Rubik's Cube Solving Robot

EEL4915—Summer 2016

Group 12

Daniel Truesdell
`Electrical Engineering`

Corey Holsey
`Computer Engineering`

Tony Verbano
`Computer Engineering`

UNIVERSITY OF CENTRAL FLORIDA

August 1, 2016

Contents

1 Executive Summary

The Rubik's Cube Solving Robot is a robotic system whose purpose is to autonomously solve a Rubik's Cube puzzle. To accomplish this task, the robot is equipped with an image sensing device and an embedded computer that is interfaced with mechanical hardware. The Rubik's Cube has inspired our project because it is a timeless puzzle that has challened both man and machine for decades.

Lots of research has been done to investigate the mathematical properties of the cube and deduce algorithms to achieve certain manipulations of the cube faces. These mathematical algorithms have enabled computers to process data of a scrambled cube and determine what physical manipulations are needed to solve it. Careful design of a robotic system allows a computer to directly apply these manipulations to a cube via a mechanical apparatus.

Our system functions in this way by first taking images a scrambled cube and performing image processing to determine the current position of each cube face. An embedded processor will run an algorithm to determine what cube manipulations are needed to solve the puzzle. Then, the processor will control a mechanical apparatus to perform these manipulations and solve the cube. The objective for this process is to accurately solve the puzzle in a timely manner.

2 Introduction

2.1 Project Description

This project features the design and implementation of a small tabletop robot capable of autonomously solving a three-sided Rubik's Cube puzzle. The robotic system consists of an embedded computer, a mechanical apparatus for physically manipulating the cube, and peripheral devices such as cameras and graphical displays. The robot is designed to receive a scrambled Rubik's Cube, visually evaluate it, and determine how to solve the cube through manipulation. The robot is equipped with the necessary hardware to quickly manipulate the cube until it is solved.

2.2 Statement of Purpose

Our motivation to engage this project stems from our enjoyment of the Rubik's cube puzzle. The multifaceted challenge of solving it autonomously incorporates our technical backgrounds in hardware and software which makes it a suitable final design project. In recent decades, many different machines of various complexities have been created to solve the cube autonomously. As with many of these existing machines, the importance of our work lies in the uniqueness of our implementation. By combining hardware and software technologies into a robotic system, we contribute a unique and well-documented approach to autonomously solving the Rubik's Cube puzzle.

2.3 Goals and Objectives

The main goal of our project is to design and build a robot that can solve a Rubik's cube. This goal can be accomplished by breaking down the project into smaller goals. First the robot should be able to visualize all the side and colors on the cube. Then with the information found run an algorithm in its database to solve the cube for all the same color on each side. After the algorithm is ran the robot then shall take the cube and spin the sides in the corresponding pattern to solve the cube. Each step stated we determine to be a goal for our project. Lastly we also determined to attach a monitor to our device that allows the users to see how many steps are left on solving the cube, the time the cube has taken to work and other functions we find applicable for our project's design.

2.3.1 Performance Requirements

Our robot's requirements are broken down into small parts of equal importance:

- The robot must solve a Rubik's cube 75% of the time.
- The robot must be able to analyze 75% of the cube's face and place it into a matrix correctly
- The robot must be able to analyze atleast 75% of the cubes colors on each face of the cube and place it into a matrix correctly.
- The robot must solve the cube in at least 15 minutes time.
- The robot must be able to rotate each face of the cube 90 degrees to the right.
- The robot must be able to rotate each face of the cube 90 degrees to the left.
- The robot must be able to rotate each face of the cube 180 degrees to the right.
- The robot must be able to rotate each face of the cube 180 degrees to the left.

Solve Time The goal of our robot is to be able to solve the Rubik's cube in a very reasonable time. The fastest Rubik's cube solving robot according to Guness World Records is set at .887 seconds. This record was set February 23, 2016 by the robot named Sub1 and build by Albert Beer.

Our robot will not be able to compete with this time, but with some fire tuning hopefully can solve the cube within a range of 60 seconds. We always strive to be the best we can, therefore if we can make our robot faster we will do everything in our power to make sure it is. Also we must take into account how the solve time may vary because of our use of the camera to visualize the cube. This process of visualizing the cube could slow down the time it takes to solve the cube altering our actual time to solve it.

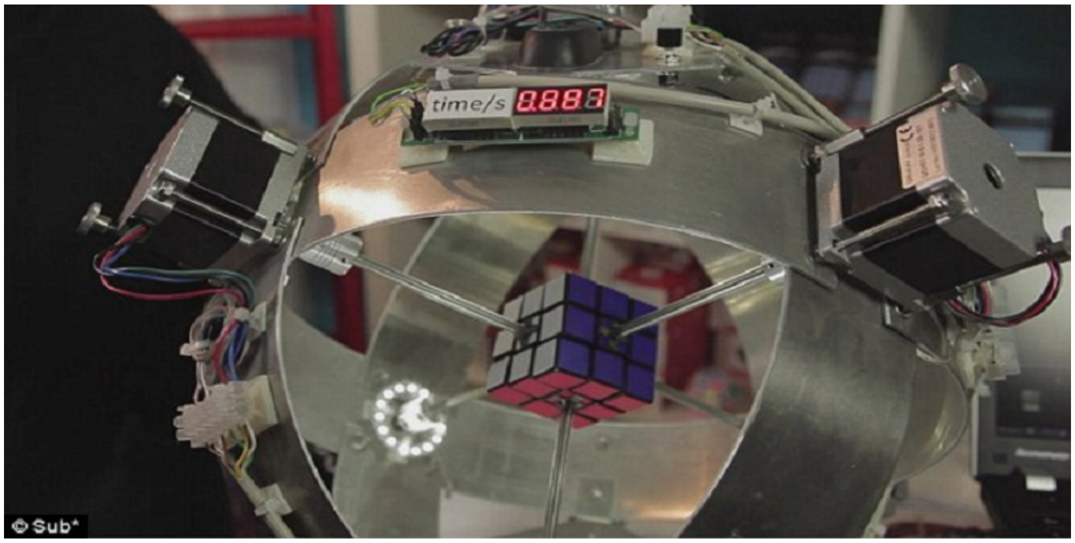


Figure 1: Sub1 with fastest time for solving a Rubik's cube

Solve Accuracy The goal of any Rubik's cube is to solve the cube for the same colors on all 6 sides. We have determined that the robot should be able to solve the cube completely without error at least 95% of the time. With that being said, that means when the program has initiated the cube must visualize the sides correctly from start to finish. Then the program must determine all the moves for the cube to be solved correctly. Lastly the robot must follow the cube pattern correctly from the program for the cube to be solved correctly. If any of the steps were to have a flaw or a mistake the robot may not be able to solve the cube therefore resulting in our accuracy to be lower than 95%.

2.3.2 Functionality

The main functions of our robot are as follows:

- Visualize a face of the cube
- Flip the cube to another face
- Arrange all the faces of the cube on a 2D plane
- Display the GUI of the cube
- Show the steps to solve the cube
- Display a timer
- Solve the cube to allow all the faces to contain the same colors in each different one (GUI)
- Rotate a side of the cube
- Solve the cube (with the mechanical arms)

Movement Movement is one of the biggest aspects of our robot. If the robot can not move a side of the cube than it will not be able to rotate the sides to solve the cube and our project will not be a success. As we researched the methods on how to build a robot to solve a Rubik's cube we found there were many different methods to rotate the sides of the cube to solve the cube. In our research we found that one of the simplest methods was to have only one arm with the ability to rotate a side of the cube. This method would require a lot of flipping the cube around to each side because one one space may rotate. We have determined this method may not be the best path for us to follow in solving the cube, because it would take to much time and we would like our robot to solve the cube in a decent amount of time. The next method we found was to have 3 rotating arms. This method seems very logical and easy to use for us. The arms would be on the left and right sides of the cube and one underneath it. This would allow us to easily flip the cube visualizing every side and also allow the rotation of each side to be quick and flawless. The next method seems to be the fastest but doesn't seem feasible in our minds. It would require every side of the cube to have a center connector piece that would fit around the middle space of the cube. This would allow the rotation of each side without flipping the device and be the most efficient way to rotate each side of the cube. Our issues with this design is that each side of the cube would have to have a camera in order to visualize each side because there would be no flipping involved. That would make our project to expensive for our likings and make the project more difficult than in has to be.

With our design choice we have decided that the robot must be able to flip the cube on all of its sides therefore making it easier to visualize and easier to solve. This movement is very simple because all it takes is the rotation from our 3 mechanical arms just in different patterns. The robot is designed to hold onto the cube with all 3 arms unless a rotation or flip is in order. When the robot decides to flip to another side the left and right arm will grasp the cube together and rotate in the same direction flipping the cube to another side. When the robot needs to rotate the face of a cube without rotating any of the sides the left and right arm shall let go of the cube and the bottom arm shall rotate on its on allowing the robot to visualize the rotated part of the cube. The only issue with the design of our cube is that only 3 faces of the cube can be rotated without flipping the cube into a new direction. This may cause our robot some delays in solving the cube due to time spent flipping the cube.

Capability The robots main goal is to solve the Rubik's cube therefore limiting its capabilities to a narrow spectrum. The first capability the robot should have is the ability to solve a standard 3 x 3 x 3 Rubik's cube. If we are capable of solving the 3 x 3 x 3 Rubik's cube and our project is finished and we have more time than we will try to design our robot to solve a more complex cube ranging from 4 x 4 x 4 to 20 x 20 x 20. These cubes would take a lot more time to write codes and algorithms for, which results in our first decision to just solve a 3 x 3 x 3.

Next our robot should be able to display a GUI for the user to see and interact with. The GUI should contain a timer so that it can track how long it takes the cube to be solved. It should layout a 2D or 3D design of the cube that the user can see and witness how the cube is solved with our algorithms. Lastly the GUI should involve a move counter to tell the user how many moves have been or need to be made to solve the Rubik's cube in person.

The robot should be capable of visualizing all the sides of the cube. This being said the robot should have a camera thats able to determine the colors on each face of the cube and send it to our GUI, that can then decipher the image and determine what colors are on that face. The robot the should be able to move the cube to another side and that face shall be then transmitted to the GUI as well with the other and properly places in our program allowing for no errors when the cube is solved.

Lastly the robot should be capable of rotating a Rubik's cubes side and flipping the Rubik's cube to all of its sides. This will be done using a 3 mechanical arm technique where the arms lie to the left and right of the cube and one underneath. Both the left and the right arm shall work together when flipping the cube to all the faces and they shall also work together to rotate the sides of the cubes that face them. The bottom arm shall rotate the cube when the left arm and right arm are not holding on. and rotate the bottom face of the cube when the left arm and right arm are holding on.

Autonomy The autonomy of our robot is a very simple design. Our robots overall minimal view is to have 6 different mechanical arms all attached to the center of each face of the cube. The top portion of the design shall be removable to take out the cube an alter then replace back into the design. The mechanical arms will all be attached to stepper motors. The motors are designed to rotate each mechanical arm 90 degrees to the left and right and 180 degrees to the right. The motors are attached to our MSP430 board through I/O connections. The MSP430 is attached to the back of our robot and is the main computer to our system.

Attached to the MSP430 will be a camera named the Pixy Cam. The Pixy cam will be attached above the front faced stepper motor looking down on the top row of the front face of the cube and looking at the bottom row of the top facing cube. These two rows will be altered during the vision process. All segments in the cube shall pass through these 2 rows therefore we found it the most applicable place to leave the camera. The camera has a processor attached to it to allow useful data to be transfered from the camera to the MSP430 board.

The MSP430 runs our main algorithm and takes in the matrix found by the camera. This is our embedded computer but attached to it is a USB port. With this USB port we can attach another computer to run applications or programs to help the understanding of our robot.

Attached to the top of the robot will be a monitor that will show our GUI and the robots actions. This monitor should be attached to the main frame so that the users can witness what the robot is doing while the robot is doing it and allow the user to easily see the time it takes the robot to solve the cube. The main frame of the robot will house all of these features and protect them from being damaged from outside elements and interferences.

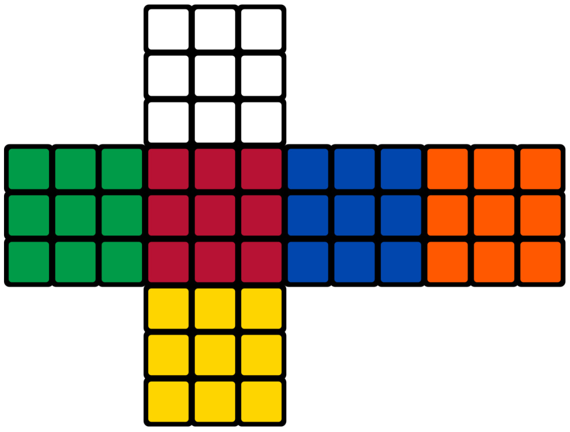


Figure 2: The Cube layout on the GUI

2.3.3 Features

The features that our robot includes are:

- GUI
- Replica of cube on display
- Timer
- Move counter
- Move list

Graphical Display Our robot is supposed to make solving a Rubik's cube easy to understand and fast at the same time. It would be hard to solve the cube at a fast pace while slowly showing off the moves it making. We found the easiest way to solve this is to add a GUI to our robot so it can show the initial cube, the moves it makes to solve the cube, and solve the cube in the mean time. This design makes it easier for the users to learn what the robot is doing without sacrificing any speed to our robots mechanics. Our GUI also will have displayed a timer for the users expense. This timer will show how long it takes the robot to solve the cube without any interruptions. The cube will also be displayed on the monitor so that the user can see the initial cube before the robot starts to solve it. The cube layout will be a 2D design that is basically a cut open version of the cube.

2.3.4 Cost Requirements

Our cost requirements are not set in stone because of our flexibility with the project. Our monitors may change or other parts may change but most likely not but we thought we would include the fact that they could to improve our product design.

Name	Serial Number	Model Number	Developer	Quanity	Price per item	Total price
Capacitor .01 uF	2721051	RadioShack	6 pieces	1.49		
Capacitor 100 uF	EKXG201ELL101ML20S	United Chemi-Con	6 piece	1.47		
Capacitor .1 uF	FK22X7R2J104K	TDK	19 pieces	1.48		
Capacitor .47 uF	ESMG451ELLR47MJC5S	United Chemi-Con	7 piece	.84		
Capacitor 1 nF			1 piece			
Capacitor 10 pf, 6 V			1 piece			
Capacitor 4.7 uF, 10 V			1 piece			
Capacitor 220 nF, 10 V			2 pieces			
Capacitor 100 nF			1 piece			
Capacitor 10 uF			1 piece			
Resistor 1 Mohm			7 pieces			
Resistor .2 ohm			12 pieces			
Resistor 20 kohm			6 pieces			
Resistor 220 kohm			6 pieces			
Resistor 470 ohm			2 pieces			
Resistor 1.4 kohm			1 piece			
Resistor 27 ohm			2 pieces			
JP4 header pin			6 pieces			
Switch SKHMPSE010			2 pieces			
LED			2 pieces			
DC Jack X1			1 piece			
Resonator CSTCR6M00G53Z			1 piece			
Serial I/O			1 piece			
JTAG			1 piece			
Mini USB Shield-UX60-MB-5ST			1 piece			
TPD3E001			1 piece			
U2B Value			1 piece			
MSP430F6659			1 piece			
U5 DRV8825-PWP-28			6 piece			
U9 TPS715A01-DRV-6			1 piece			
Pixy Cam			1 piece			
Servos			6 pieces			

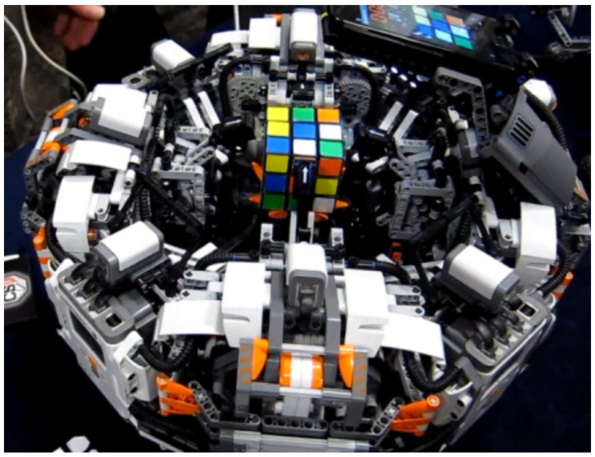


Figure 3: Modern Lego-based Robot

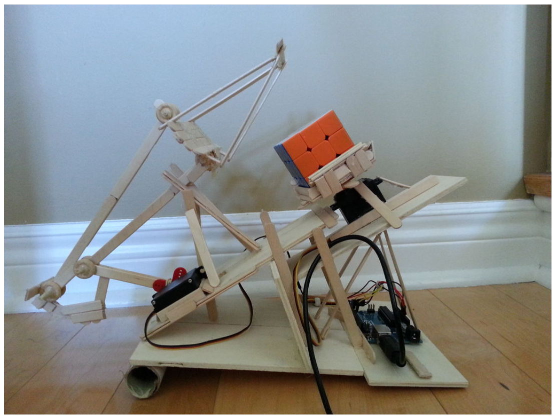


Figure 4: Popsicle Stick Robot

3 Research

3.1 Related Projects

As stated in the introduction, Cube-solving robots have been in existence for many years, with origins in the early 2000's. One of the earliest documented designs was constructed with the Lego MindStorms robotics platform, which features motors and a small microprocessor. The robot, which took months to complete, solved cubes in 15 minutes with 95% accuracy [33]. Lego-based designs are still popular today, and have broken multiple world records for solving speed. Figure 3 shows a Lego design from 2012 that solved a cube in only 5.2 seconds, a world record at the time [13]. Although Lego designs have historically been dominant, many other structural implementations have been used in recent years, ranging from popsicle sticks, shown in Figure 4, to custom machined and welded metal, shown in Figure 5. Aside from physical design, existing robots all vary in their software infrastructure, peripherals, and hardware-software interface. Some projects place emphasis on advanced programming to make up for simplistic hardware, while others focus on system portability or overall speed. An important influence on these design considerations has been time: while hardware abilities

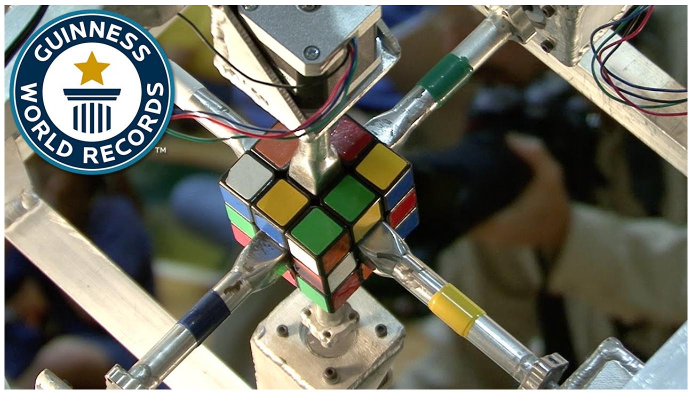


Figure 5: Metal Robot

have remained consistent over the last 15 years, there have been many changes in the enabling technologies for software development that are made evident by consistently innovative software implementations. Constant improvements in general computing performance have led to faster algorithms and faster robots.

3.1.1 Physical Design

The most simplistic designs resemble the popsicle stick robot in Figure 4 and feature one controllable rotational degree of freeom and one translational degree of freedom which is used to indirectly achieve a second rotational degree of freedom on the cube by pushing it till it rolls over.

A small step higher in complexity features two controllable rotational degrees of freedom, like the claw apparatus shown in Figure 6. These designs can manipulate any side of the cube but must first reorient the cube several times before performing a manipulation. The tradeoff for these robots is reduced hardware and increased structural simplicity at the expense of solving speed and additional software programming.

Designs of intermediate complexity resemble Figure 7 and achieve three controllable rotational degrees of freedom with three or four claws. These robots can manipulate any side of the cube with minor reorientation. These designs balance solving speed with hardware complexity and software simplicity.

The fastest and most complex designs feature three controllable rotational degrees of freedom and have the ability to manipulate any side of the cube without having to reorient it. This design is shown by Figure 5. The caveat to this design is that the cube must be physically altered in order for the robot to control it. Specifically, these robots require the middle square of each cube face to be removed or carved with a rotationally symmetric pattern of low order (like the head of a screw) so that the entire cube face can be rotated via its central axis.

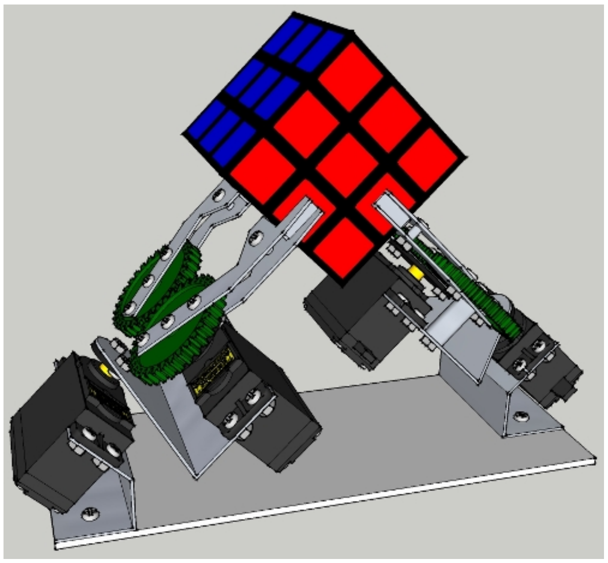


Figure 6: Two-Claw Robot with Two Rotational Degrees of Freedom

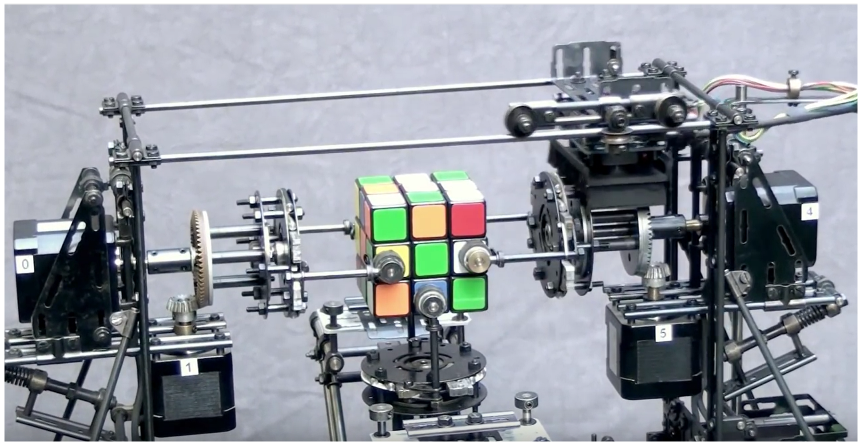


Figure 7: Three-Claw Robot with Two Rotational Degrees of Freedom

As shown by the figures, building materials are diverse, ranging from Legos to wood to metal. With the growing accessibility of 3D printers, some newer designs are now being 3D printed [39]

3.1.2 Technology

Early designs such as the original Lego MindStorms robot were innovative and still continue to inspire modern system infrastructures. The MindStorm robot creator had to design color-recognition software from scratch because there was no pre-existing computer software that worked for the purpose. After receiving images of the cube from a digital camera over a USB connection, the color-recognition software used an infrared transmitter to communicate with two Lego RCX devices containing Renesas 8-bit microcontrollers which drove the servo motors.

The modern Lego robot in Figure 3 is roughly 10 years ahead of the original MindStorms robot, and uses a smartphone camera to capture images of the cube. The two-phase, multi-threaded solving algorithm is run on the phone's 1.2GHz ARM Cortex-A9 processor and instructions are sent via Bluetooth to four Lego NXT devices each containing a 32-bit ARM7TDMI-core Atmel AT91SAM7S256 microcontroller.

The fastest robot currently in existence, which features a design similar to Figure 5, abandons the use of smaller processors and recruits the processing power of a dedicated desktop PC with Linux to run Kociemba's Algorithm (see Pg. 18) and control the motors. Additionally, every communication interface is wired in order to achieve maximum data speed.

3.1.3 Performance

Two primary performance metrics for cube-solving robots are accuracy and solving time. Each of these factors are influenced by all aspects of the robot design. Physical design options such as additional motors or grippers will improve the solving speed by reducing the number of times the cube must be reoriented prior to manipulation.

While physical and structural design undoubtedly play a role in robot performance, the solving speed and accuracy are perhaps most greatly affected by the software and programming. Modern processing platforms are capable of evaluating cube data and and producing solving instructions within seconds, at which point the goal of solving the cube is limited only by how quickly and accurately the hardware can execute the determined instructions. Many cheap motors can run at very high speeds, however their motion is only as fast and accurate as the PWM that drives them. It follows that the fastest robots all attempt to solve cubes in the fewest number of steps with the fastest manipulations possible using dedicated hardware for motor control.

As was previously mentioned, early robots achieved solve times over 10 minutes and accuracies just under 100%, while newer robots are solving cubes within seconds with 100% accuracy. The fastest robot in existence as of February 2016 has recorded a solve time of just 0.9 seconds [11].

3.1.4 Applicability

While Lego-based structures offer an interesting design solution, they are most commonly implemented by less experienced hobbyists who aren't interested in fully developing a hardware-software interface and software infrastructure. Based on the high level of our application and our technical experience, we believe that our system is best implemented with a more permanent structure with greater design flexibility such as metal or 3D-printed plastic.

In addition to their dominance in solving speed, designs like the one in Figure 5 also benefit from straightforward cube manipulation. Although our goal for system performance isn't to set a world record, these designs appeal to us because they would allow us to easily manipulate each side of the cube without a claw or gripping device.

3.2 Rubik's Cube

Breaking down the Rubik's Cube as much as possible is necessary to solve it. The Rubik's Cube is a 3x3x3 cube puzzle with 6 different colors on each side traditionally being red, blue, yellow, green, white, and orange. It has a standard size of 5.7 cm on each side which is about 2.25in. Due to there not being a center cube instead of it having 27 cubes it only has 26. The center cube is used for rotation of each side, or face, of the cube. Each face of the cube will be referenced as Front, Opposite, Left, Right, Top, and Bottom. Each color, or variation, will be assigned a side due to whichever state the cube is in.

In a solved Rubik's Cube the yellow face is towards the user then it will be referred to as the Front. The blue face will be referred to as the Left. The green face will be referred to as the Right. The orange face will be referred to as the Top. The red face will be referred to as the Base. And the white face will be referred to as the Back. Also, each face will consist of layers. The vertical layers will be referenced are the Top, Middle, and Bottom layer. While the horizontal layers sill be reference as Left, Center, and Right.

3.2.1 Mapping the Cube

It is important to throughly map the Rubik's Cube because there are 54 cubies to keep track of. This because extremely important when developing code so that each placement of acubie can be correctly noted. Also, when explaining algorithms it is important to know which exact piece of the cube is being referenced.

When breaking down the faces there are 9 cubes, or cubies, that are in place. Understanding what each cube consists of and where it is will allow for it to be mapped accordingly. In the figure below you will find an image of the Rubik's Cube mapped out as well as a definition of each mapping.

Front Face

- The Left horizontal layer, Top vertical layer, and Front face will be denoted as LTF.
- The Left horizontal layer, Middle vertical layer, and Front face will be denoted as LMF.
- The Left horizontal layer, Bottom vertical layer, and Front face will be denoted as LBF.
- The Center horizontal layer, Top vertical layer, and Front face will be denoted as CTF.
- The Center horizontal layer, Middle vertical layer, and Front face will be denoted as CMF.
- The Center horizontal layer, Bottom vertical layer, and Front Face will be denoted as CBF.

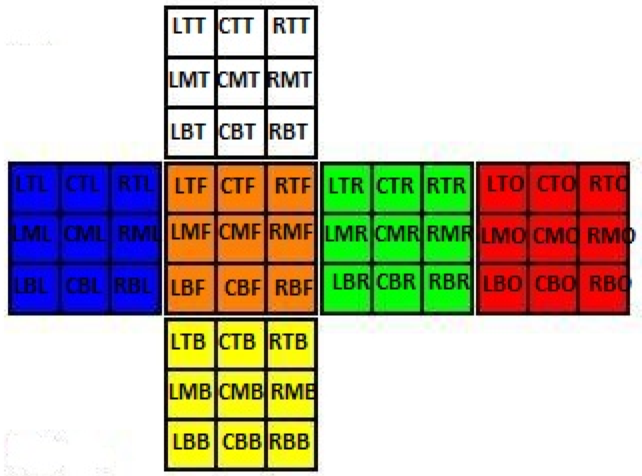


Figure 8: Mapped Cube

- The Right horizontal layer, the Top vertical layer, and Front face will be denoted as RTF.
- The Right horizontal layer, the Middle vertical layer, and Front face will be denoted as RMF.
- Thr Right horizontal layer, the Bottom vertical layer, and Front face will be denoted as RBF.

Left Face

- The Left horizontal layer, Top vertical layer, and Left face will be denoted as LTL.
- The Left horizontal layer, Middle vertical layer, and Left face will be denoted as LML.
- The Left horizontal layer, Bottom vertical layer, and Left face will be denoted as LBL.
- The Center horizontal layer, Top vertical layer, and Left face will be denoted as CTL.
- The Center horizontal layer, Middle vertical layer, and Left face will be denoted as CML.
- The Center horizontal layer, Bottom vertical layer, and Left Face will be denoted as CBL.
- The Right horizontal layer, the Top vertical layer, and Left face will be denoted as RTL.
- The Right horizontal layer, the Middle vertical layer, and Left face will be denoted as RML.
- The Right horizontal layer, the Bottom vertical layer, and Left face will be denoted as RBL.

Right Face

- The Left horizontal layer, Top vertical layer, and Right face will be denoted as LTR.
- The Left horizontal layer, Middle vertical layer, and Right face will be denoted as LMR.
- The Left horizontal layer, Bottom vertical layer, and Right face will be denoted as LBR.
- The Center horizontal layer, Top vertical layer, and Right face will be denoted as CTR.
- The Center horizontal layer, Middle vertical layer, and Right face will be denoted as CMR.
- The Center horizontal layer, Bottom vertical layer, and Right Face will be denoted as CBR.
- The Right horizontal layer, the Top vertical layer, and Right face will be denoted as RTR.
- The Right horizontal layer, the Middle vertical layer, and Right face will be denoted as RMR.
- The Right horizontal layer, the Bottom vertical layer, and Right face will be denoted as RBR.

Top Face

- The Left horizontal layer, Top vertical layer, and Top face will be denoted as LTT.
- The Left horizontal layer, Middle vertical layer, and Top face will be denoted as LMT.
- The Left horizontal layer, Bottom vertical layer, and Top face will be denoted as LBT.
- The Center horizontal layer, Top vertical layer, and Top face will be denoted as CTT.
- The Center horizontal layer, Middle vertical layer, and Top face will be denoted as CMT.
- The Center horizontal layer, Bottom vertical layer, and Top Face will be denoted as CBT.
- The Right horizontal layer, the Top vertical layer, and Top face will be denoted as RTT.
- The Right horizontal layer, the Middle vertical layer, and Top face will be denoted as RMT.
- The Right horizontal layer, the Bottom vertical layer, and Top face will be denoted as RBT.

Bottom Face

- The Left horizontal layer, Top vertical layer, and Bottom face will be denoted as LTB.
- The Left horizontal layer, Middle vertical layer, and Bottom face will be denoted as LMB.
- The Left horizontal layer, Bottom vertical layer, and Bottom face will be denoted as LBB.
- The Center horizontal layer, Top vertical layer, and Bottom face will be denoted as CTB.
- The Center horizontal layer, Middle vertical layer, and Bottom face will be denoted as CMB.
- The Center horizontal layer, Bottom vertical layer, and Bottom Face will be denoted as CBB.
- The Right horizontal layer, the Top vertical layer, and Bottom face will be denoted as RTB.
- The Right horizontal layer, the Middle vertical layer, and Bottom face will be denoted as RMB.
- The Right horizontal layer, the Bottom vertical layer, and Bottom face will be denoted as RBB.

Opposite Face

- The Left horizontal layer, Top vertical layer, and Opposite face will be denoted as LTO.
- The Left horizontal layer, Middle vertical layer, and Opposite face will be denoted as LMO.
- The Left horizontal layer, Bottom vertical layer, and Opposite face will be denoted as LBO.
- The Center horizontal layer, Top vertical layer, and Opposite face will be denoted as CTO.
- The Center horizontal layer, Middle vertical layer, and Opposite face will be denoted as CMO.
- The Center horizontal layer, Bottom vertical layer, and Opposite Face will be denoted as CBO.
- The Right horizontal layer, the Top vertical layer, and Opposite face will be denoted as RTO.

- The Right horizontal layer, the Middle vertical layer, and Opposite face will be denoted as RMO.
- The Right horizontal layer, the Bottom vertical layer, and Opposite face will be denoted as RBO.

3.2.2 Rotation Naming Convention

In the previous section, the naming convention of each face and cube was discussed (See Pg. 12). For reference on the naming convention for the faces of the Rubik's Cube and each cube of the Rubik's Cube please refer to the section regarding the Rubik's Cube. Now, each turn of the cube will indicate movement of a layer in a specified direction. Each explanation is under the assumption that the user is turning the Front face.

Right Horizontal Layer The Right horizontal layer will be regarded as R if it is turns towards you and R' if it is to be turned away from you.

Left Horizontal Layer The Left horizontal layer will be regarded as L if it is turn towards you L' if it is to be turned away from you.

Center Horizontal Layer The Center horizontal layer will be regarded C if it is turned towards you and C' if it is turned away from you.

Top Vertical Layer The Top vertical layer will be regarded as U if it is turned to the left and U' if it turned to the right.

Middle Vertical Layer The Middle vertical layer will be regarded as M if it is turned to the left and M' if it is turned to the right.

Bottom Vertical Layer The Bottom vertical layer will be regarded as D if it is turned to left and D' if it is turned to the right.

3.2.3 History

The Rubik's Cube was discovered in 1974 by Erno Rubik. The first working prototype of the cube was a wooden cube with white centers that twisted and turned. Later he added the colorful stickers and once it was scrambled the first solving of the Rubik's cube then named the Magic cube began. It went through many phases and was initially a work of art rather than a difficult puzzle. Upon making the cube he actually could not solve it initially and it took him nearly a month to figure out. However, the algorithmic methods to solve to cube have steadily progressed over time.

In 1981 Patrick Bossert wrote a famous book that gave instructions on solving cubes named "You Can Do The Cube". He utilized a layered method of solving the cube. The layered method has been further optimized since the writing of his book, but continues to meet the most widely used method. The most well known name for the layered method now is either CFOP method or the Friedrich method, named after Jessica Fridrich. Not only is this method widely used it is particularly used by speed cubers even though it is not the most efficient method when considering it takes many more turns.

A more efficient method was discovered in 2007 by Daniel Kunkle and Gene Cooperman by using computer searching methods. This allowed for the Rubik's Cube to be solved in under 27 moves! Then, just a year later Tomas Tokicki got the maximum number of moves down to 22. Eventually Herbert Kociemba determined that the Rubik's Cube can be solved in 20. His method is known as God's Algorithm [19].

3.2.4 Mathematics

The mathematics of the Rubik's Cube is important especially when analyzing the algorithms, permutations, symmetries. From the research of Kociemba it has been determined that not only the standard 3 x 3 x 3 Rubik's Cube, but any Rubik's Cube of n x n x n can be solved in $O(\frac{n^2}{log(n)})$ moves.

Permutations When considering the permutations of the standard Rubik's Cube it has eight corners and twelve edges. Since there are eight corners that means there are 8! = 40,320 ways to arrange the corners of the cubes. Of those eight corners seven of them can be oriented independently, while the eight cannot which means that there are $3^7 = 2,187$possibilities.

Similarly because there are twelve edges that means there are $\frac{12!}{2} = 239,500,800$ ways to arrange the edges. The edges are divided by 2 because an even permutation of the corners implies that the edges must also be of an even permutation. Now of the twelve edges only eleven can be moved independently while the twelfth cannot which gives a $2^{11} = 2,048$ possibilities.

The corners and edges of the cubes of each possibility must then be multiplied by each other to figure out the possible combinations.

$$8! \ * 3^7 * (\frac{12!}{2}) * 2^{11} = 43,252,003,274,489,856,000$$

Even though that is 43 quintillion possibilities there are actually more when you consider that a Rubik's Cube can only turn by it's sides which means the math would be more along the lines of this equation:

$$8! \ * 3^8 * 12! * 2^{12} = 519,024,039,293,878,272,000$$

That is 519 quintillion different possibilities! The reason it goes up so much is that the cube cannot be moved in any sequence that can rotate a single corner, swap a single pair of pieces, or an edge of the cube. Also, a standard Rubik's Cube uses solid colors on each face, but if the center cubies were to be marked that would increase the possibilities even greater! However, for the scope of this project the increase by the center cubes will not be considered.

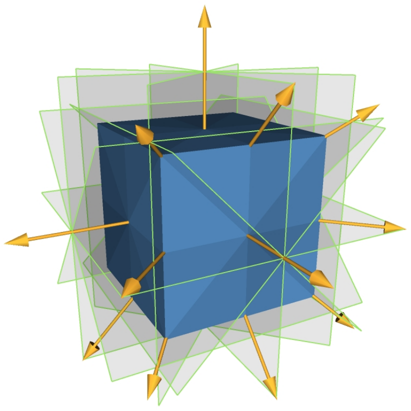


Figure 9: Cube Symmetries

Symmetries The symmetric patterns are an essential part to actually understanding the solving algorithms. There are 164,604,041,664 symmetric cubes that exist! That is a lot of possibilities, but the number greatly reduces for the most efficient algorithm being that of Kociemba. That number reduces to 32,625 different symmetric cubes that can be solved in 20 moves. Also, there is 48 possible symmetries of the cube. From geometry we know that this can be determined when one counts all of the combinations of when vertices's are chosen: 8*3*2= 48. In the figure below you will find an image of the rotational symmetrical axes of a cube.

- 1/2 rotation around an edge 6 elements
- Reflection through a place 6 elements
- 1/2 rotation around a face 3 elements
- Reflection through a plane 3 elements
- 1/4 rotation around a face 2x3 elements
- 1/4 rotation + reflection through the center 2x3 elements
- 1/3 rotation around an edge 2x4 elements
- Reflection through the center 1 element
- 1/3 rotation + reflection through the center 2x4 elements
- Identity 1 element

3.3 Algorithms

3.3.1 Kociemba's Algorithm

Kociemba's algorithm is a two-phase algorithm that is also known as God's algorithm. God's algorithm is a fancy term of most efficient algorithm in the sense that the Rubik's Cube

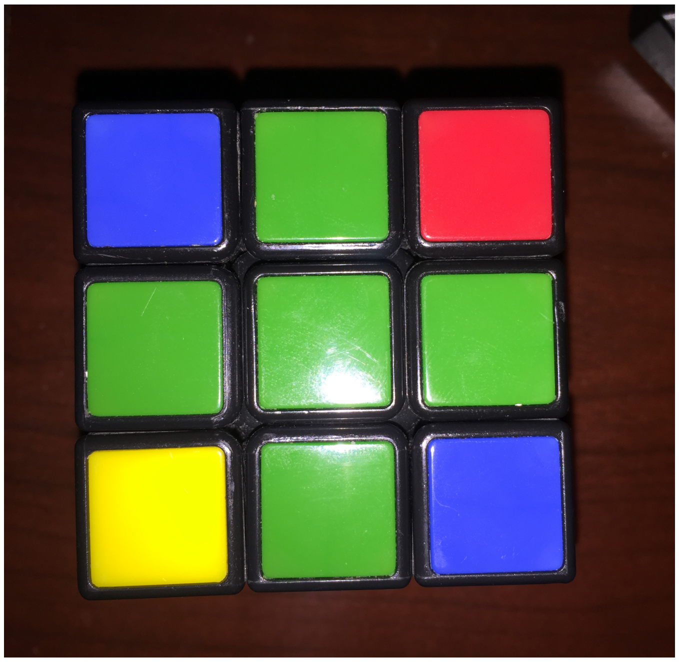


Figure 10: Base State of Layer Method

can be turned in the fewest number of times to be solved in any given state. Kociemba's algorithm allows for the popular Rubik's Cube to be solved in only 20 turns.

Kociemba's Algorithms is considered a two-phase algorithm because after an initial solution is reached additional optimal solutions are searched for. It was developed from the former most efficient algorithm being Thistlethwaite's algorithm. It solves the cube by looking at it in groups and phases. The first group is the G0 group which is the randomly mixed cube group. At this stage there are no restrictions on what moves can be made when utilizing this algorithm. To complete this state the cube needs at most 12 moves. The next group is the G1 group which is when the cube gets in a specific state of U, D, L2, R2, F2, B2. When the cube is in this state there are specific moves that cannot be made. To complete this state the cube needs at most 18 moves.

3.3.2 CFOP Method Algorithm

The most popular of the Rubik's Cube solving algorithms is the CFOP Algorithm Method. Which stands for Cross - F2S - OLL - PLL. It is also known as the Fridrich method. Jessica Fridrich is an Electrical Engineering professor at Binghamtom University and is credited for documenting and popularizing this method. She is considered a pioneer of speedcubing. This method involves solving each layer of a Rubik's Cube until it is completely solved. The layers of a Rubik's Cube can be noted as Top, Middle, and Bottom as shown below. To actually solve the Rubik's Cube using the Layer Method the user must get the Rubik's Cube into a base state. This method does not have an algorithm to get the Rubik's Cube into the base state therefore the user must be able to this on their own. The base state has been reached once there is a cross shape on one face of the Rubik's Cube with each of the cube in the

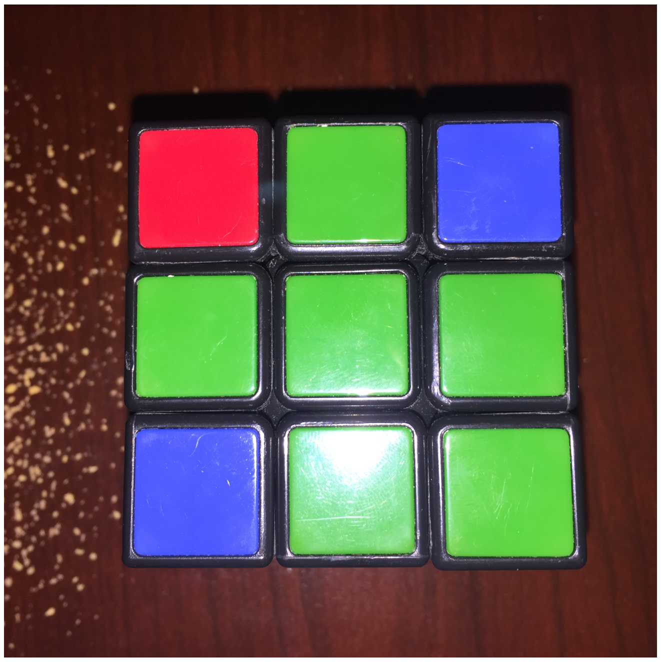


Figure 11: Corner Algorithm Initial Implementation

correct place. Once the user has obtained the base state a series of rotational algorithms must be implemented. The order of these algorithms are extremely important and each can only be implemented properly when the cube gets into a specific state. However, implemented the algorithms can only be done once the rotation naming is clear.

Corner Rotational Algorithm The Corner Rotational algorithm is a key algorithm used to solve the Rubik's Cube. It is the only one of the algorithms that is used throughout. It is the first algorithm used once the Rubik's Cube is in the Base State on the Top face. The purpose of this method is to place a cube in the correct orientation of a corner when it is in the specified column. It is used to completely solve the first face(Top) of the cube. An example of this algorithm being implemented is shown in figures 10 through 12. Above the Corner Algorithm is implemented on the Base State image and places one cube in the correct corner. This algorithm can be done six times in a row to get back to the original state of the cube before the algorithm was implemented. The actual algorithm is as follows: R D R' D' The Corner Rotational algorithm must be implemented on each corner cube of the Base State that is in an unsolved state. Once it is completed a face, Top in this case, should be solved.

Middle Layer Rotational Algorithms Middle Vertical Layer Rotational Algorithms The Middle Vertical Layer Rotational algorithms are utilized to solve the Middle vertical layer. There are only two algorithms to solve this layer which are practically inversions of each other. One is to place the FTC cube in the FMR location and the other is to place it in the FML location. Middle Layer Rotational Right Algorithm

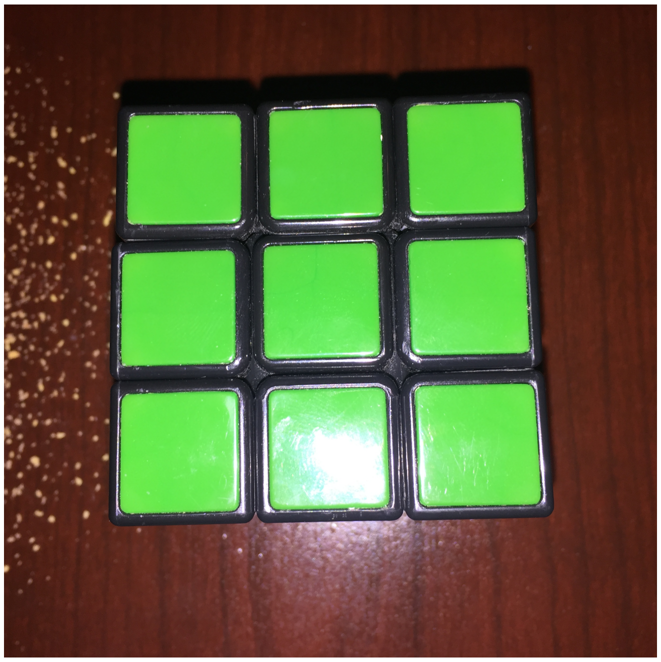


Figure 12: Face Solved

Right Rotational Algorithm The Middle Layer Rotational Right algorithm is used to solve the second layer using the CFOP Method algorithm. To implement this state the user must have correctly solved a face of the cube and have the solved face put as the Bottom face. There are only a maximum of four cubes to solve using this algorithm. In this current example we will flip the Green color solved face of the cube from the Top to now the Bottom. The Orange color FMC cube will be considered the Front for this example as well. Once that is done the user must place the cube that correctly fits the FMR positon in the FTC (Front-Top-Center) position by simply turning the Top vertical layer of the cube. If the cube is in the correct position, but the wrong orientation the user can utilize this algorithm to put any arbitrary wrong cube in the FMR position and the correctly place the cube in question in the correct position as normal. Then the algorithm is implemented as follows: U R' U' R U' F U F'

Left Rotational Algorithm The Middle Layer Rotational Left algorithm is used to solve the second layer using the CFOP Method algorithm. To implement this state the user must have correctly solved a face of the cube and have the solved face put as the Bottom face. There are only a maximum of four cubes to solve using this algorithm. In this current example we will flip the Green color solved face of the cube from the Top to now the Bottom. The Orange color FMC cube will be considered the Front for this example as well. Once that is done the user must place the cube that correctly fits the FML position in the FTC (Front-Top-Center) position by simply turning the Top vertical layer of the cube. If the cube is in the correct position, but the wrong orientation the user can utilize this algorithm to put any arbitrary wrong cube in the FML position and the correctly place the cube in question in the correct position as normal. Then the algorithm is implemented as follows: U' L U L'

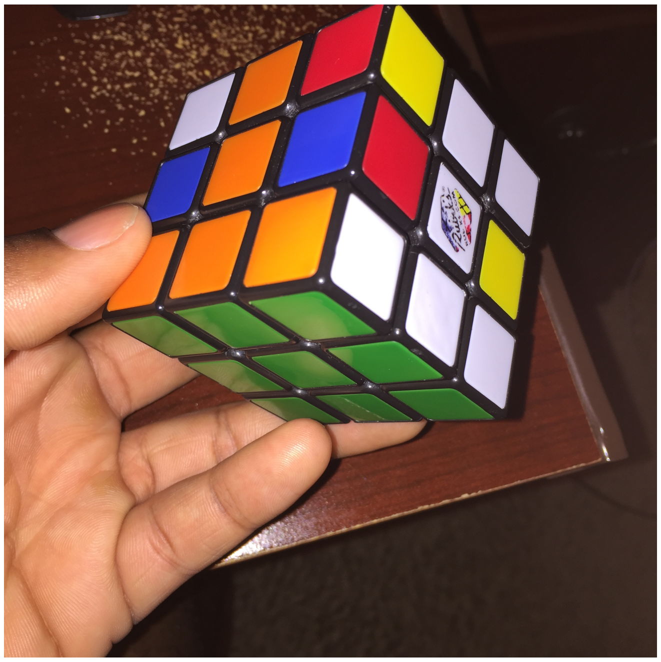


Figure 13: FMR Unsolved

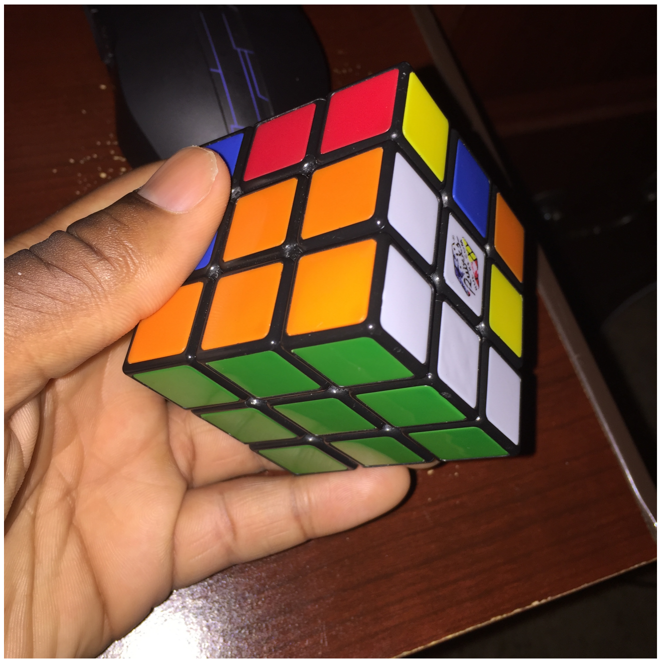


Figure 14: FMR Solved

U F' U' F Once both of these Middle Layer Rotational Algorithms have been implemented to solve the four cubes on the Middle vertical layer this state has been completed and should resemble Figure 17 with the Bottom vertical layer and Middle vertical layer solved.

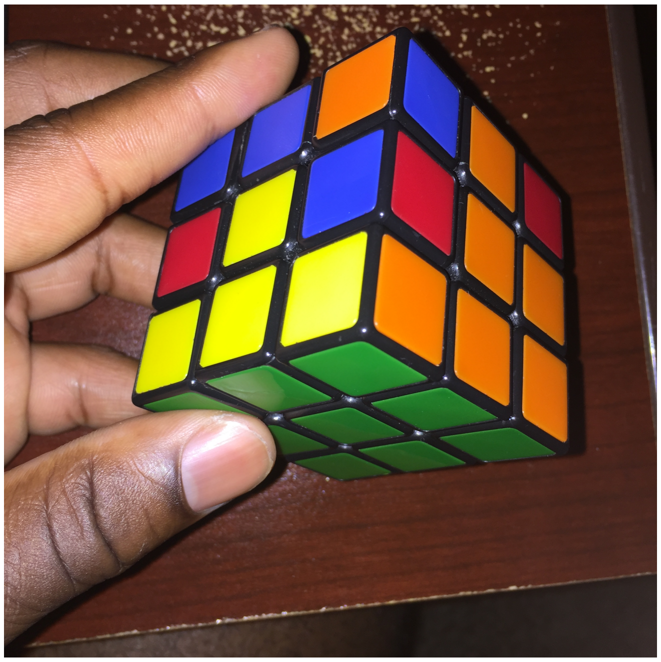


Figure 15: FML Unsolved

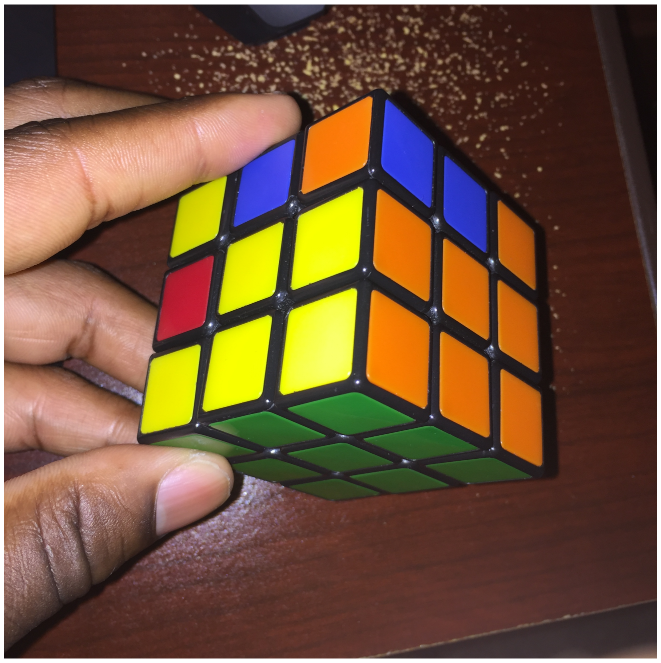


Figure 16: FML Solved

Top Vertical Layer Rotational Algorithms The Top Vertical Layer is the most involved layer of the three vertical layers Top, Middle, and Bottom. There are a number of Top vertical layer rotational algorithms. But, initially it has the same goal as the Bottom layer to get to the Base State, which is the cross. Assuming the face in question is the Front face the Base State can be further defined as the state in which the Rubik's Cube has the FTC, FMR,

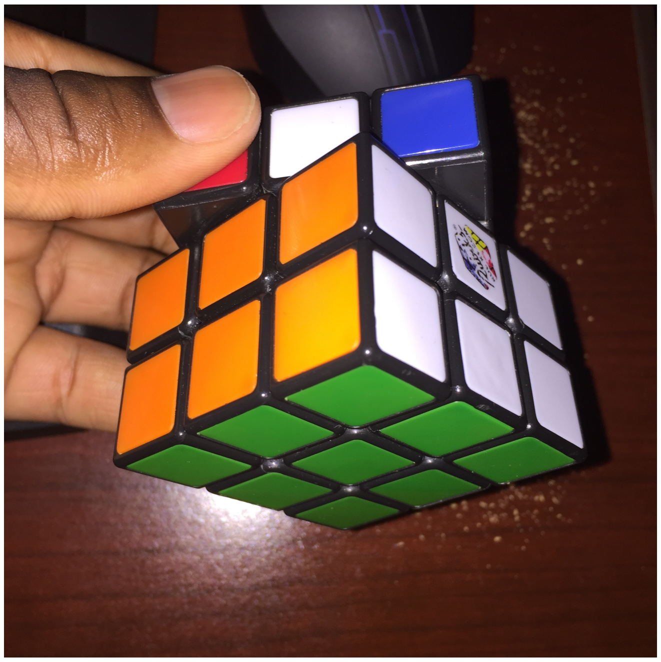


Figure 17: Middle Vertical Layer Complete

FML, FBC, and FC (Front Center) cube. The Center cubes denote which color each face should be associated with. Eventually, after all the Top vertical layer rotational algorithms are implemented the Rubik's Cube will be completely solved.

Top Cross Algorithm The Top Cross algorithm achieves the goal of the Base State. It must be the first to be implemented of all of the Top vertical layer rotational algorithms. There are only four forms the Top face can be in assuming the Green color solved face is the Bottom face. One form it can possibly be in is the TC (Top Center) cube being the only one in place. If the user is in this form the Top Cross algorithm should be implemented multiple times until it gets into the cross state. Another form it can be in is an 'L' shape in the corner of the face. If the Top vertical layer is flipped so that it is now the Front face the 'L' shape can be specifically referenced as the FTC, FML, and FC cubes being in the correct place.

However, the Top vertical layer should not actually be flipped when doing the algorithm only to understand the mapping. If the user gets into this form then the algorithm should be done two more times until it is in the cross state. Now, the other state that the cube can be in is a horizontal line across. This can be specifically mapped as the FML, FMR, and FC cube being in the correct positions if the cube were flipped so that the Top vertical layer became the Front face. Again, the cube is not actually flipped only for the understanding of the mapping. If the user is in this form the algorithm only needs to be implemented once. The actual algorithm is: F R U R' U' F .

Top Corners Permutation Algorithm The Top corners have to be should be the only ones unsolved. This algorithm is used to place the four corners into the correct place. The

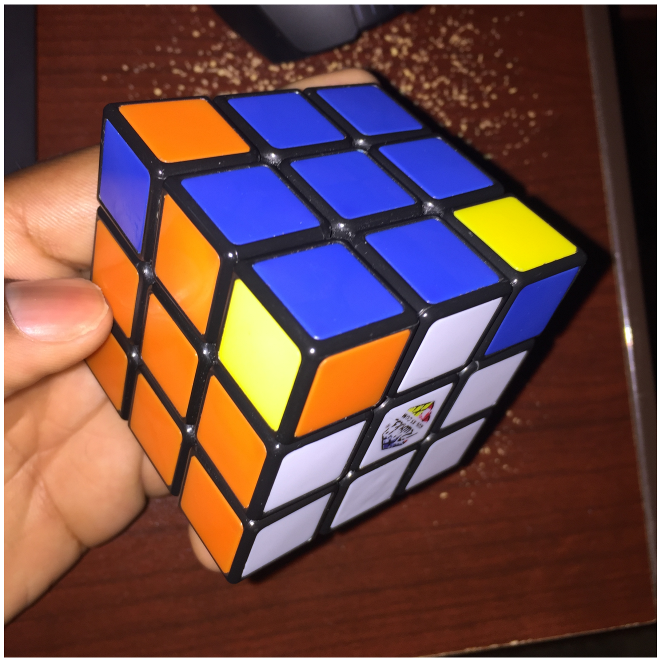


Figure 18: top vertical solved

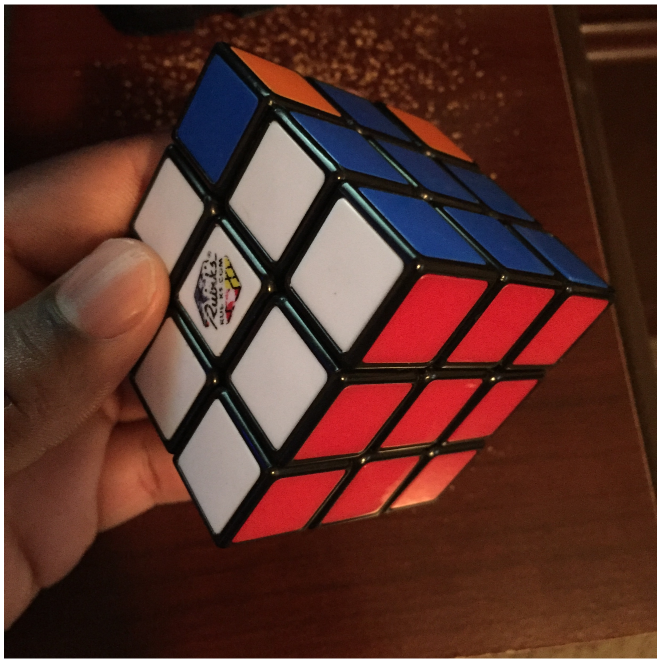


Figure 19: top corners solved

algorithm should not have to be done more than three times. The user must find a corner that is in the correct position even if it is in the incorrect orientation then have that in the bottom right corner of the top layer. If there is no cube in the correct position the user can simply choose a random corner and implement the algorithm which will result in at minimum one corner being in the correct position. The actual algorithm is: U R U' L' U R U L'

Figure 20: solved

The Final Algorithm The Final algorithm to be applied to solve the Top vertical layer and thus the entire Rubik's Cube will be solved. It is very familiar to the first algorithm used once in the Base State to solve the Bottom vertical layer in the sense that it is exactly the same. That is why it is a key algorithm as mentioned previously. The only difference is that this algorithm must be applied to each corner until the particular corner is in the correct state and then the Top vertical layer must be rotated R'. Again the algorithm is: R D R' D'

3.4 Hardware

3.4.1 Structural Platform

The structural platform of the robot refers to its building materials, method of construction, and physical layout. The following sections investigate the resources and options available to us for the structural platform of the robot.

Materials The building materials for the robot will dictate its structural integrity as well as method of its construction. Considerations for building materials include but are not limited to: cost, accessibility, ease of assembly, and structural strength.

3D Printing 3D printing for consumer use has gained popularity in recent years as the costs of buying and maintaining these machines have rapidly declined. Lower resolution printers can be found in private homes and hobby workshops, while most universities offer more advanced 3D printing equipment for student use, as is the case here at UCF.

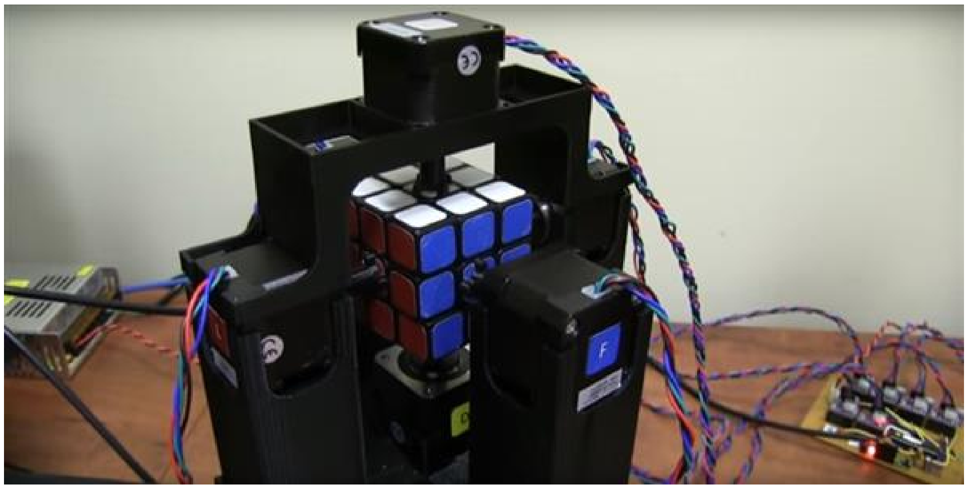


Figure 21: 3D-printed robot structure

Discussions with previous UCF design groups indicate that the total fabrication time for student print jobs averages 3-6 days which is adequate for our needs. Student design groups are allowed a certain amount of free printing every semester, and the printer is reportedly very reliable. Inspection of various completed print jobs reveals that the print resolution is high enough for most project applications.

3D printing materials are plastic-based and come in various forms which are listed and shortly described below [40]:

- **ABS (Acrylonitrile Butadiene Styrene):** Generally very strong and durable. Easy to sand down and broken pieces can be glued with ABS glue. Generates mildly irritating fumes during printing which may require ventilation or fume hood.
- **PLA (Polylactic Acid):** Biodegradable and environmentally friendly. Not as durable as ABS and becomes brittle once cooled. Does not require heated printbed or printing ventilation.
- **PVA (Polyvinyl Alcohol):** Water-soluble and high-priced. Not as frequently used as ABS and PLA.

The short fabrication time and relatively cheap cost of 3D printing make it a great option for rapid prototyping of custom physical designs, but it can also suffice as a final construction option for designs that won't be under high physical stress or temperature. Figure 21 shows a cube-solving robot that was 3D-printed.

Robotic Kits Robotics kits are a more durable alternative to 3D printing, however they offer less customizability and cannot be prototyped as quickly. Structures can be created from machined aluminum parts similar to the ones shown in Figure 22. These structures can presumably be designed and constructed in less time than it would take to design and print a structure, but the parts can cost upwards of $100 and usually need to be shipped from a warehouse.

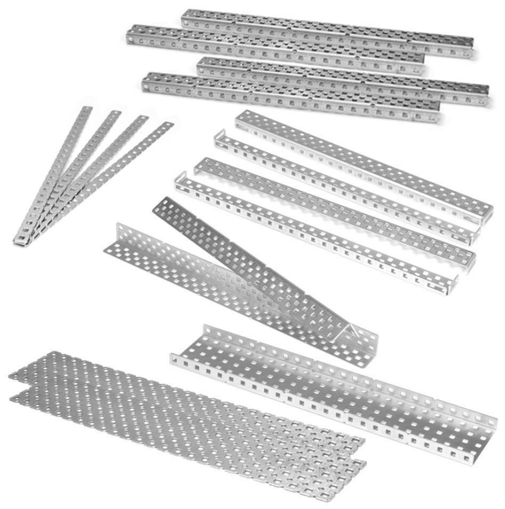


Figure 22: Aluminum robotics kit parts

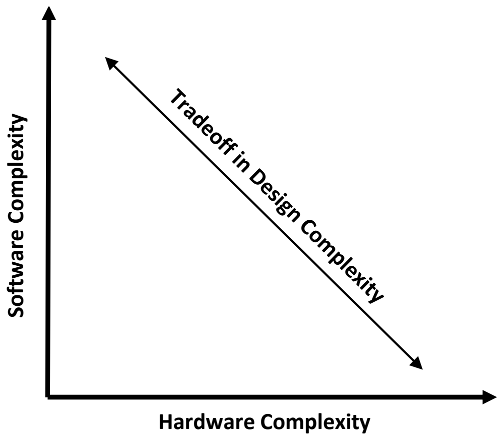


Figure 23: Tradeoff in System Complexity

Layout and Orientation As discussed in the "Similar Projects" section (see 3.1.1), the simplest robots feature a single motor and claw combination to achieve one controllable rotational degree of freedom. In order for the claw to rotate a different side of the cube, these robots use a linear actuator to push an edge of the cube until it flips over, a process that dramatically increases solving time. Moreover, structures that reorient the cube also require additional programming to accomodate this functionality and keep track of the cube's orientation

Adding additional motors or claws allows the robot to rotate a given side of the cube with less reorientation, leading to more simplistic programming. This relationship, demonstrated by Figure 23, continues to the point of having a dedicated motor for each side of the cube which allows the robot to rotate any side of the cube without any preparation or reorientation.

Another design aspect that is different in many existing robots is how the cube is oriented within the hardware. Some robots hold the cube with one face parallel to the floor, while others hold it with a corner facing the floor. Other robots such as the popsicle stick robot in Figure 4 rely on a tilted positioning that allows the cube to slide back into position after it is reoriented.

3.4.2 Processing Platforms

One feature common to all cube-solving robots is a processing platform. Processing hardware is necessary for several steps of the autonomous cube-solving process:

- Image processing
- Application of solving algorithms
- Manipulation of hardware

Many factors such as cost, complexity, and performance will influence what kind of processing platform is necessary for a robot. Some designs rely on pre-written computer software to perform image processing and thus recruit entire laptop or desktop computers as their processing platforms. The opposite end of the complexity spectrum, occupied by lego-based robots, implements image processing and solving algorithms on 8-bit processors within handheld, black-box embedded systems that shield the engineer from most technical design considerations. Our project lies somewhere between these two hardware complexities with an embedded processor on a custom PCB.

This section is dedicated to identifying and evaluating several embedded processors that could potentially be used in our robotic system. Candidate processing platforms should satisfy the project requirement for low cost and provide adequate performance for our needs. Because of our familiarity with Texas Instruments devices and because TI offers such a wide variety of processor families, we are choosing to focus our review on their device families.

TI MSP430 The MSP430 family is an ultra-low power processor family with a large variety of memory sizes and I/O capabilities. Processor speeds can range up to 48MHz and the largest devices contain up to 265KB of RAM and feature 90 GPIO pins [15]. While their specialty is in low-power operarion, the MSP430 is also a good candidate for general purpose applications because of it's straightforward implementation and large base of documentation and support.

TI MSP432 The MSP432 is the higher-performance cousin of the MSP430 that swaps out the MSP430 core for a 32-bit ARM Cortex core. Most MSP432 chip features and peripherals match those of the MSP430, including GPIO, RAM, and chip speed. Unfortunately, this chip is not available on TI's website individually, and can only be purchased as part of a development board.

Family	Device	Core	RAM	GPIO	Features	Cost
MSP430	MSP430F2131	MSP430	0.25KB	16	low-power, small footprint	$1.20
	MSP430F5528	MSP430	8KB	47	low-power, USB support	$3.55
	MSP430F6659	MSP430	66KB	74	low-power, USB support, LCD support	$6.48
MSP432	MSP432P401M	ARM Cortex-M4F	32K	84	low-power	N/A
C2000	TMS320F2837xD	Dual TMS320C28x	512KB	169	USB, enhanced PWM output	$17.03
	TMS320F28069M	TMS320C28x	100kB	54	High-resolution PWM, ROM motor control library	$12.96
Sitara	AM4376	ARM Cortex-A9	DDR2/3	192	3D graphics, SIMD coprocessor, 2x USB support	$9.00

Table 2: Processor Comparison

TI C2000 The C2000 family is a 32-bit group of processors geared toward sensing and actuation for applications in closed loop systems. Motor control is one specialty of the C2000 family, which offers devices that feature dedicated hardware for PWM generation. The C2000 hardware and software development tools make it possible to develop and fine-tune the motion, power, and feedback processing of motor setups.

TI Sitara The Sitara is an ARM-based high performance family better suited for complicated applications than the previously mentioned devices. Sitara processors, outfitted with DDR3 RAM interfaces and 1GHz clock speeds, are capable of running Linux systems. The well-known BeagleBone development board uses a Sitara processor [28].

Summary of Processing Devices Table 2 details the features of several processors from the families that are described above.

3.4.3 Camera

Image sensing is a big part of our project. For this task we had to visualize all of our options and decided which camera would fit our project the best. While doing research for our project we discovered a camera referred to as the Pixy Cam. This camera has been used in recent robotics design because of its ability to visualize objects easily while using minimum processing power on the CPU bored.

Vision sensors have been extremely useful for projects they have become a staple in autonomous robotics. The issues with most cameras though when used for Vision sensing come down to 2 main points. The first point is the whenever a robot is using vision sensing it outputs lost of data, even up to dozens of megabytes per second. Secondly processing the amount of data taken in by the camera using Vision sensing overloads many processors. If the processor can keep up with the data though it won't have enough power for other task it needs to handle.

Pixy Cam has discovered a way to solve both of these issues with its design, making it the leader in a class that has few competitors. How the Pixy Cam does this is by pairing the camera with a powerful dedicated processor. This processors takes in the images and only sends useful information to the micro-controller. This information is taken in at a fast pace and with 50 frames per second to ensure the best quality imaging for the robot. The Pixy cam also has several port interfaces that make it very easy and accessible to get information to our micro-controller without limiting the controller at all.

The main algorithm used by for vision sensing the Pixy Cam works on is called purple dinosaurs. Purple dinosaur is a color based filtering algorithm that is used to detect different objects. This method is popular because its fast, efficient, and relatively robust. This algorithm is used by RGB to calculate color hue and saturation. The only issue with this is when lighting and exposure comes into play, the purple dinosaur method sometimes fails and view the same object as two different ones. The Pixy cam has a filtering algorithm attached to its purple dinosaur method that allows the camera to detect lighting and exposure changes and neglect it.

Lastly a huge feature included in the Pixy cam is that it remembers 7 different significant colors. This is perfect for our design because the Rubik's cube only involves 6 different colors making it perfect for us to use. With this feature the Pixy cam looks for and detects these colors making our camera process less information because it will not use its power on images other than the colors we are telling it to.

Figure 24 shows the camera we decided would fit best for our project. It is referred to as the Pixy CMUcam5 designed by Carnegie Mellon Robotics Institute with partnerships with Charmed Labs. The Pixy Cam was initially started in 2014 to solve the issues of image processing that many vision sensors have had issues with in the past. Pixy cams have been used in many systems including picking up objects, chasing a ball, or locating a station designated by the operator. We feels if the Pixy Cam can be used for these task it will be more than enough to help us solve our problem of imaging.

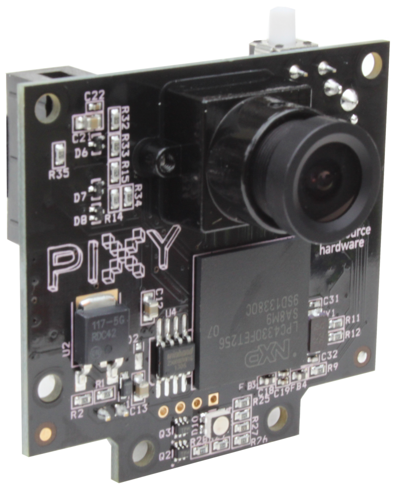


Figure 24: Pixy Cam

Motor	Wires	Max Voltage	Max Current	Price
Diligent 290-006	2	12V	200mA	\$19.99
Adafruit Metal Gear Servo	3	6V	N/A	\$19.95
Parallax Continuous Rotation Servo	3	6V	190mA	\$12.99

Table 3: Servo Motors

3.4.4 Motors

The motors are a key component of our robot and will be responsible for physically manipulating the cube in order to solve it. There are several types of motors that can be used in a robot system, each with their advantages and disadvantages. This section identifies three types of motors and discusses their features in an effort to identify a suitable motor type for our project. Here we also explore various aspects of controlling these motors and identify potential motor control methods that suit our application.

Motor Types Here we explore the functionality of DC motors, servo motors, and stepper motors. Table 5 summarizes the functional advantages and disadvantages that these motor types offer for our application

Servo Motors Servo motors generally contain three wires for power, ground, and control. Servo motors require constant power and use a constant PWM signal to move to positions within $\pm 180°$ of the home position. A neutral PWM pulse whose width is intrinsic to the motor allows it to remain in the same position even with external force applied. Optional feedback allows the shaft position to be finely tuned. Table 3 lists the specifications of a few servo motors on sale that could fit our application [10] [31] [21].

Motor	Wires	Max Voltage	Max Current	Price
Adafruit Stepper Motor	4	12V	350mA	$14.00
SureStep STP-MTR-17040	4	N/A	1.7A/Phase	$18.00
Sparkfun Stepper Motor	4	12V	330mA	$14.95

Table 4: Stepper Motors

Type	Advantages	Disadvantages
DC	High RPM Easy to operate	Lack of positioning control
Servo	Can be driven from single PWM input	Usually require feedback processing or PWM-tuning for accurate positioning
Stepper	Predetermined, reliable positioning	Usually require driver IC

Table 5: Comparison of Motor Types

Stepper Motors Stepper motors are similar to servo motors with the exception of their internal gearing. Usually equipped with four wires, stepper motors have multiple sets of internal coils that allow the shaft to be rotated bidirectionally within a central toothed gear. Actuating one set of internal coils will cause the shaft to rotate, or "step" slightly by a set amount. By engaging all the coils in different speeds or patterns, the shaft can be precisely rotated by a predetermined amount. Unlike servo motors, stepper motors will hold their current position without and external power or signals. Table 4 lists the specifications and prices of a few reasonably-sized stepper motors available online [34] [32].

DC Motors DC motors are simple two-wire motors that operate continuously in a direction dependent on the polarity of the applied power. Most DC motors run at very high speeds that can be controlled by applying a PWM signal in place of DC power.

Motor Control In considering devices such as stepper and servo motors for our project, we must also evaluate methods for interfacing and controlling them. The following paragraphs discuss the driving requirements of these motors and present several solutions for controlling them.

Stepper Motor Actuation Stepper motors require bidirectional current sourcing on four separate wires to achieve bidirectional motion as is necessary in our project. The simplest method of stepper motor actuation is to engage one coil after the other, incrementally turning the rotor. This driving method is called "wave drive" and is illustrated by Figure 25.

A problem with wave drive is that it does not recruit the maximum torque output from the motor because only one phase is active at any given time. A similar driving technique that enables maximum torque is called "full step drive", which overlaps coil currents as shown in Figure 26.

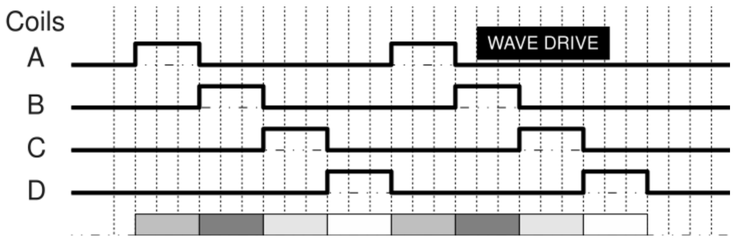


Figure 25: Stepper Motor Wave Drive

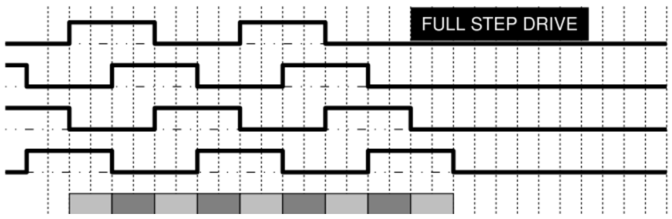


Figure 26: Stepper Motor Full Step Drive

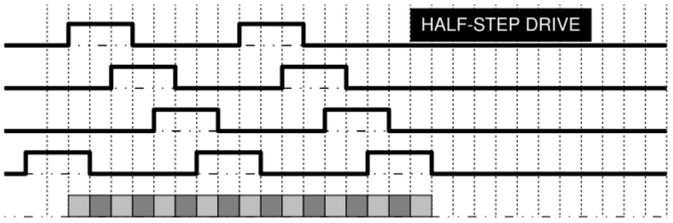


Figure 27: Stepper Motor Half Step Drive

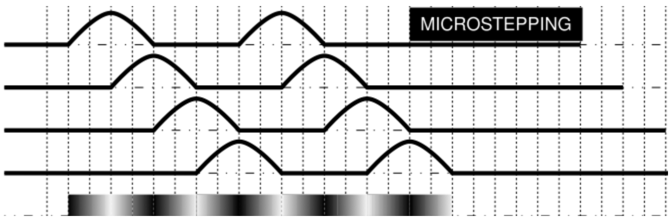


Figure 28: Stepper Motor Mircostep Drive

A more precise alternative to full step driving is “half step” driving which is depicted by Figure 27. With half step driving, the motor coils are actuated in a way such that the rotor steps in fractional (one half) increments which allows for smoother motion.

Dedicated motor-driving hardware allows these fractional-step driving techniques to be extended to a high level of precision called “mircostepping” that allows the rotor to be stepped in increments as small as $\frac{1}{32}$ of a single step. The resulting control waveforms are shown in Figure 28.

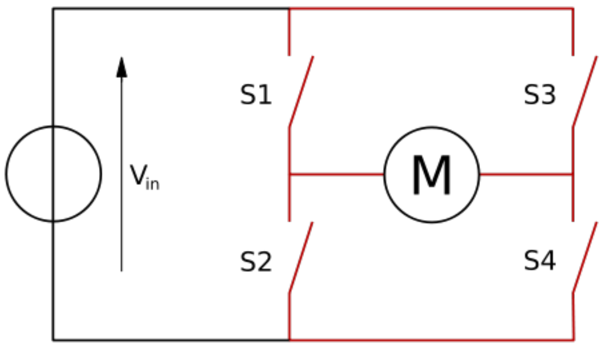


Figure 29: H-Bridge Circuit

Stepper Motor Drivers As stated in the previous section, stepper motors achieve bidirectional motion by swapping voltage polarity across the motor coils. While embedded processors can generate the PWM signals that motors require, they usually cannot source/sink an adequate amount of current to do so directly. Additionally, the inductive properties of a motor can potentially damage a fragile embedded processor without any buffer circuitry. These issues generally necessitate the inclusion of dedicated hardware for motor control. One popular method of controlling signal polarity in motor applications is with an H-bridge circuit, which is depicted in Figure 29. Controlling the internal switches allows current to be directed differently through the motor coils.

Simple motor-driving applications use single, double, or quad H-bridge integrated circuits such as the TI L293 whose functional block digram is shown in Figure 30. These devices accept PWM inputs along with H-bridge enable signals and a power supply input. This hardware works well as a buffer between a motor and a microprocessor but it does not include built-in functionality for special motor driving techniques such as half wave driving or microstepping. To help simplify system designs, some ICs provide additional hardware that can decode input signals and automatically drive an H-bridge setup to achieve microstepping functionality.

Texas Instruments' DRV family is an example of integrated motor driving and control hardware. Figure 31 shows a block diagram of a DRV implementation. Control inputs determine the step size of the rotor, direction of rotation, and decay strength of the coil current. The simplified control scheme offered by these devices make them good candidates for our project. Several components in the DRV family are listed in table 6, which details their pricing and technical specifications.

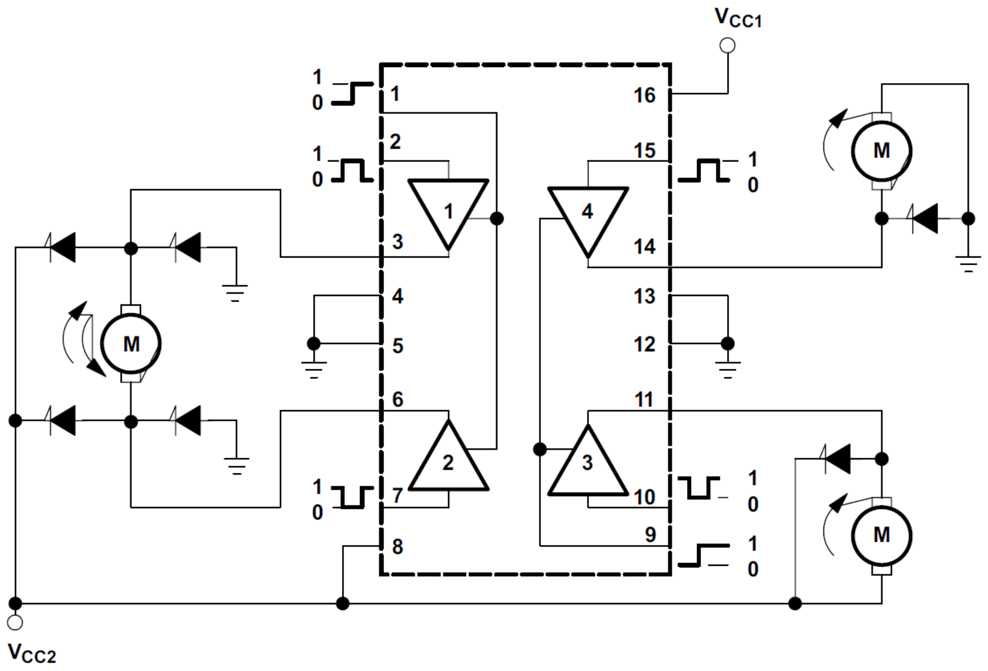


Figure 30: TI L293 Functional Block Diagram

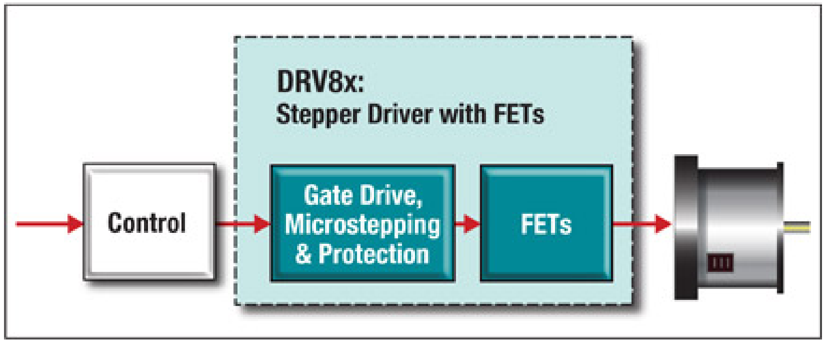


Figure 31: DRV Application Block Diagram

Device	H-Bridges	RMS Output Current	Features	Price
DRC8884	4	700mA	$\frac{1}{16}$ Microstepping, current sensing	\$1.30
DRV8818	4	1.75A	$\frac{1}{8}$ Microstepping, current sensing	\$1.85
DRV8825	4	1.8A	$\frac{1}{32}$ Microstepping, current sensing	\$1.90

Table 6: TI DRV Stepper Driver Devices

Name	Type	Dimensions	Connection	Cost
Angelelec Open Source Display	LED	8-bit x 1-bit	SPI Interface	$15.93
SparkFun Graphic Display	LCD	128-bit x 64-bit	Serial Interface	$36.95
SainSmart IIC/I2C/TWI	LCD	20-bit x 4-bit	Serial Interface	$14.99
LCD Touch Panel ILI9341	LCD	240-bit x 320-bit	SPI Interface	$5.73
LCD Display module 51/AVR/STM32/ARM/PIC	LCD	240-bit x 320-bit	SPI Interface	$4.75
Graphical LCD DIsplay GLCD	LCD	128-bit x 64-bit	SPI Interface	$10.58
909-Shield	LCD	16-bit x 2-bit	Serial Interface	$16.90

Table 7: Selection of Displays

3.4.5 Display

One of our newest features we have added to the project is a Display that can promote our robots functions in a GUI format. For this we had many questions on what we wanted to prioritize on the display and how to run it with our system without slowing down the other functions of our system. We found these displays and compared them to our system to help us decide which one may be the best fit for us.

3.4.6 Power

System power is an important design aspect that has the potential to constrain or impact many other design considerations. To begin the development of a power system for our robot, we first need to reflect on professional standards regarding electrical system safety as well as any project objectives related to power consumption and sustainability. We can dramatically narrow down our design choices because our system is expected to remain connected to a power outlet during its operation which eliminates the need for batteries. Moreover, our system's access to a power outlet allows it to consume energy more liberally than would a battery-operated system with limited charge. It follows that our design process will be limited to focus on AC-DC conversion and DC power regulation.

The next step in designing our power system is to identify the power requirements of the devices we plan on using in our robot. As shown by Tables 3 and 4, most of our potential motors would recommend a 12V input with at least 300mA peak current. Aside from the motors and their associated driver hardware, the remaining parts of our electrical system all operate in a sub-5V domain. Table 8 lists the general power options and requirements for our system components.

AC-DC Conversion AC-DC conversion is necessary for our system to be able to source power from a wall outlet. It is a multi-step process that begins with the rectification of an alternating-current (bipolar) waveform to one that is direct-current, or unipolar [43]. After the AC signal is rectified, it needs to be smoothed to maintain a constant voltage level. This can be done with a simple RC filter which will output a near constant DC voltage. [42]. While this AC-DC conversion hardware can be designed and assembled with discrete

Component	Description	V_{min}	V_{max}
Motors	Recommended voltage for most potential motors	3V	12V
Motor Drivers	DRV Family V_s supply	8.2V	45V
CMUcam5 Pixy	USB Connector (regulated)	5V	5V
	I/O Connector (regulated)	5V	5V
	Power Connector (unregulated)	6V	10V
Processor	Most TI device families (Table 2)	1.8V	3.6V

Table 8: Robotic System Power Specifications

	Linear	Switching
Function	Step down only	Step down, step up, inverting
Efficiency	Low-Medium, dependent on difference between input and output voltages	High except at low load currents
Heat Dissipation	High	Low
Complexity	Low, few external components needed	High, usually reuires several external components
Cost	Low	Medium-High
Ripple/Noise	Low	Medium-High

Table 9: Voltage Regulator Comparison

components, it can also be easily purchased as a pre-assembled device that outputs a finely tuned DC voltage. Based on the information in Table 8, we can choose a AC-DC converter device that will supply the motors with a rectified and smoothed $12V_{DC}$. This will also allow the motors to draw their required amounts of current.

DC-DC Conversion DC-DC conversion is the process of converting a DC voltage of one magnitude to a DC voltage of a different magnitude. Hardware devices that perform this task are called "voltage regulators" and can either increase (boost) or decrease (buck) the voltage level between the input and output. The following sections describe two popular types of voltage regulators, which are compared in Table 9.

Linear Voltage Regulators Linear voltage regulators operate continuously in the linear region and are known for having a clean and stable output voltages. They are only capable of stepping down voltage levels, which they achieve by acting as variable resistors within a voltage divider setup. Linear voltage regulators are limited in efficiency by the fact that extra power is dissipated as heat.

Switching Voltage Regulators As their name suggests, switching voltage regulators rapidly switches the output on and off to maintain a set voltage level. Because switching regulators can intelligently switch between and on and off state, they achieve high efficiency and are also capable of boosting the output voltage to a level higher than the input.

Device	Type	Output Voltage	Output Current	Cost
TI TPS715A	Linear	3.3V	80mA	$0.40
TI TLV701	Linear	3.3V	150mA	$0.29
TI TLV1117-50	Linear	5V	800mA	$0.19
TI TLV1117-18	Linear	5V	1A	$0.19
TI LM1086	Linear	5V	1.5A	$0.95
TI UA78M05	Linear	5V	500mA	$0.62

Table 10: Potential Voltage Regulators

As shown by Table 8, The CMUcam5 Pixy requires at least 5V, but cannot handle the $12V_{DC}$ input from the AC-DC converter, which means that we need to include a 5V step-down regulator in our system. Table 8 shows that the Processor should not receive more than 3.6V, which means that we need another step-down regular to provide this output level. Table 10 provides specifications of several devices that could fit our project's needs.

3.5 Software

The software for our robot is separated into multiple portions because its used so much in our project. We separate the code into a vision portion, an algorithm portion and lastly a GUI portion. The vision portion is designed to be written in OpenCV and work with the camera we determined would be the best for our system. The algorithm portion is to be written in C for its easy processing power and easy to implement with our device. Lastly we have determined that our GUI will be written in C most likely because it will not take up to much processing power and easy to implement through the USB port on our board.

Vision One of the biggest aspects of our project is the vision of the cube. Our project must take in the layout of the cube discriminating between the colors and placing the information found into a matrix. If the robot can not do this or messes up placing colors in the wrong spots in our matrix then our project can not solve the cube or may not find a algorithm to solve the cube either.

The optimal design for our project would include a six camera layout. If we were to have an unlimited supply of money and could afford as many parts as needed we would buy six different cameras each connected to its own PCB board. With the six camera design each arm would have its own camera. This design would is set to be the most optimal for efficiency and correctness. With the six camera design the system would take a picture of each side of the cube and interface it into the matrix making it so that the matricie could be complete and the algorithm could start right after this step.

Due to our limited budget we can only afford one camera for our robot which makes imaging the cube more difficult and less efficient. This makes the design of our software more adaptable and harder for the matrix to be formed. We decided the easiest way to integrate our cube into the algorithm was to place the camera with the ability to see just a row of three across the top of the front facing side and a row of three across the bottom of the top facing side. We determined that from this point on the cube you can turn the cube enough times to see all of the sides of the cube and place them into our algorithm.

To determine the location of all colors on the cube we drew a 2D cube and went through steps to show how we can visualize the whole cube. We will show the steps as followed explaining what moves have been done to the cube and highlighting where the camera will be looking. In our actual design our camera doesn't move, but to illustrate us rotating the cube it was easier to just move the cameras location. The cameras vision is outlined in Brown as shown in Figure 32.

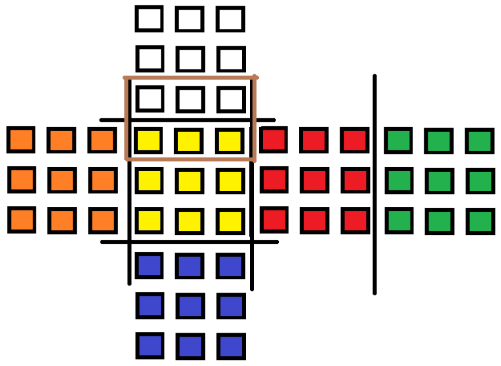


Figure 32: The brown outline symbolizes where the camera is viewing at the moment.

Also for easier explanation the cube we have illustrated has been solved just so we don't lose track of any spots on the cube. For our first step we have determined that all the centers of the cubes will be stuck in a single location therefore we can place those colors to our matrix showing the won't move. This is shown in Figure 33:

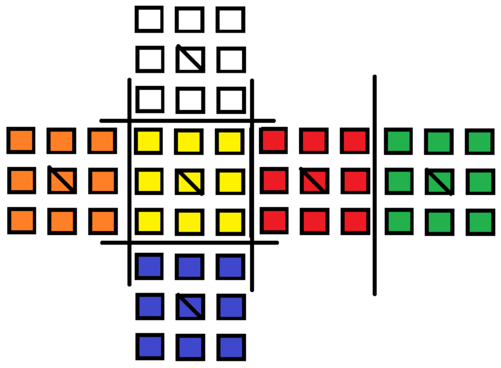


Figure 33: All centers determined

Next our camera takes its first picture of its starting location without any alterations to the cube. We use a slash through the cube to illiustrate those cubes had been taken pictures of and put into the matrix. This pictures is yellow's top side and the white's bottom side, shown in Figure 34.

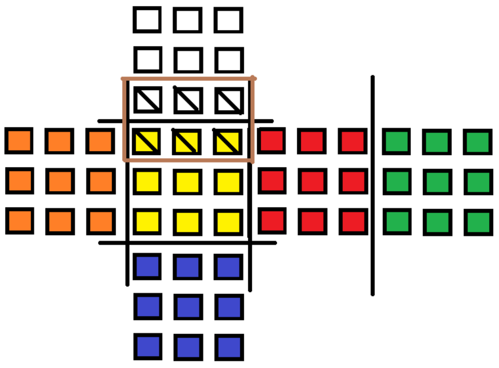


Figure 34: Inital location

The cubes front face is now rotated to the left, taking a picture of the yellow's left side and the red's right side, as shown in Figure 35.

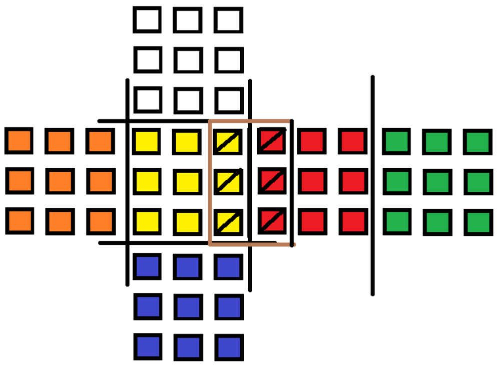


Figure 35: Yellow and red side

Next the cubes front face is rotated to the left, taking a picture of the yellow's bottom side and the blue's top side, as shown in Figure 36

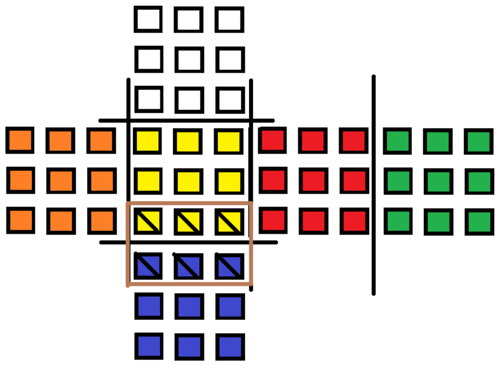


Figure 36: Yellow and blue side

Next the cubes front face is rotated to the left, take a picture of the yellow's right side and the oranges left side, as shown in Figure 37.

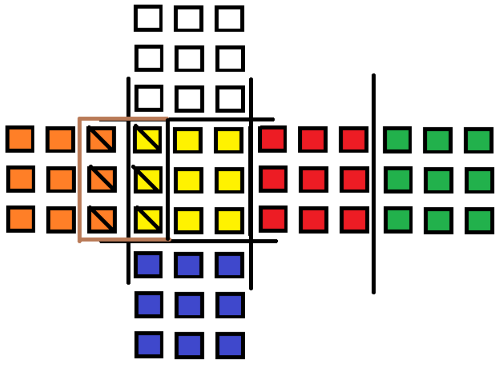


Figure 37: Yellow and orange side

Figure 38 shows all the places the camera has visualized and placed into a matrix after just those little amount of moves.

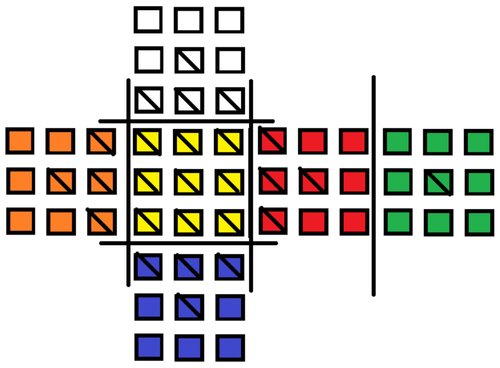


Figure 38: Front face rotation visualizations

The next few steps are harder to visualize with a 2D design. The standards are the same though. The camera will be shown on the parts that are outlined in brown and the places will be marked with a slash after they have been seen. This first images is after you turn the top face to the right, the camera will intake red's top and white's right side, as shown in Figure 39.

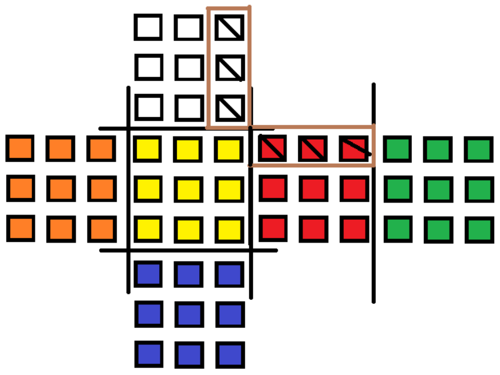


Figure 39: Red and white side

Next the cube's top face will rotate to the right, the camera will then intake green's top and white's top, as shown in Figure 40.

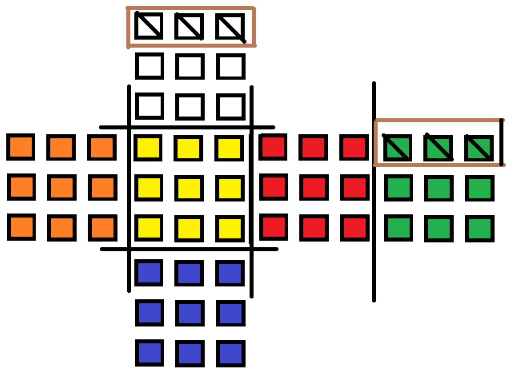


Figure 40: Green and white side

Next the cube's top face will rotate to the right, The camera will then intake orange's top and white's left, as shown in Figure 41.

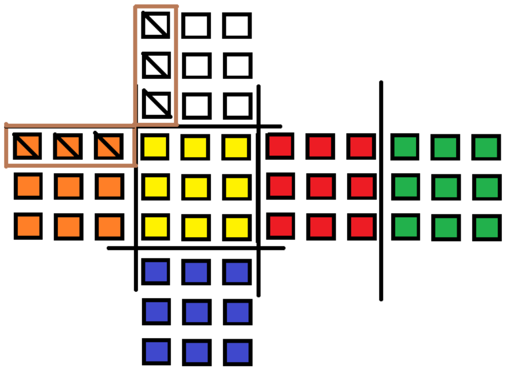


Figure 41: Orange and white side

Next the cube's top face will rotate right reseting it back to normal. From here we will rotate the right face to the left and the top face to the right. From this position the camera will intake red's right and green's left side as shown in Figure 42.

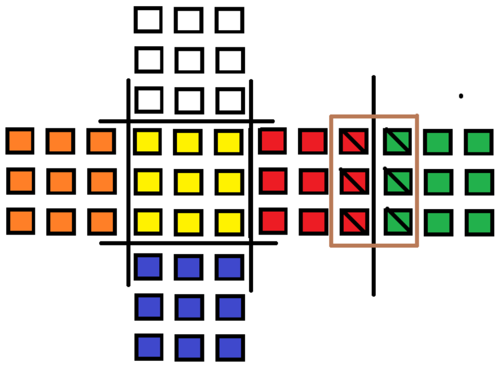


Figure 42: Red and green side

Next the cubes top face will rotate to the left, then the right face will rotate to the left and last the top face will rotate back to the right. The camera will then intake red's bottom and blue's right side as shown in Figure 43

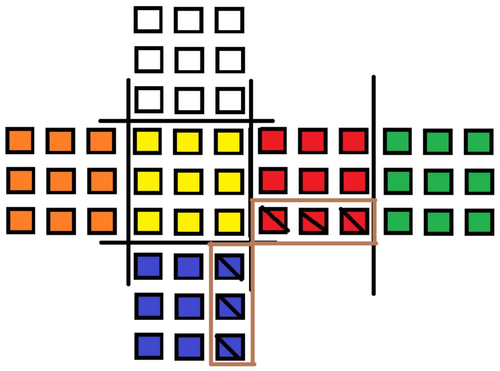


Figure 43: Red and blue side

Next the cube's top face will roate to the left then the right face will rotate left twice. After that the left face will rotate to the right and then the top face to the left. The camera will intake the image of orange's left and green's right side, as shown in Figure 44.

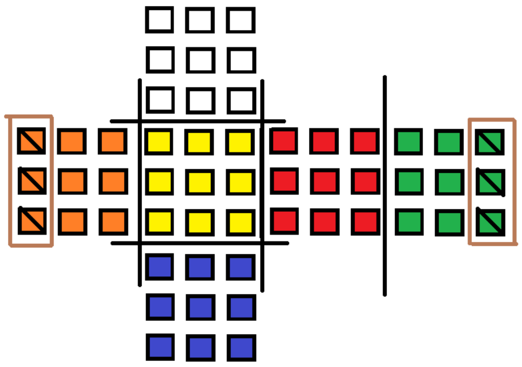


Figure 44: Orange and green side

Next the cube's top face will rotate to the right the left face to the right and then the top face to the left again. The camera will take in orange's bot and blue's left side as shown in Figure 45.

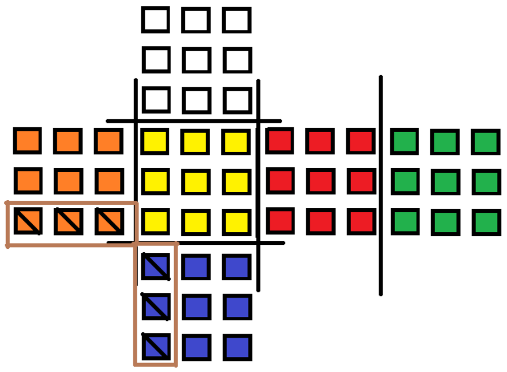


Figure 45: Orange and blue side

There now should only be two spots left unseen by our device. We could take the chance to guess what these spots could be but just to be sure we will image them to put them into the matrix as well. We do this by turning the top face to the right and the left face to the right twice. The cube should now be set back to its normal state and from here we will rotate the bottom face twice and then the front face twice. The camera should then intake blues bottom and greens bottom as shown in Figure 46.

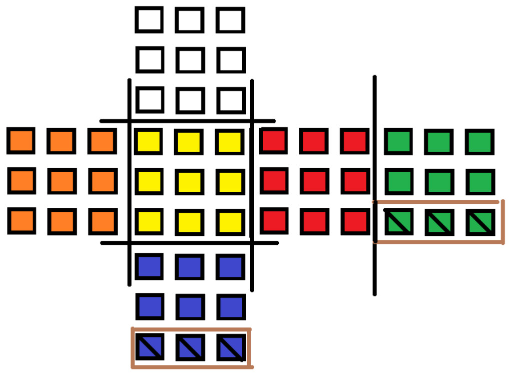


Figure 46: Blue and green side

After the last picture is taken the robot should now rotate the cubes front face right twice and then the bottom face right twice. This should reset the cube back to initial coordination, shown in Figure 47, and able to start the algorithm solving method of our design.

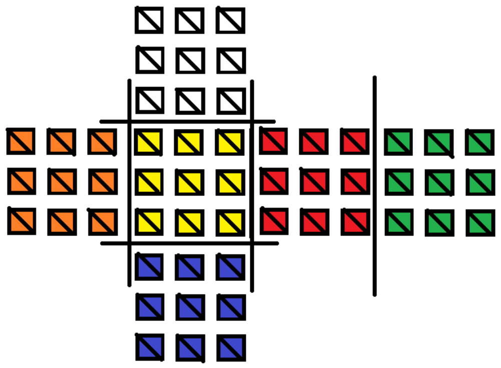


Figure 47: Initial locations with all parts being seen

Interface While using a serial protocol the Pixy cam will output detected objects every 20ms. The ports that can be used with the Pixy cam are SPI, I2C, UART, and analog/digital I/O ports. Pixy also is compatible with USB 2.0 and all these ports are easily switched and easily interfaced with the camera.

- SPI with SS - this is the same as the Arduino ICSP SPI except that it includes support for Slave Select through pin 7 (SPI SS). That is you need to drive SPI SS low before sending/receiving each byte.
- I2C - this is a multi-drop 2-wire port (pins 5 and 9 of the I/O connector) that allows a single master to communicate with up to 127 slaves (up to 127 Pixys). You can configure the I2C address through the "I2C address" parameter.
- UART - this is the common "serial port" (pins 1 and 4 of the I/O connector). Pixy receives data via pin 1 (input) and transmits data via pin 4 (output). You can configure the UART baudrate through the "UART baudrate" parameter.
- analog/digital x - this will output the x value of the largest detected object as an analog value between 0 and 3.3V (pin 3). It also outputs whether an object is detected or not as a digital signal (pin 1 of the I/O connector).
- analog/digital y - this will output the y value of the largest detected object as an analog value between 0 and 3.3V (pin 3). It also outputs whether an object is detected or not as a digital signal (pin 1 of the I/O connector).

Out of all of these ports we will be looking into using either SPI or the UART serial protocol. Luckily for the Pixy cam whether you're using SPI, I2C or UART serial, the protocol is exactly the same.

- The protocol is data-efficient binary.
- The objects in each frame are sorted by size, with the largest objects sent first.
- You can configure the maximum number of objects sent per image frame ("Max blocks" parameter).
- SPI and I2C operate in "slave mode" and rely on polling to receive updates.
- When there are no detected objects (no data) Pixy sends zeros if the interface is SPI or I2C (since Pixy is a slave, it has to send something).
- Each object is sent in an "object block" (see below).
- All values in the object block are 16-bit words, sent least-significant byte first (little endian). So, for example, when sending the sync word 0xaa55, Pixy sends 0x55 (first byte) then 0xaa (second byte).

Bytes	16-bit word	Description
0, 1	0	sync: 0xaa55=normal object, 0xaa56=color code object
2, 3	1	checksum (sum of all 16-bit words 2-6)
4, 5	2	signature number
6, 7	3	x center of object
8, 9	4	y center of object
10, 11	5	width of object
12, 13	6	height of object

Table 11: Object Block format Description

From these protocols outcomes the object block format that is sent to our board to communicate data that can be implemented into our matrix.

From research from the Pixy cams website we feel our best option for using the camera is to run the interface through UART. SPI has been known to run the images faster but has also known to have more errors in sending the correct data and less confusing code for implementation. We have discussed the idea of using SPI but we determined that we don't want to make our code any more confusing.

Object block format Table 11 shows how the object block format is sent to the board using the Pixy cam processor. It takes in objects using the purple dinosaur method and finds the color of the cube using the 4 and 5 bytes or the signature number. Then using the 6 and 7 bytes it determines the horizontal center of the object. The 8 and 9 bytes are used for the vertical center of the object. The last 4 bytes are used to find the horizontal width of the object its looking at and the vertical height of the object its looking at. We will have to use the signature number found and the width and height bytes to determine what colors are where in the matrix.

Processor The specs of the processor are as follows:

- NXP LPC4330
- dual core, ARM Cortex-M4 and Cortex-M0
- 208 MHz (both cores)
- 264k RAM (0 wait state)
- USB 2.0 high speed (OTG support)
- I2C, SPI, UART, etc.
- A/D, D/A
- Floating point unit (M4)
- SIMD instructions (M4)

Sensor The specs of the sensor are as follows:

- Omnivision OV9715
- 1/4” sensor - low light, low noise
- 1280x800, RGB bayer direct
- 25 fps full resolution, 50 fps 640x400
- end-of-life tolerant

Communicating The Pixy cam communicates by sending bits back to the board. If the information sent back to the board is then read into a program it can be read and manipulated to figure out what is located where in the system. We have broken down a list of what is read into the program when sent in.

- pixy.blocks[i].signature The signature number of the detected object (1-7 for normal signatures)
- pixy.blocks[i].x The x location of the center of the detected object (0 to 319)
- pixy.blocks[i].y The y location of the center of the detected object (0 to 199)
- pixy.blocks[i].width The width of the detected object (1 to 320)
- pixy.blocks[i].height The height of the detected object (1 to 200)
- pixy.blocks[i].angle The angle of the object detected object if the detected object is a color code.
- pixy.blocks[i].print() A member function that prints the detected object information to the serial port

Using this information we can determine the color of the cube that is being sent in using the signature function. next we can determine if there are any of the same cubes touching by looking at the width and center location on the x axis, and also the height and y location. Using these two sets of numbers we can determine if 2 or 3 of the same colors are touching or if none of the same colors are touching at all. Lastly the print function will send the information needed back to the serial port or our board and allow us to configure the cube using our algorithms function.

3.5.1 Operating System

For our robot we needed to run an operating system for our devices. When we first did our research we discovered that the MSP430 runs a RTOS or real time operating system, later on we will discuss what this means for our device. The only other part of our device that needs an operating system is the Pixy cam. Luckily the Pixy cam can learn a lot of different types of Operating systems. Through our research though we have determined that the most applicable operating system for our robots camera will be Linux.

Real Time Operating System A real-time operating system (RTOS) is a multitasking operating system which provides scheduling algorithms to help a software developer guarantee deadlines of system tasks. The most efficient algorithm that is written for performing the task switching will have the least amount of overhead. We found a list of RTOS that could work for our project. They are listed in Table 12.

uC/OS-II	Contiki	CMX-Tiny+
DioneOs	embOs	FreeROTS
PowerPac	QP	Salvo
TinyOS	ChibiOS	B.lu BOS
FunkOS	SYS/BIOS	Abassi
MSS		

Table 12: List of Potential RTOS

Through our research we have determined that the operating system that will be most applicable for our project would be QP. The reason for this is because its a very lightweight, open source, state machine-based frameworks for embedded microprocessor. This framework can be used with or without tradition RTOS and can be configured to whatever families and libraries needed for our robot.

Linux For our Pixy cam to run the PixyMon application we have determined that we would need an operating system for our device. The easiest operating system we decided to use would be Linux. Linux is an open source free library operating system that can be easily manipulated and configured to fit whatever needs. We determined any other operating system may in return be to big for our device or may not have the library needed for our program.

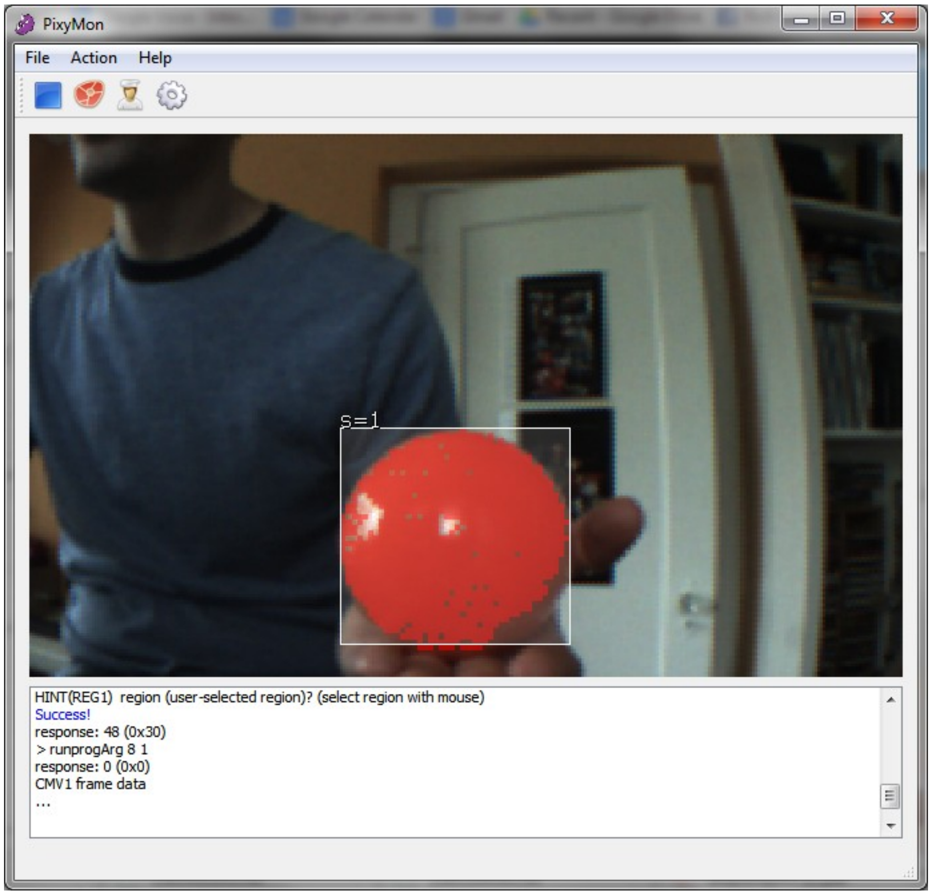
3.5.2 Languages

For our project we have determined that the best option to implement our code through programing will be to use C language. When we were determining which language should be used we looked at Java, C, C++, Vpython, OpenCV, and JavaScript. While doing research we found that Java and JavaScript when implemented on the MSP430 board they tend to take up to much space. This wouldn't be an issue if they board was only implementing an Algorithm code set but since it also has to move the arms of our robot, intake the data found from our PixyCam and lastly build and run our GUI during the process of solving the cube. With all of these functions running we would need to find a language that doesn't take up to much space and processes the code easily. Lastly we had determined the best way we could implement the vision portion of our code was to run an OpenCV program that could intake the images and process them itself. The big issue with OpenCV is it tends to take up too much processing power no matter what it is ran on and the data sent into the device is most of the time useless because it tends to include bits that are not necessary or relevant to our information. Luckily with the camera we decided to order for our robot it was designed to work with our board and has its own processor on it. This processor takes in all the data from the camera and removes all the information that is useless from the image processing and sends the useful information to the MSP430 Board.

C C has facilities for structured programming and allows lexical variable scope and recursion, while a static type system prevents many unintended operations. Function parameters are always passed by value. Pass-by-reference is simulated in C by explicitly passing pointer values. C program source text is free-format. Many later languages have borrowed directly or indirectly from C, including C++, Java, JavaScript, Limbo, C#, Objective-Cl, PHP, Python, Verilog, and etc. These languages have drawn many of their control structures and other basic features from C. Most of them are also very syntactically similar to C in general, and they tend to combine the recognizable expression and statement syntax of C with underlying type systems, data models, and semantics that can be radically different. We have decided that C will be the basis for most of our programming because of its open-source library and its free-format. Also a library known as OpenCV can be ran through it which will be used for our vision processing.

OpenCV OpenCV is a library of programming functions mainly aimed at real-time computer vision. The library is a cross-platform and free for use under the open-source BSD license. In the early days of OpenCV, the goals of the project were described as Advanced vision research by providing not only open but also optimized code for basic vision infrastructure, disseminate vision knowledge by providing a common infrastructure that developers could build on, so that code would be more readily readable and transferable, and lastly advanced vision-based commercial application by making portable, performance-optimized code available for free. OpenCV is written in C++ and its primary interface is in C++, but it still retains a less comprehensive though extensive older C interface. The applications for OpenCV include:

- 2D and 3D feature toolkits
- Egomotion estimation
- Facial recognition system
- Gesture recognition
- Human-computer interaction (HCI)
- Mobile robotics
- Motion understanding
- Object identification
- Segmentation and recognition
- Stereopsis stereo vision: depth perception from 2 cameras
- Structure from motion (SFM)



```
HINT(REG1) region (user-selected region)? (select region with mouse)
Success!
response: 48 (0x30)
> runprogArg 8 1
response: 0 (0x0)
CMV1 frame data
...
```

Figure 48: PixyMon GUI

- Motion tracking
- Augmented reality

We will only be using the object identification and segmentation and recognition aspect of this programming. But it is very helpful that OpenCV could be used for all of these different types of vision control.

PixyMon PixyMon is an application that allows you to configure Pixy and see what it sees. It runs on several different platforms including Windows, MacOS and Linux, as well as other smaller embedded systems like Raspberry Pi and BeagleBone Black. We are hoping we can implement PixyMon into our application as well to make the camera operation of our robot easier. The picture below shows how PixyMon is implemented with the Pixy cam.
We are hoping by using this program it can help us write our code to implement the making of the matrix even if we can't find a way to use it with our embedded computer.

3.6 Hardware-Software Interface

For connecting our hardware to software we need to use different types of connections. For example we need a UART connection for our Pixy cam, and a USB connection for either a computer application or graphical display. All these functions can only connect to our hardware using I/O interfaces. Without the correct I/O our system will have issues running the correct features or even getting data from the other devices.

3.6.1 Platform I/O Capabilities

The following I/O capabilities are available for our MSP430 and will be used to connect our hardware to our software:

- GPIO
- Serial
- I2C
- UART
- USB

GPIO The MSP430F6659 offers 74 GPIO pins that are mappable to various peripheral interfaces as well as the integrated USB engine and LCD driver. Configurable parameters of the GPIO pins are shown in table 13.

USB The purpose of our USB port is to implement any portion of the project we can't run in its designated port or even to allow more work to be done. The USB port has had a rise in popularity as of late in the field of technology. It now has the ability to connect to most devices therefore many devices are being made with them. This is a great aspect for us because it allows us to have a fail safe method where if a GPIO or platform I/O doesn't work with a device we can implement it with the USB port. Also as discussed in our design for the embedded computer we may use the USB to connect our device to the computer. With this connection we can run applications from our computer and allow them to run through the embedded computer.

3.6.2 Device I/O Requirements

Through our research we have determined that the following I/Os will be needed to implement all parts of our robot's design. A UART, SPI, or I2C will be needed to implement our Pixy cam, but our decision among those three ports is to use the UART. We will need to use a SPI or serial interface to implement our monitors. We will need a USB port to implement a actual computer for the program or any function that doesn't work with the port determined for it. Lastly we will need one GPIO ports for each servo motor needed to implement our design. In all we will need a total of 6 GPIO ports to run ever servo motor.

Port	Function
PxIN	Port x input. This is a read-only register, and reflects the current state of the port's pins.
PxOUT	Port x output. The values written to this read/write register are driven out the corresponding pins when they are configured to output.
PxDIR	Port x data direction. Bits written as 1 configure the corresponding pin for output. Bits written as 0 configure the pin for input.
PxSEL	Port x function select. Bits written as 1 configure the corresponding pin for use by the specialized peripheral. Bits written as 0 configure the pin for general purpose I/O. Port 0 is not multiplexed with other peripherals and does not have a P0SEL register.
PxREN	Port x resistor enable. Bits set in this register enable weak pull-up or pull-down resistors on the corresponding I/O pins even when they are configured as inputs. The direction of the pull is set by the bit written to the PxOUT register.
PxDS	Port x drive strength. Bits set in this register enable high current outputs. This increases output power, but may cause EMI.
PxIES	Port x interrupt edge select. Selects the edge which will cause the PxIFG bit to be set. When the input bit changes from matching the PxIES state to not matching it, the corresponding PxIFG bit is set.
PxIE	Port x interrupt enable. When this bit and the corresponding PxIFG bit are both set, an interrupt is generated.
PxIFG	Port x interrupt flag. Set whenever the corresponding pin makes the state change requested by PxIES. Can be cleared only by software.
PxIV	Port x interrupt vector. This 16-bit register is a priority encoder which can be used to handle pin-change interrupts. If n is the lowest-numbered interrupt bit which is pending in PxIFG and enabled in PxIE, this register reads as 2n+2. If there is no such bit, it reads as 0. The scale factor of 2 allows direct use as an offset into a branch table. Reading this register also clears the reported PxIFG flag.

Table 13: GPIO Configuration Registers

PWM Pulse-width modulation is a modulation technique used to encode messages into a pulsing signal. Even though this method can be used to encode information for transmission, its main use is to allow control of the power supplied to electrical devices. We will be using this method in method of control to turn the robot's arms when necessary. The main advantage of PWM is that power loss in the switching devices is very low.

A/D Conversion An analog to digital converter is a device that converts a continuous physical quantity to a digital number that usually represents the quantities amplitude. A ADC is defined by its bandwidth and its signal to noise ratio. It is mostly used to detect binary signals as in if the button is being pressed or not. When a microcontroller is controlled by a power source it determines the amount of voltages being fed to it will be a binary of 1 and when no voltage being sent to it as a binary of 0. By using this device we can convert

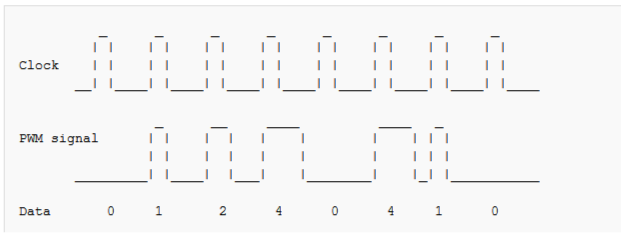


Figure 49: Modulation of PWM with clock cycle

from the analog world to the digital world. This in return allows us to interface electronics to the analog world around us.

4 Design

4.1 System

Our system is designed as a few parts that work separately without each other but when incorporated together make an outstanding robot design that can solve a Rubik's cube autonomously. The main body of our system holds 6 stepper motors that are each connected to a single face of the cube. These stepper motors are than all connected to a single PCB that sends the signals to the corresponding stepper motor when that face of the cube needs to be rotated. The PCB is connect through UART connection to our PIXY CMU5 cam that processes the information on the cube and sends frames of the cube to the PCB where it can be deciphered for the correct orientation of the cube. The information received by the PCB from the camera is than transfered to the PC through a USB connection. This USB connection is a CDC communications and basically allows for a virtual comm port to be used on the computer.

The PC takes in the information from the PIXY through the PCB's USB connection. With the information transfered to the PC it is than deciphered for the color and position of each face of the cube. After the total cube is viewed and the position of every color is found, the orientation of the cube is used to find a set of moves that allow the cube to be solved back to its correct orientation. After this certain move set is found we send back through the USB connection to the PCB the set of moves. The moves are than broken down by the PCB and send signals to each Stepper motor to solve the cube.

4.1.1 System-Level Block Diagram

The system-level block diagram for the robot is shown in Figure 50. The PC software application is designed to interact with the Pixy over a USB connection to extract color data from the cube. The MSP430 is also connected to the software application over a USB connection as a CDC device.

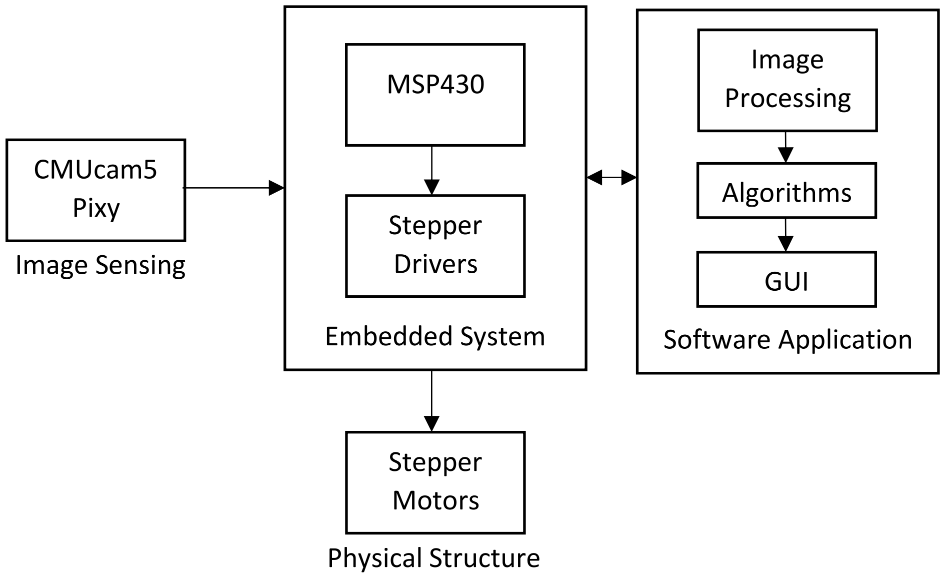


Figure 50: System-Level Block Diagram

4.1.2 Hardware/Software Integration

Our hardware integrates with our software on multiple levels. To start our Camera takes in the frame of the cube it is currently looking at and sends it through UART connection to our PCB where it then is sent to the PC to be deciphered for the correct colors and positions of the cube in the frame. The information sent is broken down into 6 blocks of 13 hexadecimal bytes that each are looked at to determine the position and color for the cube face seen by the camera. We look at 5 different bytes from each block to determine the color of that block and its X center and Y center. We then take the X center and Y center to determine the position of the color in the frame. After we find the correct spots of each color in that frame we place the blocks into an array showing its position in the cube for our software. After the software finishes its calculations to solve the cube it takes a string of moves and sends it back over the USB connection back to the PCB. The PCB than deciphers the move set and sends the correct movements to each stepper to show the cube being solved.

4.2 Hardware

4.2.1 Structural Implementation

A rendering of the physical structure is shown below in Figure 51. The structure was designed in Autodesk Inventor and can be 3D printed or projected onto 2D surfaces for laser cutting. The actual implementation of the structure will be a mixture of these two manufacturing methods. The structure is modular so that it can be easily disassembled to allow for removal of the cube: First, the top motor can be removed simply by lifting it upwards. Next, each of the side motors can be slid away from the cube thanks to elongated drill slots on the bases of the motor mounts. Once the top and side motors are removed, the cube remains resting on the bottom motor and can easily be picked up and scrambled by hand.

One of the motor mount structures has an extra attachment at the top for the Pixy camera that allows it to view the upper edge of the cube. Figure 52 shows the Pixy's viewpoint of the cube edge with the image detection overlaid on the original image.

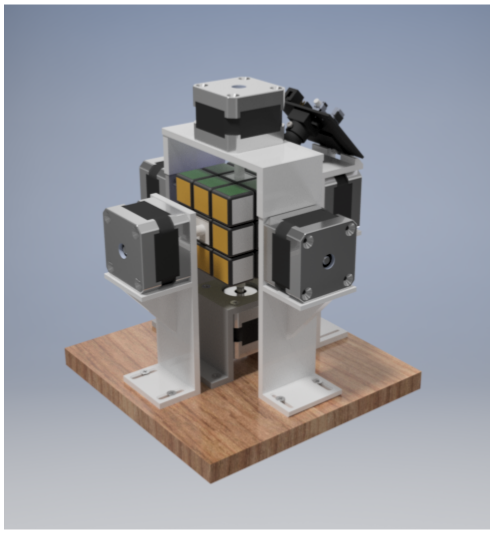


Figure 51: Render of Physical Structure

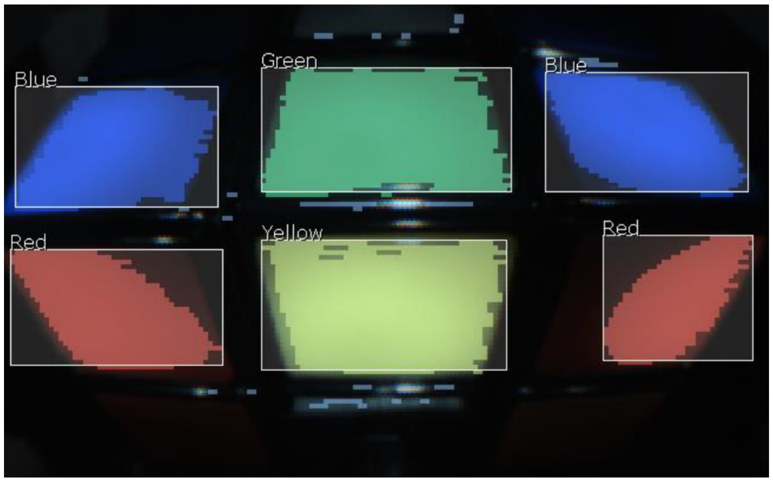


Figure 52: Pixy Image with Color Detection Overlay

4.2.2 Embedded System

The embedded system is the central electronic platform for our robot that hosts the embedded processor as well as hardware for interfacing the perihperal devices such as the motors and Pixy camera. Figure 53 shows a functional diagram of our embedded system.

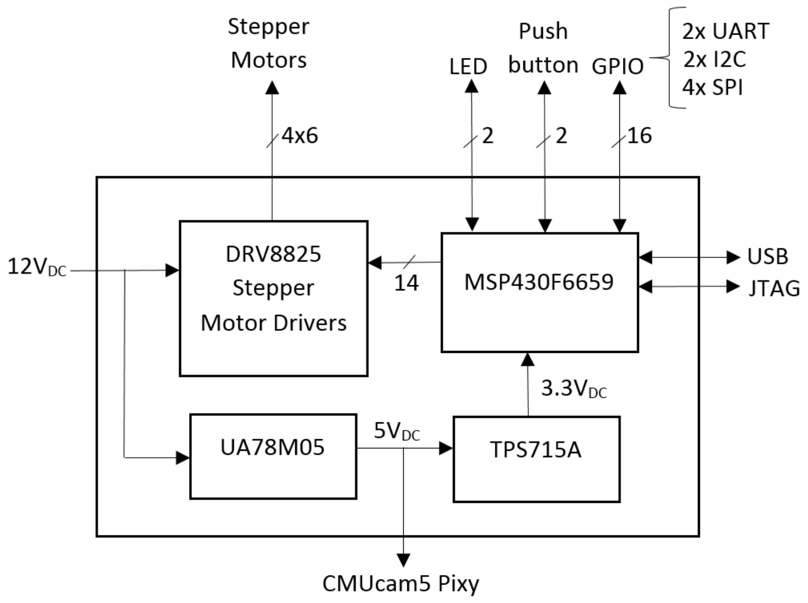


Figure 53: Embedded System Functional Diagram

Processing Platform As shown in Figure 53, our embedded system implements a TI MSP430F6659 microprocessor whose detailed specifications are listed in Table 14. We have selected this device because it has a reasonable price, sufficient memory for our needs, and allows us the option of using USB-based programming tools in the future. The following subsections detail our implementation of the of the MSP430.

Frequency	20MHz
Non-volatile Memory	512KB
RAM	66KB
GPIO	74
I2C	3
SPI	6
UART	2
DMA	6
Timers (16-bit)	4
Multiplier	32x32
BSL	USB
Min V_{CC}	1.8V
Max V_{CC}	3.6V
Active Power	$360\mu A/MHZ$
Package	100LQFP

Table 14: MSP430F6659 Specifications

GPIO The MSP430F6659 offers a maximum of 74 GPIO connections, but our application only requires a fraction of these. Table 15 lists the GPIO connections that we are using in our system. This list does not include shared functionality GPIO pins that we use for other purposes such as oscillators, JTAG connections, etc.

Purpose	Total Pins	Pin Names
Motor Driving	14	P1.0→P1.7, P2.0→P2.5
Serial I/O	16	P8.0→P8.7, P9.0→P9.7
Physical user I/O	4	P3.0→P3.3

Table 15: MSP430 GPIO Connections

USB Interface The MSP430 is interfaced with a micro USB port in our embedded system via its dedicated USB pins as shown in Figure 55. In our USB interface, we include a Texas Instruments TPD2E001DRLR (Figure 54) for electrostatic discharge protection to protect the MSP430.

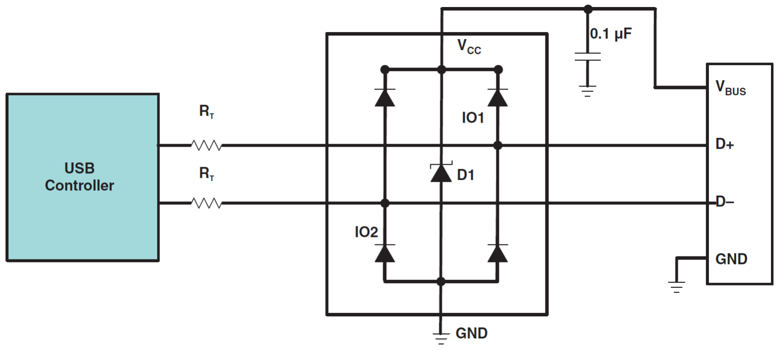


Figure 54: TPD2E ESD Protection Application

Additionally, the USB 2.0 operation requires the use of a 48MHz PLL built into the MSP430's USB engine [38]. This PLL depends on a reference clock input higher than 4MHz through one of the MSP430 system oscillator ports (XT1 or XT2). Our solution for this follows TI's design recommendations by using a 4MHz Murata CSTCRG15L, which is a cheap, high-precision ceramic resonator. Table 16 outlines the specifications of this device. The resonator schematic is shown in Figure 56.

Part Number	CSTCR4M00G15L99-R0
Frequency	4MHz to 7.99MHz
Initial Frequency Tolerance	±0.1%
Frequency Shift by Temperature	±0.08%

Table 16: Murata CSTCRG15L Specifications

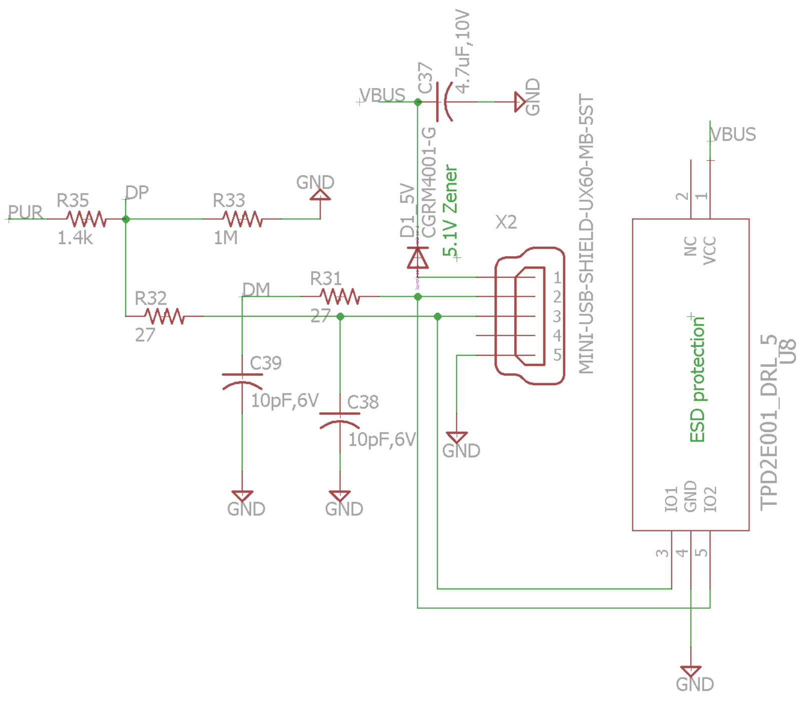


Figure 55: USB Port with ESD Protection

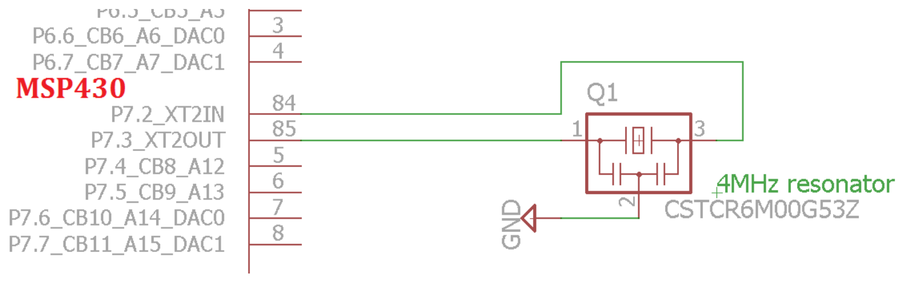


Figure 56: 4MHz Ceramic Resonator for USB engine PLL

Motor Driving We use the TI DRV8825 Stepper Motor Driver to interface the motors with the MSP430. One DRV8825 is required per motor, meaning that we need six separate DRV8825 ICs. As listed in Table 6, the DRV8825 provides $\frac{1}{32}$ microstepping. It also features a configurable decay mode and a simple control interface that is shown in the functional block diagram in Figure 57.

Most of the control signals can be driven by single pins on the MSP430 and applied to to all 6 driver units . These signals include the microstep size, decay mode, PWM input signal, and direction signal. The DRV8825 features an active low enable line that can force the unit to disable the H-bridges, leave the outputs in a high-Z state, and ignore step input commands. We can take advantage of this functionality by allowing the MSP430 to control the enable line of each DRV8825 independently. In this way, it only needs to supply one PWM signal and pick which motor driver it wants to enable. Figure 58 shows the schematic of our DRV8825 implementation.

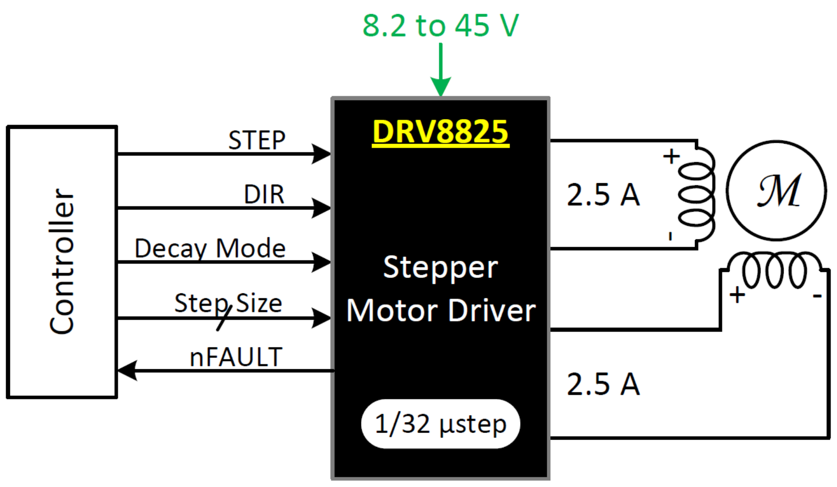


Figure 57: DRV8825 Functional Block Diagram

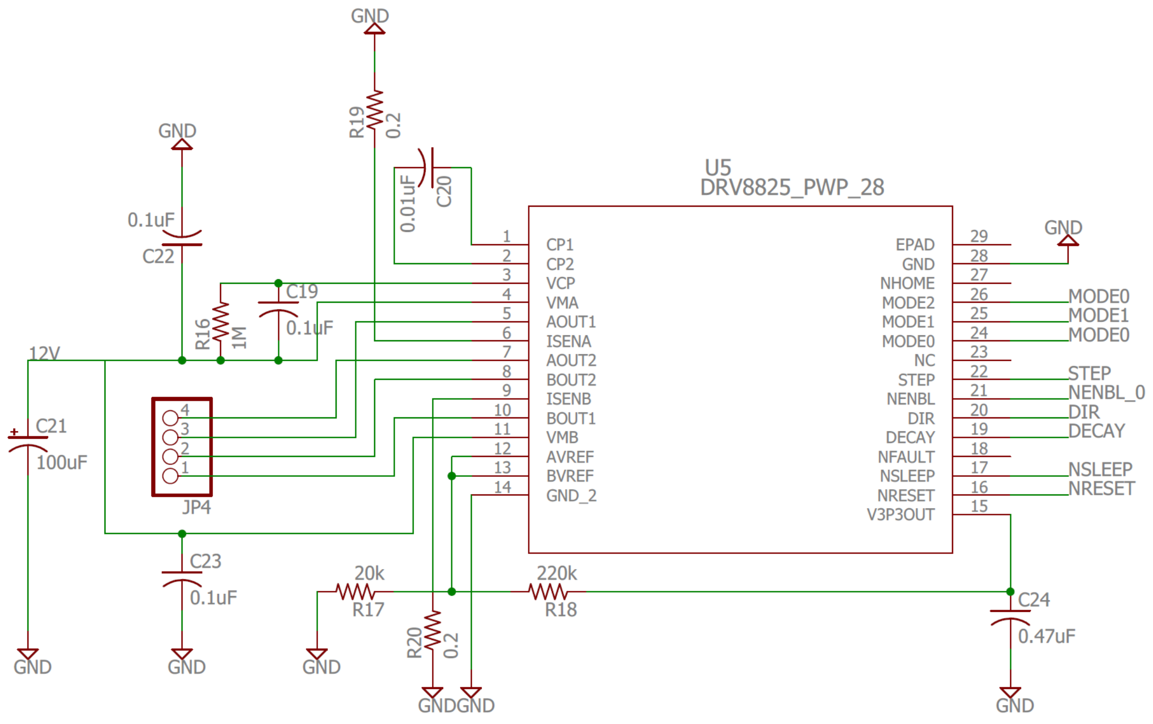


Figure 58: Motor Driver Schematic

Power Our system is designed to recieve a 12V_{dc} input which supplies the motor drivers. To supply the Pixy camera, we use a Texas Instruments UA78M05 fixed output linear voltage regulator. The specifications for this device are shown in Table 17. The 500mA output current is more than enough to supply the typical 140mA required by the Pixy, with leftover current for the MSP430 [9].

Maximum Output Current	500mA
Output Voltage	5V
Minimum Input Voltage	7V
Maximum Input Voltage	25V

Table 17: UA78M05 Low Dropout Regulator

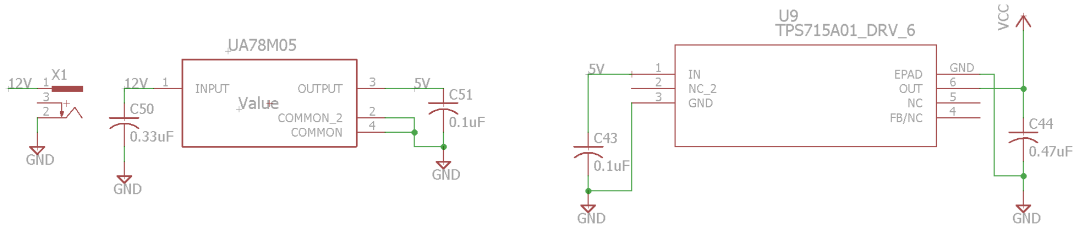


Figure 59: Linear Voltage Regulator Schematics

The MSP430F6659 has a maximum input voltage of 3.6V so it is necessary to add another step down voltge regulator that can reduce the 5V output of the UA78 to a lower level. Because the efficiency of a linear voltage regulator is inversely proportional to the voltage difference between its input and output, it makes sense to supply this sub-3.6V regulator with the 5V supply from the UA78 rather than the 12V system input supply. The Texas Instruments TPS715A works well for this purpose because its maximum output current of 80mA is more than enough to supply the $295\mu A/MHz$ requirements of the MSP430. Figure 59 shows the schematics of both linear voltage regulators.

User Feedback I/O Many development boards provide buttons, switches, and LEDs for users to interact with. These I/O devices can be useful for project development and execution by allowing users to send and receive simple feedback to and from integrated hardware. In our project, we can use the switches to test hardware functionality, run the robot, reset the robot, and have the robot scramble the cube. Figure 60 shows the simple feedback I/O schematic.

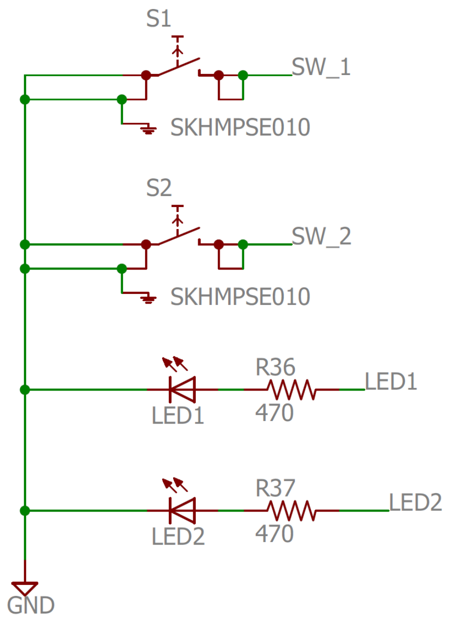


Figure 60: User Feedback I/O Schematic

4.2.3 PCB Design

The PCB supports the entire embedded system and is designed with Eagle CAD 7.4 for a two layer board. The board thickness is 1.6mm FR4 with 1oz copper [20]. Both sides of the board utilize a ground plane and the trace thicknesses are calculated to meet 10°C temperature rise. Figure 61 shows the final PCB layout. The board dimensions are 1.95 inch height and 3.33 inches width.

Several TI IC devices on the PCB utilize the PowerPAD package which features an exposed metal thermal pad on the bottom of the package [36] that is joined to the dieframe inside the plastic package. This allows for enhanced thermal diffusion through the package and out through the bottom of the PCB. The PCB design takes advantage of these packages by using thermal vias underneath the PowerPADs which can be joined with thermal paste.

4.2.4 Camera

Our robots main method for taking in data from the cube is its camera. We had many issues trying to decide which way of using the camera would be the easiest for the robot to intake data. Most importantly though with all these designs we needed the camera to intake the data without mistakes and be able to insert it into a matrix where it can solve the cubes algorithm.

Our first design was to use six different cameras, one on each end of the arm so each side could intake the data and set it into a matrix easily. This design by far is the easiest and the fastest to implement. With this design the arms can instantly intake the cubes

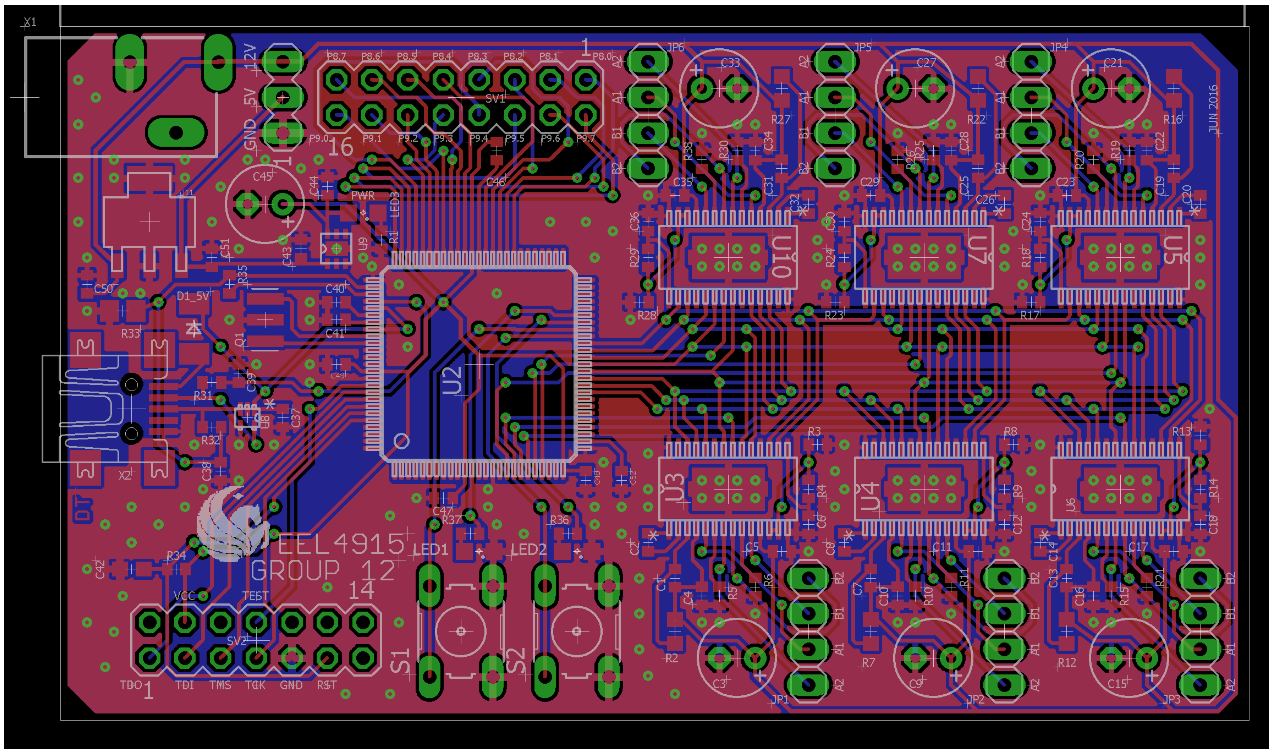


Figure 61: PCB Layout

layout and instantly start solving the algorithm for the correct cube. The issues with this design is the amount of money it would take to make it and the amount of data taken into the robots processor. With this design we would need multiple Pixy cams. The Pixy cam can also only connect to one MSP430 at a time therefore meaning we would need multiple MSP430s. This would in return would increase the price of our project substantially which is not what we were looking for in our project. Next the processing all of the data from 6 different camera's into one processor would over work our central processor and not allow it and energy to run any other function it is capable of doing. Overall this way is the most efficient way but it has many flaws that could result in our project not working.

Our next design is to implement one camera into the robot. This we feel is the best option for our project because it then in return doesn't cost to much, leaves room in our processor for other functions and then lastly can visualize the whole cube without error. In this design we have decided that if we take one Pixy cam and place it above the front face of the cube and tilt it so it can visualize the bottom row of the top face and the top row of the front face of the cube than we can manipulate the cube enough to know the position of each color on the cube. Also as long as we remember our motions we can convert the cube back to its initial scramble. This design doesn't happen to be efficient because of the time it will take to visualize the whole cube but it is cost efficient and process efficient which makes it our pick.

4.3 Programming

4.3.1 Monitor

We decided to incorporate a display into our design to allow for a GUI to be integrated into our project. With this GUI we would have more for our project to be graded on and allow us more work if we finish our project early. The only issue is we are not sure if we would have enough processing power on our board and enough funds to incorporate a display that would be pleasing to the eyes and useful for someone who doesn't know whats going on with our robot. We determined that there are three different design we could do to fulfill this goal. One design is to have a full GUI with a display of the cube and the actions the robot will be taken to solve the cube. The next idea for a design would be a half GUI. This idea is to not incorporate all the designs from the full design but still some features. Lastly we decided the bare minimum for our GUI would be just a timer display.

For the full GUI display we would incorporate multiple features into our design. The features include a display of the cube that is being solved. This display will interact and move the cube on the display as the cube is moved in real time. This GUI will also include a move counter. The move counter would taken in the total amount of moves it would take for the robot to solve the cube. Lastly the GUI would incorporate a timer onto it. This timer would start when the robot started visualizing the cube and end when the cube was solved.

For a half GUI display we would incorporate multiple features into our design that have been mentioned above but not all the same features. For this design we figured the easiest and best way to place information onto the GUI would be to just add a move counter onto the GUI. We felt as if this wouldn't take up to much space and also not increase the processing power of the MSP430 to much. Also we would add a timer onto our robot to allow the user to see how long it would take the robot to solve the cube.

The last way we would build the GUI to design our robot would be to just place a timer on the robot. The timer would still go through our embedded processor and would be tracked by the robot but it wouldn't take up barely any processing power and would be a good use for the user to tell how long the robot takes to solve the cube.

For our final design our GUI is a basic 2-dimensional Rubik's Cube that has the appropriate faces colored. The main features of it is the ability to type in the state of the cube and see the visual of it. This is implemented with the 'Set' button that colors the cube according to the values that were inputted into the text box for the cube. The reason for this button was so that the user could visually see if they typed in each face of the cube correctly.

The next main button that was implemented was a 'Solve' button. This solve button takes the string from the input text area that mapped the cube and runs Kociemba's algorithm to generate the solution for the cube. Once the solution of the cube is generated it is saved as a string.

Another button that we implemented was a ‘Scan’ button. This button was used for visualizing the cube with the camera we used. It sent a ‘!’ character to the MSP430 microcontroller over serial through bytes that would then initialize the camera to begin scanning. The camera scanned and then sent back the current 2D array of the edge of the cube it was currently seeing through the microcontroller and back to PC through the serial connection over bytes. Those byte values were read then interpreted and converted to a string that set the colors for the appropriate edges in the 2D rubik’s cube that was displayed. Waits were implemented between each read/write over serial to ensure no data was being lost. The cube would then be turned so that a different edge could be analyzed, sent over to the MSP430, analyzed by the camera (Pixy Cam), sent back over through the MSP430, read by the application, and then set the appropriate colors on the 2D rubik’s cube. This process was repeated until the entire 2D cube was filled.

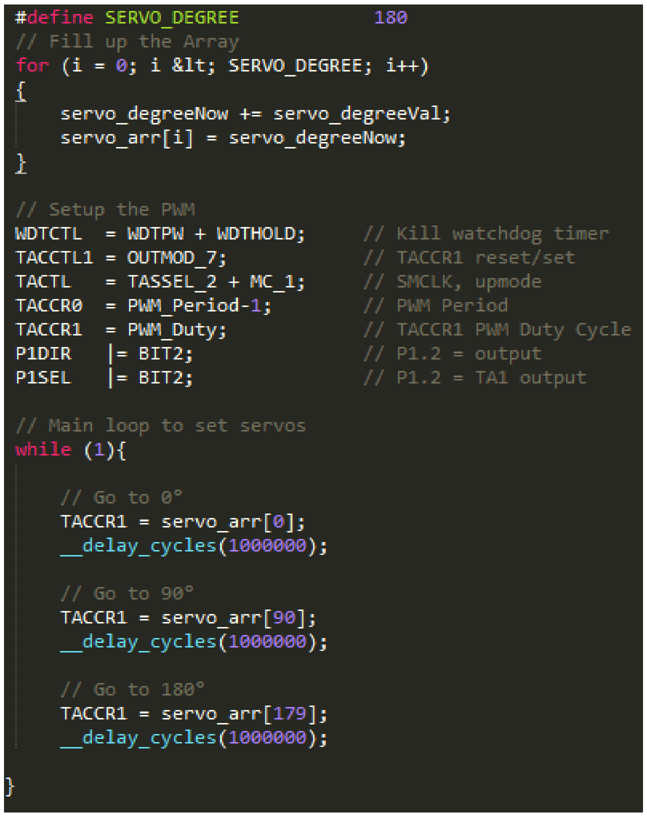
The 2D cube itself is generated by basic jbuttons in java and are non-clickable. Having a cube that could set the state by pressing each cube to change between the colors for each of the 54 cubes would take a considerable time. Not only would that take a considerable time, but mistakes of pressing a cube and not remembering which one you pressed are likely and would generate more errors. Thus, this feature was non implemented at all. The 2D cube also starts by displaying a solved state of the cube.

4.3.2 Hardware Integration

We have determined that to integrate the hardware with our software programming we could use C coding. The parts of the hardware that need to be integrated with our software include the Pixy cam with its processor, the MSP430, the robotic arms and steppers, and the display. For the Pixy cam we can integrate all of our commands through C coding and the information sent into our device is easily readable and modifiable when put into C. Next the MSP430 is ran though our operating system which is capable of C programming. The robotic arms and steppers for our device can be altered and rotated in C programing. The steppers are designed to be run with C. Lastly when designing a GUI we decided to stick to the same type of coding as the rest of the system and not confuse ourselves by integrating a different language for just one part.

In the picture above is C code to demonstrate a basic setup of the PWM. It also shows how the servo rotation will be handled by saving the rotation values into an array name servoDegree. Once the algorithm has determined the order to move the robot arms it will also tell to which degree in which the servoDegree array will be called upon for each particular arm.

For our final design of our project we had to change how our GUI was built and how we read in the PIXY cam’s information. The PIXY cam was implemented through UART connection to our PCB and therefore it has no code to run it, it basically sends information over to the PCB when the port is turned on and does not when the port is closed. The GUI had to be changed when we determined that instead of a monitor we would be using a PC to allow our algorithm to run faster and make sure we have no issues with memory from our program. To integrate our PC to our PCB we used comm port communication in our



```
#define SERVO_DEGREE            180
// Fill up the Array
for (i = 0; i &lt; SERVO_DEGREE; i++)
{
    servo_degreeNow += servo_degreeVal;
    servo_arr[i] = servo_degreeNow;
}

// Setup the PWM
WDTCTL  = WDTPW + WDTHOLD;      // Kill watchdog timer
TACCTL1 = OUTMOD_7;             // TACCR1 reset/set
TACTL   = TASSEL_2 + MC_1;      // SMCLK, upmode
TACCR0  = PWM_Period-1;         // PWM Period
TACCR1  = PWM_Duty;             // TACCR1 PWM Duty Cycle
P1DIR   |= BIT2;                // P1.2 = output
P1SEL   |= BIT2;                // P1.2 = TA1 output

// Main loop to set servos
while (1){

    // Go to 0°
    TACCR1 = servo_arr[0];
    __delay_cycles(1000000);

    // Go to 90°
    TACCR1 = servo_arr[90];
    __delay_cycles(1000000);

    // Go to 180°
    TACCR1 = servo_arr[179];
    __delay_cycles(1000000);

}
```

Figure 62: Basic setup of PWM and servo rotation

program and the PCB connects to the PC through a USB port and therefore does not need any extra code.

4.4 Software

The software programming is very important and has multiple parts and functions for our robot. If any part of these functions do not work though our robot can not succeed in solving the cube and therefore our project will be a failure. The software is broken up into three main functions. The first being the integration of the colors into an matrix that can be worked with. The second function is the algorithm of solving the cube with the matrix we found in our first function. Lastly we will have a function that can implements the rotation of the robots arms to solve the cube. If one of these functions doesn't successfully work than project may not succeed in its overall plan.

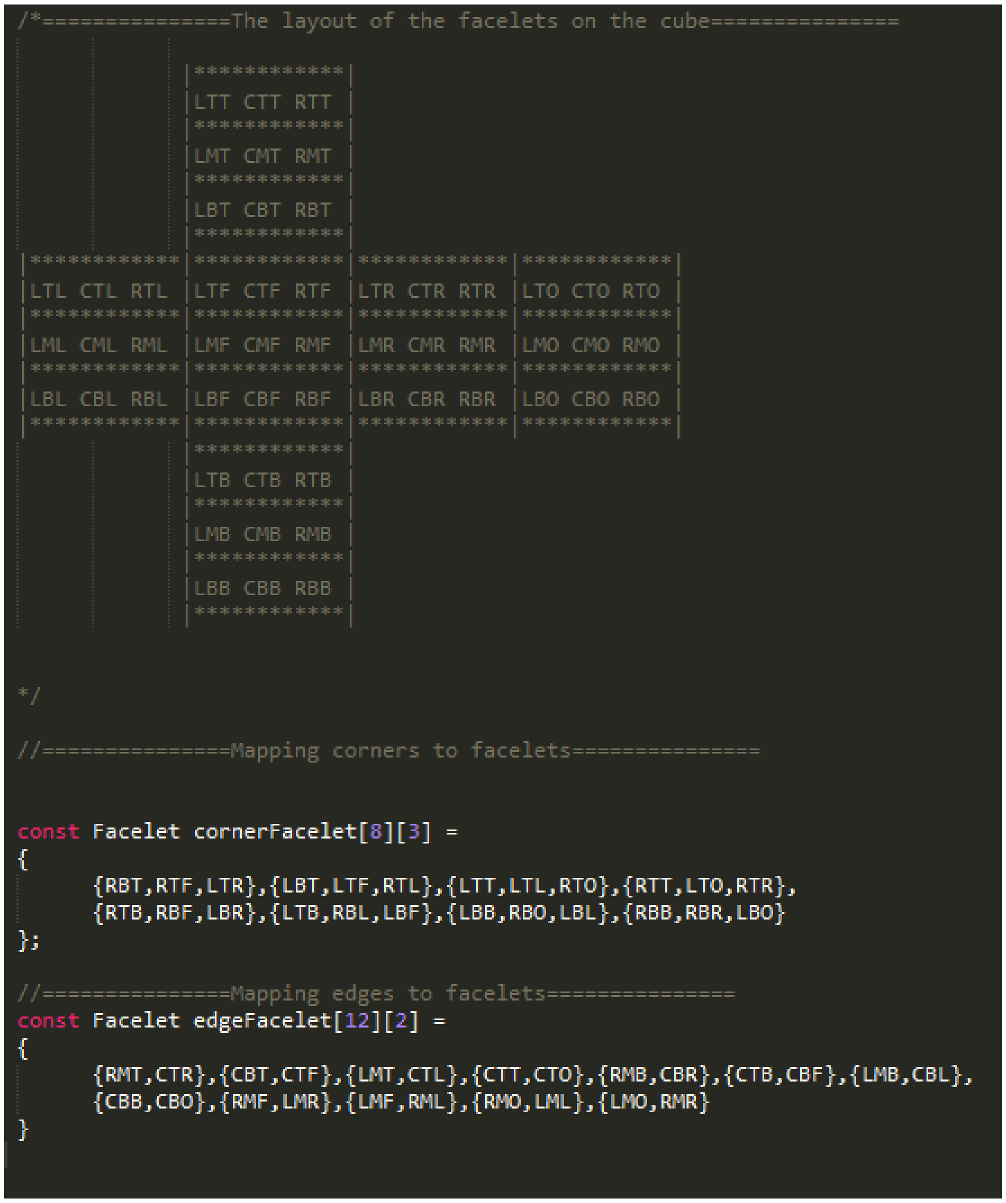
Camera Intake The first part of the software design is to determine what color cube goes where in each of the six 3x3 matrices (faces) of the cube. Each color of the cube will have already been determined by the Pixy cam and read in as a two dimensional array of integers. Each of these array of integers will correlate with a specific color. Handling the outside environment such as light must be handled by the code as well. Light is dynamic in that it differs by location so we have an algorithm that will catch when there is an outliers will help to reduce mistakes. The main mistake this algorithm prevents is assigning the wrong color to a position in the matrix.

While designing our project the cameras function changed much over the time frame of start to finish. Our final designed implemented the camera connected to the PCB through UART and the information it took in had to be sent to the PC. The camera produces 50 frames per second and with this information we can decipher the color and position of every cube in the frame. As the code runs we look at a single frame to determine the location and color at random. The frame includes a total of 6 blocks to represent 6 different cubes and 13 bytes of information for each byte. We look at 5 different bytes from each block. The first byte shows the color of that block. the next 2 bytes are the X center located in the frame and the following 2 bytes show the Y center located at the frame. We compare the X centers and Y centers to each other block in the frame and find the correct orientation for the colors in the frame. After all of the information is found for the frame in view, the robot rotates the cube and a new frame's input is taken in from the camera until the whole cube is imaged. The information found to image the whole cube is than put into six different 3 x 3 matrices depending on which face the colors lie.

Algorithm We used Kociemba's two-phase algorithm and all rights go to Kociemba for his two-phase rubik's cube solving algorithm. Another part is actually implementing the algorithm after all of the Rubik's Cube matrices are filled with the correlating colors. This process is the basis for solving the cube for its correct orientation. It takes in the matrix found from the input string on the GUI that maps the cube very a specific orientation. In the two-phase algorithm by Herbert Kociemba once a solution is made another solution is found as to find an optimum solution and not just any single solution.

Before any robotic vision can be incorporated the Rubik's Cube must be properly mapped. Once the cube has been mapped then the algorithm can be implemented. In Figure 63, the eight corners and twelve edges of the cube are mapped appropriately according to our naming convention in section 3.2.

Scrambling the cube With the design of our robot having 6 separate arms attached to the cube we have determined that it wouldn't be completely safe to constantly remove the cube from the robot and place it back in. Our design should be applicable of doing this process for the worst case scenario and we need to the cube out, but we don't prefer doing this process. Also we don't want to have to remove the cube every time its solved it would get redundant and tedious. To solve this issue we will be designing a random scrambling process that will use a random number generator to determine which face needs to be turned and which direction to turn it. The design of the program will generate a number 1 to 1000 and the number will determine which face will be turned by the following list: After the number above is found we will run another random number generator to determine how much that face will be rotated. We have determined this number can be 1 to 99 because their are only 3 possible outcomes that can happen. The first one is a 90 degree rotation to the right. The second would be a 90 degree rotation to the left. Last there can be a 180 degree rotation either way because no matter which way the the cube is turned 180 degrees



```
/*===============The layout of the facelets on the cube===============

                |************|
                |LTT CTT RTT |
                |************|
                |LMT CMT RMT |
                |************|
                |LBT CBT RBT |
                |************|
|************|************|************|************|
|LTL CTL RTL |LTF CTF RTF |LTR CTR RTR |LTO CTO RTO |
|************|************|************|************|
|LML CML RML |LMF CMF RMF |LMR CMR RMR |LMO CMO RMO |
|************|************|************|************|
|LBL CBL RBL |LBF CBF RBF |LBR CBR RBR |LBO CBO RBO |
|************|************|************|************|
                |************|
                |LTB CTB RTB |
                |************|
                |LMB CMB RMB |
                |************|
                |LBB CBB RBB |
                |************|


*/

//===============Mapping corners to facelets===============


const Facelet cornerFacelet[8][3] =
{
      {RBT,RTF,LTR},{LBT,LTF,RTL},{LTT,LTL,RTO},{RTT,LTO,RTR},
      {RTB,RBF,LBR},{LTB,RBL,LBF},{LBB,RBO,LBL},{RBB,RBR,LBO}
};

//===============Mapping edges to facelets===============
const Facelet edgeFacelet[12][2] =
{
      {RMT,CTR},{CBT,CTF},{LMT,CTL},{CTT,CTO},{RMB,CBR},{CTB,CBF},{LMB,CBL},
      {CBB,CBO},{RMF,LMR},{LMF,RML},{RMO,LML},{LMO,RMR}
}
```

Figure 63: Mapping of the Corner and Edges

the same outcome will result. We decided not to incorporate 270 degree turn because that is the same as turning the cube 90 degrees in the opposite direction. We have determined that if you use 40 moves to randomize how the cube is oriented before the start, would be enough to make sure the cube is not in a layout to easy for the robot to handle. We also are going to delete these moves after they happen therefore making sure that the robot doesn't just reverse the move set to mix up the cube. If both of these of sets do seem to come in to question the integrity of if the robot is actually solving the cube we will show that the robot can solve a cube that it doesn't mix up.

Face	Beginning Number	Ending Number
Front Face	1	166
Top Face	167	333
Bottom Face	334	500
Right Face	501	666
Left Face	667	833
Back Face	834	1000

Table 18: Number Value for each face

Degrees	Direction	Beginning Number	Ending Number
90 degrees	Right	1	33
90 degrees	Left	34	66
180 degrees	Either	67	99

Table 19: Number Value for rotation

Software to Hardware Our last function for our software will be our software to hardware communication. This is needed throughout the process of solving the cube. At the beginning we are going to need the communication between the camera and board. This process will help build our matrix and will lead to solving the cube. The camera will also need the cube turned during the image processing therefore we will need the arms to move during this whole process. After the algorithm is solved we will need the robot to turn the cube in the direction made by the algorithm so this in another portion that needs the communication between the software and hardware. If for some reason the software doesn't communicate with the hardware our robot can not solve the cube and our project will be a failure.

Our final design had to incorporate only had to incorporate a few functions between the hardware and software. Our PIXY CM5 cam connected to our PCB through UART. To get information from our camera we needed to send a '!' command to our PCB to inform it that it was time to receive the frame from the camera. Next we take that frame from the PCB and use it on our PC. We use this information to find the movements needed to solve the cube and send it back to the PCB. Before we send this information back to the PCB we have to send a command of 2 that shows the PCB we will be making the cube steppers rotate. Next our PCB reads in a string of movements sent by the PCB to solve the cube. It rotates each face based on the letter that represents its face. When an apostrophe (') command comes in it takes the face that has been recently rotated and rotates it the opposite direction.

4.4.1 High-Level Software System Architecture

The design for our high-level software architecture that we felt was best to describe our system is a component system with event driven processes. We decided it would be a component based system because each component is necessary to work with each other. One component is used to drive to the next component but each component can work on its own, they just need each other to solve the goal of the project. Next we also believed that the system was

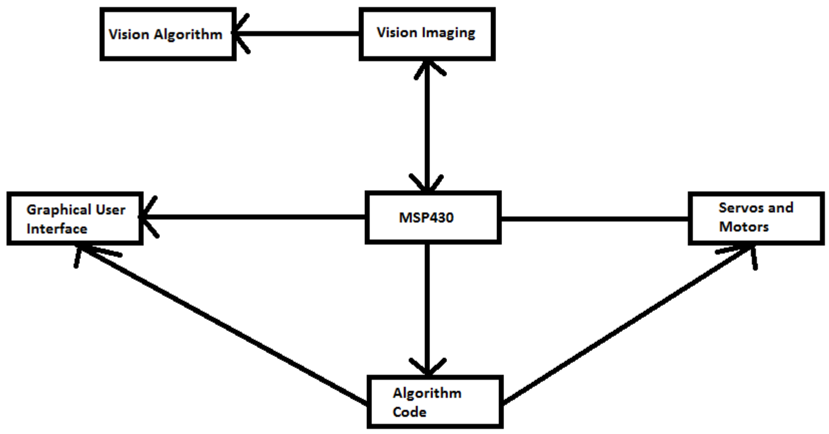


Figure 64: The design for our High-level Software Architecture

also event driven process. It happens to be event driven because each task is reliant on its previous task. We can prove this by showing that the arms can't rotate to solve the cube without the algorithm, and the algorithm for solving the cube can't be done until the matrix is made and the matrix can't be made without the camera intaking the images it can process. Therefore we have determined that our system is a component based system with even driven processes.

Our final designs high-level architecture has not changed much from the beginning. Every part of our system connects to our MSP430 and sends information back and forth between it. The main difference is that our algorithm code runs off our GUI interface and that shows that our GUI is the main part for solving the code. All of our information relays through our MSP making it the most essential part of our software architecture

4.4.2 Software Integration

For the design of our project we have two different options on how to run our project. We have determined that we could build a program on a computer and run the application from our computer to our embedded computer. This way of integration would be the most efficient way and easy to see implement. The issue with this plan is it would limit our capabilities with the GUI to just using our IO pins. Our other way of integrating our code would by putting the application on the MSP430. This way would be an ideal design because we wouldn't have to run any applications through our computer. We also would have an open spot to run a graphical display. This gives us more opportunity to allow our project to smaller and easily transportable. Our final design decision is not fully determined but one of these plans will be used in the final design.

Our final design for our project implemented running our algorithm and GUI code through the computer and sending instructions to our PCB to solve the cube. Our software starts by sending an instruction to our PCB to intake a single frame from our camera. We take this single frame and send it to the computer where we decipher the bytes received to find the orientation of the cube. While the orientation of the cube is looked for we update our GUI to show the orientation of the cube on a 2D display. After the orientation is found we take the layout of the cube and run it through our algorithm code to find the set of moves needed to solve the cube. The moves are displayed on the screen and sent to our PCB where it is parsed through by our PCB. The parsed instructions tell which stepper to move and which direction. The moves are done one at a time till all the moves are done.

4.4.3 Algorithm Implementation

The design we decided to take for our algorithm Implementation is based on how the camera takes in the images of the cubes. The Pixy cam is to designed to work as the purple dinosaur method. This basically means it intakes 7 significant colors and determines them as keys. We will be able to take in the 6 different colors on each side of the cube include green, white , orange, blue, red, and yellow. Given this information we will place the color locations into a matrix. This matrix should be 6 different 3 by 3 matrices. After the matrices are made our algorithm will be put into place. It will take the results of the matrices and determine the fastest and most efficient way to solve the cube. It will keep track of the amount of moves it must take to solve the cube and what moves need to be taken to solve the cube. After the algorithm is ran and the cube is determined able to be solved the directions to solve the cube shall be sent to the servos. The servos then will work together to solve the cube. Once the values have been retrieved from the Pixy camera the code will map the necessary values to the six matrices. Then the algorithm will be implemented from here.

Robot Vision The design we decided to take for our algorithm Implementation is based on how the camera takes in the images of the cubes. The Pixy cam is to designed to work as the purple dinosaur method. This basically means it intakes 7 significant colors and determines them as keys. We will be able to take in the 6 different colors on each side of the cube include green, white , orange, blue, red, and yellow. Given this information we will place the color locations into a matrix. This matrix should be 6 different 3 by 3 matrices. After the matrices are made our algorithm will be put into place.

Our final design based the camera on our robot using its UART connection to communicate with our PCB. The robot was set up to look at 6 blocks of the cube at one time. The bottom 3 of the top face and the top 3 of the right face. We take the information from the frame showing the 6 blocks and send it to the PC. The information sent shows each block color, its X position, and Y position. We look at each blocks X position and compare it to the other blocks to find the two left most blocks. After we find the two leftmost blocks we compare to find which block is higher and which block is lower. After all the positions are found for each cube we place the signature colors in a string based on their placement using the X and Y position. We then send this string to our algorithm to make 6 different 3 x 3 matrices to show off the total orientation of the cube.

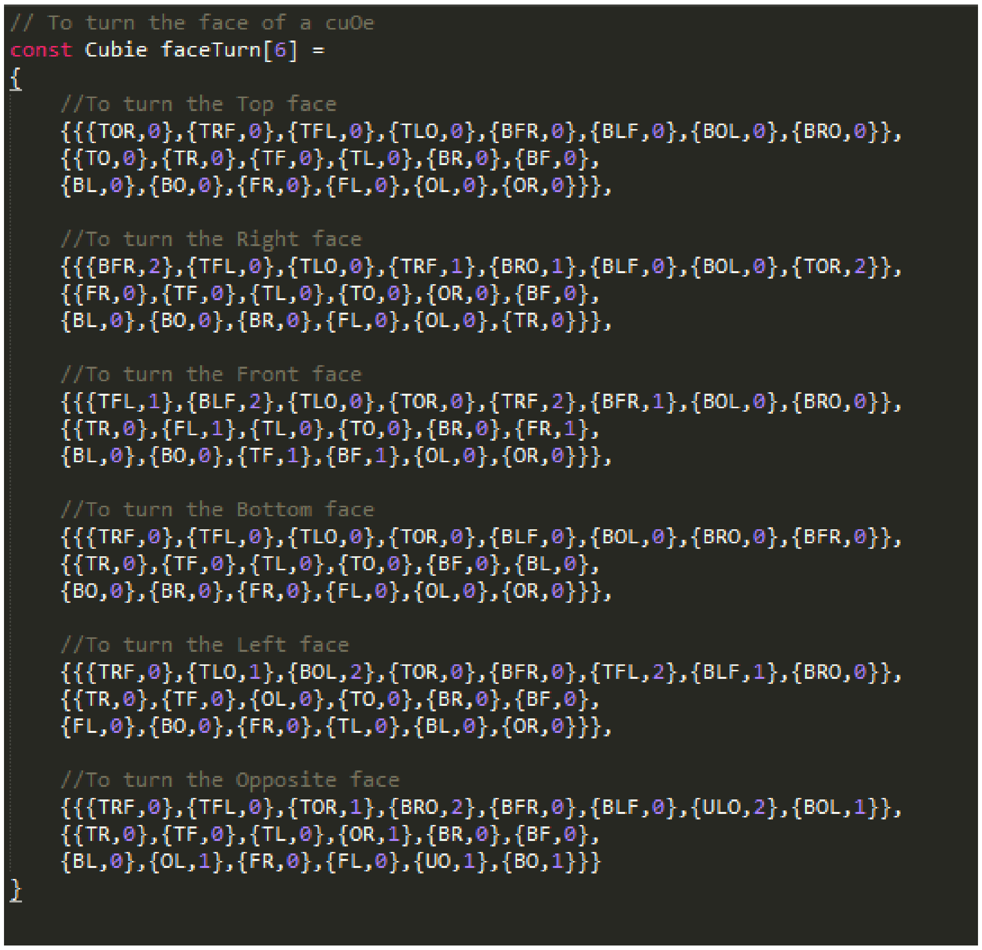
Kociemba's Algorithm Kociemba's Algorithm begins from coordinating the cube. Which is the initial mapping of the matrices. Then symmetries of the cube are identified. Once the symmetries of the cube have been successfully identified then the pruning of the cube comes into play. In the pruning process a table is made which will check for parities within the cube. It will also take in multiple possible solutions in order to improve upon efficiency. After the cube has been properly pruned them the optimum solution is found. Also, there will be an initial declaring of functions to be called by all of the C programs. This declaration of function will allow for the identity of the cube to be easier identified. The algorithm should then have successfully solved the Rubik's Cube.

All rights are given to Herbert Kociemba for his two-phase rubik's cube solving algorithm. An open-source java package is on his website that was utilized in our hardware intensive project in which we modified for our specific needs [12]. It begins from coordinating the cube which is the initial mapping of the matrices. Then there are two main groups that must be found being G1 and G2 to generate the solution. Both groups look for solutions while not looking for solutions that either group has previously generated. In the G1 group it generates millions of tables to find solutions and can be reduced greatly due to symmetries that can be found in this state. The G2 group only has a certain set of moves that it can use <U,D,R,R,L,L,F,F,B,B>. Both groups generated millions of lookup tables that use a lot of RAM and are slow. The speed for this algorithm comes from the pruning process. In the pruning process a table is made and it checks for parities within the cube. It also takes in multiple possible solutions in order to improve upon efficiency. After the cube has been properly pruned then the optimum solution can be found. However, speed was also important to us therefore we do not look for the most optimum solution with 20 moves, but with 24 instead. The algorithm should then have successfully solved the Rubik's Cube.

CFOP The CFOP algorithm is less efficient when done by a computer, but easier to implement. However, the same initial mapping of the cube must be properly done by the robotic vision just as with the Kociemba algorithm. To begin an initial Mapping of LMF, CMF, RMF, CTF, CBF must be found. The edgeFacelet function will have need to have those cube in there correct position according to our mapping. Multiple scans of the camera may be necessary to get to this Base State of the algorithm. We will always solve for the Green color first. We chose Green because it was the one we began learning with when we doing our research on this method and decided to simply stay consistent. Also, solving for the same color first will produce less error of getting the cube to the Base State to actually start the algorithm.

Once the Base State is produced then the cornerFacelet function for the Front face, being the Green side, must be solved for. The values that must be found and properly mapped to the Front Face are LTF, RTF, LBF, RBF. Next, the second layer is taken into account to be solved.

The second layer, whcih is the Middle layer, will only have four cubes to be solved for at most because there are only eight cubes that make up the Middle layer and only half of them will be utilized. There will be a function to find the following values [RTL,LBT],



```
// To turn the face of a cuOe
const Cubie faceTurn[6] =
{
    //To turn the Top face
    {{{TOR,0},{TRF,0},{TFL,0},{TLO,0},{BFR,0},{BLF,0},{BOL,0},{BRO,0}},
    {{TO,0},{TR,0},{TF,0},{TL,0},{BR,0},{BF,0},
    {BL,0},{BO,0},{FR,0},{FL,0},{OL,0},{OR,0}}},

    //To turn the Right face
    {{{BFR,2},{TFL,0},{TLO,0},{TRF,1},{BRO,1},{BLF,0},{BOL,0},{TOR,2}},
    {{FR,0},{TF,0},{TL,0},{TO,0},{OR,0},{BF,0},
    {BL,0},{BO,0},{BR,0},{FL,0},{OL,0},{TR,0}}},

    //To turn the Front face
    {{{TFL,1},{BLF,2},{TLO,0},{TOR,0},{TRF,2},{BFR,1},{BOL,0},{BRO,0}},
    {{TR,0},{FL,1},{TL,0},{TO,0},{BR,0},{FR,1},
    {BL,0},{BO,0},{TF,1},{BF,1},{OL,0},{OR,0}}},

    //To turn the Bottom face
    {{{TRF,0},{TFL,0},{TLO,0},{TOR,0},{BLF,0},{BOL,0},{BRO,0},{BFR,0}},
    {{TR,0},{TF,0},{TL,0},{TO,0},{BF,0},{BL,0},
    {BO,0},{BR,0},{FR,0},{FL,0},{OL,0},{OR,0}}},

    //To turn the Left face
    {{{TRF,0},{TLO,1},{BOL,2},{TOR,0},{BFR,0},{TFL,2},{BLF,1},{BRO,0}},
    {{TR,0},{TF,0},{OL,0},{TO,0},{BR,0},{BF,0},
    {FL,0},{BO,0},{FR,0},{TL,0},{BL,0},{OR,0}}},

    //To turn the Opposite face
    {{{TRF,0},{TFL,0},{TOR,1},{BRO,2},{BFR,0},{BLF,0},{ULO,2},{BOL,1}},
    {{TR,0},{TF,0},{TL,0},{OR,1},{BR,0},{BF,0},
    {BL,0},{OL,1},{FR,0},{FL,0},{UO,1},{BO,1}}}
}
```

Figure 65: The C code of what all cubies will be affected by a Face turn

[RMT,CTR], [CBR,RMB], [CBL,LMB]. Now the Middle layer should be solved and we will proceed to move on to the Top layer.

The third layer, which is the Top layer, will look for the top facelets to be in the following states: only facelet CMT in the correct position, only facelets CMT, CTT, LMT in the correct position, facelets LMT, CMT, and RMT in the correct positions, and the goal of CMT, LMT, RMT, CTT, and CBT in the correct positions. In the case of only facelet CMT being in the correct position then the matrix will be manipulated so that it gets to only CMT, CTT, LMT in the correct places. Which will lead to LMT, CMT, RMT in the correct positions. Which will finally lead to a cross of CMT, LMT, RMT, CTT, and CBT in the correct positions. Finally the corners of the matrix will be manipulated until the cube gets into a solved state.

4.4.4 Detailed Design

Our full design method of the software portion is a very long list of task after task. The task should be taken in order and the next task shall only be completed after the previous task is

Face	Rotation	Signature Number	Rotation Number
Blue	90 degrees, right	6	1
Red	90 degrees, left	3	2
Blue	90 degrees, left	6	2
White	180 degrees	2	3
Yellow	90 degrees, right	1	1
Green	180 degrees	4	3
Red	90 degrees, right	3	1
Orange	90 degrees, right	5	1

Table 20: Example of our solution array compared to move that needs to be made

full accomplished. The program starts by entering its random scramble phase. This phase will generate a mixture of spins and turns to all sides of the cube that will in the end mix up the cube to a scrambled the orientations of the colors. After this program is ran we then can start the visual portion of solving the cube. The robot will run the purple dinosaur algorithm designed by the developers of the Pixy cam and the visualization algorithm designed by our group. With the purple dinosaur algorithm we will intake bits that show the location of significant colors and their width and height. The width and height functions will be debugged to determine if more than one of the same color are next to each other. During the process of vision processing is done the robot will be filling up 6 different matrices each one coordinating to a different face of the cube. These matrices will then be added one to each other to determine the total layout of the cube.

After the process of placing the colors into a matrices is completed, the algorithm portion of our code will begin. The program will run a single algorithm at a time to solve the cube for its correct orientation. This portion may result in more than one algorithm that can solve the cube. If this issue is reached than the program will pick which ever algorithm takes the least amount of moves. The algorithms moves will be saved into a 2 D array where it will determine what face needs to be turned by using the color signature as a description of the face and a number between 1-3 to show how that rotation is made. We have determined that 1 would correspond to a 90 degree rotation to the right, a 2 would result in a 90 degree rotation to the left, and last a 3 would be 180 degree rotation. The signature numbers will be set according to how the colors are set into our purple dinosaur algorithm but for example we could determine yellow to be 1 if its the front face of the cube. The reason why we can do this is because we don't expect the center of the cubes to be moved out of the positions that they start at because our cube rotates sides from that position and the cube doesn't ever actual rotate. The table below shows how the 2 D array for rotating sides will be laid out: The table above is just an example of what will be done when the solution to solve the cube is being found. The reason why in our design we only have 3 different moves to rotate the cube is because those 3 moves can do all moves to one face. For example rotating the face of the cube 270 degrees to the right is the same as rotating the face 90 degrees the opposite direction. Also rotating 180 degrees to the right is determined the same as rotating 180 degrees to the left.

After this array is made it will be sent back to the MSP430 for the execution by the robot's arms. Each instruction will be sent to the corresponding servo that fits with the significant number at the center of each side. The next part of our code will just be the rotation of each side till all instructions found during our algorithm phase and met. The instructions will go in one at a time therefore not allowing any issues of rotating adjacent sides either breaking an arm of the robot or breaking a side of the cube.

After all instructions are met we will determine if the robot has solved the cube to its 100 % correctness or if the programs routine needs to be ran again for an issue of the cube not being 100 % correct.

The last part of our detailed design of our software we are going to integrate a graphical display with our robot. This graphical display will have functions to allow the users to feel more involved with the robots actions. If everything goes as plan our graphical display shall show a 2 D display of the cube, a move list showing off which moves are being taken by the robot to solve the cube, a move counter to show how many moves are needed to solve the cube to 100 % correctness, and lastly a timer to show how long it takes the robot to solve the cube. We have discussed among each other that it may slow down the process of solving the cube if it takes time to show how its solving the cube. Therefore we are thinking about designing two different modes of solving. Both design modes are exactly the same to solve the cube the difference is how long it takes to solve it. If we slow down how fast the instructions are fed into the servos we can show the user exactly what moves are taken place. This function may be useful for people to watch our robot function at a much slower pace keeping track of the moves and helping them determine how to solve the cube. The other design is to not worry about the showing of the moves list so the robot doesn't need to take its time to solve the cube. This would increase the time it takes to solve the cube and make it easier for the robot to function so this will be our first design direction.

For our final design, we utilized Kociemba's algorithm on a PC application. All rights to Kociemba for his two-phase rubik's cube solving algorithm. The application prompts you to type in what port you want to use on each initial startup. It has a GUI that can input the colors of the cube white(W), red(R), green(G), yellow(Y), orange(O), blue(B) by inputting the first letter of the specified color. The face must be inputted in the order W,R,G,Y,O,B and that sets the cube once the 'Set' button is pressed. Then once the 'Solve' button is pressed the algorithm takes in that string letters and runs Kociemba's algorithm on it to generate a solution string. This solution string of letters is converted to bytes and sent over via serial communication to the robot. The robot receives that byte information and translate it into the appropriate ASCII letter to turn the specified motor for that letter. The robot is programmed such that each motor turns based on what letter is sent to it and in the specific order the letter is sent to it. The letters that are set to each of the six motors are U/W, R/R, F/G, D/Y, L/O, B/B.

The GUI also has a scan mode that works by sending a start signal to the MSP430 which is an '!'. This prompts the MSP430 to tell the Pixy camera to begin scanning the current edge. The current edge is scanned and sent back to the application over serial communication.

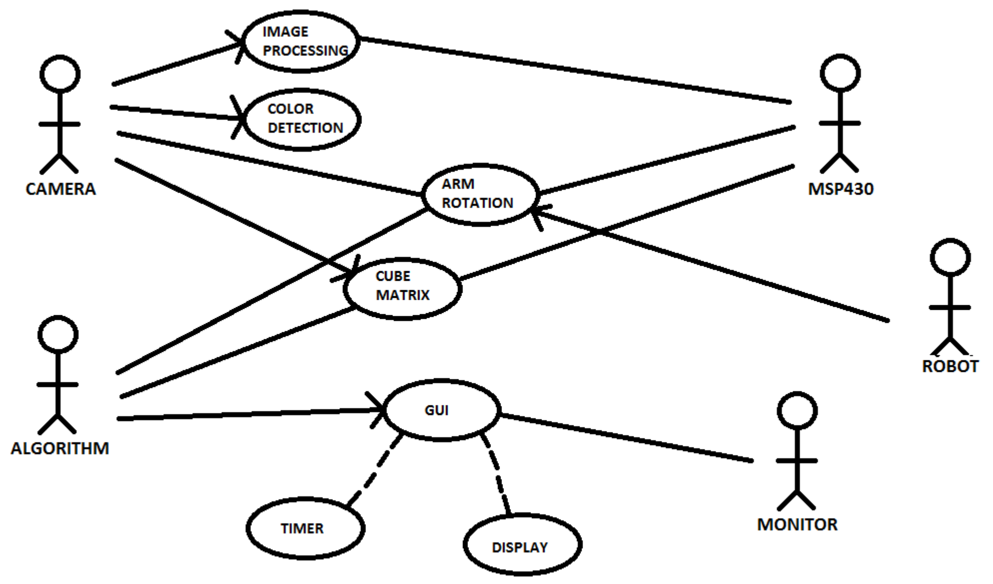


Figure 66: Usecase diagram made to describe the vision concept of our project

The PC application gets these values and stores them then waits for two seconds. After the wait is completed it sends over a string to turn the motors to a different edge. That edge is saved and sent back over again and that process is repeated until each of the twelve edges are properly stored so that the rubik's cube is properly mapped. A button is pressed on the application to solve that specific mapping of the cube. Then the solution string is generated and sent over via serial communication to turn each specified motor in the string.

Usecase Diagrams The Usecase diagram shown above describes how we all of our parts integrate with each other making the basis for our design the MSP430. There are 5 different sources in this Usecase diagram. The five different sources are Camera, MSP430, Algorithm, Robot, and Monitor. We decided that this design covered all the aspects of our vision controls and how it integrated with the other aspects of our project. As you can see the Camera is used for the image processing, and designing the cube matrix. It is also used for color detection in our implementation of the matrix. Lastly it needs the robots arm rotation to finish its programing. The MSP430 is used to take in the image processing and help design the cube matrix. It also sends power and processing speed to our robot to allow arm rotations. Our algorithm is used to solve our cube so it intakes the cube matrix and is used to determine what arm rotations are necessary for our robot. Lastly we have our monitors as a source. This basically doesn't output to much material besides our GUI who has two representing factors which are the timer and the display of the cube.

Class Diagrams The class diagram above shows the communication between each program side of our system. The system starts off with vision processing with the Pixy cam. The pixy cam takes it raw data and sends it to the MSP430 to place in a matrix that can be solved later. The Pixy cam also needs to communicate with the servos so that it can visualize the

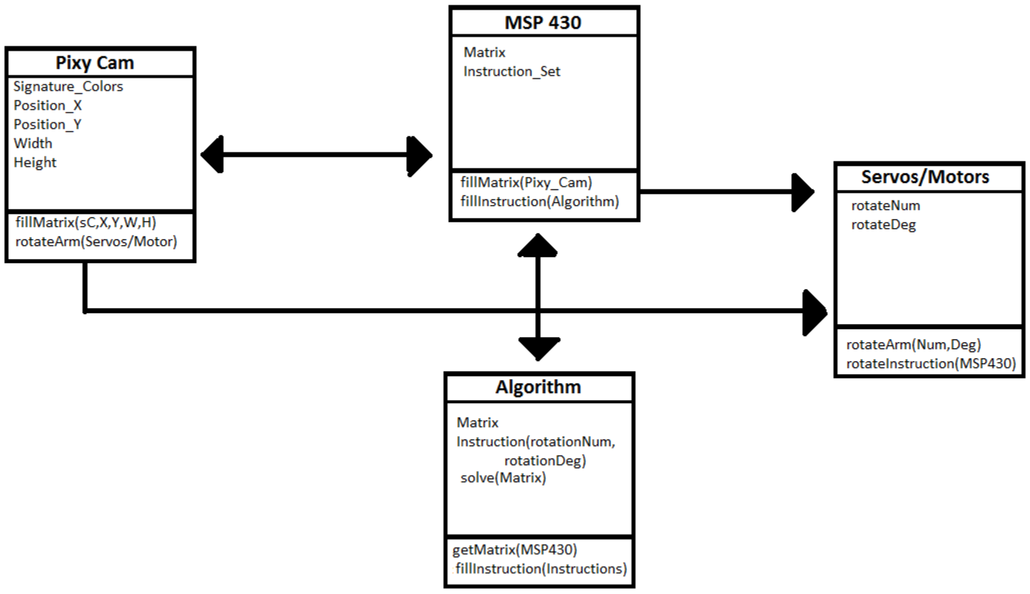


Figure 67: Simple Class diagram that describes our system

whole cube. The servos are fed an algorithm during the visualization phase because we plan to process the cube the same way ever time. After the full matrix is sent to the MSP430 from the camera the matrix is sent to the algorithm process of our code. The algorithm process takes the matrix and solves it to find instructions that will allow our cube to be placed in the correct orientation with the same colors. This instruction set is then sent back to MSP430 to process for the servos/motors. The servos intake the instructions one step at a time and rotate the corresponding arm to the corresponding angle.

4.4.5 MSP430 Programming

As discussed in previous sections, the MSP430 programming enables serial communication between the embedded system and the PC application. The MSP430 code is based on example code developed by TI with additions for handling and parsing through input strings for relevant data. Further additions are made to control the stepper motor drivers such as simple PWM generated signals tuned to a specific frequency. Two versions of MSP430 code are listed in Appendix E in Section 9.5. One version contains both stepper driving ability and Pixy data retrieval for use over a 57600 baud UART connection with the PC. The other version of the MSP430 code contains just the stepper driving functionality but enables the USB port on the board so that it can connect to a PC as a CDC.

5 Schedule

Table 21 lists some of the primary design and testing tasks for the project, as well as the prescribed time frames to accomplish them.

Category	Task	Testing Time
Robot Structure	Design Structure	2 weeks
	Prototype Motor Driver Circuit	1 week
	Prototype Motor Driver PWM and control code	1 week
	Design Structure in CAD	2 weeks
	3D Print Structure	1 week
	Assemble and Test Structure	1 week
GUI	Design and Compile GUI	3 weeks
	Rubiks Cube Algorithm	2 weeks
	Develop and Test Serial Communication Output	2 week
	Develop Image processing data I/O and connectivity	2 weeks
Image Processing	Program and Configure Pixy	3 weeks
	Test Serial Communication Output	2 week
	Develop data parser and visualization matrix	2 weeks

Table 21: General Project Development Schedule

6 Prototype Testing

6.1 Hardware Testing

Table 22 lists the essential hardware prototype tests that will evaluate and ensure proper design functionality. The first tasks to be run for hardware prototyping include assembly and configuration of the stepper motor driver circuit. This testing should conclude that the DRV8825 Stepper Driver IC is adequate to power and control the stepper motors used in this project. The next testing should evaluate the control interface for the DRV8825 and establish a frequency bandwidth over which the step input is valid for the chip. The MSP430 should be able to supply a frequency within this range, otherwise the DRV8825 will not function in the embedded system.

The last testing needed to validate the motor control functionality is to ensure that the output signals of the MSP430 are capable of driving the inputs of all six DRV8825's simultaneously. The problem that could occur is that the MSP430 output pins may not have adequate drive strength to overcome the combined capacitive load of the six DRV8825 input gates. In such a scenario, the gate voltage on the DRV8825 input pins would not reach logic high before the next clock edge in the MSP430. With the relatively low step signal frequencies, this should not be an issue.

Category	Component	Tests
Electronics	Motor Control	Assemble prototype circuit with stepper motor
		Test valid frequency range for pulsed step input
		Test maximum motor coil current output
		Test frequency of MSP430 step output
		Test MSP430 control signal applied to 6 stepper driver IC's to ensure adequate pin drive strength
	MSP430	Development and compilation of stepper driver control signals
		Serial input and output to console
		Serial input to MSP430 to control stepper motor drivers
	Camera	Test Pixy color recognition for all cube colors
		Analyze serial output from Pixy for useful data
Structure	Motors	Test Motor rotation in all directions on all sides of cube
	Structure	Test fitment of motors to structure
		Test viewing angle of Pixy
		Ensure Cube can be removed easily from structure

Table 22: Hardware Prototype Testing Plan

6.2 Software Testing

Robot Vision Test The software has to be tested in multiple steps. The first step will be to test that the robotic vision is being correctly handled. The software has to correctly interpret the values identified by the pixy cam for each side of the cube. An initial test of identifying the cube color values of a solved cube will be done initially. A solved cube should be much easier to detect and result in the least amount of error due to the solid color on each side. When tested the each integer value passed by the pixy camera to the code must be correctly mapped. After it has passed an initial solved cube test it should be tried in different lighting conditions. This will help determine the best and worse lighting conditions that the robot vision done by the pixy camera can handle.

Next, the more difficult test should be done being that of an unsolved cube. The varying color qualities of each cube will make error more likely. However, this error has to be held to a great minimum or the code with incorrectly interpret the cube resulting in a failed attempt of solving it. After, the cube has been identified and mapped to each matrix properly in a unsolved state it will now be tested against lighting conditions again. Light can cause color values to spike so making sure the code handles this conditions well will increase the success rate of it.

Algorithm Test Once the robot vision done by the Pixy camera has been properly identified and mapped the code must then manipulate the values to find a solved state. The solving algorithm will be tested to see the percentage of times it can solve the Rubik's Cube succesfully. The algorithm itself should have the least percentage of error because the pixy camera has moving parts as well as the hardware has moving parts.

Hardware Communication Test Hardware communication test is essential because if the code cannot tell the robot to move properly then it will result in the Rubik's Cube not being solved. The fist hardware communication test would be making sure the pixy camera can be properly rotated to analyze the entire cube. Making sure the pixy camera moves in all directions desired to capture the cube is essential.

The next communication test will be to make sure that the code can turn each of the six arms of the robot. The code should be able to turn each arm 90 degrees, 180 degrees, 270 degrees, and 360 degrees in both clockwise and counterclockwise motions. It is essential for each arm to move according to specification for the Rubik's Cube to successfully be solved.

Finally, the timing of the arms to turn after the code tells it to is incredibly important. The timing of each arm should be measured and accounted for when the code changes algorithms into arms movements. The arm movement must be in sync or an arm could get jammed trying to go at the same time as another arm causing an error in solving the cube.

6.2.1 Environment

Operating System We will be ensuring that the Linux operating system boots up and is running. Once it is booted up the Integrated Development Environment (IDE) being CodeBlocks will be launched to make sure it has no errors.

6.2.2 Device Integration and Connectivity

- Test the connection with UART and the MSP430
- Test the connection with UART and the Pixy Cam
- Test the connection with the MSP430 and the Pixy Cam
- Test the connection with SPI and the MSP430
- Test the connection with SPI and the monitor
- Test the connection with the MSP430 and the monitor
- Test the connection with USB and the MSP430
- Test the connection with USB and a computer
- Test the connection with MSP430 and a computer
- Test the connection with I/O ports and the MSP430
- Test the connection with I/O ports and servos
- Test the connection with MSP430 and servos

6.2.3 Algorithms

- Test if the algorithm can solve an easily messed up cube
- Test if the algorithm can solve solve a severely messed up cube
- Test if the algorithm can solve the cube with 1 method
- Test if the algorithm can solve the cube with multiple methods
- Test if the algorithm can pick which method has least amount of moves
- Test if the algorithm can save instructions into a list
- Test if the algorithm can send instructions to MSP430
- Test if the algorithm's instructions are easily debugged
- Test if the algorithm can parse through matrix

6.2.4 Software Integration Testing

- Test if instruction set can be worked into servos
- Test if camera's code can be taken into the MSP430
- Test if MSP430 can push the matrix to the algorithm
- Test if the algorithm can be ran on the MSP430
- Test if the GUI is operational on the display
- Test if the the ports push and pull data

7 Standards

In this section we take a look at some of the many electrical standards that have been created and specifically how they affect the design constraints and requirements specifications of our product.

Standard	Summary
IEC60027	Part 1: Letters symbols to be used in electrical engineering
IEC60038	IEC standard voltages
IEC60050	International electro-technical vocabulary
IEC60059	Standard current ratings
IEC60071	Insulation coordination
IEC60072	Dimensions and output ratings
IEC60073	Indicating lamps
IEC60076	Power transformers
IEC60083	Plugs and socket-outlets for domestic and similar general use
IEC60088	Standard related current (2 to 63 A) of fuse links for low voltage
IEC60112	Methods for determining the comparative and the proof-tracking indices of solid insulating material under moist conditions
IEC60113	Diagrams, charts, and tables
IEC60146	Semiconductor converts
IEC60157	Low voltage switchgear and controlgear
IEC60185	Current transformers
IEC60186	Voltage transformers
IEC60189	Low-frequency cables and wire with PVC insulation and PVC sheath
IEC60228	Conductors for insulated cables
IEC60255/ BS142	Electrical protection relays
IEC60269	Low-voltage fuses
IEC60270	Partial discharge measurements
IEC60337	Control auxiliary switches, relays, and push buttons
IEC60478	Stabilized power supplies, DC output
IEC60479	Effects of current on human beings and livestock
IEC60529	Classification of degrees of protection provided by enclosures
IEC60536	Part 1: Classification of electrical and electronic equipment with regards to protection against electric shock. Part 2: Guideline to requirements for protection against electric shock
IEC60606	Application guide for power transformers
IEC60617	Graphic symbols for diagrams
IEC60688	Electrical measuring transducers for converting AC electrical quantities into DC electrical quantities

Standard	Summary
IEC60896	Stationary lead-acid batteries. General requirements and methods of test
IEC60906	IEC systems of plugs and socket-outlets for household and similar purposes
EN55022	Limits and methods of measurement of radio interference characteristics of information technology equipment
IPC-2615	Printed circuit board dimensions and tolerances
IPC-D-325	Documentation requirements for printed circuit boards
IPC-ET-652	Guidelines and requirements for electrical testing of unpopulated printed circuit boards
IPC-2612	Sectional requirements for electronic diagramming documentation (schematic and logic descriptions)
IPC-2221	Generic standard on printed board design
IPC-2223	Sectional design standard for flexible printed circuit boards
IPC-FC-234	Pressure sensitive adhesives assembly guidelines for single-sided and double-sided flexible printed circuits
IPC-4101	Laminate prepreg materials standard for printed circuit boards
IPC-4202	Flexible base dielectrics for use in flexible printed circuitry
IPC-4203	Adhesive coated dielectric films for use as cover sheets for flexible printed circuitry and flexible adhesive bonding films
IPC-4204	Flexible metal-clad dielectrics for use in fabrication of flexible printed circuitry
IPC-A-600	Acceptability of printed circuit boards
IPC-A-610	Acceptability of electronic assemblies
IPC-6011	Generic performance specification for printed boards
IPC-6013	Specification for printed wiring, flexible and rigid-flex
IPC-6202 / PAS-62123	Performance Guide Manual for single and double sided flexible printed wiring boards
IPC-TF-870	Qualification and performance of polymer thick film printed boards
ANSI/ISEA 105-2011	Hand protection selection criteria
ANSI/ASSSE A10.1-2011	Pre-Project and Pre-Task Safety and Health Planning
USB	Universal Serial Bus standard
RS-232	Serial port communication standard
I2C	Serial port communication standard
SPI	Serial Peripheral Interface standard

8 Safety and Ethics

8.1 IEEE Code of Ethics

We will set our ethics standards to align with those of IEEE Code of Ethics.

1) To accept responsibility in making decisions consistent with the safety, health, and welfare of the public, and to disclose promptly factors that might endanger the public or the environment;

2) To avoid real or perceived conflicts of interest whenever possible, and to disclose them to affected parties when they do exist;

3) To be honest and realistic in stating claims or estimates based on available data;

4) To reject bribery in all its forms;

5) To improve the understanding of technology; its appropriate application, and potential consequences;

6) To maintain and improve our technical competence and to undertake technological tasks for others only if qualified by training or experience, or after full disclosure of pertinent limitations;

7) To seek, accept, and offer honest criticism of technical work, to acknowledge and correct errors, and to credit properly the contributions of others;

8) To treat fairly all persons and to not engage in acts of discrimination based on race, religion, gender, disability, age, national origin, sexual orientation, gender identity, or gender expression;

9) To avoid injuring others, their property, reputation, or employment by false or malicious action;

10) To assist colleagues and co-workers in their professional development and to support them in following this code of ethics.

9 Appendices

9.1 Appendix A –Abbreviations

Abbreviation	Meaning
AC	Alternating Current
DC	Direct Current
FPS	Frames Per Second
GB	Gigabyte
GPIO	General-Purpose Input/Output
Hz	Hertz
IDE	Integrate Development Environment
kHz	kiloHertz
MCU	Microcontroller
MSP430	Texas Instruments board
PCB	Printed Circuit Board
PWM	Pulse-width modulation
RAM	Random Access Memory
RTOS	Real Time Operating System
SPI	Serial Peripheral Interface
UART	Universal Asynchronous Receiver/Transmitter
USB	Universal Serial Bus
UML	Unified Modeling Language

9.2 Appenix B –References

[1] 1.8 inch/cog 128 x 64 graphic lcd. http://www.ebay.com/itm/like/291037052683?lpid=82&chn=ps&ul_noapp=true.

[2] 2.2 inch 240 x 320 spi tft lcd. www.ebay.com/itm/like/171756246358?lpid=82&chn=ps&ul_noapp=true.

[3] Analog to digital conversion. https://learn.sparkfun.com/tutorials/analog-to-digital-conversion.

[4] Angelelec diy open source led display. http://www.amazon.com/Angelelec-Compatible-Principle-Interface-Automatically/dp/B01E6XXO5E/ref=sr_1_10?ie=UTF8&qid=1461631444&sr=8-10&keywords=spi+led+display.

[5] C (programming language). https://en.wikipedia.org/wiki/C_%28programming_language%29.

[6] Cmucam5 pixy overview. http://www.cmucam.org/projects/cmucam5.

[7] Cmucam5 pixy porting. http://cmucam.org/projects/cmucam5/wiki/Porting_Guide.

[8] Cmucam5 pixy wiki. http://cmucam.org/projects/cmucam5/wiki.

[9] Cmucam5 pixyspec. http://www.cmucam.org/projects/cmucam5/wiki/Wiki?version=35.

[10] Diligent 290-006. http://www.mouser.com.

[11] Give up, humanity. this robot can solve a rubik's cube in under 1 second. http://www.digitaltrends.com/cool-tech/rubiks-cube-robot-2/.

[12] Herbert kociemba's homepage. http://kociemba.org/.

[13] Lego robot : Fastest rubik's cube solver. https://hotgears.wordpress.com/2012/11/04/lego-robot-fastest-rubiks-cube-solver/.

[14] Low-level i/o. http://www.tinyos.net/tinyos-2.x/doc/html/tep117.html.

[15] Msp430 product search. http://www.ti.com/lsds/ti/microcontrollers_16-bit_32-bit/msp/products.page.

[16] Msp430 real time operating systems overview. http://processors.wiki.ti.com/index.php/MSP430_Real_Time_Operating_Systems_Overview.

[17] Olimex. http://www.mouser.com/ProductDetail/Olimex-Ltd/SHIELD-LCD-16X2/?qs=J7x7253A5u648zrOBSewkA%3D%3D&gclid=CjwKEAjwgPe4BRCB66GG8PO69QkSJAC4EhHh7LmyrSeHEiSohfatSrrMnB3jqZFJYVnyv-zo4BhqxRoCCMPw
wcB.

[18] Opencv. https://en.wikipedia.org/wiki/OpenCV.

[19] Optimal solutions for rubik's cube. https://en.wikipedia.org/wiki/Optimal_solutions_for_Rubik%27s_Cube#Kociemba.27s_algorithm.

[20] Osh park fabrication services. http://docs.oshpark.com/services/.

[21] Parallax (futaba) continuous rotation servo. http://www.robotshop.com/en/parallax-futaba-continuous-rotation-servo.html.

[22] Pixy serial protocoll. http://cmucam.org/projects/cmucam5/wiki/Pixy_Serial_Protocol.

[23] Pixy smart vision sensor. http://www.amazon.com/Pixy-CMUcam5-Smart-Vision-Sensor/dp/B00IUYUA80.

[24] Pixymon overview. http://cmucam.org/projects/cmucam5/wiki/PixyMon_Overview.

[25] Pulse width modulation. https://en.wikipedia.org/wiki/Pulse-width_modulation.

[26] Rubik's cube wikipedia. https://en.wikipedia.org/wiki/Rubik%27s_Cube.

[27] Sainsmart iic/i2c/twi serial. http://www.amazon.com/SainSmart-Serial-Module-Shield-Arduino/dp/B00A61SEU6/ref=sr_1_4?ie=UTF8&qid=1461631059&sr=8-4&keywords=spi+lcd+graphics.

[28] Sitara processors overview. http://www.ti.com/lsds/ti/processors/sitara/overview.page.

[29] Sparkfun serial graphic lcd. https://www.sparkfun.com/products/9351.

[30] Spi tft lcd touch panel. http://www.ebay.com/itm/like/171983887298?lpid=82&chn=ps&ul_noapp=true.

[31] Standard size - high torque - metal gear servo. https://www.adafruit.com/products/1142.

[32] Stepper motor with cable. https://www.sparkfun.com/products/9238.

[33] Stumped by rubik's cube? let the lego robot solve it. http://www.nytimes.com/2001/10/11/technology/circuits/11RUBI.html?pagewanted=all.

[34] Surestep stp-mtr-17040. http://www.automationdirect.com.

[35] Ti msp430. https://en.wikipedia.org/wiki/TI_MSP430#MSP430x6xx_series.

[36] Ti powerpad made easy. http://www.ti.com/lit/an/slma004b/slma004b.pdf.

[37] Tpd2e001 low-capacitance 2-channel esd-protection for high-speed data interfaces. `http://www.ti.com/lit/ds/symlink/tpd2e001.pdf`.

[38] Usb module. `http://www.ti.com/lit/ug/slau284d/slau284d.pdf`.

[39] Watch this robot solve a rubik's cube in less than 2 seconds. `http://motherboard.vice.com/read/watch-this-robot-solve-a-rubiks-cube-in-less-than-2-seconds`.

[40] What material should i use for 3d printing? `http://3dprintingforbeginners.com/filamentprimer-2/`.

[41] Steven Keeping. Understanding the advantages and disadvantages of linear regulators. `http://www.digikey.com/en/articles/techzone/2012/may/understanding-the-advantages-and-disadvantages-of-linear-regulators`.

[42] Krishnavedala. Rc filter image. `https://commons.wikimedia.org/wiki/File:RC_filter.svg`.

[43] Wdwd. Half-wave rectifier image. `https://commons.wikimedia.org/wiki/File:Gratz.rectifier.en.svg`.

9.3 Appendix C –Programs Used

Software	Usage
Eagle 7.4.0	Schematic and PCB editing
TeXstudio	Documentation
TeXworks	Documentation
SourceTree	Project collaboration
CodeBlocks	Documentation
Sublime Text Editor	Documentation
Microsoft Paint	Documentation
Code Composer Studio 5.5.0	MSP430 Code Development
RealTerm 2.0.0.70	Serial Communications development and testing
Ultra Librarian Lite 8.1.23	Package generation for PCB layout

9.4 Appendix D –Bill of Materials

Part	Value	Package	Description
C19	0.1uF	C0402	CAPACITOR, American symbol
C20	0.01uF	C0402	CAPACITOR, American symbol
C21	100uF	E2,5-6	POLARIZED CAPACITOR, American symbol
C22	0.1uF	C0402	CAPACITOR, American symbol
C23	0.1uF	C0402	CAPACITOR, American symbol
C24	0.47uF	C0402	CAPACITOR, American symbol
C37	4.7uF,10V	C0603	CAPACITOR, American symbol
C38	10pF,6V	C0603	CAPACITOR, American symbol
C39	10pF,6V	C0603	CAPACITOR, American symbol
C40	220nF,10V	C0603	CAPACITOR, American symbol
C41	220nF,10V	C0603	CAPACITOR, American symbol
C42	1nF	C0603	CAPACITOR, American symbol
C43	0.1uF	C0402	CAPACITOR, American symbol
C44	0.47uF	C0402	CAPACITOR, American symbol
C45	10uF	E2,5-6	POLARIZED CAPACITOR, American symbol
C46	100n	C0603	CAPACITOR, American symbol
D1_5V	CGRM4001-G	SOD-123_MINI-SMA	Molded plasitc,JEDEC SOD-123/Mini SMA
JP4		1X04	PIN HEADER
LED1		SML0805	LED
LED2		SML0805	LED
Q1	CSTCR6M00G53Z	CSTCR6M	Resonator

R16	1M	M1206	RESISTOR, American symbol
R17	20k	R0402	RESISTOR, American symbol
R18	220k	R0402	RESISTOR, American symbol
R19	0.2	R0402	RESISTOR, American symbol
R20	0.2	R0402	RESISTOR, American symbol
R31	27	R0402	RESISTOR, American symbol
R32	27	R0402	RESISTOR, American symbol
R33	1M	M1206	RESISTOR, American symbol
R34	47k	R0402	RESISTOR, American symbol
R35	1.4k	R0402	RESISTOR, American symbol
R36	470	R0402	RESISTOR, American symbol
R37	470	R0402	RESISTOR, American symbol
S1	SKHMPSE010	SKHMPXE010	6.2 X 6.5mm TACT Switch (SMD)
S2	SKHMPSE010	SKHMPXE010	6.2 X 6.5mm TACT Switch (SMD)
SV1		MA08-2	PIN HEADER
SV2		MA07-2	PIN HEADER
U1	LM1086_KTT_3	TS3B	
U2	Value	PZ0100A_N	
U5	DRV8825_PWP_28	PWP28_5P18X3P1	
U8	TPD2E001_DRL_5	DRL5	
U9	TPS715A01_DRV_6	DRV6_1P6X1	
X1		SCD-014-A	Power Jack 2.5mm
X2	MINI-USB-SHIELD-UX60-MB-5ST	UX60-MB-5ST	MINI USB Connector

9.5 Appendix E –MSP430 Code

9.5.1 USB interface

```
/* --COPYRIGHT--,BSD
* Copyright (c) 2012, Texas Instruments Incorporated
* All rights reserved.
*
* Redistribution and use in source and binary forms, with or without
* modification, are permitted provided that the following conditions
* are met:
*
* *  Redistributions of source code must retain the above copyright
*    notice, this list of conditions and the following disclaimer.
*
* *  Redistributions in binary form must reproduce the above copyright
*    notice, this list of conditions and the following disclaimer in the
*    documentation and/or other materials provided with the distribution.
*
* *  Neither the name of Texas Instruments Incorporated nor the names of
*    its contributors may be used to endorse or promote products derived
*    from this software without specific prior written permission.
*
* THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
* AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO,
* THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
* PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
* CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
* EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
* PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS;
* OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY,
* WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR
* OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE,
* EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
* --/COPYRIGHT--*/
/*
* ======== main.c ========
* LED Control Demo:
*
* This USB demo example is to be used with a PC application (e.g. HyperTerminal)
* This demo application is used to control the operation of the LED at P1.0
*
* Typing the following pharses in the HyperTerminal Window does the following
* 1. "LED ON" Turns on the LED and returns "LED is ON" phrase to PC
* 2. "LED OFF" Turns off the LED and returns "LED is OFF" back to HyperTerminal
* 3. "LED TOGGLE - SLOW" Turns on the timer used to toggle LED with a large
*    period and returns "LED is toggling slowly" phrase back to HyperTerminal
* 4. "LED TOGGLE - FAST" Turns on the timer used to toggle LED with a smaller
*    period and returns "LED is toggling fast" phrase back to HyperTerminal
*
+----------------------------------------------------------------------------+
* Please refer to the MSP430 USB API Stack Programmer's Guide,located
* in the root directory of this installation for more details.
* ----------------------------------------------------------------------------*/
#include <intrinsics.h>
#include <string.h>

#include "USB_config/descriptors.h"

#include "USB_API/USB_Common/device.h"
#include "USB_API/USB_Common/types.h"               //Basic Type declarations
#include "USB_API/USB_Common/usb.h"                 //USB-specific functions

#include "F5xx_F6xx_Core_Lib/HAL_UCS.h"
#include "F5xx_F6xx_Core_Lib/HAL_PMM.h"

#include "USB_API/USB_CDC_API/UsbCdc.h"
#include "usbConstructs.h"

//Function declarations
void TurnMotors(char*);
VOID Init_Ports (VOID);
VOID Init_Clock (VOID);
VOID Init_TimerA1 (VOID);
BYTE retInString (char* string);

//Global flags set by events
volatile BYTE bCDCDataReceived_event = FALSE;   //Indicates data has been received without an open rcv
      operation

#define MAX_STR_LENGTH 64
char wholeString[MAX_STR_LENGTH] = "";          //The entire input string from the last 'return'
unsigned int SlowToggle_Period = 20000 - 1;
unsigned int FastToggle_Period = 1000 - 1;

/*
* ======== main ========
*/
VOID main (VOID)
{
```

```
WDTCTL = WDTPW + WDTHOLD;                                  //Stop watchdog timer

Init_Ports();                                              //Init ports (do first ports because clocks do change
    ports)
SetVCore(3);
Init_Clock();                                              //Init clocks

USB_init();                    //Init USB

Init_TimerA1();

//Enable various USB event handling routines
USB_setEnabledEvents(
kUSB_VbusOnEvent + kUSB_VbusOffEvent + kUSB_receiveCompletedEvent
+ kUSB_dataReceivedEvent + kUSB_UsbSuspendEvent + kUSB_UsbResumeEvent +
kUSB_UsbResetEvent);

//See if we're already attached physically to USB, and if so, connect to it
//Normally applications don't invoke the event handlers, but this is an exception.
if (USB_connectionInfo() & kUSB_vbusPresent){
USB_handleVbusOnEvent();
}

__enable_interrupt();                                      //Enable interrupts globally
while (1)
{
BYTE i;
//Check the USB state and directly main loop accordingly
switch (USB_connectionState())
{
case ST_USB_DISCONNECTED:
__bis_SR_register(LPM3_bits + GIE);                        //Enter LPM3 w/ interrupts enabled
_NOP();                                                    //For Debugger
break;

case ST_USB_CONNECTED_NO_ENUM:
break;

case ST_ENUM_ACTIVE:
__bis_SR_register(LPM0_bits + GIE);                        //Enter LPM0 (can't do LPM3 when active)
_NOP();                                                    //For Debugger

if (bCDCDataReceived_event){                               //Some data is in the buffer; begin receiving a
    command
P3OUT|=BIT3;
char pieceOfString[MAX_STR_LENGTH] = "";                   //Holds the new addition to the string
char outString[MAX_STR_LENGTH] = "";                       //Holds the outgoing string

//Add bytes in USB buffer to theCommand
cdcReceiveDataInBuffer((BYTE*)pieceOfString,
MAX_STR_LENGTH,
CDC0_INTFNUM);                                             //Get the next piece of the string
strcat(wholeString,pieceOfString);
cdcSendDataInBackground((BYTE*)pieceOfString,
strlen(pieceOfString),CDC0_INTFNUM,0);                     //Echoes back the characters received
P3OUT&=~BIT3;

if (retInString(wholeString)){                             //Has an enter been recieved?

TurnMotors(wholeString);

for (i = 0; i < MAX_STR_LENGTH; i++){                      //Clear the string in preparation for the next one
wholeString[i] = 0x00;
}
}
bCDCDataReceived_event = FALSE;
}
break;

case ST_ENUM_SUSPENDED:
__bis_SR_register(LPM3_bits + GIE);                        //Enter LPM3 w/ interrupts
_NOP();
break;

case ST_ENUM_IN_PROGRESS:
break;

case ST_NOENUM_SUSPENDED:
__bis_SR_register(LPM3_bits + GIE);
_NOP();
break;

case ST_ERROR:
_NOP();
break;

default:;
}
}  //while(1)
} //main()
/*
* ======== TurnMotors ========
*/
void TurnMotors(char* sequence){
int w=0;
int i;
int j;
int stringlength=strlen(sequence);
P3OUT|=BIT2;                                    // LED2 ON
P1OUT&=~BIT7;                                   //       TURN ON nRESET (ACTIVE LOW)
for(w=0;w<stringlength;w++){
int motor=sequence[w];
int dir=sequence[w+1];
if(dir==0x27){P1OUT|=BIT4;}                     //Set DIR=1 if needed (Set P1.4)
else{P1OUT&=~BIT4;}                                     //Otherwise DIR=0 (Clear P1.4)
switch(motor){
case 0x46:                 //CONDITION FOR "F"
i=0;
j=0;
P1OUT&=~BIT3;                       // Initialize Step to logic low
P2OUT&=~BIT1;                       //Turn nENBL1 off (Clear P2.1)
P1OUT|=BIT7;                        // Turn off nRESET (active low)
for(j=0;j<1625;j++){}    // Quick Delay
for(i=0;i<99;i++){
P1OUT^=BIT3;               // Toggle STEP (P1.3)
for(j=0;j<1625;j++){}
}
P2OUT|=BIT1;                        //toggle nENBL1 on (P2.1)
P1OUT&=~BIT3;                       // Initialize Step to logic low
P1OUT&=~BIT7;                       // turn on nRESET (active low)
break;

case 0x4c:                 //CONDITION FOR "L"
i=0;
j=0;
P2OUT&=~BIT2;                       //Turn nENBL2 off (Clear P2.2)
P1OUT&=~BIT3;                       // Initialize Step to logic low
P1OUT|=BIT7;                        // Turn off nRESET (active low)
for(j=0;j<1625;j++){}
for(i=0;i<99;i++){
P1OUT^=BIT3;               // Toggle STEP (P1.3)
for(j=0;j<1625;j++){}
}
P2OUT|=BIT2;                        //toggle nENBL2 on (P2.2)
P1OUT&=~BIT3;                       // Initialize Step to logic low
P1OUT&=~BIT7;                       // turn on nRESET (active low)
break;

case 0x52:                 //CONDITION FOR "R"
i=0;
j=0;
P2OUT&=~BIT3;                       //Turn nENBL3 off (Clear P2.3)
P1OUT&=~BIT3;                       // Initialize Step to logic low
P1OUT|=BIT7;                        // Turn off nRESET (active low)
for(j=0;j<1625;j++){}
for(i=0;i<99;i++){
P1OUT^=BIT3;               // Toggle STEP (P1.3)
for(j=0;j<1625;j++){}
}
P2OUT|=0xff;                        //toggle nENBL3 on (Set P2.3)
P1OUT&=~BIT3;                       // Initialize Step to logic low
P1OUT&=~BIT7;                       // turn on nRESET (active low)
break;


case 0x42:                 //CONDITION FOR "B"
i=0;
j=0;
P2OUT&=~BIT4;                       //Turn nENBL5 off (Clear P2.4)
P1OUT&=~BIT3;                       // Initialize Step to logic low
P1OUT|=BIT7;                        // Turn off nRESET (active low)
for(j=0;j<1625;j++){}
for(i=0;i<99;i++){
P1OUT^=BIT3;               // Toggle STEP (P1.3)
for(j=0;j<1625;j++){}
}
P2OUT|=0xff;                        //toggle nENBL3 on (Set P2.4)
P1OUT&=~BIT3;                       // Initialize Step to logic low
P1OUT&=~BIT7;                       // turn on nRESET (active low)
break;

case 0x55:                 //CONDITION FOR "U"
i=0;
j=0;
P2OUT&=~BIT0;                       //Turn nENBL0 off (Clear P2.0)
P1OUT&=~BIT3;                       // Initialize Step to logic low
P1OUT|=BIT7;                        // Turn off nRESET (active low)
for(j=0;j<1625;j++){}
for(i=0;i<99;i++){
P1OUT^=BIT3;               // Toggle STEP (P1.3)
for(j=0;j<1625;j++){}
}
P2OUT|=0xff;                        //toggle nENBL1 on (P2.0)
P1OUT&=~BIT3;                       // Initialize Step to logic low
```

```
P1OUT&=~BIT7;                      // turn on nRESET (active low)
break;

case 0x44:               //CONDITION FOR "D"
i=0;
j=0;
P2OUT&=~BIT5;                      //Turn nENBL5 off (Clear P2.5)
P1OUT&=~BIT3;                      // Initialize Step to logic low
P1OUT|=BIT7;                       // Turn off nRESET (active low)
for(j=0;j<1625;j++){}
for(i=0;i<99;i++){
P1OUT^=BIT3;             // Toggle STEP (P1.3)
for(j=0;j<1625;j++){}
}
P2OUT|=0xff;                       //toggle nENBL3 on (Set P2.5)
P1OUT&=~BIT3;                      // Initialize Step to logic low
P1OUT&=~BIT7;                      // turn on nRESET (active low)
break;
}
}
P3OUT&=~BIT2;
P1OUT&=~BIT7;    //
}
/*
* ======== Init_Clock ========
*/
VOID Init_Clock (VOID)
{
//Initialization of clock module
if (USB_PLL_XT == 2){
#if defined (__MSP430F552x) || defined (__MSP430F550x)
P5SEL |= 0x0C;                                        //enable XT2 pins for F5529
#elif defined (__MSP430F563x_F663x)
P7SEL |= 0x0C;
#endif

//use REFO for FLL and ACLK
UCSCTL3 = (UCSCTL3 & ~(SELREF_7)) | (SELREF__REFOCLK);
UCSCTL4 = (UCSCTL4 & ~(SELA_7)) | (SELA__REFOCLK);

//MCLK will be driven by the FLL (not by XT2), referenced to the REFO
Init_FLL_Settle(USB_MCLK_FREQ / 1000, USB_MCLK_FREQ / 32768);   //Start the FLL, at the freq indicated
      by the config
//constant USB_MCLK_FREQ
XT2_Start(XT2DRIVE_0);                                          //Start the "USB crystal"
}
else {
#if defined (__MSP430F552x) || defined (__MSP430F550x)
P5SEL |= 0x10;                                        //enable XT1 pins
#endif
//Use the REFO oscillator to source the FLL and ACLK
UCSCTL3 = SELREF__REFOCLK;
UCSCTL4 = (UCSCTL4 & ~(SELA_7)) | (SELA__REFOCLK);

//MCLK will be driven by the FLL (not by XT2), referenced to the REFO
Init_FLL_Settle(USB_MCLK_FREQ / 1000, USB_MCLK_FREQ / 32768);   //set FLL (DCOCLK)

XT1_Start(XT1DRIVE_0);                                          //Start the "USB crystal"
}
}

/*
* ======== Init_Ports ========
*/
VOID Init_Ports (VOID)
{
//Initialization of ports (all unused pins as outputs with low-level
P1DIR |= 0xf8;                                                  // P1 all outputs
P1OUT |= BIT6 + BIT7;                                           // nSLEEP and nRESET = 1

P2DIR |= 0xff;                                                  // P2 OUTPUTS: 0011|1111
P2OUT |= 0xff;                                                  // all nENBL high to disarm chips

P3DIR |=0xff;                                                           //LEDS
P3OUT |=0x00;
}

/*
* ======== UNMI_ISR ========
*/
#pragma vector = UNMI_VECTOR
__interrupt VOID UNMI_ISR (VOID)
{
switch (__even_in_range(SYSUNIV, SYSUNIV_BUSIFG))
{
case SYSUNIV_NONE:
__no_operation();
break;
case SYSUNIV_NMIIFG:
__no_operation();
break;
case SYSUNIV_OFIFG:
```

```
UCSCTL7 &= ~(DCOFFG + XT1LFOFFG + XT2OFFG); //Clear OSC flaut Flags fault flags
SFRIFG1 &= ~OFIFG;                          //Clear OFIFG fault flag
break;
case SYSUNIV_ACCVIFG:
__no_operation();
break;
case SYSUNIV_BUSIFG:

//If bus error occured - the cleaning of flag and re-initializing of
//USB is required.
SYSBERRIV = 0;                                      //clear bus error flag
USB_disable();                                      //Disable
}
}

/*
* ======== Init_TimerA1 ========
*/
VOID Init_TimerA1 (VOID)
{
TA1CCTL0 = CCIE;                                            //CCR0 interrupt enabled
TA1CTL = TASSEL_1 + TACLR;                                  //ACLK, clear TAR
}

/*
* ======== retInString ========
*/
//This function returns true if there's an 0x0D character in the string; and if so,
//it trims the 0x0D and anything that had followed it.
BYTE retInString (char* string)
{
BYTE retPos = 0,i,len;
char tempStr[MAX_STR_LENGTH] = "";

strncpy(tempStr,string,strlen(string));                     //Make a copy of the string
len = strlen(tempStr);
while ((tempStr[retPos] != 0x0A) && (tempStr[retPos] != 0x0D) &&
(retPos++ < len)) ;                                   //Find 0x0D; if not found, retPos ends up at len

if ((retPos < len) && (tempStr[retPos] == 0x0D)){           //If 0x0D was actually found...
for (i = 0; i < MAX_STR_LENGTH; i++){                   //Empty the buffer
string[i] = 0x00;
}
strncpy(string,tempStr,retPos);                         //...trim the input string to just before 0x0D
return (TRUE) ;                                         //...and tell the calling function that we did
     so
} else if ((retPos < len) && (tempStr[retPos] == 0x0A)){    //If 0x0D was actually found...
for (i = 0; i < MAX_STR_LENGTH; i++){                   //Empty the buffer
string[i] = 0x00;
}
strncpy(string,tempStr,retPos);                         //...trim the input string to just before 0x0D
return (TRUE) ;                                         //...and tell the calling function that we did
     so
} else if (tempStr[retPos] == 0x0D){
for (i = 0; i < MAX_STR_LENGTH; i++){                   //Empty the buffer
string[i] = 0x00;
}
strncpy(string,tempStr,retPos);                         //...trim the input string to just before 0x0D
return (TRUE) ;                                         //...and tell the calling function that we did
     so
} else if (retPos < len){
for (i = 0; i < MAX_STR_LENGTH; i++){                   //Empty the buffer
string[i] = 0x00;
}
strncpy(string,tempStr,retPos);                         //...trim the input string to just before 0x0D
return (TRUE) ;                                         //...and tell the calling function that we did
     so
}

return (FALSE) ;                                            //Otherwise, it wasn't found
}

/*
* ======== TIMER1_A0_ISR ========
*/
#pragma vector=TIMER1_A0_VECTOR
__interrupt void TIMER1_A0_ISR (void)
{
}
```

9.5.2 UART interface

```
/* --COPYRIGHT--,BSD_EX
* Copyright (c) 2012, Texas Instruments Incorporated
* All rights reserved.
*
* Redistribution and use in source and binary forms, with or without
* modification, are permitted provided that the following conditions
* are met:
*
* *  Redistributions of source code must retain the above copyright
*    notice, this list of conditions and the following disclaimer.
*
* *  Redistributions in binary form must reproduce the above copyright
*    notice, this list of conditions and the following disclaimer in the
*    documentation and/or other materials provided with the distribution.
*
* *  Neither the name of Texas Instruments Incorporated nor the names of
*    its contributors may be used to endorse or promote products derived
*    from this software without specific prior written permission.
*
* THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
* AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO,
* THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
* PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
* CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
* EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
* PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS;
* OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY,
* WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR
* OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE,
* EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
*
*****************************************
//
//                MSP430F6659
//             -----------------
//         /|\|                 |
//          | |                 |
//          --|RST              |
//            |                 |
//            |     P8.2/UCA0TXD|------------> PC RX
//            |                 | 57600 - 8N1
//            |     P8.3/UCA0RXD|<------------ PC TX
//
//   P. Thanigai
//   Texas Instruments Inc.
//   October 2012
//   Built with IAR Embedded Workbench Version: 5.40 & CCS V5.2
//*****************************************
//                MSP430F6659
//             -----------------
//         /|\|                 |
//          | |                 |
//          --|RST              |
//            |                 |
//            |     P9.2/UCA0TXD|------------> PIXY RX
//            |                 | 57600 - 8N1
//            |     P9.3/UCA0RXD|<------------ PIXY TX
//
//   P. Thanigai
//*****************************************
#include <msp430.h>

void Port_Mapping(void);
void TurnMotors(char*);
int main(void)
{
WDTCTL = WDTPW | WDTHOLD;                      // Stop WDT

P8SEL |= 0x0c;                        // P8.3 to UCA1RXD, P8.2 to UCA1TXD
P9SEL |= 0x0c;                        // P9.3 to UCA2RXD, P9.2 to UCA2TXD

P1DIR |= 0xf8;                                    // P1 all outputs
P1OUT |= BIT6 + BIT7;                       // nSLEEP and nRESET = 1

P2DIR |= 0xff;                                        // P2 OUTPUTS: 0011|1111
P2OUT |= 0xff;                                        // all nENBL high to disarm chips

UCA1CTL1 |= UCSWRST;                 // **Put state machine in reset**
UCA1CTL1 |= UCSSEL_2;               // SMCLK
UCA1BR0 = 18;                       // 1MHz 576000
UCA1BR1 = 0;                        // 1MHz 576000
UCA1MCTL = UCBRS_1 | UCBRF_0;    // Modln UCBRSx=1, UCBRFx=0,m

UCA2CTL1 |= UCSWRST;                           // **Put state machine in reset**
UCA2CTL1 |= UCSSEL_2;                          // SMCLK
UCA2BR0 = 18;                                   // 1MHz 57600
UCA2BR1 = 0;                                   // 1MHz 57600
UCA2MCTL = UCBRS_1 | UCBRF_0;    // Modln UCBRSx=0, UCBRFx=0,m
```

```
UCA1CTL1 &= ~UCSWRST;                          // **Initialize USCI state machine**
UCA1IE |= UCRXIE;                              // Enable USCI_A0 RX interrupt

UCA2CTL1 &= ~UCSWRST;                          // **Initialize USCI state machine**
UCA2IE |= UCRXIE;                              // Enable USCI_A0 RX interrupt

char framestring[250];
char cubestring[40];
int y=0;
int k=0;
int q=0;
int within_frame=0;
int mode=0;
int w=0;
int x=0;
for(;;){
if(UCA1IV==2){   //IF NEW UART RX ON USCIA1
if(UCA1RXBUF!=0x00){      //If RX!=null
if(mode==2){
cubestring[k]=UCA1RXBUF;
k++;
if(UCA1RXBUF==0x0d){
for(y=k;y<41;y++){
cubestring[y]=0x00;}
TurnMotors(cubestring);
k=0;
mode=0;
}
}
if(UCA1RXBUF==0x21){


scan:     for(w=0;w<250;w++){
while(!(UCA2IV==2)); //IF NEW UART RX ON USCIA2
framestring[w]=UCA2RXBUF;
}
int frame_start;
int frame_end;
int frame_start_found=0;
int temp;
for(x=0;x<250;x++){
temp = framestring[x];
if(temp==0x55){
if(framestring[x+1]==0xAA)
if(framestring[x+2]==0x55){
if(framestring[x+3]==0xAA){
if(frame_start_found==0){
frame_start=x+2;
frame_start_found=1;}
else{
frame_end=x;
x=250;
}
}
}
}
}
unsigned int objnum=0;
unsigned int total_objects=0;
unsigned int m;
unsigned int signature_code[6];
unsigned int x1_position[6];
unsigned int x2_position[6];
unsigned int y1_position[6];
unsigned int y2_position[6];
for(m=frame_start;m<(frame_end);m++){
if(framestring[m]==0x55){
if(framestring[m+1]==0xAA){
signature_code[objnum]=framestring[m+4];
while (!(UCA1IFG&UCTXIFG));
UCA1TXBUF=signature_code[objnum];
if(signature_code[objnum]>0x06){goto scan;}
x1_position[objnum]=framestring[m+6];
while (!(UCA1IFG&UCTXIFG));
UCA1TXBUF=x1_position[objnum];
x2_position[objnum]=framestring[m+7];
while (!(UCA1IFG&UCTXIFG));
UCA1TXBUF= x2_position[objnum];
y1_position[objnum]=framestring[m+8];
while (!(UCA1IFG&UCTXIFG));
UCA1TXBUF=y1_position[objnum];
y2_position[objnum]=framestring[m+9];
while (!(UCA1IFG&UCTXIFG));
UCA1TXBUF= y2_position[objnum];
objnum++;
}


}
}
}
if(UCA1RXBUF==0x32){
mode=2;
}
}//end (if new UART!=null)
}//end (if new UART input)
}//end (inf loop)
}//end (main)

void TurnMotors(char* sequence){
int w=0;
int i;
int j;
int stringlength=strlen(sequence);
for(w=0;w<stringlength;w++){
int motor=sequence[w];
int dir=sequence[w+1];
if(dir==0x27){P1OUT|=BIT4;}                         //Set DIR=1 if needed (Set P1.4)
else{P1OUT&=~BIT4;}                                         //Otherwise DIR=0 (Clear P1.4)
switch(motor){
case 0x46:                  //UART CONDITION FOR "F"
i=0;
j=0;
P2OUT&=~BIT1;                           //Turn nENBL1 off (Clear P2.1)
P1OUT&=~BIT3;                           // Initialize Step to logic low
for(j=0;j<250;j++){}
for(i=0;i<100;i++){
P1OUT^=BIT3;                // Toggle STEP (P1.3)~450Hz
for(j=0;j<250;j++){}
}
P2OUT|=BIT1;                            //toggle nENBL1 on (P2.1)
P1OUT&=~BIT3;                           // Initialize Step to logic low
break;

case 0x4c:                  //UART CONDITION FOR "L"
i=0;
j=0;
P2OUT&=~BIT2;                           //Turn nENBL2 off (Clear P2.2)
P1OUT&=~BIT3;                           // Initialize Step to logic low
for(j=0;j<250;j++){}
for(i=0;i<100;i++){
P1OUT^=BIT3;                // Toggle STEP (P1.3)~450Hz
for(j=0;j<250;j++){}
}
P2OUT|=BIT2;                            //toggle nENBL2 on (P2.2)
P1OUT&=~BIT3;                           // Initialize Step to logic low
break;

case 0x52:                  //UART CONDITION FOR "R"
i=0;
j=0;
P2OUT&=~BIT3;                           //Turn nENBL3 off (Clear P2.3)
P1OUT&=~BIT3;                           // Initialize Step to logic low
for(j=0;j<250;j++){}
for(i=0;i<100;i++){
P1OUT^=BIT3;                // Toggle STEP (P1.3)~450Hz
for(j=0;j<250;j++){}
}
P2OUT|=0xff;                            //toggle nENBL3 on (Set P2.3)
P1OUT&=~BIT3;                           // Initialize Step to logic low
break;


case 0x42:                  //UART CONDITION FOR "B"
i=0;
j=0;
P2OUT&=~BIT4;                           //Turn nENBL5 off (Clear P2.4)
P1OUT&=~BIT3;                           // Initialize Step to logic low
for(j=0;j<250;j++){}
for(i=0;i<100;i++){
P1OUT^=BIT3;                // Toggle STEP (P1.3)~450Hz
for(j=0;j<250;j++){}
}
P2OUT|=0xff;                            //toggle nENBL3 on (Set P2.4)
P1OUT&=~BIT3;                           // Initialize Step to logic low
break;

case 0x55:                  //UART CONDITION FOR "U"
i=0;
j=0;
P2OUT&=~BIT0;                           //Turn nENBL0 off (Clear P2.0)
P1OUT&=~BIT3;                           // Initialize Step to logic low
for(j=0;j<250;j++){}
for(i=0;i<100;i++){
P1OUT^=BIT3;                // Toggle STEP (P1.3)~450Hz
for(j=0;j<250;j++){}
}
P2OUT|=0xff;                            //toggle nENBL1 on (P2.0)
P1OUT&=~BIT3;                           // Initialize Step to logic low
break;

case 0x44:                  //UART CONDITION FOR "D"
i=0;
```

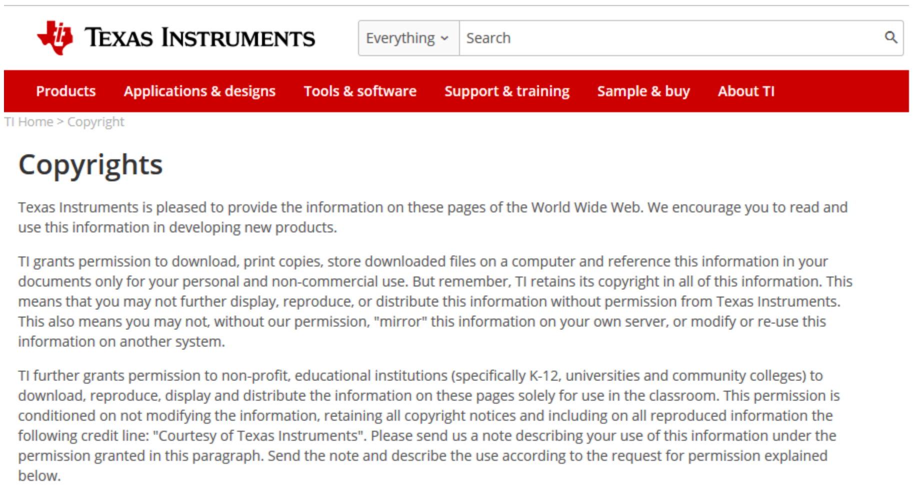
```
j=0;
P2OUT&=~BIT5;                        //Turn nENBL5 off (Clear P2.5)
P1OUT&=~BIT3;                        // Initialize Step to logic low
for(j=0;j<250;j++){}
for(i=0;i<100;i++){
P1OUT^=BIT3;               // Toggle STEP (P1.3) ~450Hz
for(j=0;j<250;j++){}
}
P2OUT|=0xff;                         //toggle nENBL3 on (Set P2.5)
P1OUT&=~BIT3;                        // Initialize Step to logic low
break;
}
}
}


void Port_Mapping(void)
{
// Disable Interrupts before altering Port Mapping registers
__disable_interrupt();
// Enable Write-access to modify port mapping registers
PMAPPWD = 0x02D52;

#ifdef PORT_MAP_RECFG
// Allow reconfiguration during runtime
PMAPCTL = PMAPRECFG;
#endif


// Disable Write-Access to modify port mapping registers
PMAPPWD = 0;
#ifdef PORT_MAP_EINT
__enable_interrupt();                           // Re-enable all interrupts
#endif
}
```

9.6 Appendix F –Permissions Used



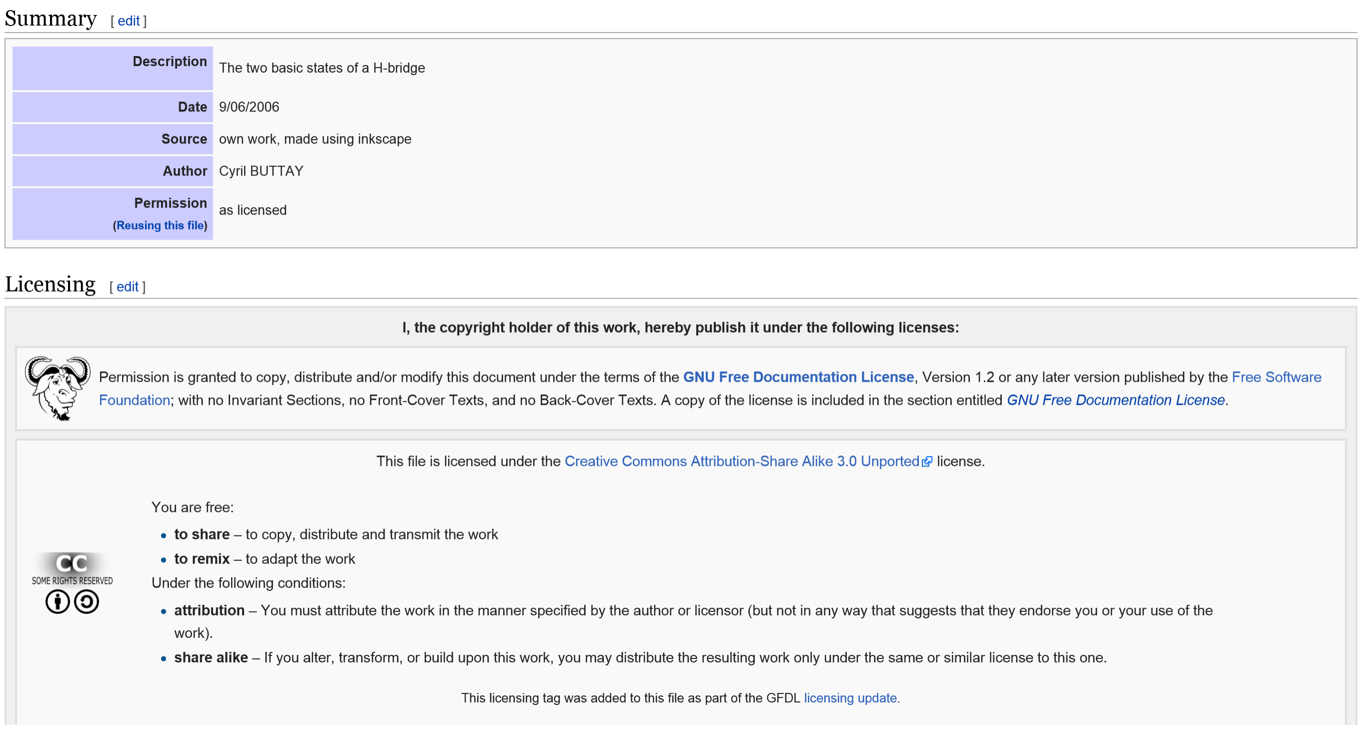
Copyrights

Texas Instruments is pleased to provide the information on these pages of the World Wide Web. We encourage you to read and use this information in developing new products.

TI grants permission to download, print copies, store downloaded files on a computer and reference this information in your documents only for your personal and non-commercial use. But remember, TI retains its copyright in all of this information. This means that you may not further display, reproduce, or distribute this information without permission from Texas Instruments. This also means you may not, without our permission, "mirror" this information on your own server, or modify or re-use this information on another system.

TI further grants permission to non-profit, educational institutions (specifically K-12, universities and community colleges) to download, reproduce, display and distribute the information on these pages solely for use in the classroom. This permission is conditioned on not modifying the information, retaining all copyright notices and including on all reproduced information the following credit line: "Courtesy of Texas Instruments". Please send us a note describing your use of this information under the permission granted in this paragraph. Send the note and describe the use according to the request for permission explained below.

Figure 68



Summary [edit]

Description	The two basic states of a H-bridge
Date	9/06/2006
Source	own work, made using inkscape
Author	Cyril BUTTAY
Permission (Reusing this file)	as licensed

Licensing [edit]

I, the copyright holder of this work, hereby publish it under the following licenses:

Permission is granted to copy, distribute and/or modify this document under the terms of the **GNU Free Documentation License**, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled *GNU Free Documentation License*.

This file is licensed under the Creative Commons Attribution-Share Alike 3.0 Unported license.

You are free:

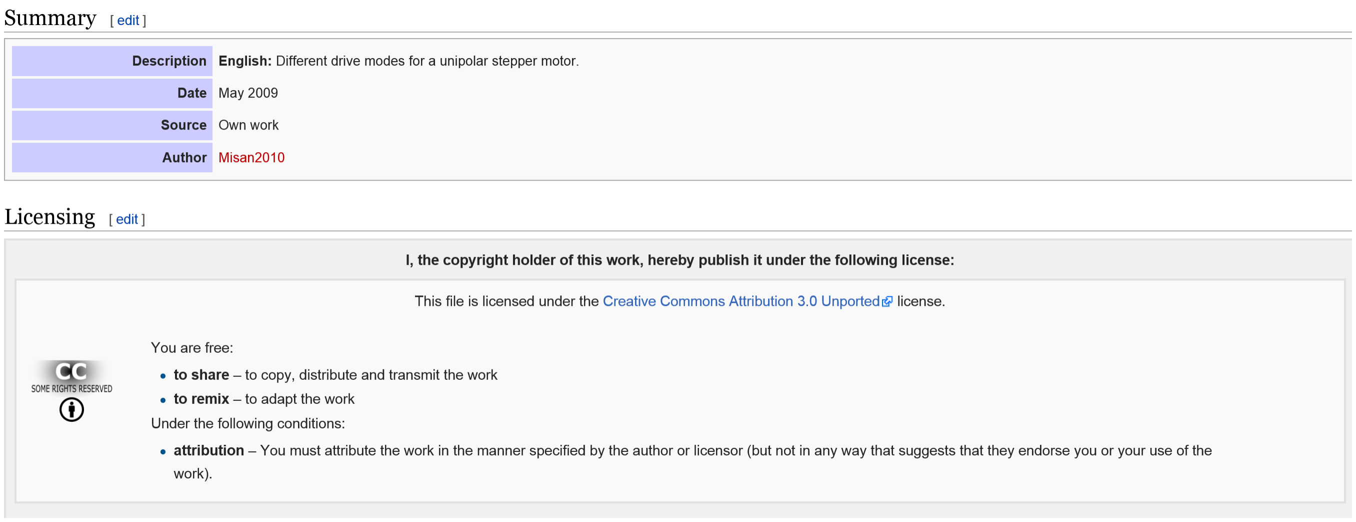
- **to share** – to copy, distribute and transmit the work
- **to remix** – to adapt the work

Under the following conditions:

- **attribution** – You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).
- **share alike** – If you alter, transform, or build upon this work, you may distribute the resulting work only under the same or similar license to this one.

This licensing tag was added to this file as part of the GFDL licensing update.

Figure 69

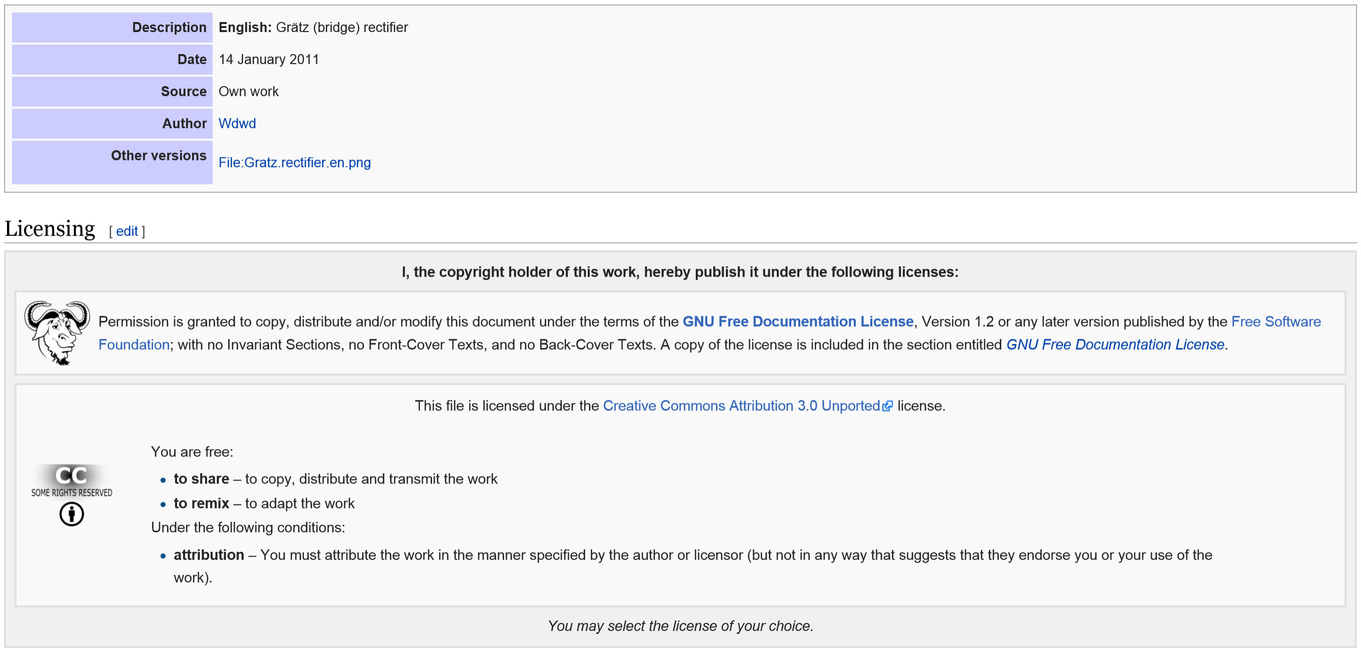


Summary [edit]

Description	**English:** Different drive modes for a unipolar stepper motor.
Date	May 2009
Source	Own work
Author	Misan2010

Licensing [edit]

Figure 70



Description	**English:** Grätz (bridge) rectifier
Date	14 January 2011
Source	Own work
Author	Wdwd
Other versions	File:Gratz.rectifier.en.png

Licensing [edit]

Figure 71