown below in figure 3.12.1, and the layout tool design rules are shown below in table 3.12.1.



**Fig 3.12.1 - .65mm BGA Fan-out Technique**



**Table 3.12.1 - .65mm BGA layout design rules**

## 3.13 - Operating System

In developing CyberChess, the development team sought an operating system that was lightweight, flexible, and, most importantly, readily usable. CyberChess required an operating system that was as bloat-free as possible so as few things as possible would get in the way of the system running smoothly. As all modern, featureful open source operating systems are UNIX-based, with the most powerful options being based on the Linux kernel, the development team reviewed a selection of popular Linux distributions which offered support for the ARM architecture. Fortunately even after moving away from the iMX223 to using ATMega-based microcontrollers, this research remained relevant, allowing straightforward integration of the Raspberry Pi as the mainframe without any additional learning curve.

### Arch Linux ARM

Arch Linux is a Linux distribution based on the idea of providing simple, accessible customizability to the end user. The majority of system customization is maintained in simple textual configuration files. Pacman, the main package manager of the distribution, provides simple, powerful functionality in locating and installing binary programs of any of thousands of packages available in the officially supported repositories. The Arch User Repository (AUR) further extends this by providing access to even more packages that are voluntarily maintained by users, and access to this repository is simplified by any of a number of programs that act as wrappers around Pacman, enhancing its functionality.

Arch Linux ARM is developed and maintained by a separate team from that which actively works on the main Arch Linux distribution. However, these teams cooperate, resulting in Arch Linux ARM being a true port of the distribution to the ARM CPU architecture. Arch Linux ARM is fully supported on any processor that supports ARMv5TE or higher, including the OLinuXino iMX233 and the Raspberry Pi.

Arch Linux ARM ultimately became the final choice for CyberChess due to the benefits of having significant previous experience using the Arch Linux operating system on a day to day basis. The extensive documentation present on the official Arch Linux website was quite useful, thoroughly addressing the few issues that arose in the context of the CyberChess project.

### Debian ARM

Debian has always been known to support more hardware architectures than other Linux distributions and includes many packages specific to each ARM version. For the purposes of CyberChess’s internal applications, arm-Linux-gnueabi in Lenny in the armel section would be a necessity, as it supports ARMv4t and up. The most advanced version of Debian supports ARMv7 and up as well as the Thumb-2 instruction set.

Debian was ultimately not used mainly for sake of the team having a greater familiarity with the Arch Linux environment. It is a very solid environment in its own right and very well could have served as the core operating system of CyberChess.

### Ubuntu ARM

Ubuntu is one of the most popular desktop Linux distributions, being designed to be readily accessible to new users with no previous experience with POSIX environments. It is built upon Debian, and thus bears a certain underlying resemblance to that distribution. Being designed with ease-of-use in mind, Ubuntu hides much of the system configuration from the user, giving them ready access to less powerful GUI-based controls while restricting further access. This was one of the main reasons for avoiding the Ubuntu ARM distribution in the creation of CyberChess - despite much of the team having had experience with Ubuntu, its reliance on bloated user-friendly GUIs and tendency to hide background mechanics left it unsuitable for the lightweight environment desired on CyberChess’s mainframe.

Ubuntu ARM is a complete Linux ARM distribution with thousands of packages available for both desktop and server. The Ubuntu Netbook Remix (UNR) brought about a very promising feature that made use of the Enlightenment Foundation Libraries to power a 2D user interface with features similar to that of a 3D interface. UNR went away with the release of Ubuntu 11.04 as Ubuntu ARM and Desktop were merged. The current Ubuntu ARM port supports ARMv7 and up.

### Gentoo

Gentoo is one of the most flexible Linux distributions, officially supporting at least 10 CPU architectures, including ARM. The distribution is based on a philosophy of giving the user extreme control over optimizing their environment. While binary packages do exist and can be installed on a Gentoo system, the most common method of program installation involves using the Portage software distribution system to download sources and compile them client-side. These programs are optimized by the user’s use of various compilation flags to indicate to the compiler what features are and are not necessary. In this manner, the user can optimize individual programs to contain nothing more than what is needed for their purposes. While this gives the user explicit and powerful control over optimizing the system, it comes at the price of relatively time-consuming upkeep - every update to a program requires complete recompilation of that program, any change of USE flags (universally applied compilation flags that can a user can override as needed) requires recompilation of all affected programs.

### Android

Android is a Linux-based operating system designed for use in devices like smartphones and tablet computers. The main platform for Android is the ARM architecture and would have been an excellent fit for CyberChess if the project had made use of a more advanced processor such as the ARM Cortex-A or Cortex-A8. While Android was certainly viable, performance would not have been close to the level desired. If CyberChess had made use of Android it would probably have been necessary to modify the hardware to include a DSP to offload all multimedia processing to this processor.

## 3.14 - Shells

Shells provide user access to the functionality of the operating system’s kernel, allowing for direct control over the system. A number of shells exist for interacting with the Linux kernel, including the historically beloved Bourne Shell, the Bourne Again Shell (Bash) which builds upon it, and the recently popular Z shell (zsh). While CyberChess ultimately operates without the users having access to the underlying shell, it was necessary to pick a strong shell to facilitate productive development. Additionally, the system still makes use of shell scripts to automate behavior at boot up and shut down, a vital element of the final product to implement well.

The Bourne Shell is effectively the canonical shell of all UNIX-based operating systems, having been the primary shell of UNIX Version 7. Any modern \*NIX machine can be expected to have support for executing shell scripts written for the Bourne Shell, and many modern shells, including bash and zsh, provide full support for executing these scripts with little modification. While the Bourne Shell has undeniable historical value to the development of \*NIX systems, its limited functionality and low availability of abstractions made it an impractical choice for the CyberChess system.

The Bourne Again Shell, more commonly simply called Bash, was created as a replacement for the Bourne Shell and has become the standard shell included on most Linux distributions. With support for a greater number of features, bash is effective not only for interaction with the kernel, but also for basic prototyping of program ideas. As the default shell of most Linux distributions (in fact, most will simply use a symbolic link to have the path for the Bourne Shell instead use bash), bash was a strong and obvious choice for the shell to use. The final CyberChess product made use of bash scripts to initiate its core functionality.

In the actual development environment that was formed on the Raspberry Pi, the team made use of zsh, another shell designed as an extension and reworking of the Bourne Shell taking inspiration from various shells that have both historically built upon it and competed with it. Zsh offers complete compatibility with Bourne Shell scripts when invoked as /bin/sh as well as extreme customizability throughout all of its features. These additional features were not a key part of developing CyberChess, but the small niceties made for a more pleasant development experience.

## 3.15 - Speech Recognition

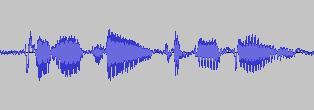
The main component of the project that makes CyberChess stand out from other automated chess games is the speech recognition that serves as input to the other engines. CyberChess required an independent voice recognition system in order to recognize and process voice commands. PocketSphinx, an open source voice recognition engine produced at Carnegie Mellon was identified as the most efficient choice for the project. Being written strictly in C and open source, the project was readily integrated into the CyberChess system.

### How Speech Recognition Works

Understanding speech recognition first requires understanding of the nature and structure of speech. Speech is a sequence of stable states mixed with dynamically changed states. Throughout this sequence, similar classes of sounds called phones and diphones (parts of phones between two consecutive phones) create words. From a software perspective, sub phonetic units, different substates of a phone, are often emphasized. Usually three or more regions per phone can be found. Phones create subwords like syllables and subwords form words. Words and non-linguistic sounds like breathing, “um” and “uh”, cough, etc form utterances, the audio chunks between the pauses.

To recognize what was said, the audio waveform is split into utterances and paired with the closest match. An example audio waveform of “PB2 to B3” is shown below in Figure 3.15.1. First a feature vector, typically taken from 10 milliseconds of audio and composed of 39 numbers, represents the speech in a comparable form. Then a model, a mathematical object that gathers common attributes of the spoken word is compared against and the best model used.

Three models are used to find the match: an acoustic model, a phonetic dictionary, and a language model. The acoustic model is the acoustic properties of each senone. The phonetic dictionary contains a mapping from words to phones. The language model is used to restrict word search. PocketSphinx was chosen specifically for the reason that it provides us with the acoustic model for the English Language as well as the tools to setup a language model and phonetic dictionary specifically for the CyberChess system.



**Fig 3.15.1 - Audio Waveform of “PB2 to B3” recorded using Audacity**

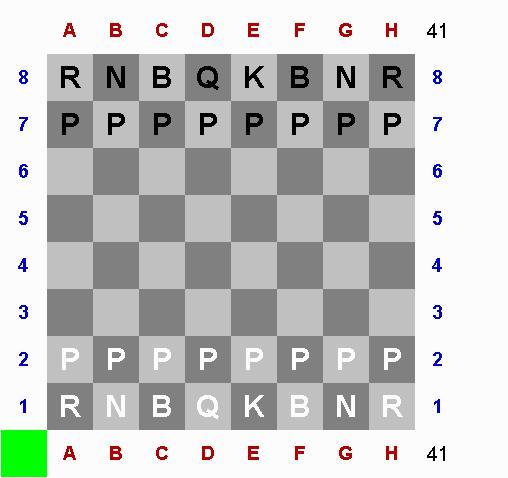
The language model and dictionary files are created using an online converter provided by the CMU Sphinx Toolkit. To create them, one must first create a *Corpus.txt* file which will contain every possible phrase or command, one per line. This file is uploaded via a web browser and converted into a randomly generated 4 digit .lm and .dic file. These files are then referenced at runtime to compare inputs from the microphone. CyberChess made use of such a dictionary with a very limited vocabulary to ensure maximum chance of recognizing commands.

### Problems with PocketSphinx

In testing the language model and dictionary files for CyberChess, it was discovered that PocketSphinx had some difficulty in recognizing the difference between the letters B, D, and E, almost always defaulting to B. Valid solutions included retraining the acoustic model with a large sample of accurate recordings or altering the language used to communicate with CyberChess such that this confusion would not appear. Ultimately, the decision was made to alter the coordinate system to use the NATO phonetic alphabet in place of letters for files.

Retraining the acoustic model would have required recording of each command and word at a sampling rate of 16KHz in mono with single channel. These recordings would then generate acoustic feature files which would have been tweaked until the speech recognition engine was better at recognizing the different letters. Since PocketSphinx worked near perfectly for CyberChess out of the box besides this small problem, adapting the acoustic model would have required a lot of unexpected work.

Using words in place of letters was chosen as a simple solution that would alleviate this problem, at the cost of having a slightly unnatural syntax for user interaction. In order to give an additional level of redundancy to the system for checking that user input was heard correctly, CyberChess has the standard move syntax involve the name of the piece being moved, allowing the system to check whether the space named contains the piece named.



**Fig. 3.15.2 - Complex Naming System**

### Additional Speech Recognition Problems and Research

An additional hurdle CyberChess must regularly deal with is use in loud or noisy environments. While games of chess usually occur in a fortuitously quiet setting, CyberChess still needs to function with apparent background noise. To help counter this problem, Bluetooth Headsets, Cell Phones, wired headsets, and standard microphones were investigated as possible solutions.

Bluetooth headsets would have required incorporation of a Bluetooth module into the board, an additional facet that was not considered worth their built-in background noise reduction. A cell phone based solution could have been accomplished wired or wirelessly, either through an auxiliary cable, Wi-Fi, or Bluetooth. A wireless implementation would have again required addition of more hardware and a wired implementation would have required additional software for the cell phone in order to be able to process the data.

As this additional work was seen as not worth the minor benefits (PocketSphinx still performed well enough in relatively loud environments), standard microphones were determined to be sufficient for the purposes of the project. In the final prototype stage, Rock Band microphones were utilized, offering sufficient services as long as the microphones were held close to the user when they were speaking. In future iterations, wired headsets would be a welcome addition, offering the user a hands-free option that keeps the microphone close to the user’s mouth throughout gameplay.

### Speech Recognition Alternative - Sphinx4

Sphinx4 comes from the same open source project at Carnegie Mellon University that created PocketSphinx. Sphinx4 is written entirely in Java and serves cloud-based systems with deep interaction with NLP modules, web services and cloud computing best. Since CyberChess relies on live speech recognition embedded into the Linux platform, Sphinx4 was rejected as being a less useful choice.

### Speech Recognition Alternative - eSpeak

eSpeak is an open source speech synthesizer that was developed to handle multiple languages and run in either a Linux or Windows environment. One of the main features of eSpeak is the small amount of memory required for the software with all of its data being stored in 1.4MB. eSpeak is available in either a command line program or a shared library format to be used in another program and is written in the C programming language. Not only does eSpeak handle text to speech but it also supports text to phoneme codes as well, allowing it to be integrated into another speech synthesizer engine’s acoustic modeling tools.

### Speech Recognition Alternatives - Android

Another alternative to PocketSphinx considered in the development of CyberChess was use of Android and its built in library for Speech Recognition. The Android SDK can integrate speech input with an application that uses the RecognizerIntent. This intent prompts the user with a “Speak Now” input and sends the data to Google’s servers to be processed. Currently, Google can provide speech recognition for English, Mandarin Chinese, and Japanese.

Using the Android SDK to provide speech recognition would have required use of Android as the core Linux distribution in addition to adding a Wi-Fi module to connect to Google’s servers. Alternatively, Android Smartphones could have been utilized to take advantage of their built in connectivity to process commands, but this would have required additional software written for the phone.

### Speech Recognition Alternatives - Windows

Both Windows Vista and Windows 7 come with built in Speech Recognition software. It has support for dictation of documents and emails as well as voice commands for nearly every operation on the computer including starting and switching between applications, controlling the operating system, and filling out forms on the web. Windows Speech Recognition comes with support for English (US), English (UK), German, French, Spanish, Japanese, Chinese (Traditional), and Chinese (Simplified). It also modifies itself to accommodate both headset and desktop microphones. The use of Windows Speech Recognition would have required a Windows Environment to be tied into the CyberChess system. This could have been accomplished through use of a more advanced mainframe, but would have taken away from the presentation of maintaining everything within the CyberChess enclosure. For this reason, Windows Speech Recognition was not chosen.

## 3.16 - Board Representation (Logical)

### Language

Software production always requires the programmer to pick the best tools for the job. The most important tool to the project is the programming language itself - each language provides different benefits and, unfortunately, limitations. Each software element of CyberChess could have readily benefitted from a different language, and thus inspection of various languages was imperative. Looking towards being able to use benefits of previous knowledge, C, Common Lisp, Java, and Python were most heavily investigated. Cursory research of other alternatives included looking into C++ and JavaScript, two of the most popular languages for software development and web development respectively.

All languages that were looked into are relatively well known throughout the world of software, and likely need little introduction. C, as one of the oldest and most prolific system programming languages, is universally recognizable as a staple of programming. It offers the benefit of speed, with code compiled directly down to native machine code and control over the lower levels of execution placed directly in the hands of the programmer. Operating at this low level allows for heavy optimization, but comes with the responsibility of manual memory management - unlike many of its successors, C expects the programmer to allocate and free memory as needed, having no concept of garbage collection.

C’s widespread use has also resulted in plentiful documentation of the various features of the language, as well as a ready supply of examples of implementations of various data structures and algorithms. Use of C for large projects, however, can quickly become a challenging task, as the language is still very low level - it compiles to efficient machine code because it is quite close to it, acting effectively almost as shorthand. This means that many concepts that other languages have abstracted away - a simple example being iterating through and acting upon the elements of a linked list - must be implemented by hand. While other libraries beyond C’s standard library do exist to fill in missing functionality, having such additional external dependencies is not seen as desirable for a project intended to be lightweight.

In direct relation to CyberChess, C provides little in the way of abstractions to aid in creation of the software, but it does give the programmer extensive control over low-level operations. Many efficient implementations of chess programs make use of such operations, playing around at the bit level to minimize memory usage. As a procedural language, C has no built-in support for object-oriented programming - a paradigm which may seem fitting for creating a virtualization of chess. However, such a paradigm can be affected with creative use of structs, data structures that can contain any combination of data types, including other structs, and function pointers, variables that refer to functions by their memory location. This involves much more effort and intentionality than a language with this functionality built in, but the benefits of efficiency and having unhindered access to low-level mechanics could well be worth the added complexity. In an effort to make CyberChess’s chess engine as efficient and low-level as possible, an object-oriented paradigm was dismissed in favor of extensive use of bit-twiddling. With this paradigm, the direct access to memory provided by C made it an obvious choice. Additionally, the ATMega line of microcontrollers are programmed in a slightly customized variety of C/C++, meaning the entirety of the code could be (and was) written in a single language.

A variety of other languages were strongly considered for implementing CyberChess’s software, particularly the chess engine. Common Lisp sits on the other end of the spectrum as a language that has a plethora of abstractions ready for the programmer’s use. In its interactive form, Common Lisp makes use of a rather advanced version of a Read-Eval-Print Loop, often referred to as a REPL, in which code is first read, then evaluated, then the output is printed. While this does not sound overly revolutionary, and a number of other languages have such a construct, it proves to be quite powerful in Common Lisp. Programmers can implement reader macros to alter the behavior of the Read part of the REPL, literally changing the syntax of the language to meet their needs. Before code is evaluated, user-defined macros expand to write entire sections of code for the programmer based on a template they previously defined. As a Lisp dialect, the code is composed entirely of S-expressions, nests of lists infamously enclosed in parentheses.

This uniform representation of code and data allows macros to flawlessly nest within the code itself with no worry of the code breaking. Mark Jason Dominus, a prominent Perl programmer writes about Common Lisp’s use of macros as the main form of assignment - setf, the assignment operator, is itself a macro that not only expands to a call to the appropriate function, but can be readily updated to reflect assignment to a data type constructed by the programmer. This flexibility and the accompanying expressiveness is the core strength of Common Lisp - Peter Seibel’s *Practical Common Lisp* demonstrates the creation of a unit testing framework in a total of 26 lines of readily understandable code. Unfortunately, this power comes with some serious drawbacks - compiled Common Lisp applications tend to be extremely large, as the entire language must be present within the application. Their performance is not up to par with most compiled languages. Common Lisp also suffers from its own flexibility - the community surrounding it sees frequent mentions of the fact that the triviality of implementing something that is good enough for a single user’s purposes has resulted in very few libraries being produced that fit any more generalized purpose.

Common Lisp does not offer simple access to lower level operations and hides all memory management in its abstractions. As such, the efficiency of any program written for CyberChess would have effectively fallen to the whims of the compiler, tempered by those suggestions the programmer is permitted to make. The expressive power of the language, which includes a fully formed object-oriented suite, known as the Common Lisp Object System (CLOS), was certainly a desirable feature. Considering the limitations of final product existing in an embedded system, this power was not worth the inability to easily influence the space efficiency of the program.

Java is often considered to be in the same family of languages as C, with heavy influence visible in its syntax. While it supports various paradigms, it is often used for its solid support of object-oriented programming. Java is a memory-managed language, with a relatively advanced Garbage Collector (GC) that progresses objects from a young generation to an old generation which is checked for collection less often to improve efficiency. Most of the memory management is hidden from the programmer and low-level operations are not as readily available as in C, nor quite so obscured as in Common Lisp.

While some compilers can compile Java down to native machine code, the standard practice is to compile to an intermediate bytecode which is run on the Java Virtual Machine (JVM) as a means to make these compiled files universally usable. While this is useful for applications that are expected to proliferate throughout a number of machines, CyberChess was designed with the software and hardware hand in hand. Thus, the benefit of universal usability was effectively moot - while it may normally be nice to be certain that the program will act exactly the same regardless of the production environment not being identical to the final deliverable, CyberChess’s need to target only a single platform made this an unimportant aspect of language selection. Unfortunately, as virtual machines can never run quite up to speed with their physical counterparts, the speed loss this universality would have required persists. Just-In-Time (JIT) compilers attempt to offset this by caching code and compiling it to native machine code right before it is to be executed, and good JIT compilers have begun to approach the speeds of natively compiled code. Java does offer an extensive set of libraries, providing excessive functionality to the programmer. The object-oriented nature of Java is certainly appealing for the design of a chess engine, but the speed lost to running on the JVM without Just-In-Time compilation was deemed to be too great to be worth investing in.

Python is another multi-paradigm language that has steadily been gaining popularity. While it is most often used as an interpreted language, it can, like Java, be compiled to intermediate bytecode and executed on a virtual machine. Python was designed to be highly expressive and powerful, with simple, English-like syntax representing large abstractions. It is often informally described as “batteries included” - it features an extensive standard library that provides functionality beyond that of any of the other three languages described above. Despite *The Zen of Python*’s (PEP 20) advice that “There should be one - and preferably only one - obvious way to do it,” multiple implementations of chess engines in Python can readily be found. The expressivity and ready supply of resources make Python a strong contender for use in designing the virtual implementation of the rules. The main drawbacks that resulted in its dismissal were again speed and the lack of access to lower level operations.

C++ is popular for software design due to its ability to compile down to native machine code, like C, with support for numerous paradigms, similar to Java and Python. It is effectively, though not actually, a superset of C, and thus most valid C programs compile and execute the same in C++. Programs written in C++ are roughly equivalent in execution time to their C counterparts and C++ is often used as the standard to be compared to in benchmarks between languages. The main detractor for use of C++ in CyberChess was lack of prior experience - while learning a new language is always desirable, it can be a trying activity in the midst of creating a project. The final design of the chess engine was such that C++’s additional features would not have drastically simplified any of the code.

With every modern web browser supporting a full implementation, JavaScript is considered to be the native language of the internet. It offers many of the same features that Java, Python, and C++ provide. The main advantage of its use would have been easy potential to bring CyberChess’s engine to the web. As this was not a major design goal, it was not a strong deciding factor, and while JavaScript is certainly a full-featured and growing language, it did not offer anything important the other considered languages lack.

### Board Data Structure

Research into board representation was initially in the form of attempting to implement the board in various ways to find those which were the easiest and most flexible to work with. A simple array implementation was investigated and, while relatively easy to deal with, was ultimately dismissed as being unnecessarily inefficient. Keeping track of a set of piece positions without an actual virtual representation of the board was attempted as well and found to work decently, but having a virtual board to interact with was found to be more natural.

Early on, one of the key design decisions for the chess engine within CyberChess was to aim for the most compact and efficient processing of the chess game itself. The engine should of course be a minor processing concern, and as such an efficient data structure for board storage was required. Any such data structure should make checking the validity of generated moves direct and non-intensive. Various data structures have been widely used in other chess engine projects, but two implementations stood out due to usability, efficiency, and popularity - the 0x88 method and bitboards.

0x88 represents each piece location as a single byte, with file and rank corresponding to the first and second nybble (or vice versa, as the programmer desires). This effectively allows reference to four adjacent boards, with only one being the valid board. While this may seem to be wasteful, simply logical-ANDing a location with the hexadecimal value 0x88 will indicate whether a test location exists on the valid board - if either of those bits are set, the location is in a file or rank on one of the adjacent boards. This simplifies both programmatic generation of moves and checking the validity of moves being requested by a player.

Bitboards store a current board state in a collection of 64-bit integers, using each bit as a flag for the presence of a piece on a one-dimensional representation of the board. This provides the benefit of being able to perform a wide variety of operations with simple, efficient bit-shifting. Unfortunately, the benefits are reduced in non-64-bit processors, due to their inability to handle entire bitboards each clock cycle. Bitboards also have a high time versus space tradeoff - the most time-efficient usage relies on making use of a static set of pre-generated bitboards rather than dynamically creating them as needed. Simply having bitboards representing all possible movement vectors takes 32 kibibytes of storage, with numerous other possible sets adding additional speed benefits at the cost of more memory. The benefits of bitboards are seen most in applications that need to generate many moves, such as chess AI systems.

Ultimately, the virtual board within CyberChess made use of a custom representation inspired by attempting to make a hybrid of the above two implementations. A single board is represented as an array of eight 32-bit integers, easily handled by a 32-bit processor. These integers are divided into eight nybbles, each corresponding to a single spot on the chessboard. These four bits are divided into a single color bit and three bits to identify the type of piece present. This representation loses out on the ease of determining if a move is to a valid spot on the board offered by 0x88, but due to the overwhelmingly passive nature of the final engine, this is not an issue. Requests for move locations are limited on multiple levels throughout the CyberChess system and thus a request for a location off the board would require an extreme anomaly.

### Algorithms

The final software implementation of the rules of the game is responsible for checking that any given move is both valid and legal. There are various means of performing these checks, but the implementation within CyberChess seeks to minimize time spent on them.

Checking move validity is simply a question of whether the destination exists on the board. Determination of this is effectively built into the board representations addressed above - 0x88 board representation requires only a simple logical-and operation to determine a location’s validity, while bitboards and the eight integer board simply do not permit invalid moves by nature of only ever operating on the valid board. As such, there are no real clever algorithms to further enhance the offerings of these representations.

Checking move legality is a more involved process, requiring knowledge that the desired move obeys the piece’s movement patterns, is not in any way obscured, and that it does not result in breaking any rules of the game. Verifying movement patterns is fortunately simple for many pieces - means of doing so are addressed in table 3.j.1 below. All listed methods require very little processing time, relying heavily on checking equality of values and simple mathematical operations, two things processors are designed to be very efficient with. These methods were built into CyberChess’s chess engine and cooperate smoothly with the eight integer board.

Bitboard representation would simply logical-and the requested move with a representation of valid moves for the respective piece type at its current location (an array of such bitboards would take 3 kibibytes of memory). In order to ensure the game progresses properly, an attempt to move to the same space the piece currently occupies must register as illegal.

|  |  |  |
| --- | --- | --- |
| Piece | Movement Pattern - Patterns assume all spaces are unblocked unless otherwise noted. | Programmatic Method |
| Pawn | The Pawn has the most complicated movement pattern to define, being dependent on its location and the type of move:   * At any spot, the pawn may move forward one space in its file. * At its initial spot, the pawn may move forward two spaces in its file. * A pawn may only capture a piece that is in one of its adjacent files, one rank ahead of it.   Additionally, what is considered to be forward differs based on color, and thus color must also be addressed. | The proper direction of a Pawn is determined with a conditional assignment based on the color identified in the Pawn’s nybble. With this knowledge, the system simply runs through a series of equality checks based on adding appropriate values to the rank and file. |
| Rook | The Rook moves only in straight horizontal or vertical lines. | Simply XORing the results of checking that the file and rank of the source and destination are equal returns a truth value corresponding to the validity of the move. |
| Knight | The Knight moves in an L-shaped pattern, advancing two spaces in one direction and one space perpendicularly. | A naive approach might sum the absolute values of the differences between the destination and source file and rank and equate this to the value 3. This would allow the Knight to move three spaces in straight lines. Instead, the CyberChess engine checks whether the sum of the squares of these distances is 5, ensuring that only moves matching the Knight’s movement pattern register as valid, while also removing the need for the slightly more intensive operation of finding absolute value. |
| Bishop | The Bishop moves only in straight diagonal lines. | Equating the squares of the differences between destination and source files and ranks returns the validity of the move. This is supplemented by a check that these distances are not 0 to ensure the piece is actually moving. |
| Queen | The Queen moves in straight lines in any direction. | ORing the results of the move validity for a Rook and Bishop is all the engine does to properly determine the validity of a move for the Queen. |
| King | The King moves a single space in any direction. | Checking that the square of the file difference and the square of the rank difference are each less than or equal to 1 determines the validity of the move. This is, like the Bishop, be supplemented by a check that the distances are not both 0. Castling is handled by a separate function, but is initiated by a move in which the King moves 2 spaces in its home rank. |

**Table 3.16.1 - Piece Movement Patterns and Algorithms**

Checking that a move is not obscured requires additional computation. With a bitboard representation, bitboards containing 1s at each spot between two points and 0s elsewhere would be available, taking up a total of 32 kibibytes. Using a logical-and operation with the current board state would indicate whether the requested move is blocked. In a 0x88 representation, the spaces between the two spots must be iterated through in a simple loop, checked one by one. While this is a comparatively slow process, the loop will never need to check more than 6 spots, which is still a rather inconsequential processing load. This latter option was selected, being simple to integrate with the eight integer representation. This loop is actually built into the function that determines the validity of Bishop and Rook moves, eliminating any additional function call overhead.

Finally, the move’s legality relies on its obedience of the more abstract rules. The availability of an en passant capture is indicated on the board using a piece identification that is not representative of any actual, physical chess piece. These markers are set on the board any time a pawn advances two spaces and are cleared during each subsequent turn to ensure that the capture is only valid for its intended duration. The validity of castling requires that the King and the castling Rook have never moved and that the spaces between them be empty. Knowledge of the movement of one of the two pieces is maintained by simply having a 4-bit word that acts as flags for the availability of each possible castling move (white and black, king and queen side), mimicking the storage of this knowledge in Forsyth-Edwards Notation (FEN). Every move, the engine determines whether this flag should be modified and unsets the relevant bits if needed. Castling, like any movement of the King, also requires that no space the King would be moving through be in check, the most complicated of the abstract rules to address. This actually required a reworking of the function which determined whether a King was threatened - while it initially directly accessed the location of the Kings, it had to be rewritten to allow an empty spot to be checked for threats from only a single color. This was a simple rewrite, but was only recognized late in the development process.

Verifying that a space is not threatened requires ensuring that no enemy piece can reach that square in one move. For bitboards, this requires a series of logical-and operations, performed on the bitboards representing the current location of each piece type of the opposite color and that type’s movement pattern starting from the square in question. In the 0x88 and eight integer representations, this can be accomplished for any arbitrary spot in two main ways, and can be checked after each move in one additional, less processor-intensive way. The most demanding method starts at the space in question and attempts every movement pattern available from that spot - up to 35 positions, each of which must be dynamically generated. If knowledge of the location and type of all enemy pieces is maintained, it is instead possible to iterate through this list and simply attempt the move using each piece as the start location. This requires additional information storage - namely, the list being iterated through must be maintained and updated after each move.

While movement of the King requires a thorough check like those described above, movement of any other piece can be simplified to checking along the vector that continues in the direction from the King to the other piece’s start location. If this vector is not a straight line on the board, the move does not have the potential to put the King in check. Otherwise, simply proceeding along that line will encounter any piece that could potentially render the move illegal. Similarly, it must be determined if a player’s move leaves their opponent in check. This is readily handled by attempting to move from the destination square to the opponent’s King’s square as well as following a similar vector, from the opponent’s King through the source square, to look for pieces that now threaten the King. Despite the perceived efficiency of these solutions, a naive approach of iterating through the spots on the board and determining if the piece inhabiting that square could successfully move to the possibly threatened square was used in the final implementation of CyberChess. Despite a minor hit to efficiency, this was one of the main elements that contributed to a simple, clean code base in the finished product.

# 4 – Initial Hardware Design

Originally, the group planned to build a CyberChess system run completely by an ARM processor with an embedded Linux operating capability. The digital signal processing for speech recognition and audio feedback would have been implemented outside of the chip using a DAC and ADC. The Chess Engine and the Board Movement and Sound Engine would have all been programmed into a single processor. The motor drivers and LED drivers were much smaller than the ones that ended up in the final design. This section lays out the research for all of the main electronic components in the initial design plans for CyberChess.

## 4.1 - ARM9 AM1808 Processor

The ARM9 AM1808 is the microprocessor we chose to be the core of CyberChess. There were many reasons we needed a processor like this one. First of all, it is an inexpensive processor. The processing power in the AM1808 is plenty for CyberChess, without being too over the top (like the Cortex-A8 in respect to this project). The AM1808 contains 37 registers total, status registers occupy 7 of them leaving 30 general purpose registers. It has a total of 144 GPIOs, which (along with our various drivers) should be enough to support all of our systems (LEDs, motors, switches, etc.). Twenty of the pins (PINMUX0-PINMUX19) are capable of multiplexing, each with a 4-bit field. This may allow for the control of more outputs (such as numerous LED drivers) without the requirement for a large amount of pins. As mentioned in Chapter 3, the AM1808 has a uPP to interface with the DAC and ADC converters. It’s internal Direct-Memory-Access controller allows the uPP to operate without much interference to the microprocessor’s other operations. Figure 4.1.1 (below) shows the AM1808 Functional Block Diagram.

**Figure 4.1.1 - AM1808 Functional Block Diagram**

### Accessing GPIO

The ARM9 AM1808 used for CyberChess comes with 144 GPIO pins which will be responsible for handling the input and output to the microprocessor. The GPIO can be accessed through the Linux userspace or it may be implemented into the C source code. The GPIO sysfs interface must be supported by the Linux kernel used in order to manage the GPIO from the Linux shell. Linux kernels 2.6.30 and above come with the GPIO support already included. The next step in handling the GPIO is determining how many pins are available on the given SoC. Before handling a GPIO signal, the pin must be exported to the interface using the following prompt with N being the number of the pin accessed.

echo N > /sys/class/gpio/export

The next step in accessing the GPIO is determining the direction of the pin. Enter out for an output pin and enter in for an input pin.

echo out > /sys/class/gpio/gpioN/direction

echo in > /sys/class/gpio/gpioN/direction

The final command for handling the gpio pins is for writing a value of either high or low to an output pin. To perform the write function the following is entered with a “0” for a low value and “1” to write a high value to the pin.

echo 1 > /sys/class/gpio/gpioN/value

echo 0 > /sys/class/gpio/gpioN/value

For the GPIO to be accessed throughout the source code it must contain the following header.

#include <Linux/gpio.h>

Then the pin must be allocated and released after use using the following functions respectively. Neither function will work when provided an invalid GPIO pin.

int gpio\_request(unsigned gpio, const char \*label);

void gpio\_free(unsigned gpio);

The direction and value of the GPIO may be altered using the functions given below.

int gpio\_direction\_input(unsigned gpio);

int gpio\_direction\_output(unsigned gpio, int value);

### PLL

The PLL clocks on the AM1808 are divided into PLL0 and PLL1. PLL0 is used as a reference clock for external devices (such as the data converters). The PLL1 is used in the mDDR/DDR2 controller, separate from the PLL0. The two types of clock outputs on the microprocessor are the domain clocks (SYSCLK) and the auxiliary clock (AUXCLK). The AM1808 provides two serial clocks (pins D19 and G19) for Serial Peripherals 1 and 2, DDR positive and negative clocks (pins W8 and W7, respectively), external memory interface (EMA) clock (pin B7), UPP channel A and channel B clocks (pins U17 and G1, respectively), and a video display port clock (pin K4). Internal clock division and multiplication will allow the AM1808 to interface with different devices of varying clock input requirements. PLL0 and PLL1 both require a supply voltage of 1.2 V through pins L15 (PLL0\_VDDA) and N15 (PLL1\_VDDA), grounded at pins M15 and M17.

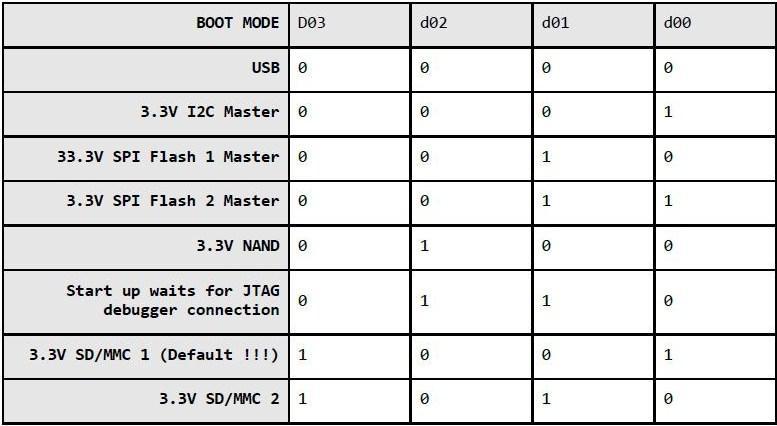
### UPP

The Universal Parallel Port on the AM1808 will be used to interface with the two data converters in the system. The AM1808 comes with two channels (A and B) uPP pins. Channel A will be assigned to the ADC and channel B to the DAC. Pins U17 and G1 will be linked to the bit clock pins on the ADC and DAC, respectively. The data from both devices will be sent/received via two of the UPP\_D pins (out of 16).

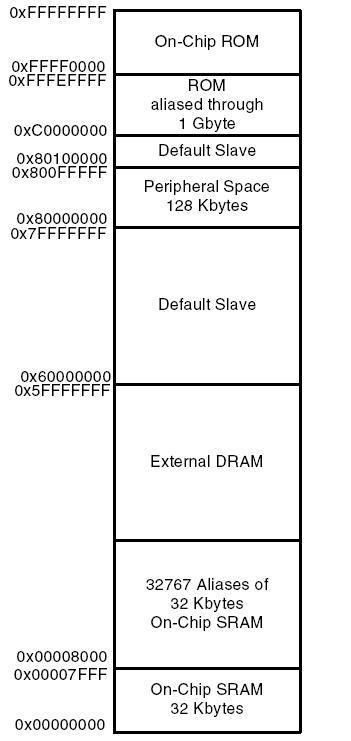
## 4.2 - OLinuXino iMX233 Development Board

The OLinuXino iMX233 is an open source single board computer that will be used as the main development board during the design phase of CyberChess. It was decided that a development board that contained an ARM9 processor would be best suited for prototyping due to the final PCB environment of CyberChess containing an ARM9 AM1808 processor. The iMX233 fits the needs of this project, containing an ARM926 processor that runs at 454 MHz with 32 KB of on-chip RAM and 64 KB of ROM. The OLinuXino was also chosen as the development board for CyberChess due to its compatibility with a Linux OS environment and its relatively low price compared to other platforms in consideration. There are two main debugging options available for the iMX233. The first is using the UART interface for debugging and can be done using a USB to serial cable. The second option is the JTAG interface, which will not be considered due to the pins of the JTAG interface being multiplexed with the microSD signal. This leaves the SD slot of the OLinuXino inoperative. The iMX233 comes with 40 general purpose I/O pins that allow the user to add additional hardware as well as help debugging. Each pin can be accessed individually through a female to male jumper or a 40 pin ribbon cable can be attached to access all pins.

The OLinuXino iMX233 is capable of booting the operating system from up to eight different locations based on the connection of four jumpers located on the bottom of the platform. The default location is set to read from an SD/MMC. Since an SD card was already chosen to be used as memory, no additional soldering will be needed for the iMX233 to boot properly. Below are the boot locations based on active jumpers on the development board.



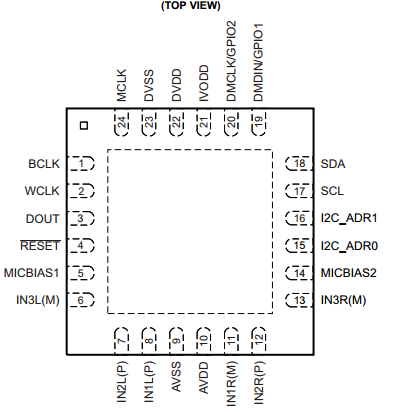
**Fig. 4.2.1 - OLinuXino iMX233 Boot Locations Based on Memory Used**



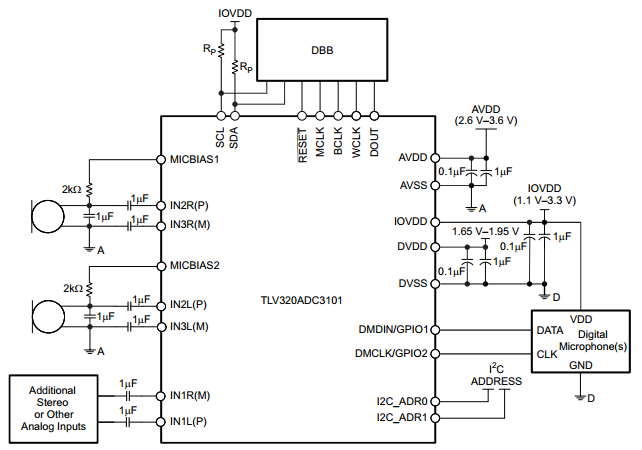
**Fig. 4.2.2 - OLinuXino iMX233 Memory Map**

## 4.3 - TLV320ADC3101 ADC

The TLV320ADC3101 is the analog-to-digital converter that will be used in the hardware for CyberChess. We chose this device because one of the main applications is voice and audio processing. This is an essential tool in implementing speech recognition into CyberChess. The TLV320ADC3101 can convert stereo audio signals into digital through six audio inputs. With this many inputs, and sampling rates ranging from 8 to 96 kHz, this ADC should be suffice for the sole purpose of converting the human voice into computer language. The TLV320ADC3101 is packed with many digital filters and an analog gain control that allows the user to adjust the EQ and signal strength of the voice input. Figure 4.3.1 (below) shows the pin layout for the ADC.



**Figure 4.3.1 - TLV320ADC3101 Pin Layout**



**Figure 4.3.2 - Typical Circuitry**

Pins 7 and 8 will be connected through the PCB to a mono phone connector used for audio input. Pins 11 and 12 will be hooked up to a second phone connector. These two pairs of pins will be the destination of the voice signals going through microphones white and black (separately) before the signal is converted to digital. The CPU will be programmed to ignore one channel while listening to the other, and vice versa. All pins associated with digital microphones or GPIOs will be unused. This includes pins 19, 20, 21, 22, and 23. The clock inputs (pins 1,2, and 24) will be connected to SYSCLK outputs on the AM1808.

The TLV320ADC3101 can operate in I2S mode, where all the data is sent through a single pin (DOUT), and the bit and word clocks determine what channel information is being sent. In standard I2S mode, a word clock low corresponds to information coming from the left channel, and a word clock high corresponds to the right channel. The period of the word clock is equal to the sampling period. The audio clocks can operate at frequencies of 8 kHz, 11.025 kHz, 12 kHz, 16 kHz, 22.05 kHz, 24 kHz, 32 kHz, 44.1 kHz, 48 kHz, 88.2 kHz, and 96 kHz. A reference clock must be provided through either the BCLK or MCLK pins. The TLV320ADC3101 has an internal PLL clock to generate the appropriate audio clocks from a wide range of MCLK inputs. The MCLK inputs accepted can range from 512 kHz to 50 MHz. The following functions show how the sampling frequency is determined based in the PLLCLK\_IN (MCLK or BCLK).

fS = (PLLCLK\_IN × K × R) / (NADC×MADC×AOSR × P)

P = 1, 2, 3,…, 8

R = 1, 2, …, 16

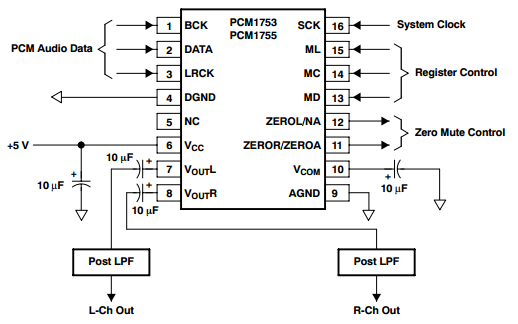
K = J.D

J = 1, 2, 3, …, 63

D = 0000, 0001, 0002, 0003, …, 9998, 9999

## 4.4 - PCM1753 DAC

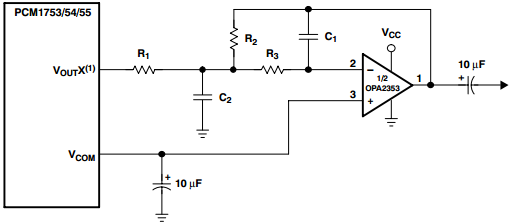
The PCM1753 is a CMOS DAC with 16- to 24-bit audio processing capabilities. This converter can reconstruct digital data sampled at up to 200 kHz. With a dynamic range of 106 dB and a three pin serial control port that allows the user access to some internal functions, the PCM1753 is ideal for the sound system that will be implemented in CyberChess.



**Figure 4.4.1 - PCM1753 Pin Connections**

The serial interface on the PCM1753 includes the BCK, DATA, and LRCK pins (1, 2, and 3, respectively). The system clock (SCK, pin 16) should be synchronous with the BCK and LRCK pins. The DATA pin receives the mp3 (or other digital audio format) information from the processor. The BCK determines the rate at which the bits of data are streamed, and the LRCK determines which channel the data belongs to. In standard data format, the left channel corresponds to LRCK high, and the right channel corresponds to LRCK low. The BCK may operate at 32, 48, or 64 times the sampling frequency (LRCK frequency) to determine how much information comes through in one cycle.

The amount of information that can be sent in one sampling period is directly related to the frequency of the bit clock. However, due to the limitations of clock speeds (in this case, the bit clock), the sampling frequency can only go so high before the amount of information sent per cycle reaches a limit.



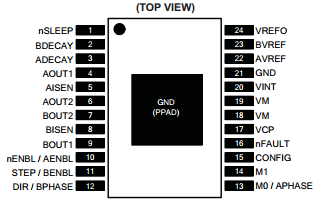
**Figure 4.4.2 - PCM1753 Output Filters**

Figure 4.4.2 suggests an output lowpass filter that can be used to amplify and filter each channel while the analog output is sent to the speakers. The values of the capacitors and resistors in this basic filter are dependent on the electrical ratings of the audio speakers as well as the desired range of frequencies that would optimize the sound quality.

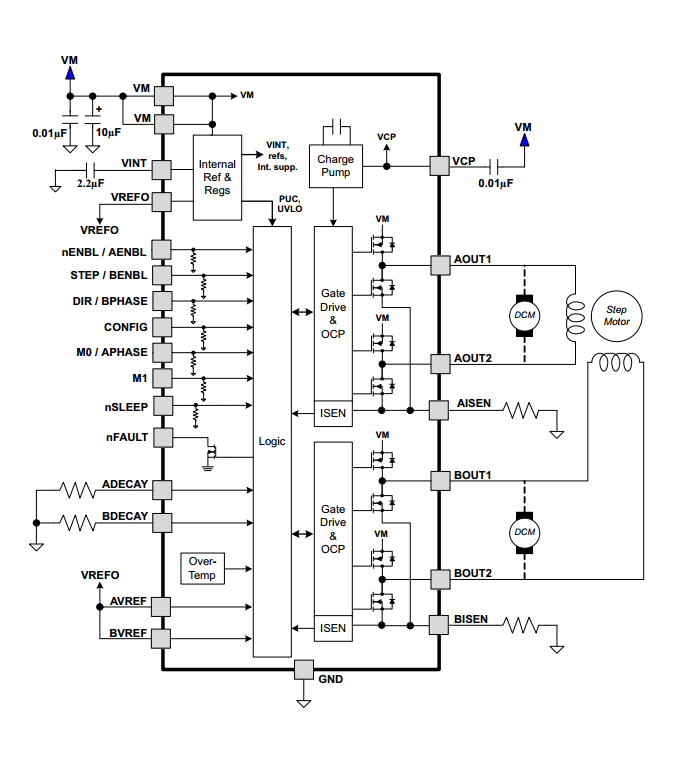
It is recommended that the digital and analog portions of the printed circuit board be powered by different power sources to keep digital switching noise from interfering with analog components performance. If a 5V digital power supply must be used for both the analog and digital sections of the PCB, an inductance may be placed in between the digital supply and analog supply connection to prevent coupling. A RF choke or ferrite bead will work as an inductor in this case.

## 4.5 - DRV8834 Stepper Motor Driver

The DVR8843 is a dual H-bridge motor driver with the ability to drive two DC motors or (Lone stepper motor. Because we are interested in driving two stepper motors with this part, we will need two of these drivers. The driver has two control modes: Step/Direction, with up to 1/32-step microstepping, and Phase/Enable, which allows the driver to drive external references. We were mainly interested in the Step/Direction mode. With 1/32 stepping, we can get a fairly smooth motor movement from the DVR8843.



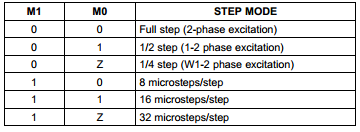
**Figure 4.6.1 - DRV8834 Pinout**



**Figure 4.6.2 - DRV8834 Schematics**

The two figures above show the pinouts (Figure 4.6.1) and basic wiring schemes (Figure 4.6.2) under standard operating conditions for the DRV8834. In Phase/Enable mode, pins 10 and 11 can be turned on to enable output from AOUT1 (4), AOUT2 (6), BOUT1 (9), and BOUT2 (7). These outputs go to the windings on the stepper motor. By sending a high or low signal to BPHASE (12) and APHASE (13), a lag or lead can be created between the A and B outputs. The switching of phases in windings A and B (see Figure 4.5.1) determine which direction the motor will rotate. The M1 pin (14) is used to determine fast or slow decay of the stepper motor. The Phase/Enable mode is intended for input from other devices that are basically pre-programmed to send high, low, and high impedance signals at a rate independent of the DRV8834 stepping capabilities. In essence, the stepping of the motor is controlled by a source other than the driver, and the DRV8834 is simply a medium to convert logic high and low into driving currents or no currents.

In indexing mode, the DRV8834 operates on one of its stepping modes, determined by the CPU, in a specified direction, also determined by the CPU. The M0 and M1 pins will each be connected to its own tri-state GPIO from the AM1808. The digital signal sent through these connections will address one of the 6 stepping indexes in the driver (see Table 4.f.i, next page). The DIR pin (12) will specify whether the output currents will increment or decrement to the next signal step (note that if the signal through phase A is incremented, the signal through phase B must be decremented; and vice versa). The STEP pin (11) is used as a clock (not necessarily with a constant frequency). The output steps are incremented or decremented on the rising edges of the STEP pin input. The frequency of the stepper motor is directly related to the frequency of the input into the STEP pin. In 1/32 microstepping mode, there are 238 steps per revolution of the motor. So, the frequency of the stepper motor is equal to the frequency of STEP divided by 128.



**Table 4.6.1 - DRV8834 Step Mode Truth Table**

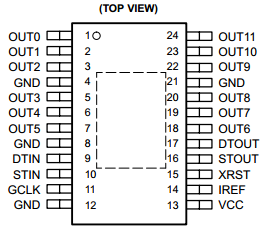
The DRV8834 takes a power supply voltage anywhere from 2.5 to 10.5 V (the absolute maximum is 11.8 V). The maximum output current from the OUTA and OUTB pins is about 2.12 A, at 10.8 V power supply. To match the rated current of 1.7 A and torque of 48 N·cm in the ROB-10846 Stepper Motor, the power supply voltage must be reduced to below maximum voltage. The STEP input can be used to control the acceleration of the motor. However, the motor must have a decent amount of torque to move the weight of the other pieces on the XY-plotter. During the testing phase, optimal power supply conditions will be determined for each stepper driver (exclusively from each other, because each stepper motor will carry a different weight).

The digital input voltages must be below 0.5 V, and the input highs must be at least 2.5 V. Therefore, pins 1 and 10-15 will all be linked to 3.3 V GPIOs on the microprocessor. Pin 15 is the CONFIG pin, which switches the driver mode between Phase/Enable (low) and Indexing (high).

The stepper motors in CyberChess will be run by drivers in 1/32 indexing mode. So the CONFIG pin on both drivers will constantly receive a logic high. The nSLEEP input (pin 1) will be used to power up the device after other priority systems have turned on. The nSLEEP and CONFIG pins on both drivers could possibly be routed from a single GPIO on the processor. The nENBL pin (10) disables/enables the output stage of the stepper motor without stopping the internal logic processes from the M1, M0, DIR, and STEP pins. When nENBL is high, the output stage is disabled; when it’s low, the output stage remains enabled. Because there is no apparent reason to disable only the output stage for CyberChess applications, the nENBL pin should remain low. Therefore, nENBL will not be connected to anything.

## 4.8 - TLC5930 LED Driver

The TLC5930 is a constant-current sink driver that can operate currents between 0.2 to 40 mA. This means the driver is capable of running our COM10866 LEDs at the desired current of 20 mA. The driver has 12 output pins, which are divided into four groups of R, G, and B outputs. This means that four RGB LEDs can be operated from one driver. Since we need 81 LEDs, we will need 21 TLC5930 drivers. The DS-link input allows packet operation so that the TLC5930 can operate with minimal pin use from the CPU. Three types of brightness adjustment functions allow the TLC5930 to change the brightness of all diodes at once or individually. The device utilizes pulse width modulation to allow 10 bits of gray scale data to go to each output. This allows for a very wide selection of intensities in each LED, and therefore a lot of color choices in our RGB LEDs. The gray-scale clock can be operated at up to 25 MHz, internally or externally.



**Figure 4.8.1 - TLC5930 Pin Layout**

The output currents that pass through the LEDs are about 168 times the magnitude of Iref. Therefore, we can control the constant outputs simply by adding one resistor between the IREF pin and ground. This resistance is calculated by taking the reference voltage Viref (1.23 V) increased by a factor of 168 and dividing it by the desired output current (20 mA for our LEDs). This yields an Riref of about 10.3 kΩ. That is the size of the resistor we will be connecting between pin 14 and ground.

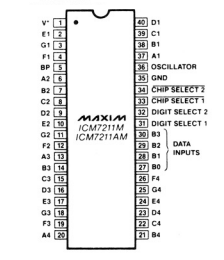
## 4.9 - COM-08642 Reed Switches

The COM-08642 is a simple reed switch that shorts a circuit when coming in contact with a magnetic field, and leaves a circuit open otherwise. There will be 64 reed switches in CyberChess (one for each of the 64 squares on the play space and 16 in each of the two capture zones). The common end of each reed switch will be connected all into a single 5 V input from the power supply. The other end on each of the reed switches will be connected to separate GPIOs on the microprocessor. The rated current of the COM-08642 is 1.2 A. Assuming the resistance in the switch is negligible, the rated current requires a resistor of at least 4.17 Ω to be connected between each reed switch and the voltage source. A resistor of about 10 Ω will be used to keep the current well below rated value.

## 4.10 - Memory (SD Card | RAM)

The operating system as well as our code, speech recognition libraries, and additional data will be stored on a 2 GB Secure Digital Card.The random access memory chip that will be used is a Dynamic 512Mb (32Mx16) 333 MHz DDR2 1.8v chip with an access time of 7.8 microseconds. (Mouser)

## 4.11 - ICM7211M LCD Panel Driver

The ICM7211M is an LCD 4-digit panel driver designed to interface with a microprocessor. Using multiplexing, the ICM7211M reads information sent from the processor to the pins (31 and 32) to select a digit (2 bits for a total of 4 digits) and the four data pins (27-30) to write the binary hexadecimal number desired. The chip select pins (33 and 34) are used to enable the writing in data from the CPU. The ICM7211M has an internal RC oscillator with an oscillation frequency of 19 kHz, which is usually an acceptable frequency for most applications. An external clock can be connected through pin 36, but is probably unnecessary. The Backplane pin (5) will be connected to four 200 pF capacitors, each connected to four parallel groups of pins; 2-4, 6-20, 21-26, and 37-40. 

**Figure 4.11.1 - ICM7211M Pinout**

The pinout is shown on the next page in Figure 4.11.1. The driver is rated at 5 V supply voltage, supplied to pin 1, grounded through pin 35. Two of these drivers will run independently of each other in the CyberChess design. Each one will be assigned to one of the two LCD panel displays.

### Powering the Colon

The ICM7211 does not have an output to support the colon (pin 28 on the FE0202) on our panel display. To solve this problem, one extra GPIO will be linked from the AM1808 to pin 28 in both displays when timed games have been selected.

## 4.12 - Power Supply/Voltage Regulation

Table 4.12.1 (below) shows the power ratings for all of the active devices on the final PCB. Devices with ranges of acceptable power ratings have been assigned specific voltages which are consistent with their respective ratings and should be optimal for the performance of the whole system.

|  |  |  |
| --- | --- | --- |
| **Device** | **Voltage (V)** | **Current (mA)** |
| AM1808 | 1.8/3.3 | N/A |
| TLV320ADC3101 | 3.3 | N/A |
| PCM1753 | 5 | 16 |
| DRV8834 | 10 | 4 |
| ROB-09064 | 5 | N/A |
| TLC5930 | 3.3 | N/A |
| COM-08642 | 3.3 | N/A |
| SDRAM | 1.8 | N/A |
| ICM7211M | 5 | N/A |

**Table 4.12.1 - PCB Device Supply Ratings**

In order to accommodate all of the voltage requirements on the PCB, a power supply will be routed through three different voltage regulators: LM1117-1.8, LM1117-3.3, UA7810, and one UA7805. The main power supply will be 15 V, and it will supply four different power groups: A-D. The voltage regulators should significantly reduce the ripple from the main power source. The circuit diagrams below shows how each group of devices will be connected to the power supply.

|  |  |  |  |
| --- | --- | --- | --- |
| **Group A** | **Group B** | **Group C** | **Group D** |
| AM1808 (1.8 V)  SDRAM | AM1808 (3.3 V)  TLV320ADC3101  TLC5930  COM-08642  CD54HCT237 | PCM1753  ROB-09064  ICM7211M | DRV8834 |

**Table 4.14.2 - Power Supply Groups**

## 4.13 - Other Hardware

### M74HC238 Decoder

The M74HC238 operates at a supply voltage between 2 and 6 V. It will be supplied in power group B (see Table 4.14.2). Operating with a supply of 3.3 V, the minimum voltage for input high is approximately 2.5 volts, and the maximum for input low is approximately 0.8 V. The minimum voltage for output high is about 3.2 V, and the maximum voltage for output low is 0.1 V.

For the purposes of this project, only pins 1-3 and pin 6 will be connected to the processor. OE0 (pin 6) is the Enable pin; and A0, A1, and A3 are the pins used to address a certain output (pins 7 and 9-15) to switch to high, while the other outputs remain low.

0 = Low

1 = High

X = Don’t Care

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **OE0** | **A0** | **A1** | **A3** | **Y0** | **Y1** | **Y2** | **Y3** | **Y4** | **Y5** | **Y6** | **Y7** |
| 0 | X | X | X | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |

**Table 4.13.1 - M74HC238 Decoder Truth Table**

The output pins will be sent connected through each row of reed switches (with every reed switch in that row in parallel with each other, see Figure 6.1.2). When a reed switch is activated, the output signal will be retrieved by one of twelve GPIOs connected to the other side of the reed switches.

## 4.14 – Design Change

The group found out that implementing the ARM controlled system discussed in this section was a very difficult task for a first project. The members had little to no experience with the design process, so trying to integrate a DSP system, a feedback system, hardware control, and a software engine onto one customized microprocessor mandated that the group have a lot of milestones in a very limited amount of time. An accident occurred in the lab which caused the iMX233 to malfunction, and the time lost waiting for repairs would have been detrimental to the group’s schedule. A new design plan had to be formed which would allow the group to finish the project on time. The next section discusses the new hardware plan.

# 5 - CyberChess Hardware

When the ARM based system was unsuccessful, the group switched to the AVR platform for controlling the hardware. The digital signal processing and external memory interfaces were no longer necessary, the reed switch system was eliminated, and the motor and LED drivers were switched for components that were more popular and documented much more thoroughly. The new system became the final system in CyberChess, consisting of a Raspberry Pi for the chess engine and an Atmel microcontroller for the board lighting and movement engine. This section describes in depth the main electronic and mechanical components in CyberChess using AVR technology.

**5.1 - Atmel ATMega 2560**

The Atmel ATMega 2560 was selected as the primary microcontroller for CyberChess. It was chosen over the ATMega 1280 because the development board (Arduino Mega) for the 2560 was easier to obtain. The 1280 and 2560 are identical in every design aspect except for memory space. The ATMega 1280 has 128 kB of onboard memory while the 2560 has 256 kB.

The ATMega chip supports communication between 4 serial lines (UART), an SPI interface, a 10-pin JTAG interface, and 86 I/O pins including numerous pulse-width modulation pins. ATMega supports Arduino language, which was the language used to code the motor and LED control systems.

The ATMega does not have support for USB communication, so an FTDI USB-to-Serial cable was used to convert USB data from the Raspberry Pi into UART serial data that the chip could read. Because the USB-to-Serial conversion was not set up with an ATMega 16U2 chip, the Arduino IDE was unable to burn a bootloader to the ATMega chip, so all code was uploaded using the SPI interface, via an AVR Programmer.

The ATMega operates at 5 V and a maximum frequency of 16 MHz, so the respective crystal oscillator was linked to the XTAL ports of the chip.

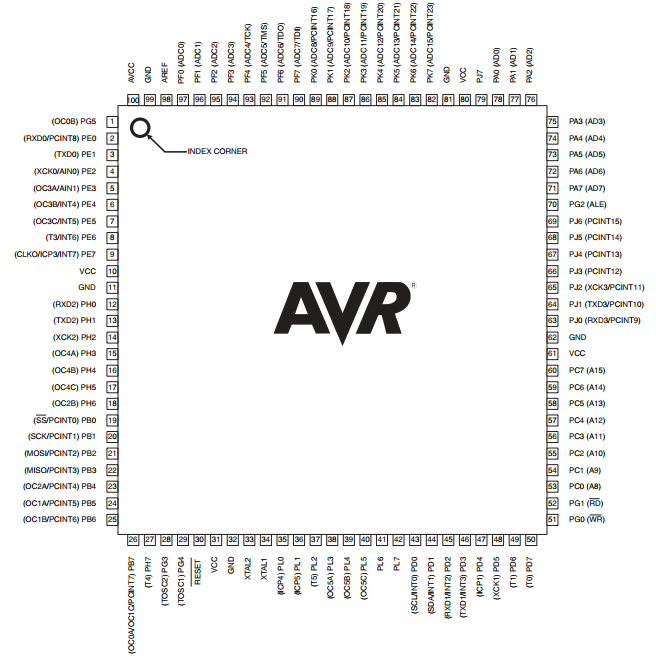


Figure 5.1.1 - ATMega 1280/2560 Pinout

## 5.2 - Big Easy Drivers

A pair of Big Easy Drivers were used to drive the stepper motors based on the Arduino code. The primary logic inputs are Step, Direction, and Enable. Direction controls the direction in which the stepper shaft spins. When the Direction pin is set high, the motor will turn in the counterclockwise direction, the motor will turn clockwise when set low. Step controls the speed at which the motor shaft spins and should be seen as a periodic square wave with a 50% duty cycle. Each step the motor takes is triggered on the rising edge of the square wave form. Enable is active low and should be set to high at all times when the drivers are not supposed to be spinning the motors. The power should not be supplied to the drivers until Enable is set, because when the Enable pin floats, unnecessary current flow can cause the device to heat up and malfunction due to the low internal resistance of the motors when not running.

The Big Easy Drivers can output currents of up to 2 A and are rated for a supply voltage of between 7 and 35 V. Because the ATMega chip is being powered at 5 V by the Raspberry Pi, the proper current and voltage cannot be supplied to the Big Easy Driver from the same power plane that provides for the ATMega. A separate power source of about 7.5 V was used to supply the Big Easy Drivers. An onboard voltage regulator allows the Big Easy Driver to operate the motors at a single current regardless of the input voltage (assuming the input voltage is within the range of 7 to 35 V). A potentiometer allows the user to set the desired current based upon what current sense resistor is attached to the model. For the Big Easy Driver v1.2, the following equation was used to determine the output current.

Because the stepper motors in CyberChess were rated for 1.7 A, we set the potentiometer to allow for 1.4 A (slightly lower than the rating) to be sent through the stepper motor coils. The output current was determined by measuring the reference voltage through a multimeter while adjusting the potentiometer until the calculated value needed was reached. It is important to note that the current supplied to the bi-polar stepper motors is not supplied unto a single coil at a time, but is split amongst coils providing 0.7A per coil to the motors.

The Big Easy Drivers have other optional logic inputs to adjust microstepping settings. If these inputs are left unused, the Big Easy Drivers operate at their lowest microstep option of 1/16 stepping. Therefore, those pins went unused so the Big Easy Drivers could provide the smoothest signal possible.



Figure 5.2.1 - Big Easy Driver

## 5.3 - 74HC595 Shift Registers

The 74HC595 chips are basic 8-bit shift registers which come in dual inline pin packaging (DIP). The 74HC595 shift registers were used to cut down the number of GPIO needed to run both the LED array and chess clocks of CyberChess. The shift registers have 3 logical input lines: data, clock, and latch. The clock is linked to a GPIO on the microcontroller and used to synchronize the shift register with the microcontroller so that serial data is not lost. The clock on the microcontroller operates at 16 MHz, so data is sent through the data pin at the same rate. The latch pin is set to high when the data stored in the shift register is ready to be sent to the outputs (Q0 - Q7). If more than a byte of data is sent, the excess data shifts through the Q7’ pin. This feature allows the shift registers to be daisy-chained, where Q7’ of one shift register is linked into the data pin of another register, effectively creating one 16-bit shift register.

For the purposes of CyberChess, two shift registers were daisy-chained together to control the segment activation of the 7-segment displays. Six shift registers are daisy-chained and used to control the lighting of the RGB LEDs from the anode side. Three shift registers control the reds, greens, and blues of the top four rows while the other three control those of the bottom three rows. The microcontroller effectively communicates with one 16-bit shift-register and one 48-bit shift register.

The shift registers require 5 V to operate. The enable pin is active low and can be constantly grounded for the purposes of CyberChess. The shift registers are powered by the same 5 V plane as the microcontroller because the current output is not too excessive.

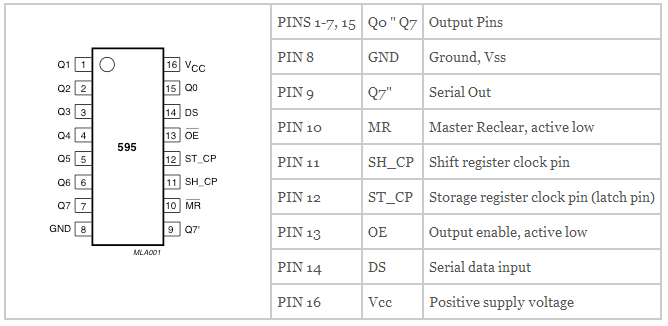
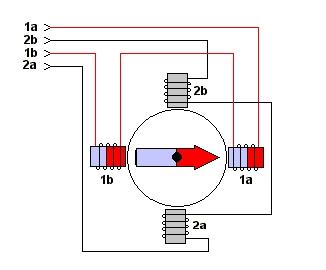


Figure 5.3.1 - Shift Register Pinout

## 5.4 - ROB10846 Stepper Motor

The ROB10846 stepper motor was selected to be used as the primary motors to drive the XY-plotter in the movement system of CyberChess. This motor was chosen because it provided the highest torque for the lowest cost. The torque of the motors were taken into consideration due to the additional weight of the Y axis being placed onto the X axis motors. At a max input voltage of 3V and a current of 1.7A, the ROB10846 provides its highest level of torque at 48N·cm. The current required to drive the motor was a key factor in which motor driver would be selected to control the stepper motors of the XY-plotter.

Figure 5.4.1 (below) shows the wiring diagram for the ROB10846 for reference when connecting the stepper motor to the driver. The stepper driver is discussed in the next section.



**Figure 5.4.1 - Basic Stepper Motor Schematics**

In order to help determine how much space will need in the XY-plotter compartment, it is also helpful to know how much space the stepper motor will be taking up. The long side of the stepper motor is 48 mm. The side orthogonal to the rotary arm is 43x43 mm. A sufficient amount of space will be allowed for the stepper motors in both directions. The drive shaft of this particular stepper motor is 5mm and will affect the decision on which circular gear will be chosen to drive the motors along the vex rack. Each stepper motor will be powered by its own Big Easy Driver motor driver. To connect the motors to the driver simply take the black and green cable of the stepper motor, they are labeled A and A respectively, to pin 4 and pin 6 of the motor driver. Two other wires must be connected to the Big Easy Driver in order to drive the steppers, they are the red and blue wires labeled B and B respectively, and must be attached to pin 9 and pin 7 of the motor driver.

There is also a way to determine the correct wiring of the stepper motor if its wires are not labeled. Begin by plugging in all four wires of the stepper motor into the driver in any arbitrary order and then send a pulse to drive the motor. If the motor is moving erratically or not at all, switch a wire from phase A with a wire from phase B. If the motor is rotating in the opposite direction than desired, switch the wires belonging to one of the motors phases. This method is applicable to the ROB10846 motor selected for CyberChess but will not be needed due to the motor being properly labeled.

## 5.5 - ROB09064 Servo Motor

The servo motor is a pretty simple part. It comes with a 3 pin power and control cable. The individual wires/pins are brown (ground), red (power), and orange (control). The power cable can be operated from 4.8 to 6 V directly from the power supply. To control the servo, different length pulses must be sent through the control wire. The ROB09064 will turn 60 degrees in time periods ranging from 160 milliseconds (operating at 6 V) to 200 milliseconds (operating at 4.8 V). The stall torques at 4.8 and 6 V are 5.2 and 6.5 kg\*cm, respectively. These operating conditions should be sufficient to support the weight of our primary magnet and the small length of wood that holds it.

The ROB09064 can be operated by sending 5 V pulses through the control cable. The servo motor changes its angle based on the length of the pulse it receives. A pulse time of 1 ms will cause the servo motor to turn to its 0 degree (minimum) position. A pulse time of 2 ms will turn the servo to its 180 degree (maximum) position. Any pulse time between 1 and 2 ms will turn the servo to an angle proportional to that time. For the purposes of CyberChess, we are only interested in its minimum and maximum positions.

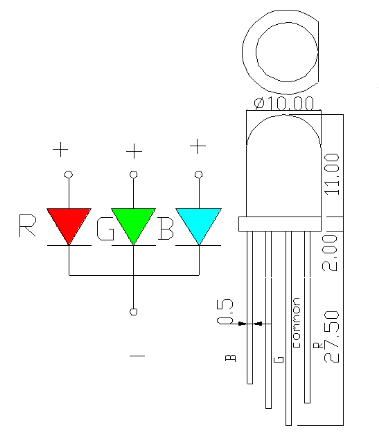
The servo has to be given a pulse every 20 ms to either retain its position or move to another position. For example, to keep the servo at minimum position for a long time, a pulse of 1 ms must be sent once every 20 ms. This pulse width control can be accomplished through software run by the microprocessor. No driver is needed for the servo motor. The control wire will be linked directly to a single output pin on the CPU.

It is probably safe to assume that CyberChess players will spend a lot more time thinking about their moves than the XY-plotter will spend executing the moves. Therefore it is also safe to assume that the servo motor will spend most of the gameplay in the release position (the position where the magnet is facing down so that no chess piece can be grabbed) as opposed to the hold position (where the magnet is facing up to grab a piece). Because the servo motor must be constantly fed pulses, the release position should correspond with the minimum position of the servo motor in order to save power by using the least amount of pulse time as possible.

The ROB09064 dimensions are 41x20x38 mm, which is important in determining the amount of space needed in the XY-plotter compartment. It weighs 41 g, which is light enough to be supported by the stepper motors which will be controlling its location.

## 5.6 - COM-11120 RGB LEDs

We chose the COM-11120 LEDs for their size and color variety. The RGB LED consists of three diodes, one for each of the three main colors. The forward voltages for each of these colors are as follows: 2-2.2 V for red, and 3.1-3.3 V for green and blue. The maximum current rating is 20 mA, so the LEDs should be operated at around 18 mA.



**Figure 5.6.1 - COM11120 Dimensions and Pin Layout**

Figure 5.6.1 (above) shows the pin layout and chip dimensions. The LEDs are small enough that there will be no problem fitting them underneath our chessboard. The height of the LED is important in determining the crawl space between the two glass panes we plan to use. In order for the LEDs to operate in forward bias, the ground pin will be in series with a resistor leading back to the microcontroller and the three anode pins of the LED are connected to shift registers awaiting logic signals to determine which LED to light. The final CyberChess design will have 64 of these RGB LEDs, one under each corner of each square on the top glass pane. The purpose of the LED array is to provide visual feedback to the users of CyberChess.

## 5.7 – FTDI USB to Serial Cable

The FTDI cable used in CyberChess is responsible for communicating serial data sent from the Raspberry Pi to the Atmega2560 using RS-232 protocol for UART devices. This particular model of FTDI has one channel for data transmission labeled TX0 and RX0 and two control lines labeled CTS (clear to send) and RTS (request to send). The I/O pins of this cable are configured to operate at a potential of 5V which is supplied by the VCC pin of the cable.

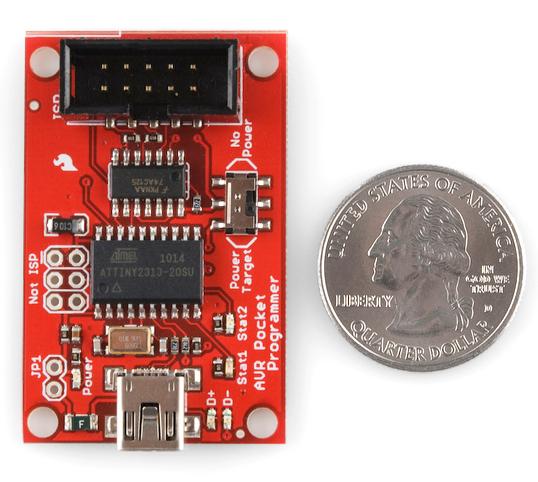


**Figure 5.7.1 FTDI USB-to-Serial cable**

## 5.8 - AVR Pocket Programmer

The AVR Pocket Programmer was used in CyberChess to program the microcontroller on the printed circuit board. By connecting the Pocket Programmer to the circuit board using a pinned out SPI interface, the team was able to upload developed software. The Pocket Programmer had option for turning on and off the power supplied to the board, which served useful in powering the parts of the board to test initial code procedures and actions.

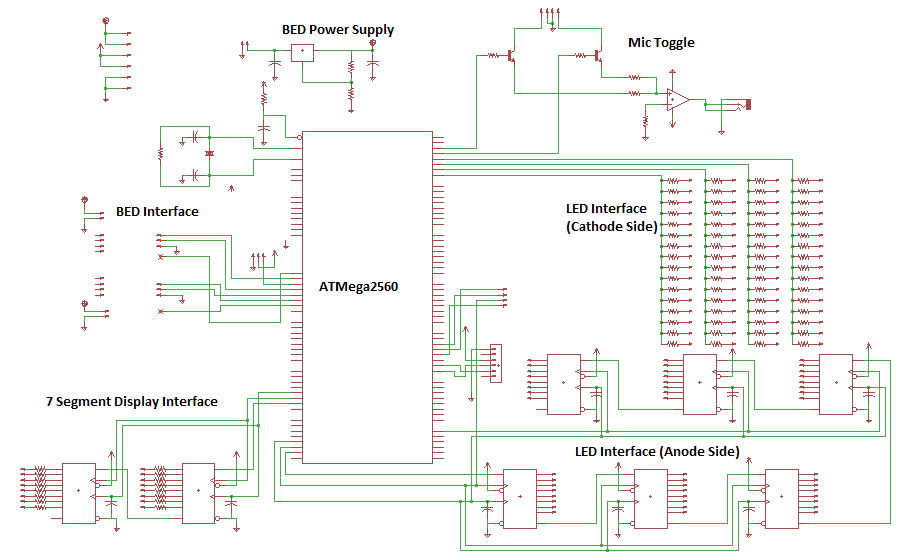
Initially the team had planned on using the Arduino Mega 2560 as an AVR Programmer but complications with memory limitations created a need for the AVR Pocket Programmer.



**Figure 5.8.1 tinyUSB AVR Programmer**

## 5.9 - Schematic and Printed Circuit Board (PCB)

The schematic for the Printed Circuit board can be seen below.



**Figure 5.9.1 - PCB Schematic**

The printed circuit board was created using the above schematic and the final layout, traces, and design can be seen on the next page.

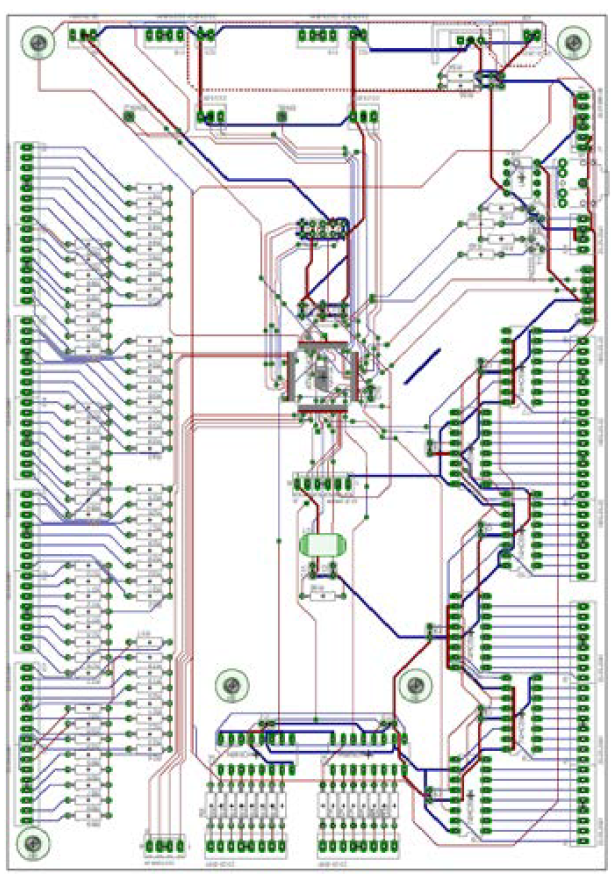
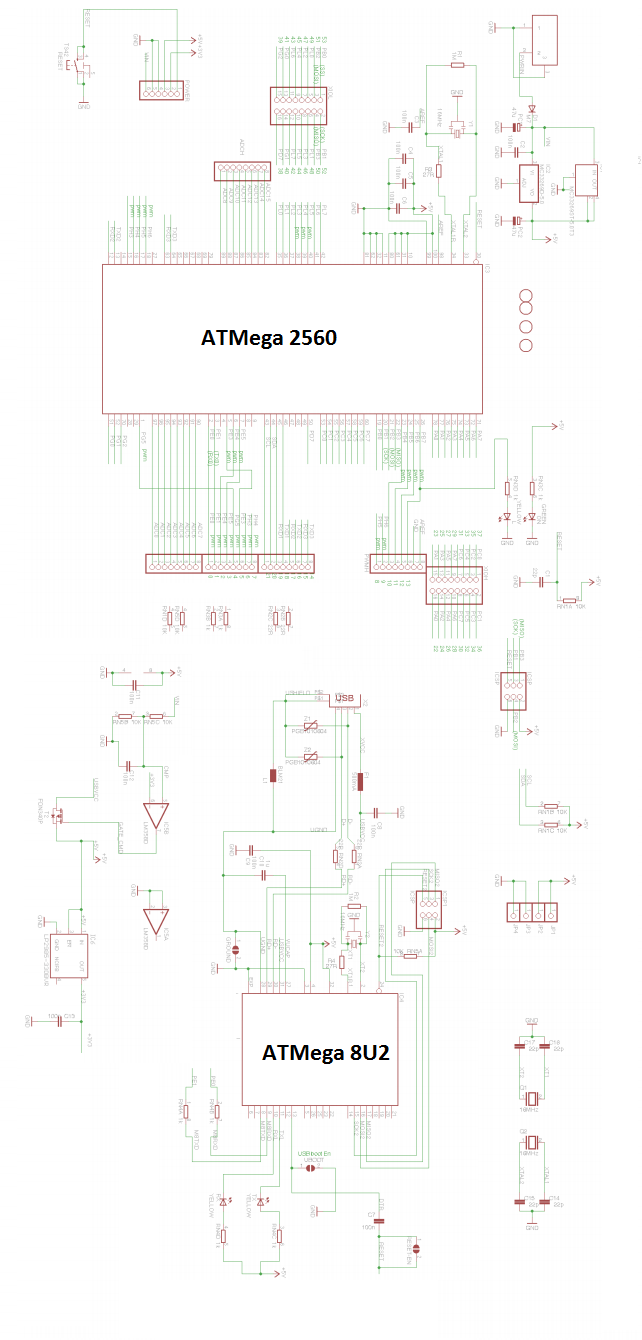


Figure 5.9.2 - PCB Eagle Layout

## 5.10 - Arduino Mega

The Arduino Mega was selected to aid in the development of CyberChess hardware control. The Arduino Mega comes with an ATMega 2560 pinned out to female headers on the edges of the board. Using the Arduino Mega allows the ATMega to be tested as an easily programmable microcontroller without having to solder any connections. The Arduino Mega can also be used as an AVR programmer in many cases. However, for the purposes of CyberChess the AVR Pocket Programmer mentioned in section 4.8 sufficed.

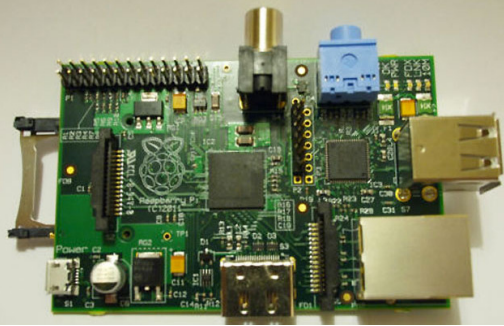
The Arduino Mega has an ATMega 8U2 for USB-to-Serial conversion. The ATMega 8U2 is a smaller microcontroller than the ATMega 2560, however it supports the communication protocols that the 2560 lacks, those of which are essential for interfacing with many external processors.



**Figure 5.10.1 - Arduino Mega Schematic**

## 5.11 – Raspberry Pi

The Raspberry Pi houses the speech recognition, audio output, and chess engine software. It communicates with and powers the 5 V digital components on the PCB via USB port. It has an HDMI out as well as enough USB ports to support a keyboard for debugging purposes. The Raspberry Pi is rated for 5 V input at no more than 1 A.



**Figure 5.11.1 - The Raspberry Pi**

# 6 - CyberChess Software

The software is responsible for making all of the hardware work. The Chess Engine provides a virtualization of the game of chess, allowing the computer to know where pieces are at all times, which pieces can move where, and how to know if a player is in check or lost the game. The Board and Movement Engine is comprised of the XY-plotter, LED Control Systems, and sound systems. This engine is tightly integrated into the Chess Engine and allows for the virtualization to become a reality. Finally the Speech Recognition ties them together allowing a user to give a command to the Chess Engine and having the Board Engine initiate and finish the move.

The vast majority of software within the CyberChess system is written in the C programming language. C was chosen over the other investigated options for its fine control over memory management and low-level operations, allowing simple, direct access to hardware without much overhead. Well-written C code tests extremely well in benchmarks, especially for functionality that is highly important to CyberChess, such as bit manipulation and, again, memory management.

## 6.1 - Operating System

The operating system running on the Raspberry Pi mainframe is Arch Linux ARM. This is a port of the Arch Linux distribution to the ARM architecture which stays true to the Arch philosophy of a simple, controllable operating system. Arch is a full-featured rolling-release distribution of Linux that permits the end user to control the entire system as they see fit. This fits nicely with CyberChess - it provided a full production operating system during development and readily switched to become dedicated to its task of just running the CyberChess software.

## 6.2 - Chess Engine

The Chess Engine is a programmatic implementation of the rules of the game to ensure user-requested moves are valid and the game proceeds properly. It maintains an internal representation of the current game state, including piece location, en passant availability, castling possibility, and which color next moves. The engine provides a single C header file, “chess.h”, which guarantees a simple API permitting access to knowledge about the game state and the ability to request a move.

As the project does not require generation of excessive numbers of possible moves, the engine was not necessarily designed with efficiency in this area as a key priority. Even a particularly poorly implemented engine would have easily outpaced the speed at which users could issue new commands. However, the engine is still designed with the goal of high efficiency, as a learning exercise and to permit flexibility in any future use of the engine for other purposes or projects.

### Data Structures

The engine makes use of three key data structures. Game state is contained in a structure composed of a representation of the board and a word made up of various bit flags corresponding to game information. The board has a space-efficient representation in which each row is contained in a single 32-bit integer value. This gives each spot in the row 4 bits, or a single nybble, to store piece information, split into one color bit and three piece-identification bits. With this implementation, much of the general control of the board is accomplished entirely with bit shifting, taking advantage of the relevant speed efficiency. This does have the drawback of having to search the entire board space to determine piece location for such functions as check detection. This is partially addressed by maintaining knowledge of the locations of the two Kings, updated whenever either of them are moved.

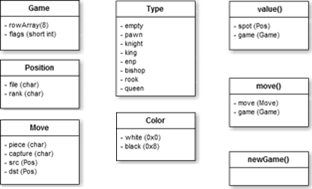
Positions are an elementary data structure composed of a single 8 bit character value, split into two signed 4 bit fields containing the file and rank that are being referenced. This data structure was inspired by the flexibility of the 0x88 board representation, which similarly represents positions in 8 bits. While no use of the eponymous 0x88 ANDing is present within the chess engine, treating each field as a signed value allows for simple iteration, terminating as soon as a negative value is encountered.

Moves are a basic data structure containing two Position data structures - a reference to the starting location, the ending location - and two nybbles containing the identities of the pieces in each. This is sufficient information to readily test both the legality and the validity of the move while also slightly improving the efficiency of move execution by having piece information available for manipulation.

Each of these abstractions can readily be transformed to or from a string of text, allowing simple text-based interfacing with the other software within CyberChess. While the core functionality of all of the engines are compiled into a single binary within CyberChess, this allows for simple extensibility, making use of UNIX pipes to directly pass their textual output to the input of another program, reducing internal communication to extreme simplicity.

The board to some degree makes use of the 0x88 method for determining valid positions, as described above in reference to the Position data structure. This was chosen over bitboards for simplicity in writing the engine - the benefits of bitboards will be lost to the fact that large numbers of board states will not often be generated. For the purposes of further experience and experimentation, the engine may at some point be re-implemented with bitboards. If this does come to pass, the two engines will be run through a set of identical tests to determine which really is a better choice for the CyberChess system.

A summary of data structures and methods is shown below in figure 6.2.1.



**Fig. 6.2.1 - Summary of Chess Engine Data Structures and Methods**

### Game Saving and Loading

Upon request, CyberChess will save the current game state to a file which can later be reloaded or read by an interfacing system. The game state is written to a simple text file stored at a consistent location within the system. Subsequent game saves will overwrite this file, a simple way to ensure that system meets the requirement of being able to load one previous game. The output is a straightforward dump of all data stored within a Game data structure, separating various pieces of information with unrelated characters to ensure that, for instance, the integers corresponding to the board layout are not misread upon a new game load. Further expansion may at some point reorganize the output into a modified version of Forsyth-Edwards Notation (FEN) which stores the positions of the pieces on the board, the color of the next player, castling availability, en passant availability, and move counts. This notation is quite readable and easy to edit both manually and programmatically, allowing for simple game analysis. The save game output will only be generated and saved at the request of the users, not unnecessarily wasting any processing power on keeping it up to date.

CyberChess may at some point be extended with notation of the game as it goes on to allow for further external analysis and reconstruction of a game. The progress of the current game would make use of a condensed form of Portable Game Notation (PGN), recording each move with Coordinate notation, slightly modified to differentiate between movement and capture, rather than the usual Standard Algebraic Notation. This modification to PGN makes code for future parsing of the file simple to implement as each move is recorded in an incredibly consistent form. This PGN movetext would be updated after each move by simply appending to the file until the game is completed, at which point it would be saved with a time-based identifier allowing for future inspection. By simple parsing of this file, it would be readily possible to have the system replay a completed game as it was initially played.

### Timer

CyberChess implements a simple chess clock that permits users to play either with an hourglass timer or a time limit on their individual moves, on each player, or on the entire game. The hourglass timer grants each player a set amount of time at the outset of the game and effectively gives the current player’s time to their opponent - much like an actual hourglass. This style of clock encourages rapid play in order to starve the opponent of their time and not grant them any additional time.

With an individual move timer, each turn a player is allotted a certain amount of time; if they fail to make a move in this time, they lose the game. Giving each player a set amount of time for the entire game is in line with tournament-style time controls; here, each player must ration their time as they see fit, again losing when they run out of time. The final timer option is a simple timed game, in which the game will end after a set amount of time. This option is mainly for when users want to play only for a set time due to external restrictions and does not necessarily end the game when time runs out - should the time run out, the system offers to save the game to be returned to later.

Should the clock be updated in a future iteration of CyberChess, it will make use of a delay to offset the effects of having to wait for pieces to move. When each player has a set amount of time for the entire game, the clocks will use a compensation delay, either a simple delay or a Fischer delay, to make up for the time spent moving the piece. A simple delay waits a set amount of time before reducing the player’s clock while a Fischer delay adds to their clock at the beginning of the turn, allowing them to potentially increase their total time by calling out a move quickly. To better facilitate blitz (4 to 15 minutes per player) and lightning (under 3 minutes per player) games, the system may be updated to increase the time allotted to the simple/Fischer delay or permit a player to call out a move before the previous move has been physically completed, halting their timer and switching turns at that point.

## 6.3 - Board, Movement and Sound Engine (BMSE)

The Board and Movement Engine was responsible for providing the low-level software interface between the virtual and physical games of chess. It contained the functions for moving the XY-plotter, adjusting the LEDs on the board, and triggering the use of audio files. It took input from the Chess Engine and instructed the physical hardware to react appropriately - visually, audibly, and most importantly, with the proper physical movement. The engine could be broken down into three smaller, specialized engines addressing these three main areas.

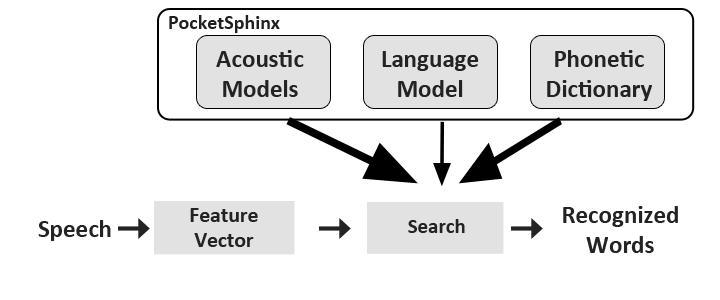
The LED lighting system is encompassed inside the Board and Movement Engine and contains all of the functions for manipulating the LEDs. As CyberChess makes extensive of RGB LEDs, this system had been designed to support the creation of a wide range of effects. This aspect of the engine allowed for control of the appropriate values for each individual LED, requiring interfacing with the entire array of 64 LEDs.

The audio engine selected the appropriate audio file to be played as necessary. Most audio support simply utilized the functionality of the operating system, so the audio engine simply needed to contain a set of conditional statements or a switch-case to determine which file was to be played. To create a more seamless play experience, the engine was initially expected to be expanded to queue audio files that are relevant to current play, expiring files that remain in the queue for longer than they are relevant to what is going on in the game and prioritizing their placement in the queue based on their importance to gameplay.

The control of the physical movement of the pieces is the most important area the engine had power over. This portion of the engine determined the most direct path for movement of a piece from its starting location to its destination and then issued the appropriate signals to have the stepper motors actually execute the move.

## 6.4 - Speech Recognition Engine

The Speech Recognition Engine was composed of several C files as well as a language model and dictionary file that adapted PocketSphinx specifically to the vocabulary necessary for operation of the CyberChess system. The main C file for PocketSphinx was called PocketSphinx\_Continuous.c and is shown in block form in Figure 6.4.1. This code first parsed the input speech for model-specific features from the built in feat.params library. The code then read in a binary model definition file and broke down the input into CI-phones, CD-Phones, estimate phones, CI-sen, SEN, and Sen-Seq. These sequences and phones were then compared using memory-mapped I/O for senones and a dict.c file. The possible words and cross words were found and then narrowed down using the language model (.LM) and dictionary (.dic) files.

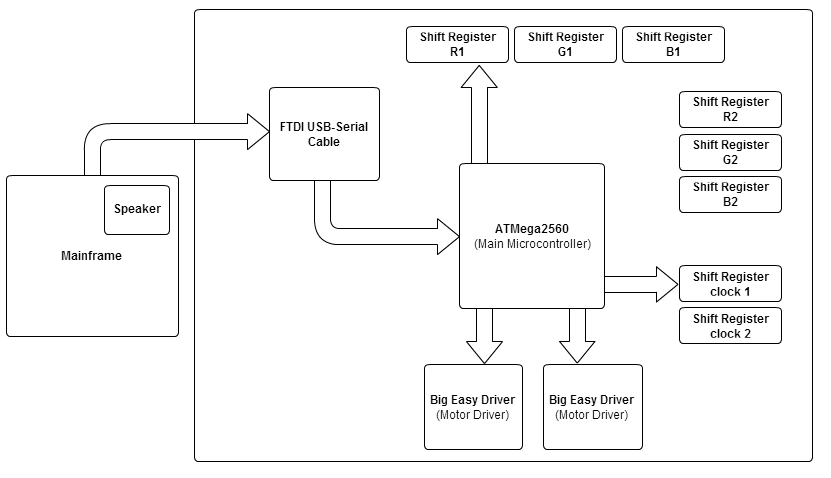


**Fig 6.4.1 - Speech Recognition Engine**

|  |  |
| --- | --- |
| **Command** | **Action** |
| **“[Piece] [Piece Location] to [New Location]**” | Moves the piece at *PIECE LOCATION* to the stated *NEW LOCATION*. |
| **“Possible move [Piece Location]”** | Lights up the squares that the piece at *PIECE LOCATION* can move to. |
| **“Save Game”** | Saves game to file. |
| **“End Game”** | Asks opposing player for confirmation. |
| **“New Game”** | Asks opposing player for confirmation. |
| **“Draw”** | Asks opposing player for confirmation. |
| **“Resign”** | Asks player for confirmation. |

**Table 6.4.1 - Speech Commands**

# 7 – Hardware Design Summary



**Fig 7.1.1 - Hardware Block Diagram**

## 7.1 - Movement System

The main goal of cyber chess was to create an automated chessboard that responds to voice commands. The users of cyber chess therefore do not need to move the chess pieces manually. Instead each chess piece will have a thin neodymium magnet attached to its base to be used in both the movement and sensor system. In the movement system the magnets on the chess pieces will interact with another stronger, cylindrical neodymium magnet, 0.5 inch diameter by 0.3 inch height, which lies beneath the chess board and sensor board. This larger magnet is attached to the movement system which is a basic XY-plotter.

The XY-plotter will consist of the following:

* + 2 - 68 oz\*in (400 steps/rev) Bipolar Stepper Motor
  + 1 - 6.5 kg\*cm Servo Motor
  + 2 - Sets of Toothed Rack Gear
  + 1 - 1.5” Diameter 36 Tooth Circular Gear
  + 42” x 30” plywood base
  + 1 - 42” Linear motion kit
  + 1 - 30” Linear motion kit
  + 1 - 42” x 3”x .75” X-Axis plywood vex rack platform
  + 1 - 30” x 3” x .75” Y-Axis plywood vex rack platform
  + 1 - 30” x 5” x .75 Y-Axis Arm
  + 1 - Hot Glue

The first step in design of the movement system was to attach the X-Axis linear motion kit to the 43” x 30” base board. This kit will provide the X-Axis direction of motion and ensured that it flowed in a fluid motion. This first step was simple but crucial to the project overall. If the track was not exactly parallel to the base board, the accuracy of the XY-plotter would have been significantly lowered and the magnet might not have been able to reach all the desired boundaries of the playing board.

Next the vex rack was aligned along the inside of the X-Axis and was glued in place. Then a 5mm hole was drilled in the center of the 1.5” diameter circular gear so it could be placed tightly onto the shaft of one of the stepper motors. To ensure that the gear had a solid connection to the shaft of the motor, high strength epoxy was used to lock it in place. The Stepper motor in turn was glued into place into section equal to the size of the motor on a piece of 1.5”x 4”x 3.5” wood that will be mounted on top of the Y-Axis arm. The height the motor was placed was important because the team needed the circular gear to be properly mated with the vex rack so there is no complications. Next a block of wood equal to the size of the piece the stepper motor is mounted in, was placed on the opposite drawer track to complete the X-Axis of the plotter.

The design of the Y-Axis is identical to the X-Axis described before. A piece of 30”x 4” x .75” wood and one 30” vex track will serve as the range for the Y-Axis of motion. The team first placed more vex rack alongside the linear motion track and glued it in place. The second stepper motor was then mounted on a block of wood as before and secured it to the Y Axis linear motion kit track at the right height to where the gears mesh as before. A servo motor was also attached on top of the 1.5”x 4”x 3.5” piece of wood the Y-Axis stepper motor is attached to. This servo motor controls the range of motion for the larger neodymium magnet. The Y-Axis was then connected to the X-Axis by attaching the 4”x 30” x 1.5” x axis on top of the 1.5”x 4”x 3.5” wood secured to the linear motion slides on both ends of the playing board. The main neodymium magnet was attached to the wooden arm magnetically by attaching it to a metal cap fitted to the end of the wood. The magnetic attachment (as opposed to glue) made it easier to remove the magnet for testing and diagnostic purposes. This completed the basic design of the plotter.

The two bipolar stepper motors were driven by Big Easy Drivers acquired through SparkFun. These motor drivers were connected to the ATMega 2560 processor and were implemented to control the motion and direction of the magnet base. The servo motor did not need a driver and was connected to the microprocessor directly to control the rotation of the magnet when it was needed.

## 7.2 - Sound System

The primary function of the sound system in CyberChess was to provide the users of the game feedback based on their voice or manual commands given. It was important that correct feedback was given because the system depended heavily on the speech recognition function of CyberChess for efficiency. The sounds system tied in mainly with the chess rules engine and responded when a user made an illegal move, there was interference with the playing board, or the match was won. The sound system consisted of both input taken from the user and output relayed to the speakers for feedback.

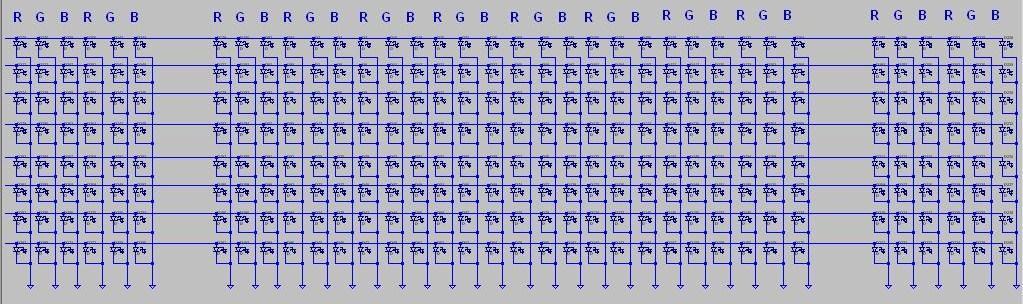
The main input of CyberChess’s sound system was two Rock Band microphones that took in the user’s voice when giving a command. This analog signal was then passed through the speech recognition engine which created a text output. This text was then sent to a parser which interpreted and processed the information.

The output of the sound system followed a reversed process of the input to the sound system. The microprocessor took the digital string given from the input and broke off the appropriate feedback that was to be given to the user based on the command. Once the feedback was selected the string was passed through the to the speakers to produce speech.

## 7.3 - Lighting system

In order to stay in theme of the project, CyberChess was laced with L.E.D’s around the playing board to give the game a more futuristic feel. The L.E.D.’s were also there for more than just pleasing the eye. They also served as a guideline for how to play chess if the user was a beginner at the game and unfamiliar with the rules. Using the speech recognition, a user could call out a chess piece at a given location ( i.e. Pawn Alpha 2 ). The board would then light up locations on the board that marked a valid move for the chess piece. There were L.E.D’s located in the center of every playing square to make sure the playing board was properly lit. To do this 64 L.E.D’s were needed. All of the diodes were connected to a shift register which in turn was connected to the microprocessor.

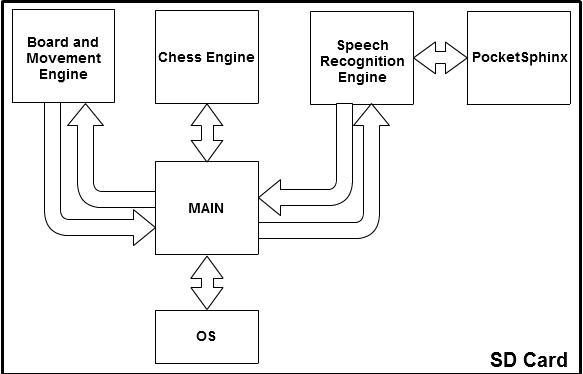
The L.E.D’s were arranged in a row/column multiplexed array to cut down the amount of GPIO used by the microprocessor. Figure 6.1.3 shows the array of L.E.D’s in the CyberChess design.



**Fig. 7.3.1 - multiplexed L.E.D. array**

# 8 – Software Design Summary

The software for CyberChess was broken down into four main components: the main engine, the board and movement engine, the chess engine, and the speech recognition engine. The board and movement engine oversaw all of the physical actions taking place by the hardware. The chess engine was responsible for all of the virtual chess logic and rules. The speech recognition engine was used to control the other two engines, and the main engine was used to connect all the engines together into a cohesive software system. The breakdown of this software system can be seen below in fig 6.2.1. Also shown in this figure is the Operating System (OS) block.



**Fig. 8.1.1 - Software Block Diagram**

The main program facilitated communication between the other pieces of the CyberChess software, taking output from each block and routing it to the appropriate destination to maintain a smooth gameplay experience. As the largest part of this program’s facilities involved functionality that can be handled entirely by file redirection and sequential piping of program output, this overarching control was implemented as a shell script, beginning execution at startup by way of being included in the sequence that is executed each time the system is booted up. The program utilized a simple infinite loop, constantly checking for output from the various blocks to be redirected to its appropriate destination. This design allowed for system shutdown to be handled cleanly and simply as well - the appropriate code was simply sequentially followed by the infinite loop, automatically executing upon exit from the loop.

## 8.1 - Chess Engine

The chess engine was responsible for all of rules, moves, and logic involved with the game of chess. This engine took all of these abstractions and applied them to a virtual space where moves could be made on a simulated board. The important input for the chess engine came from the speech recognition engine in the form of a textual representation of a move. After a vocalized command was given, it was parsed and passed through the main program and into the chess engine to be computed. If the move was not possible the user was notified, otherwise the chess engine carried out the logic and gave outputs back to the main, to be sent to the board and movement engine where the physical changes could be controlled.

The chess engine additionally handled saving and loading games, maintaining a game record in one file that was ultimately archived and writing a textual representation of the game state on demand to another file. This latter file was what was consistently loaded from any time the system and asked to load a game, allowing for the possibility of users loading custom games by overwriting this file.

The board and movement engine controls all of the hardware-based systems including the movement system and the lighting system. The board and movement engine takes inputs from the main program, which interacts with the chess engine and speech recognition, to determine what moves to make, which lights to turn on, and which sounds to send out through the speaker.

When the board was turned on, an activation and welcome sequence was initiated with sounds and lights. All of the LEDs would flash and rotate through the available colors while a Welcome Message is played through the speakers.

When it was a player’s turn, their half of the board would light up with their corresponding team color (green for white, blue for black). If a player does not vocalize a move command within the time allotted in a timed game, an audible and visual countdown will occur. The lighting system would light up with their team color five rows up from their first rank and darken one row each second during the last five seconds of their turn. The sound system would also play a spoken countdown as well.

When a player puts another player in check, a pulsating lighting sequence around the threatening piece would occur and a “check sound” would play through the sound system. The path between this piece and the threatened King would also be lit to aid the player in deciding how to react to the threat.

When a game ended, a specific lighting and sound sequence would initiate. The respective sequence would differ based on how the game ended - checkmate would result in the winning player being congratulated and the lights being focused on their color. A resignation would similarly signal which side has won. In the event of a draw, a more balanced sequence would play, indicating that neither side successfully won the game.

## 8.2 - Speech Recognition Engine

First the speech was taken in through the wired microphone provided for the player. This speech was converted to a feature vector. PocketSphinx and its libraries took over from this point, providing us with the acoustic models to compare against during the search, to find out which words were spoken. These recognized words were then plugged into the Speech Recognition Engine’s Move Definer to finalize the move to be done by the Chess Engine. The Chess Engine took this move and passed its converted instructions to the movement engine and the move was completed.

# 9 - Project Prototype, Construction, and Coding

## 9.1 - Parts Acquisition and Bill of Materials

The parts for CyberChess will be acquired over the course of senior design I up until the start of the spring semester on January 7, 2013. All of the components needed are in stock and will not require any longer than a week to ship. Below is the list of all products needed to create CyberChess along with the name of the supplier of the component as well as the model number.

|  |  |  |  |
| --- | --- | --- | --- |
| **Qty** | **Description** | **Supplier** | **Part Number** |
| 2 | 68 oz.in (400steps/rev) Bipolar Stepper Motor | Sparkfun | ROB10846 |
| 2 | Dual Bridge Stepper Driver | Texas Instruments | DRV8834 |
| 1 | Servo Motor | Sparkfun | ROB09064 |
| 1 | 24 Bit Analog to Digital Converter with 92dB SNR | Texas Instruments | TLV320ADC3101 |
| 1 | 24 Bit Digital to Analog Converter with 106dB SNR | Texas Instruments | PCM1753 |
| 1 | 456 MHz ARM9 Microprocessor | Texas Instruments | AM1808 |
| 1 | Single Board Linux Computer | Olimex | OLinuxino-iMX233 |
| 1 | 2 GB Secure Digital memory card | SanDisk | SDSDQM002GB35A |
| 2 | (16 pack) Rack Gear | Vex Robotics | 276-1957 |
| 100 | RGB LEDs | Sparkfun | COM-10866 |
| 24 | LED Driver | Texas Instruments | TLC5930 |
| 96 | Reed Switches | Sparkfun | COM-08642 |
| 1 | DRAM 512MB 333MHz | Digi-Key | IS43DR16320B3DBLI |
| 2 | 7 Segment LCD display | Purdy | FE0202 |
| 2 | LCD Panel Driver | Intersil | ICM7211M |
| 2 | Phone Jack for Audio Input | Shenzhen Lanho | PJ-380 |
| 3 | 24” Drawer Slide | Gatehouse | LS12224 |
| 1 | 1.5” Diameter 36 Tooth Circular Gear | Vex Robotics | 279-2169 |
| 1 | 36” x 30” x .093” Acrylic Sheet | Optix | MS-06 |
| 1 | 5V Low-Dropout Linear Regulator | Texas Instruments | UA7805 |
| 1 | 10V Low-Dropout Linear Regulator | Texas Instruments | UA7810 |
| 1 | 3.3V Low-Dropout Linear Regulator | Texas Instruments | LM1117-3.3 |
| 1 | 1.8V Low-Dropout Linear Regulator | Texas Instruments | LM1117-1.8 |
| 1 | 3 to 8 Decoder | Sparkfun | M74HC238 |
| 2 | JFET Op-amp | Texas Instruments | TL082 |
| 2 | 23mm Hobby Speaker | Abra | SPK110 |

**Table 9.1.1 - Parts acquisition**

## 9.2 BOM

The bill of materials for CyberChess is displayed below in table 7.a.ii. Each component is listed below along with its model, unit price, quantity, and total price. The total estimated budget for CyberChess is $412.40. N/A in the table means the part was able to be sampled for free or the part was already owned by a group member.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Qty** | **Description** | **Part Number** | **Unit Price** | **Total Price** |
| 2 | 68 oz.in (400steps/rev) Bipolar Stepper Motor | ROB10846 | $16.95 | $33.90 |
| 2 | Dual Bridge Stepper Driver | DRV8834 | Sampled | Sampled |
| 1 | Servo Motor | ROB09064 | $12.95 | $12.95 |
| 1 | 24 Bit Analog to Digital Converter with 92dB SNR | TLV320ADC3101 | N/A | N/A |
| 1 | 24 Bit Digital to Analog Converter with 106dB SNR | PCM1753 | N/A | N/A |
| 1 | 456 MHz ARM9 Microprocessor | AM1808 | N/A | N/A |
| 1 | Single Board Linux Computer | OLinuxino-iMX233 | $56.50 | $56.50 |
| 1 | 2 GB Secure Digital memory card | SDSDQM002GB35A | N/A | N/A |
| 2 | (16 pack) Vex Rack Gear | 276-1957 | $19.99 | $39.98 |
| 100 | RGB L.E.D’s | COM-10866 | $0.86 | $63.64 |
| 24 | LED Driver | TLC5930 | N/A | N/A |
| 2 | 7 Segment LCD display | FE0202 | N/A | N/A |
| 2 | LCD Panel Driver | ICM7211M | $1.33 | $2.66 |
| 3 | 24” Drawer Slide | LS12224 | $8.23 | $24.69 |
| 1 | 1.5” Diameter 36 Tooth Circular Gear | 279-2169 | $12.99 | $12.99 |
| 1 | 36” x 30” x .093” Acrylic Sheet | MS-06 | $19.98 | $19.98 |
| 1 | 5V Low-Dropout Linear Regulator | UA7805 | N/A | N/A |
| 1 | 10V Low-Dropout Linear Regulator | UA7810 | N/A | N/A |
| 1 | 3.3V Low-Dropout Linear Regulator | LM1117-3.3 | N/A | N/A |
| 1 | 1.8V Low-Dropout Linear Regulator | LM1117-1.8 | N/A | N/A |
| 1 | 3 to 8 Decoder | M74HC238 | $0.95 | $0.95 |
| 2 | JFET Op-amp | TL082 | N/A | N/A |
| 2 | 23mm Hobby Speaker | SPK110 | $0.99 | $1.98 |
| 1 | ATMega2560 | ATMEGA2560-16CU | Sampled | Sampled |
| 1 | Arduino Mega 2560 |  | $59.99 | $59.99 |
| 1 | Printed Circuit Board |  | $30.00 | $30.00 |
| ? | Miscellaneous (magnets, screws, small parts) |  | ~$70.00 | ~$70.00 |
|  |  |  | Total | ~$430.21 |

**Table 9.2.1 - Bill of Materials**

## 9.3 - Plexiglas Chess Board Construction

All dimensions were determined by first embedding the magnets in the bases of the chess pieces and testing how distance between the chess pieces affected the magnetic forces between them. It was determined that the minimum center-to-center distance at which two chess pieces did not cause each other to move to be 1.5 inches, which thus became half the distance of one chess square. No chess piece should ever come closer than half a square to another chess piece during the course of a CyberChess game. Using this half square distance, the final dimensions of the Plexiglas panels were determined to be roughly 36” X 30”. Since the size of each individual square on the top pane will be 3”, the total play space should be 24x24 in2.

On the left and right side of the play space were capture zones for captured pieces to move to the side. Sufficient space was designated for two columns of eight squares (although these squares will not be marked) for each capture zone, which allowed for 3x24 in2 on each side of the play space.

## 9.4 - Encasing Construction

The wood enclosure for CyberChess was needed to accommodate our movement system’s hardware so we began by measuring the final board play area, capture zones, as well as the movement hardware to be encased. It was important to note that our movement system would be incapable of reaching certain areas, so the board and encasing was designed with enough space to create an outer buffer of areas unreachable. Only the 4 outer walls and the bottom of the encasing were created as the top of the encasing will be provided by the chess boards. The four outer walls were held together by Velcro straps, which allowed for easy disassembly and access to hardware and the movement system.

## 9.5 - PCB Vendor and Assembly

The printed circuit board (PCB) was housed and electrically connected all of our hardware components.

Eagle was the software used to create our PCB design as it had good support from the industry and many tutorials available. We made a decision to solder our parts onto the board ourselves since it not only gave us experience but helped the project’s budget. The main motivation initially behind having the components soldered on for us was their microscopic size, as it would be very difficult to properly solder our components without the proper equipment. Having them attached by our PCB manufacturer would increase the cost of our board assembly and since we did not have funding, we initially thought about attempting to solder the parts ourselves. Since we switched a majority of our hardware components, the process of soldering them became more reasonable as they were breadboard size.

In addition to Eagle, OrCad and Allegro were looked into since we did not locate all part layouts for Eagle initially. OrCad and Allegro were to be used to export these layouts into Eagle where the rest of the design would take place. Alternatively we looked into using PCB Artist as it is completely free and is supported fully by our manufacturer.

To actually have the board printed we used 4pcb.com because they offered special discounts for engineering students. Additionally, they offered 24 hour tech support which served to be helpful for the few complications that we ran into.

Coding the software for CyberChess was an involved process, ensuring each program operated properly individually and in concert with each other piece it needed to interact with. As such, the coding process was to be intimately tied to the test plan detailed below in Section 8.3.

## 9.6 - Chess Engine

The chess engine went through a number of iterations as the data structures for the game state and moves were refined and relevant code was refactored. As many elements of the engine, such as checking movement validity and executing captures, were rather straightforward to implement, a basic skeleton of the engine was created for each conceived data structure. These skeletons were then fleshed out with attempts at the more complicated functions, such as check detection, and the data structures were further refined as issues arose with these functions. When the engine and relevant data structures were solidly defined in such a way as to minimize such issues, the engine started undergoing more thorough testing as outlined in the referenced test plan.

The functionality required for saving and loading games was implemented with very basic code which needed only minor changes based on refining the data structures used. These functions were needed to respectively create and parse a textual representation of the current game state, using the previously established modified Forsyth-Edwards Notation. Selecting file location to write from and read to was trivial, as was actually writing and reading from this file.

## 9.7 - Board and Movement and Sound Engine

The board and movement engine interacted with various hardware components. This engine, being responsible for a considerable variety of functions, was constructed in a piecemeal fashion, with support for lighting, motors, and audio all addressed individually. There was considerable possibility that each subsection would be divided out into its own engine, at which point the board and movement engine would act more as an interface between the rest of the software and these engines than as an engine itself.

# 10 - Implementing the Design

## 10.1 - Prototype Plan

Prototyping can serve to be very important in determining what design decisions may need to change. Our entire prototype ran using our development board and the same Arch Linux ARM distribution we would be using in the final build. The hardware components were connected via breadboard and for prototyping purposes only, a monitor, keyboard, and mouse were connected to the development board to allow for interaction.

The prototype lacked the enclosure and instead was setup similar to a table, with the Plexiglas boards on top of a support structure allowing easy access to the movement system and hardware for adjustment and control.

## 10.2 - Build Plan

### Hardware Specific Prototyping

An area that will required the most tweaking and optimization was our Movement System. This system was comprised mainly of our stepper motors, vex racks, and our motorized magnet. To ensure that this system was as quiet and fast as possible, the team prototyped this system first. The play area of the game was created using a peg board and Plexiglas combo. Once this play area was acquired and created, it was secured in place using braces and mounts. The movement system was then created below it and wired into the breadboard which was connected to our development board’s general purpose I/O. This prototype allowed us observe our magnet strengths and their effects, providing the feedback necessary to tweak and modify the movement system for the better.

## 10.3 - Test Plan

CyberChess was and is a complex combination of hardware and software built from the ground up specifically for our needs. This meant there could arise problems within the hardware or software that needed testing or debugging. The environment in which all speech-related tests occurred needed to be as quiet as possible due to the fact that our input is the user’s voice. To make sure our system could accommodate background noise, tests were also done with these conditions. Lighting in the test environment is not very important but should be substantial enough to allow all of the hardware actions to be seen easily. The size of the test environment was also negligible but was big enough to house the CyberChess unit as well as four individuals.

### Software Testing

CyberChess contains many software components that required individual tests. A large portion of these tests had to be completed prior to testing the hardware because a large chunk of the hardware tests required working software. Each element of the software was first tested individually to check that prepared input resulted in anticipated output. When two pieces that regularly interact were both functional, they were further tested in conjunction with each other to make sure they cooperated smoothly. As each pair proved operational, longer chains of programs were be tested, building up to tests of the entire system as a whole.

### Chess Engine Tests

As the chess engine operates independently of the speech functions, it could be tested entirely within the production environment alongside its development. An ncurses-based interface was designed to interact with the chess engine in the production environment. This allowed the testers to interact with a visualization of the virtual chess board, testing alterations to the engine by playing with the game.

The design decision to include the ability to request possible moves proved to be fortuitous in testing the solidity of the chess engine implementation. By having the ncurses overlay highlight spots listed by the output of the possible moves function, it was possible to easily set up tests and observe whether the engine was properly limiting moves. This was rigorously tested especially for Pawns and Kings, as their movement options vary dependent on their location and the state of the game. The majority of testing of the chess engine was small scale, manually generated tests accompanied by simply playing the game and testing the possible moves available to pieces at random. One iteration of the engine demonstrated strange edge case bugs during this testing, which lead to a rewrite from scratch of roughly half of the move logic in an attempt to ensure the error was not held on to. All testing that followed this rewrite showed nothing outside of expectations.

In the development of the code for check detection, en passant handling, and castling, a battery of manually constructed tests were performed, setting up the board in a variety of ways that were expected to have these be valid and not. Fortuitously, these never showed any holes in the logic.

### Board and Movement Engine Tests

The Board and Movement Engine was tested first in a terminal in order to ensure that expected output was being obtained. Ascertaining this to be the case, it was then tested in conjunction with the hardware.

Testing of the Board Engine along with the hardware addressed the individual pieces of the engine first on their own, and then together. The lighting portion of the engine was programmatically instructed to light each LED in sequence to ensure that output was handled as expected. Each LED was directed to cycle through the set of colors that are required for the game in a set order to again check that every LED was being accurately controlled. Having established that this control was properly implemented, more complicated sequences of lighting the LEDs were be tested, culminating in tests of each of the sequences to be used during actual gameplay.

The audio control of the engine was tested by simply having the system play each of the audio files in sequence. Once this system had been shown to be reliable, the testing advanced to having the audio play at the same time as the corresponding LED sequences.

The movement system required the most rigorous testing. Various paths were manually created and provided as input to the engine in order to check that the system properly moved as required. Following confirmation of this, actual moves were generated and used as input, checking that the engine properly parsed a move into an efficient path. Once this proved successful, testing moved on to bringing the entire engine together - test input involved moves that the entire system must participate in, mainly the capturing of pieces. This testing was accompanied by observations determining that the proper audio was used, proper lighting sequence was displayed, and the more complicated moves were executed properly.

### Speech Recognition Engine Tests

The Speech Recognition Engine was tested by having a test user speak loud and clear into a microphone. To ensure every component of the engine was working properly, the tests took place in two distinct phases. The first covered all of the vocal commands using piece location place holders to ensure the command is understood. The next test covered every possible piece and location configuration to ensure the system understands every possible move.

The commands were operated in a terminal and documented in the table below with a simple yes if the command was recognized and no if the command was not recognized. The Recognized As column served as a place to document what was actually recognized by the engine in the case where the output is not what the test user spoke.

|  |  |  |
| --- | --- | --- |
| **Command** | **Recognized?** | **Recognized As (If No <-)** |
| *command speech* | *yes/no* | *if no what was it recognized as?* |

These commands were repeated multiple times to attempt to determine the accuracy with which they were heard by the speech recognition engine. On the whole, the system correctly interpreted individual words roughly 90% of the time. Unfortunately, longer commands such as movement suffered from the compounding of possibility of error – moves were properly interpreted only about 60% of the time due to additional words being interpreted or individual words being misheard.

All possible piece and location configurations were then tested to make sure the engine could properly decipher every possible combination. The table below shows how this information was documented.

|  |  |  |
| --- | --- | --- |
| **Piece/Location** | **Recognized?** | **Recognized As (If No <-)** |
| Rook Alpha One |  |  |
| Pawn Alpha Two |  |  |
| etc. |  |  |

Pieces were overwhelmingly properly recognized, as were the files of piece locations. The numbers corresponding to the ranks of the pieces were the most common part of the utterance that PocketSphinx inaccurately interpreted, often mixing in meta-commands instead of the numbers.

### Hardware Testing

### Basic Hardware

Upon acquisition of each major device in CyberChess, the component would be tested by running certain hardware specific code on the Arduino Mega to check the functionality of the component. This section discusses the testing of each device and what the results were.

### Big Easy Drivers

The Big Easy Driver has a potentiometer to control the resistance across its onboard voltage regulator and thus control the output current. Because the stepper motors were rated at 1.7 A, we set the potentiometer such that Vref on the regulator was as follows:

VREF=.88xIo

We got a reference voltage of about 1.4 V. In order to test the motor driver with this reference voltage, we wrote a small Arduino sketch that simply set DIR to either high or low constantly, set ENABLE low, and oscillated between high and low on the STEP pin. We wired the stepper motors and found that the optimal STEP frequency was about 2 kHz. We tested the torque of the motors by placing heave pieces of wood that would be similar in weight to the XY-plotter on top of the motors. The motors had enough torque to drag the weight at a fairly slow pace. We also tested how many steps the B.E.D.’s required to move the motors 3” (the length of a chess square). We found that it took around 4050 steps to cross one square. Therefore, in the code each motor driver would be given 4050 oscillations of the STEP pin for each square the motor has to move through.

### Shift Registers

The shift registers were very straightforward. We tested them with 8 LEDs by sending serial bytes using the Arduino shiftOut function. The shiftOut takes a decimal number input and sends out its binary conversion to the bits of the shift registers most significant bit first (MSBFIRST) or least significant bit first (LSBFIRST) depending on the programmer’s specifications. When we shifted out ‘255’ LSBFIRST or MSBFIRST, all of the bits went high and all of the LEDs lit. When we shifted out ‘0’ LSBFIRST or MSBFIRST, all of the bits went low and no LEDs lit. When we shifted out any power of 2­­­p (where p is an arbitrary integer between 0 and 7); the pth+1 LED would light up in LSBFIRST, and the 8-pth LED would light up in MSBFIRST.

## 10.4 - Extending the Design

CyberChess exists as a fully playable, voice controlled, automated chessboard at this time. However, its design does not necessarily have to stop there. The development team has come up with and looked into a variety of ideas for extending the system, whether to make it more complete or to create an even more unique product. The following are a few of the possibilities for future iterations of CyberChess.

### LCD Display/Video Out

CyberChess currently provides all direct feedback to the user through a combination of visual effects and audio output. Throughout the development process, the idea of providing a visual display in the form of a LCD panel on the control side of the enclosure has been a regular point of discussion. This display would provide more detailed visual feedback to the players, allowing for improved interaction during initial game setup and an additional means of communicating information about current gameplay. Implementing this addition would require supplementing the current software with another overlay of the base chess engine featuring logic to display whatever is desired on the screen.

### Wi-Fi

The addition of Wi-Fi connectivity to the CyberChess system would offer a wide range of additional features, most of which could be considered interesting extensions, but none of which would significantly alter core functionality. Potential extensions would include the ability to upload save data to external storage to permit simplified access to multiple save files and possibly transfer of games between CyberChess units. Game info including win/loss data and transcripts of games played could be shared through social networks. For those seeking true novelty in their gameplay, a CyberChess unit could be remotely accessed by an opponent, allowing for an online game of chess to have a physical element for at least one of the participants.

### Camera

Supplementing the CyberChess system with a camera would permit the unit to monitor its surroundings and the current state of gameplay. A camera system would allow CyberChess to not only monitor its surroundings, but would provide additional functionality such as win and loss photos, taking pictures of the two users upon game completion. In combination with the addition of Wi-Fi connectivity, this addition would offer even more in the realm of social media, allowing two physically separate users to enjoy a game of chess with Skype-like interaction.

### Sensors

While it was seriously considered during initial development, the idea of providing some means of sensing the presence and location of pieces was removed early on in actual construction due to the lack of readily implementable uses for such a system. With the addition of a camera capable of monitoring the current state of the board, the more interesting design options offered by a sensing system become viable. This combination would allow CyberChess to intelligently correct failed movements and perform initial board setup, two features which would greatly improve the user experience. Visual recognition of pieces would require investigation into Computer Vision and a major addition to the software, possibly requiring more powerful hardware than is currently present in the system. As such, this would be a major undertaking that would warrant a second version of the CyberChess project.

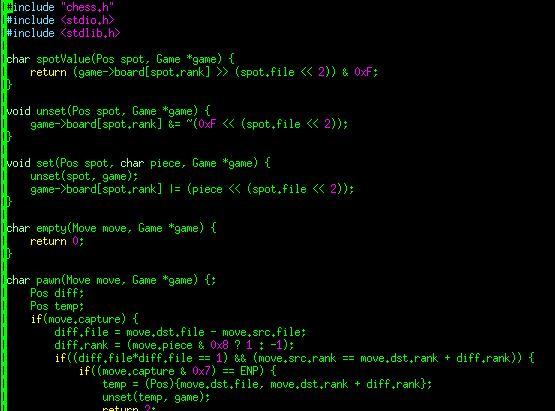
### AI

Integrating an artificial intelligence component into CyberChess would allow for single player enjoyment of the game. While CyberChess is currently designed for use by two players, an AI component could adequately mimic an opponent, providing a full experience for the lone user. There are many implementations available for chess artificial intelligence, both open source and commercial. For the purposes of CyberChess, an open source engine would be the best fit - the vast majority of the components that make up the CyberChess system, both hardware and software are open source. Chesstools is one such open source chess engine with a built-in AI framework that is available on Google code. The AI framework is a compact minimax design that utilizes alpha-beta pruning and transposition tables. A Python extension module named Psyco exists that would be used to significantly improve the performance of the bot within this engine should it be included in a future version of CyberChess. (<http://code.google.com/p/chesstools/>).

# 11 - Facilities and Equipment

## 11.1 - IDE and SDK

The vast majority of coding for CyberChess took place within the development environment using Vim, a powerful text editor that has been in constant for multiple decades. Vim is a spiritual successor to vi, which traces its heritage through ex back to ed, one of the first Unix text editors, originally authored by Ken Thompson. While Vim can be supplemented by numerous plugins to turn it into a very versatile editing ecosystem, the fundamental goal was simple editing of the code and thus such extensibility was mostly ignored. As the team desired to leave the mainframe, which rapidly became the main coding environment, as clean of unrelated clutter as possible, a vanilla installation of Vim on the Raspberry Pi was the main tool used. A screenshot of the Vim environment as well as code specific to CyberChess can be seen below in Figure 9.1.1. All C code written for CyberChess was compiled by the GNU Compiler Collection in the production Arch Linux ARM environment and tested therein.



**Figure 11.1.1 - Vim Text Editor Screenshot with CyberChess C Code**

Prior to the migration of development onto the Raspberry Pi itself, the majority of the programming work took place on an existing 64-bit Intel Arch Linux installation. As all code was written in ANSI C, which is completely portable between architectures, this resulted in a seamless transition into the ARM environment.

Development of sketches for the ATMega 2560 took place in equal parts within Vim and in the Arduino IDE, an IDE provided freely by the Arduino project. The Arduino IDE is primarily a simple, relatively featureless text editor built with support for direct uploading of code to the wide variety of chipsets supported by Arduino. It provides the ability to verify the syntactical correctness of sketches in addition to compilation and uploading of user programs. Through integration with avrdude, an AVR programmer command line utility, the IDE also supports use of AVR programmers to indirectly program a chip.

## 11.2 - Development Board

The OLinuXino iMX233 requires additional components to power and maintain a proper environment for the development of CyberChess. The power supply for the development board will be a 5V wall wart with a max output current of 1A. The debugging interface of the iMX233 will require a SJTAG interface programmer as well as a monitor and keyboard. In order to be usable as practical production environment, the board will require a Secure Digital card with a Linux image flashed upon it.

## 11.3 - Chess Board

The chess board for CyberChess consists of two main parts, the Plexiglas playing board and the enclosure that holds the board in place and houses the XY-plotter. The playing board is a 36” x 30” Plexiglas sheet that lays atop a perforated board painted to resemble a chessboard. The enclosure for the playing board was constructed from wood using a variety of basic woodworking tools. Each wall of the enclosure was outfitted with Velcro that allows the construction to be easily dismantled.

# 12 - Abnormal Operation

## 12.1 - Motors

When troubleshooting the bi-polar stepper motors, one should first measure the resistance of both phases of the motor to determine if the motor has become defective. For each phase of the motor, connect the leads of a multimeter to both wires leading to the coil and check for a small resistance of about 1.7 - 2.4Ω. If the resistance measures anywhere in the kΩ and MΩ region, the coil has become an open circuit and needs to be replaced.

If the stepper motor passes this first test, check for an open circuit between phases by testing each wire leading to phase A with both wires leading to phase B and vice versa. The resulting resistance should be infinite, indicating there is no current shared between phases. If this test produces a resistance in the 0 - 10Ω region, there is a short between phases and the stepper motor needs to be replaced. After these tests have been applied to verify the motors are not defective, it is time to check the wiring of the motor to see if that is the cause of the abnormal behavior of the stepper motor.

The servo and stepper motors needed loose wiring to allow the motors to reach every location of the playing board. This can lead to kinks in the wiring, cutting off current to the motor which will cause sporadic behavior in the way the motor rotates the shaft.

If the shaft of the bi-polar stepper motor vibrates when powered but does not turn, check the orientation of the stepper motor wires connected to the Big Easy Driver. This normally occurs when the output to one phase on the Big Easy Driver shares a wire from both coils of the stepper motor. To remedy this, switch one wire going to phase A with both wires of phase B and observe the behavior. The stepper motor should resume its normal operation once the wires have been aligned in the proper order.

If the stepper motor is rocking back and forth (continuously switching directions), the wiring of the motor is correct but the current is being cut off to one of the coils. Check the wiring connections as this is usually a result of faulty wiring or a bad connection between the stepper motor and the Big Easy Driver. If the servo motor displays the same rocking behavior, again check for a bad connection.

Should the shaft of the stepper motors rotate opposite to the direction expected, simply switch the cathode and anode wires on the Big Easy driver.

## 12.2 - Memory Leaks & Segmentation Faults

The Raspberry Pi used in the demo model of CyberChess is unfortunately prone to memory leaks and relevant errors. In efforts to trace the problem, no consistent error within the logic devised by the development team was identified. The current leading theory is that the speech recognition processing is encountering a race condition that results in a bad memory access. Unfortunately, this results in the CyberChess environment crashing with little warning. If the program exits cleanly after faulting, the mainframe will send a signal to the microcontroller such that it can visually notify the user of the error and reboot into a new game of chess. Should the program instead hang due to the memory error, gameplay will simply appear frozen to the end user. If the user has a USB keyboard on hand and ready access to the Raspberry Pi that acts as the CyberChess mainframe, they can plug in the keyboard, press “Ctrl+C” and then type “sudo reboot” in order to reboot the system. Otherwise the user may simply cycle the power to CyberChess in order to force the Raspberry Pi to reboot. This is less favorable due to the possibility of corrupting the SD card that acts as the Raspberry Pi’s main memory, but as that is highly unlikely, it is considered a valid option.

# 13 - Operators Manual

**Step 1:** First make sure CyberChess is properly wired. The bi-polar stepper motors should be connected to the female header pins attached to the phase output on the Big Easy Drivers as seen in Figure 12.1 below.

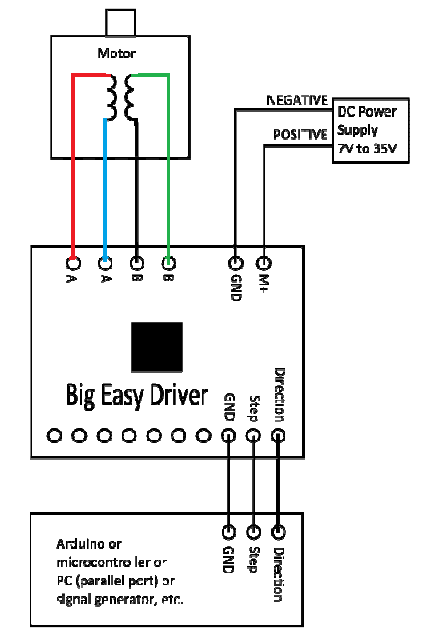


Figure 12.1 - Big Easy Driver Connections

The orange wire on the servo motor should be connected to the servo signal output located above the B.E.D.’s on the PCB. The red wire on the servo motor should be connected to the 5V output from the LM7805 voltage regulator, while the black wire should be grounded. In order to set up the 7 segment display showing the chess clock, connect the wires corresponding to each segment in the display to the shift register for the chess clocks on the PCB in the order of A to G. The final header pin on the chess clock shift register is for the colon on the 7 segment display. Since the shift registers for the clocks are daisy chained together, only one set of digit select pins from the 7 segment displays need to be connected to the PCB. The second chess clocks digit select pins are then put in series with the first clocks digit select pins. Figure 12.2 shows the pinouts of the 7 segment display and the shift registers and Table 12.1 shows what pins to connect. Some pins on the 7 segment display are excluded because they do not need to be used. This may not be the case with excluded pins on the shift register or microcontroller.

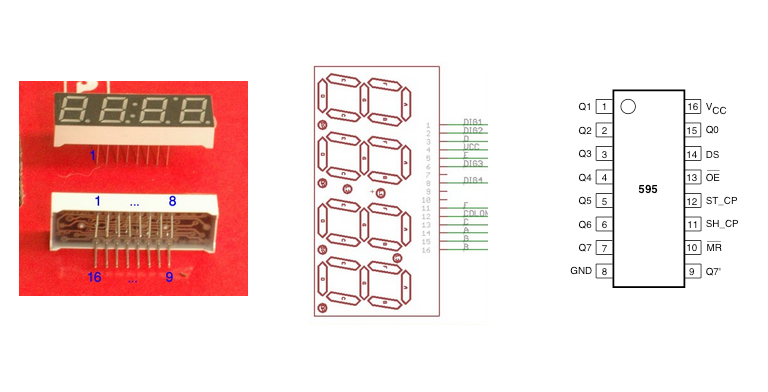


Figure 12.2 - 7 Segment Display/Shift Register Pinouts

|  |  |  |
| --- | --- | --- |
| **7 Segment Display** | **Shift Register** | **ATMega 2560** |
| 1 (DIG1) |  | 6 (PE4) |
| 2 (DIG2) |  | 7 (PE5) |
| 3 (D) | 3 (Q3) |  |
| 5 (E) | 4 (Q4) |  |
| 6 (DIG3) |  | 1 (PG5) |
| 8 (DIG4) |  | 5 (PE3) |
| 11 (F) | 5 (Q5) |  |
| 12 (COLON) | 7 (Q7) |  |
| 13 ( C ) | 2 (Q2) |  |
| 14 (A) | 15 (Q0) |  |
| 15 (G) | 6 (Q6) |  |
| 16 (B) | 1 (Q1) |  |

Table 12.1 - 7 Segment Display Pin Assignment

The anode side of the RGB LED’s in the LED array need to be wired into the LED shift registers in the following order, where B is for the bottom half and T is the top half of the array: BR0 - BR7, BG0-BG7, BB0-BB7, TR0-TR7, TG0-TG7, TB0-TB7. The cathode side of the LEDs in the array are grouped into eight wires representing each row on the board. The order of the eight wires in each row is not important when connecting to the PCB, what is important is that the rows are separated from the others and are connected to the female header pins on the PCB in the following order: R1, R5, R2, R6, R3, R7, R4, R8. CyberChess is ready for power once the stepper motors, servo motor, chess clocks, and the LED array are all properly attached to the PCB. Table 12.2 (below) assigns pins from ATMega 2560 to the corresponding rows on the chessboard according to their traditional chessboard names.

|  |  |
| --- | --- |
| **Rows** | **ATMega 2560** |
| 1,5 | 75 |
| 2,6 | 76 |
| 3,7 | 77 |
| 4,8 | 78 |

Table 12.2 - LED Matrix Row Mapping

Table 12.3 (below) shows the assignment of the columns on the LED Matrix to the shift register (of an arbitrary color and set of rows) according to their traditional chessboard names.

|  |  |
| --- | --- |
| **Column** | **Shift Register** |
| A | 15 (Q0) |
| B | 1 (Q1) |
| C | 2 (Q2) |
| D | 3 (Q3) |
| E | 4 (Q4) |
| F | 5 (Q5) |
| G | 6 (Q6) |
| H | 7 (Q7) |

Table 12.3 - LED Matrix Column Mapping

Because each column is split into 2 sets of 4 rows (top half and bottom half) and each of those sets of columns can be split into 3 colors, it is important to make sure each of the 6 required shift registers is assigned correctly to its proper color and row according to the CyberChess program. The header pins next to the shift registers should be labeled with a letter and four numbers to signify the color and rows assigned to that set of headers. The labeling syntax is R for red, G for green, and B for blue and the numbers will be 1234 for rows 1,2,3, and 4 or 5678 for rows 5,6,7, and 8. (Ex. R-1234 is Red, rows 1,2,3, and 4). The wires are color coordinated such that red wire corresponds to the red LEDs, green wire corresponds to the green LEDs, and yellow wire corresponds to the blue LEDs. The user must check how each set of wires is routed to know which 4 rows each wire is linked to.

**Step 2:** To ensure CyberChess will boot up into the proper environment, check that all external accessories are appropriately connected. A powered USB hub is used to grant the Raspberry Pi sufficient USB ports to properly run the system. Ensure that the USB hub is plugged into the Raspberry Pi and that the two microphones are plugged into this hub. Plug the USB end of the FTDI cable into the Raspberry Pi directly and attach the opposite end such that the following pairs match up: GND/Black, CTS/Brown, VCC/Red, TX0/Orange, RX0/Yellow, RTS/Green.

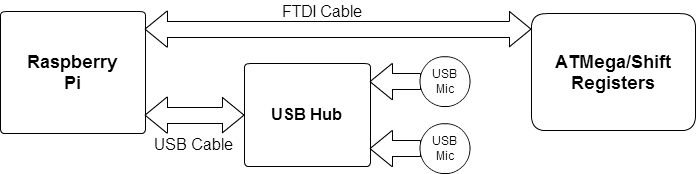


Figure 12.3 - USB Communication Setup

**Step 3:** To turn on CyberChess, plug in the two wall adapters needed to power CyberChess. The Raspberry Pi is powered by a USB-micro connected to a wall outlet. The Raspberry Pi powers the ATMega 2560 and the shift registers with 5 volts through the FTDI cable. Because the Raspberry Pi is unable to provide the current needed to run motors, the servo motor and Big Easy Drivers are powered by a separate wall outlet. A wall adapter at 7.5 V is plugged into the jack on an orange vector board, which is connected to the power terminals of the servo motor and Big Easy Drivers.

The Raspberry Pi must be powered on first, and the program should be given a few seconds to boot before powering the motor section. The Big Easy Drivers should never be powered before receiving the proper logic signals (which ultimately are provided by the power going to the Raspberry Pi). If the Big Easy Drivers are powered before their Enables are initialized, unwanted current will flow through the drivers and heat the drivers. If this goes on long enough, the Big Easy Drivers could malfunction and stop working permanently.

**Step 4:** Once CyberChess has been powered, a new game will be started once issuing the proper command into the first player’s microphone. The board will display an LED pattern, along with playing audio, indicating that a new game has begun. If a previous game was saved, the users will have the option to load a previous game and its settings by issuing the command “Load”. If both users would like to start new game, one of the users must give the command “New”. Both users will then decide on the settings of game such as the volume of the speakers, the lighting of the LED array, and if a chess timer will be used at all.

**Step 5:** Throughout the game, the current player will dictate their move into a microphone using the standard format of “[Corresponding Piece Military Alphabet Pronunciation]][Current Square] to [Destination Square] (e.g. “Bravo 1 to Charlie 3” for at “B1 to C3”). Should the player have requested a move that is either illegal or invalid, the problem will be communicated through the speaker system, audibly informing the player that their move cannot be made. This notification will be accompanied by use of the LEDs to visually show the problem. The player may ensure that they will request a valid and legal move by asking for all possible moves for a piece using phrasing along the lines of “Show possible moves for [Piece Corresponding Piece Military Alphabet Pronunciation][Current Square]” (e.g. “Show possible moves for Delta 3”).

**Step 6:** Either player may offer a draw at any time, which will end the game if the other player accepts. In any of these instances, the board will display the appropriate lighting sequence and gameplay will terminate. Additionally, either player may choose to resign at any time and, after receiving confirmation that the player does really want to resign, the game will end and the winner will be indicated.

**Step 7:** Game Conclusion takes place in the normal means available in any standard chess game. If a checkmate occurs, CyberChess will display an end of game lighting sequence and audibly inform the players of the outcome, acknowledging the winner

**Step 8:** It is important to correctly power off CyberChess as to not damage the SD data connected to the Pi or the Big Easy Drivers by providing no logic. When deciding to end the game the user must disconnect the wall adapter responsible for powering the motors before giving the command “End” to shut down the Pi.

# 14 - Administrative Content

## 14.1 - Milestones

Below are specific deadlines intended to keep progress on design and implementation of CyberChess moving steadily forward such that a working prototype is available for presentation by mid-April. All future dates are, of course, tentative, but will be treated as binding so as to encourage constant work. Past dates are included to demonstrate progress already made.

**October 15, 2012**

*Initial Design and Initial Research*

The overarching idea of the project has been firmly established and initial problems and decisions are being addressed. While no final decisions have been made about parts, hardware or software, the basic ideas of what will be needed and how they will interact are known. Investigation and comparison of the various products available to meet these needs has begun.

**November 15, 2012**

*Research Phase Complete*

While it is recognized that design ideas may yet change to deal with problems or opportunities for improvement that arise during implementation, a basic design has been drawn up. Based on conclusions drawn from research in the previous phase, decisions about specific hardware and software for this design have been made. Both research and the decisions based on it are prepared for documentation.

**December 6, 2012**

*Research and Design (Senior Design 1 Documentation) Finalized*

The documentation is to be turned in at the end of the Fall 2012 semester which concludes on December 6, 2012. Having our Senior Design 1 Documentation completed a few days before gives us time to proofread, make changes, and print the final copy to be professionally bound. Each member of the team is expected to contribute their fair share of the total 120 pages of documentation, which equates to 30 pages per team member.

**March 7th 2013**

*All Parts Acquired*

This date marks the beginning of the Spring 2013 semester when the team will be in Senior Design II. The remainder of this semester will be dedicated to developing the working prototype of CyberChess, and thus materials and relevant equipment should be available so that work can immediately begin. The initial parts, which it is recognized may be replaced as better alternatives are discovered, are outlined in the hardware section of this document.

**March 30, 2013**

*Testing of Individual Parts Completed*

Each part must be tested individually to ensure that it runs properly on its own - should it fail as an individual, there is no reason to expect it to work in concert with the other pieces of the project. This testing should not take a significant amount of time and should be completed rapidly so that any problems may be addressed immediately. Successful completion of this testing will allow for continuation of the build process.

**April 5, 2013**

*Prototype Created*

Using all of the initial parts and the development board the team will create the prototype. This prototype will be used to further develop the hardware and software and test their implementations.

**April 11, 2013**

*Testing Phase 1 Completed*

This milestone marks the completion of the testing phase using the prototype hardware. The ending of test phase 1 means that all of the tests have been passed for both hardware and software. Any test that is failed will be addressed by way of modifying the design and prototype until the test is passed. As such, at the conclusion of this phase, there will effectively be a working development prototype.

**April 17, 2013**

*CyberChess Final Product Created*

After creating and testing the prototype, the final hardware will be created and implemented with the software uploaded to it. This milestone includes the creation of the printed circuit board, the soldering of our parts, as well as construction of the enclosure.

**April 29, 2013**

*Senior Design II Documentation and Testing Phase II*

Upon constructing the final CyberChess product, the Senior Design II documentation will be finalized in order to be presented to the review board alongside the product.

**April 12, 2013**

*Final Presentation*

This milestone marks the completion of the final presentation. This milestone can only be reached if every previous milestone in this section is completed. The team’s graduation is dependent on being successful during this presentation. Being successful requires that the product meets all of the specifications and requirements that were initially agreed upon as well as the requirements of the review board.

|  |  |  |
| --- | --- | --- |
| **Date** | **Milestone** | **Met?** |
| 10/15/2012 | Initial Design and Initial Research | Yes |
| 11/15/2012 | Research Phase Completed | Yes |
| 12/4/2012 | Research and Design Documentation (Senior Design 1 Documentation) Finalized | Yes |
| 3/7/2013 | All Parts Acquired | Yes |
| 3/30/2013 | All Parts Individually Tested | Yes |
| 4/5/2013 | Prototype Created | Yes |
| 4/11/2013 | Testing Phase 1 Completed | Yes |
| 4/12/2013 | Final Presentation | Yes |
| 4/17/2013 | CyberChess Final Product Created | Yes |
| 4/19/2013 | Final Demo | Yes |
| 4/29/2013 | *Senior Design II Documentation and Testing Phase II* | Yes |

**Fig 10.1.1 - Milestone Review Table**

## 14.2 Budget and Finance Information

All financial issues with CyberChess were directly handled by the team members. The total cost of the project came out to approximately $430 and has been split amongst all group members evenly.

## 14.3 - Summary and Conclusion

The team has also learned valuable lessons about the importance of time management in the midst of large projects. Without properly addressing time management, there is simply no way this project could ever come together in time to match the deadlines dictated above. This became yet more evident through the production of the system presented on Senior Design day. The project required large commitments of time and energy from all members of the team.

In addition to learning to budget time, learning to budget financial resources was key in effectively implementing CyberChess. As the project was entirely funded by the team, it was quite important on a personal level to guarantee that associated costs remain low. Thorough research of available options ultimately lead to discovery of cost-effective solutions to meet the requirements and specifications of CyberChess.

All of this collaborative effort has had the positive impact of giving every member of the team significant experience with working on a team in which each participant brings a different skillset to the table. Learning to make use of each member’s strengths and cover for their weaknesses has allowed this project to evolve to a state that no individual member ever could have assembled alone. This is excellent preparation for entering a similar environment in the real working world where such collaboration is expected on a daily basis.

All of this learning and new experience is excellent itself, but it is made all the better by having something to show for it. The team is excited to have realized the design described in this document and was greatly relieved to have been able to present a working prototype on Senior Design Day in April 2013. With a collection of elements that interested each team member, CyberChess truly became an excellent investment.