Self Solving/Scrambling Rubik's Cube for Learning and Training

Byron Lathi

Colin Choi

Walter Uruchima

ECE 445 Final Report - Fall 2022

TA: Li Qing Yu

Project No. 09

7 December 2022

Abstract

The self solving and self scrambling Rubik's Cube is a functional toy-puzzle with integrated motors and controllers. The components are all held inside of the custom cube and are battery powered. The system is operated by a microcontroller which is able to store imputed moves into memory and then reverse them through motor control. Additionally, it can also perform the scrambling process on its own. This paper serves to overview the entirety of the development process for this project.

0 | Contents

1 | Introduction

1.1 Background

Designed and created by Hungarian teacher, Erno Rubik, the Rubik's Cube has been a smash hit for puzzle solvers across the globe. Since its release to stores in 1980, over 450 million Rubik's cubes have been sold making it the most popular and best-selling toy in history. Along with these incredible numbers, Rubik's Cube has also found itself a dedicated group of avid solvers, commonly known as 'cubers'. From the beginning, these people have been learning, working, and competing for faster times in solving the cube. With the current world record for a single 3x3 Rubik's cube solve sitting at 3.47 seconds, it is clear that modern cubers have become incredibly quick at solving the once impossible puzzle.[1]

The standard Rubik's Cube consists of six faces each with a 3x3 grid of colored squares. In its solved state, all 9 of the squares per side will show the same color. In a scrambled state, the colors are scattered and each face will show an array of colors. The standard color set is red, blue, white, orange, green, and yellow. Each of the sides revolve around the center square within the 3x3 grid and can be rotated in either direction.

1.2 Problem

Naturally, the puzzle can be difficult to solve when attempting it for the first time. Beginners are often daunted by the algorithms used to solve the cube and struggle with memorizing them as well. It can almost even feel like learning a new language with all of the strings of letters, numbers, and apostrophes that 'instructionalize' the solving algorithms. There are many tools and resources online that can be used to aid the learning process, but these are not always perfect. It is often unclear in depicting the exact steps required to solving the cube. Especially for a beginner, starting out on the cubing journey can be complicated.

On the other end of the spectrum, skilled cubers often find scrambling the cube to be a cumbersome task. You have to randomize which sides to turn and which way to rotate them by. There must be at least around 20 steps worth of randomization in order to achieve a high quality scramble[2]. However, years of practicing cubing leads to random biases during the scrambling process. Turns become less random and cubers often end up giving themselves suboptimal scrambles. These make for worse practice and slow down improvements to skill.

1.3 Solution

We have worked towards creating a Rubik's Cube that is capable of scrambling and solving itself. Our device contains six integrated motors (one for each side of the cube) with controllers for each of them. They're controlled by a microcontroller and powered by batteries. Bi-directional switches are located on each of the motor controller PCBs to aid with rotation monitoring and precision turning control. Users are able to interface with the cube to initiate solving and scrambling settings by quickly rotating specific sides back and forth.

The cube is programmed to use the hardware inside of it to solve itself regardless of whatever state it might be in. This allows beginner cubers to visually see the necessary steps to solve the cube, and gain better hands-on experience with the correct algorithms. With this, beginners do not have to sift through dense tutorials for suboptimal solving algorithms.

Experienced cubers are also able to extend their skills through better scrambles. The cube is programmed to scramble itself to the optimal 20 moves with (pseudo) random processes[2]. This prevents experienced users from gravitating towards internally biased scrambles from their years of continued cubing.

1.4 Visual Aid

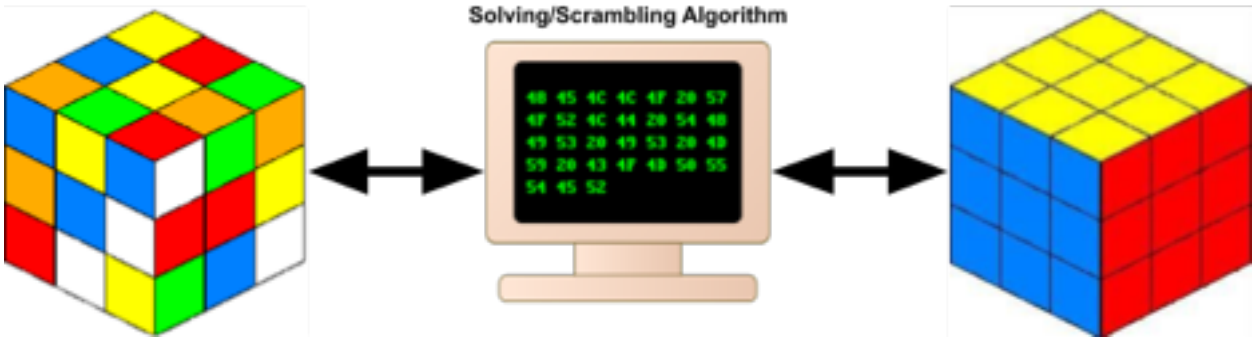


Figure 1: Project Functionality Visual Aid

1.5 High-Level Requirements List

1. The cube must be able to function as a normal Rubik's cube would, independent of the electronics inside of it.
2. The cube must be no larger than 150mm x 150mm x 150mm
3. The cube must be able to solve and scramble itself in under a minute.

2 | Design

2.1 Block Diagram

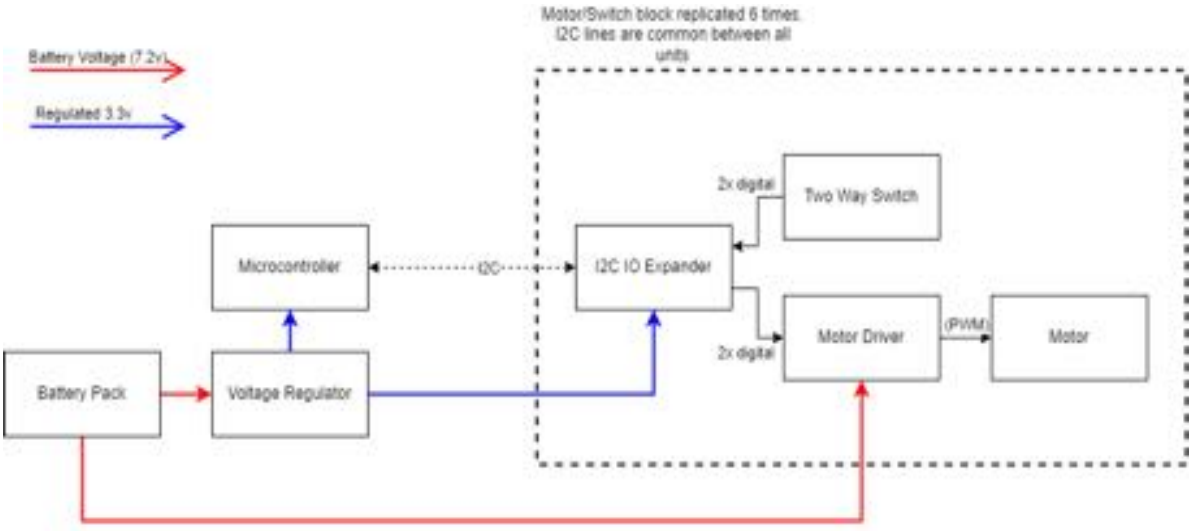


Figure 2: Block Diagram of Project Systems

2.2 Physical

The overall cube structure operates in the same general method as a traditional Rubik's Cube. Pieces are interlocked with each other and are anchored around each face's central square. In our case, the central square is also sleeved over a motor shaft via friction fit. The original design called for a screw to hold the piece in place, but was not implemented in the final design. Internally, the cube's core holds six motors which holds everything together.

Faces are aligned by way of a bi-direcitonal switch. The switches are soldered to a PCB which is screwed into the motor gearbox. These are activated by nubs placed radially equidistant at the base of the center face shaft. The nubs force the switch to actuate every 90 degrees as the center face rotates. In this way, rotational direction can be determined based on the direction of switch activation and precision turning can be achieved based on nub location.

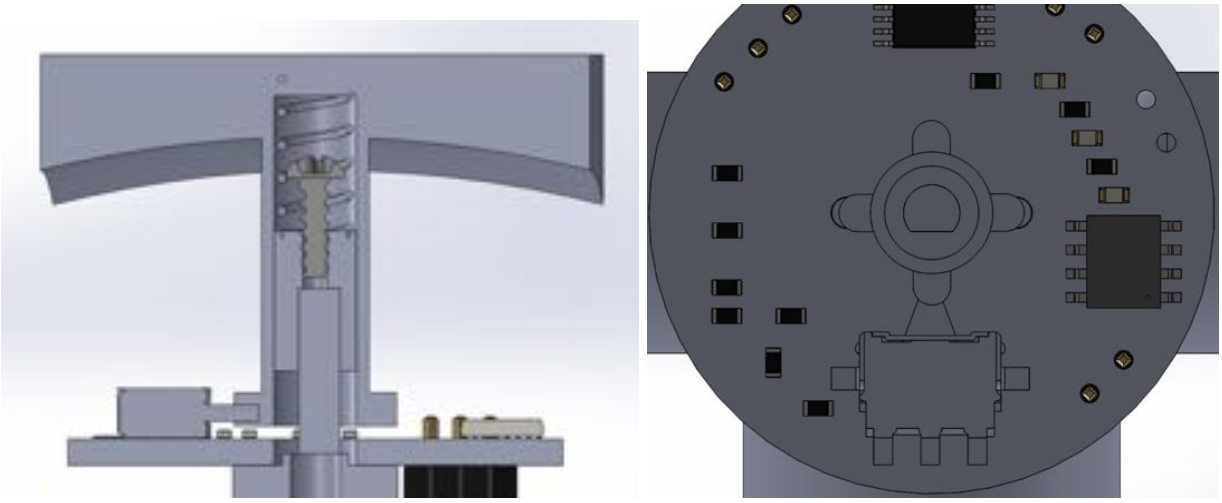


Figure 3: Center Face Connection Design (left) and Switch Module (right)

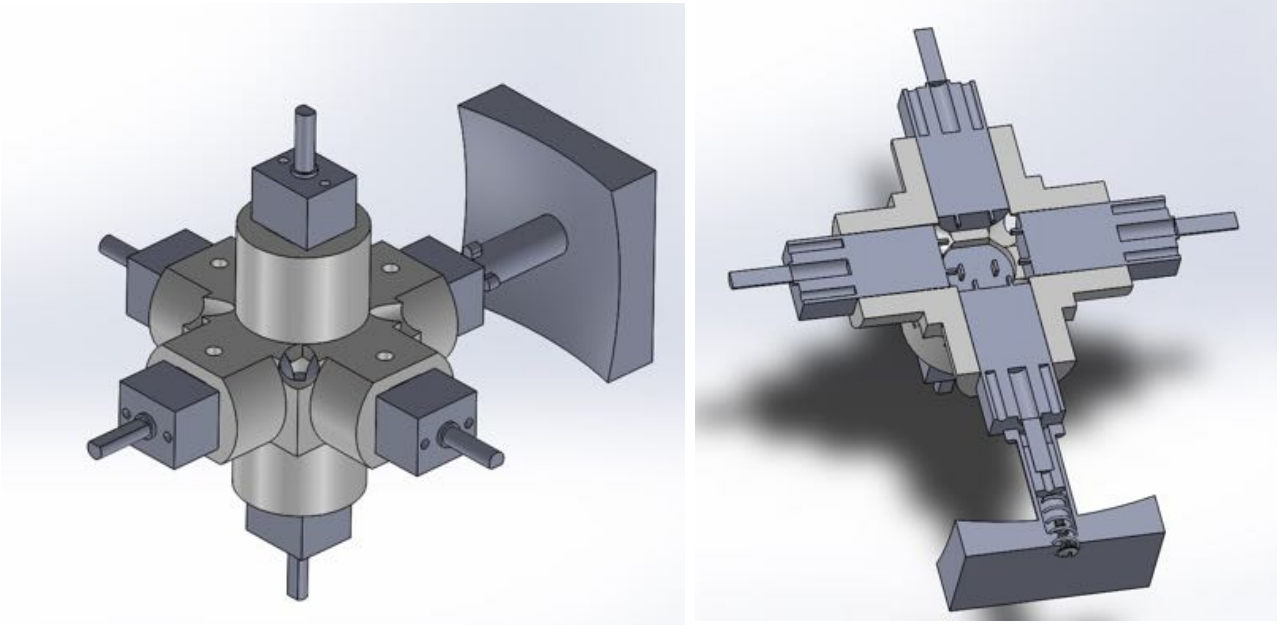


Figure 4: DC Motor Hub Core (left) and Core Sliced View (right)

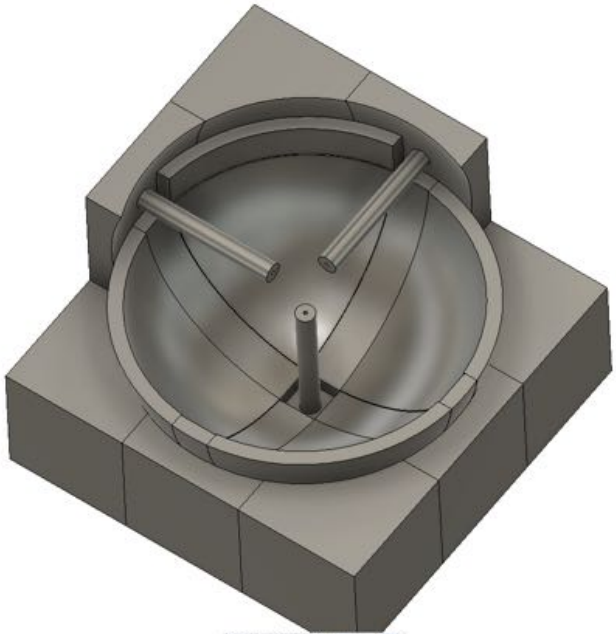


Figure 5: CAD of Outer Cube Shell

2.3 Subsystem Overview

2.3.1 Switch Module

The switch modules contain a bidirectional switch, an I2C I/O expander, and a motor driver. The switch is a Panasonic ESE24, which features separate outputs for both left and right switching directions. The I/O Expander is a PI4IOE5V9554, which has 3 address pins allowing up to 8 bit states to be used in the same system[3]. It contains 8 I/O pins, of which only 4 are needed. The ZXBM5210 Motor Driver allows the motor to be driven in forward and reverse directions as well as be set in coast and brake states[4]. This motor driver has a maximum output current of 1.5A, which is well above the 0.67 A stall current of the motors. Additionally, the driver contains an internal PWM oscillator which can be tuned to lower the motor output power if needed. The maximum supply voltage is 18 volts, which is well above the nominal motor voltage of 6V. Each motor driver drives one mini gear motor. The motors have a 1:298 gear ratio, rotating at approximately 45 RPM at 6V, with a stall torque of 70 oz-in.

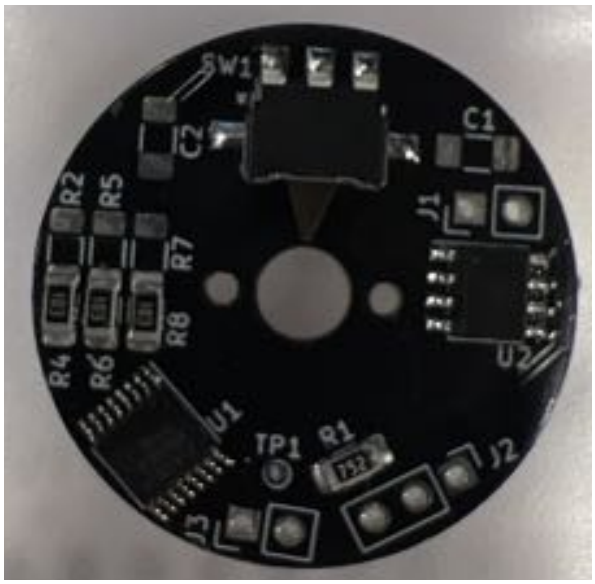


Figure 6: Switch Module PCB with Components

2.3.2 Power and Control System

The control module contains both the microcontroller and the voltage regulator. The motors are powered by two 3.2V LiPo batteries resulting in a nominal battery voltage of 7.4V. This is used to power the motors through the switch modules. As the system only ever powers a single motor for ~0.5 seconds at a time, this does not put any major strain on the output capacity of the batteries. However, this is far too much power to be supplied to the microcontroller. As such, the original plan was to use a 3.3V regulator to step down the voltage so as to not over power the controller.

2.3.3 Software System

The software system controls the motors and is also responsible for keeping track of the state of the cube at all times. This is taken care of by the STM32 microcontroller which is capable of communicating over the I2C bus. It takes input from the bi-directional switches and can set motor states to each face over the I2C bus.

Two forms of user interaction are available from its solved state. A user can choose to either free scramble the cube on their own, or trigger the automated scramble. These algorithms can be triggered by quickly rotating a specified side back and forth. Two different sides correspond to triggering the solve and scrambling algorithms respectively.

The default mode for the motors is set to “coast” so that the user can rotate the cube freely. In this way, the cube can be scrambled without the self scrambling feature. This allows for more freedom to control the cube directly if the user chooses. Alternatively, for an initiated self scramble, the software system generates a random number to determine the face ID number that will be rotated. The motor setting is then changed to “forward”, while all others will be set to “brake”. This process is repeated for 20 turns, and motors are set back to “coast”.

In either case, a basic back tracing algorithm is implemented to solve the cube. As it is scrambled, the rotations are stored in an array in the STM32's memory. When a self solve is triggered, the rotations are played back in reverse order to return the cube to its original state. Through this, we are able to solve the puzzle regardless of what state the cube is in after its scramble.

2.4 Verification and Testing

The verification and testing process for the project can be broken down into three categories: Hardware, Software, and Mechanical. The hardware testing and verification timeline started with the motors, controllers, and switches. The primary goal when we first got our components together was to make sure the motor would be able to respond or provide signals that we could use. Originally, we were planning on testing this all through our switch module PCB. However, once we realized the delays were going to prevent this testing route from being successful, we were forced to explore other options. Because our ICs are all SMD components, we had to work through soldering them to pin headers through perfboards in order to make them usable.

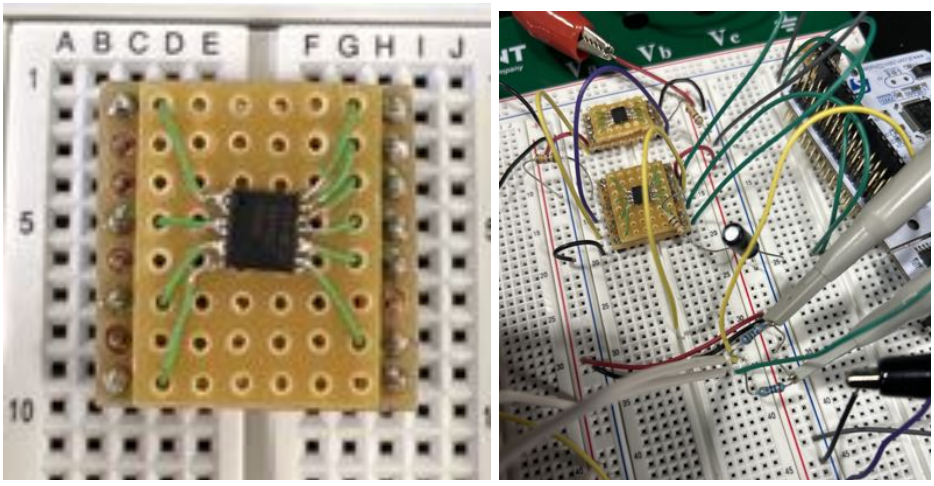


Figure 7: IC Soldered to Pin Headers (left) and Breadboard Implementation (right)

Luckily, once we were able to get across this issue, we were able to connect everything successfully. Our first tests were to make the motor rotate in the

direction of the actuation of the switch. While this was not the actual intended functionality of the components, it was enough proof of concept for us to feel comfortable with proceeding with development. After the PCBs had arrived, we were able to start connecting the motor to the center facepiece and the switch module. We then tested the actual functionality and made sure that the PCBs read the correct rotational direction and that the motors stopped rotating after 90 degrees. From there we were able to start wiring up all six motors together and make sure we were able to communicate with each of them individually over I2C. While there were initial issues with noisy outputs, we were later able to solve this by resoldering some wires and cleaning up connections.

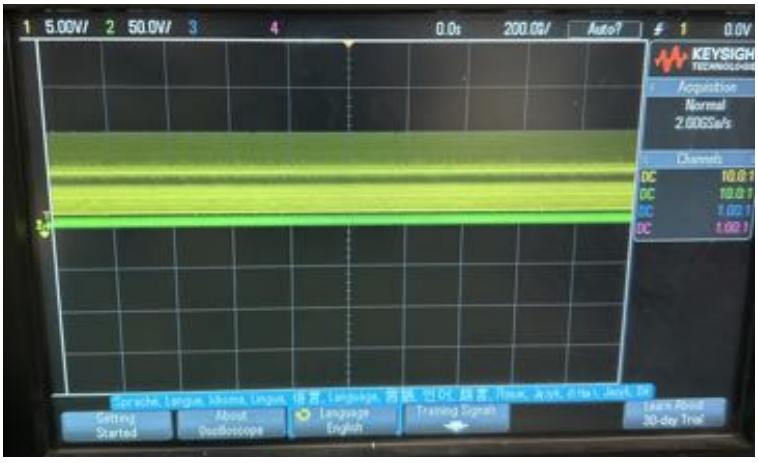


Figure 8: Noisy Oscilloscope Reading

After that, it was primarily testing the software component of the project. Throughout the testing period of the hardware, we were simultaneously doing minor checks on its compatibility with our software. As a result, the merging process between the two systems went fairly smoothly. Our initial checks to make sure the software was receiving readings on the motor rotations and also sending the correct ones paid dividends at the end. The switch register allowed us to read and send precise 90 degree turns. The one major issue we ran into was problems with debouncing. When the switch was returned to its neutral position it sometimes overshot itself and registered an extra erroneous actuation. We were able to solve this through adding hold timers in software and adding capacitors in hardware.

Both of which would have likely solved the issue individually but together provided further reliability.

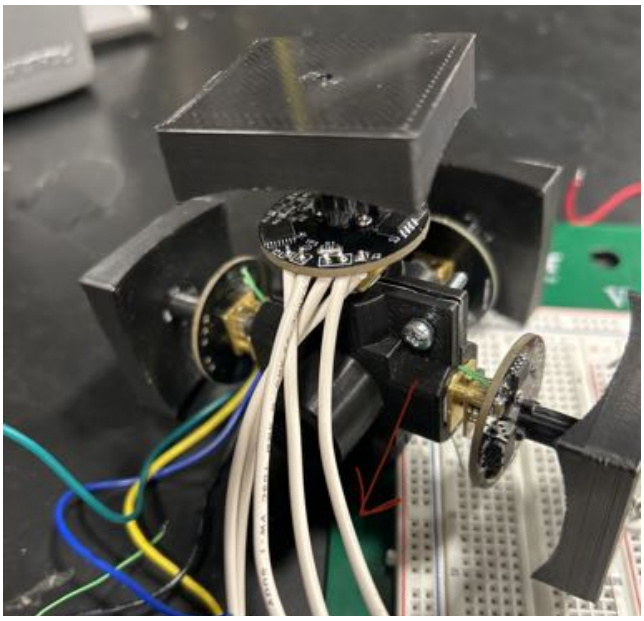


Figure 9: Early Assembly of Inner Core

For the mechanical portion of the project. We spent a significant amount of time working on the 3D printing and cleaning them up. While the center face pieces came out surprisingly well, the corner and side pieces did not. There was a ton of time spent on trying to get them to a workable level, but ultimately we decided it was not realistic to accomplish. The quality level of the 3D print was simply too low and would not allow for reliable enough rotations.

3 | Cost and Components

3.1 Cost Analysis

The total cost of the project can be separated into two expenses. Mainly, the labor and materials cost. Labor can be calculated based on the total hours being put into the project and the average salary for electrical and computer engineers.

Assuming an average salary of $40/hr * 2.5 hrs/day * 75 days * 3 members = $22,500 in total labor costs[5].

The material costs are further split between the components list and 3D printing costs. Our original estimates for a full print of the cube was set around $40 given 125mm sides with 15% infill at $0.15/gram[6]. Unfortunately, this was underestimating this cost by some margin. The center face pieces were relatively cheap printing at around $2 a piece. But, the corner and side pieces combined for a cost of $80 to print them all. Given there were a few iterations of the center face prints, the total cost of 3D printing came out to a little under $100. We did not account for this great of an expense in our project budget, so it ended up being covered through our own dollars.

By contrast, with our components list, we save more money than we originally anticipated. While the motors were predictably still our largest expense, the other components ended up costing less than we thought they would. Mainly, we saved money by not having to pay for our microcontroller. Group member Byron, having been a former intern at STMicroelectronics, was able to source the STM32F4 without any cost associated with us. The batteries also were cheaper than we thought. Despite the original prediction of $20 for two, we ended up paying $10 for five. It was also convenient that we were able to use the SMD resistors, capacitors, and wiring that were stocked within the course lab room for free as well. With all these costs accounted for, the total cost of the project stands at $22,500 + $100 + $114.45 = $22,714.45.

3.2 Components List

Description	Quantity	Total Price	Link
Micro Gear Motor	6	$77.70	Link Link (alternative)
Motor Driver	10	$9.73	Link
Two Way Switch	10	$6.04	Link

I2C Expander	10	$10.98	Link
STM32F4 MCU	1	$0.00	Link
Battery	5	$10.00	Link

4 | Ethics and Safety

4.1 Ethics

As engineers working towards a brighter future and a better planet, we adhere to the IEEE Code of Ethics throughout all engineering practices. For the duration of the project and going forward during work in the real world, we will ensure integrity, responsibility, and ethical practices in our work environment. All members of our team are mutually respected and treated as such. We will work to ensure the ethical nature of our project is not tainted and it will likewise refrain from producing unethical outcomes for others. We do not foresee many points of ethical contention with our project, but that will not prevent us from remaining weary of any possible conflicts[7].

4.2 Safety

There are a few main safety concerns with our project. Namely with all electronics there is danger of electrical shock. As such we will work to create a non conductive housing for the electrical components to prevent injuries or accidents of this nature. This too comes with considerations for battery safety. We must be cautious when working with such components as they are prone to cause major

accidents when not kept a watchful eye over. We will work to make sure batteries are in good, working condition when in use and stored away properly when not. This will all ensure a safe workplace for our team and others around us. We will also consider implementing temperature sensors and a hard limit switch to turn off the device in case the battery reaches hazardous conditions.

As a final safety concern, we recognize that there is a potential issue for people with long hair. Our project contains strong turning motors with crevesaces that allow for hair to get caught. Users with longer hair are at a higher risk for an accident of this nature. To prevent this as much as possible, we will work to minimize gaps in the hardware and warn users of this hazard when applicable.

5 | Conclusion

5.1 Successes and Challenges

Overall, the project was quite successful. While the delays with the PCBs and the issues with 3D printing prevented us from having the perfect product, the most important goal was still achieved. We were able to develop a product that could solve and scramble itself. Obviously, this felt a little unsatisfactory without it all being perfectly contained within a cube. However, there was still a significant amount of great work that was done. First and foremost, the PCB designs were fantastic. Barring issues with the screws grounding the PCB at times, they functioned as perfectly as we could have expected them to. The printing design for the center face nubs also ended up being far more precise than we had initially anticipated. In what was expected to be a several week long project of printing, testing, and redesigning the center face to work properly, we got it nearly on the first try. Altogether, the switch module, motor, and center face combo was a major success. It allowed us to have precision motor control as well as rotational readings.

The software component of this project was also a major success. While we did not have enough time to work through optimized solving algorithms for the cube like we originally wanted to, we also understood that was a long shot goal and knew backtracking would be much more realistic. In the end, the backtracking worked fantastically. In what may seem like a simple idea, the software implementation for the backtracking algorithm ended up being far more complicated than we had thought. Despite this, we were able to have a consistently functional backtracking algorithm with motor control for the cube. The user movement detection also worked incredibly well. The cube is able to detect every move that a user performs on the cube and stores it into memory well. All of this together combines to help us achieve the lofty goal of self solving and scrambling.

The biggest challenges associated with the project ended up being 3D printing and delays. While we did expect 3D printing to be a big hurdle, we did not realize just how big it would be. Ironically, the relatively simple design of the corner and side pieces ended up being the most troublesome prints. The low quality of the printer created major tolerance issues when rotating the cube. Because of this, unless the cube was rotated to absolute perfection, it would get caught on itself and prevent further rotations. As a result we had to scrap the corner and side pieces for the final demo.

The delays in our project caused significant problems for our development timeline. We could not perform almost any of the major testing that we needed to do without the PCBs. In a much more obvious way, we also could not implement the control module and power system at all due to the PCB delays. It simply was not possible to do the testing and verification for the entirety of the subsystem given the PCB came in within a few days of the final project demo. While we attempted to put the subsystem together, we failed in the sense that we ran out of time to complete all the testing to feel confident that it would work with consistency.

5.2 Future Revisions

Most of the revisions that could be done revolve around simply having more time. The addition of the control module and power system would likely be able to be implemented given even just one more extra week. However, that would simply be completing the rest of our base project. In terms of additional revisions that we did not initially have as a part of our base project, the largest would be optimized solving algorithms. We would want to program the cube to read the cube state and use different solving strategies to optimize its solve time. This would require a lot more time invested into the software behind the cube, but could result in vastly reduced solve times.

One of the more minor revisions that could be made would be adding bevels to the 3D print. The main issue with the 3D prints was that the quality was too low. This can be partially worked around by editing bevels onto the corners of each of the pieces. This would allow for the tolerances to increase and make it possible to turn the cube even with suboptimal turns.

Another minor change would be to edit the switch module PCB to prevent the grounding issue. This is very straight forward as all we would have to do is move some of the trace lines to be further away from the screw down points. Despite the simplicity of this change, it would have prevented quite a bit of headache during the development process and would also make the entire system more reliable going forward.

5.3 Broader Impact

In seeing many of the other projects, one might believe that ours was not necessarily as impactful as others. With so many of them being directly to human health and safety, their impact can be directly seen. While our project does not contain that explicit help towards human health and safety, we believe that our project can still help people in a positive way. While a self solving/scrambling Rubik's cube may not change the world, its impact should not be underestimated. The Rubik's cube puzzle itself has long since been helpful in providing recreational

entertainment to the masses for years. Our project will help others pursue their passion and help others begin their journey into a wonderful community of great people all devoted towards the goal of solving the world's most popular puzzle. With that, there is simply no way of measuring the possible positive impact on mental health of providing a loving and passionate community to join. What is an extremely low impact on world resources, can ultimately end up making an extremely high impact on people's passion. And for that, this project contains an infinite ceiling of positive impact for people around the world.

6 | References

[1] "The Inventor of the Rubik's Cube Took This Long to First Solve It." *CBCnews*, CBC/Radio Canada, 13 Mar. 2021, https://www.cbc.ca/radio/undertheinfluence/the-inventor-of-the-rubik-s-cube-took-this-long-to-first-solve-it-1.5945283#:~:text=Soon%2C%20Rubik's%20Cube%20become%20part,the%20pandemic%20has%20boosted%20sales.

[2] "God's Number Is 20." *God's Number Is 20*, http://www.cube20.org/.

[3] *How Much Do Electrical Engineer Jobs Pay per Hour? - Ziprecruiter* https://www.ziprecruiter.com/Salaries/Electrical-Engineer-Salary-per-Hour.

[4] *Description Pin Configuration - Diodes Incorporated*. https://www.diodes.com/assets/Datasheets/PI4IOE5V9535.pdf.

[5] *ZXBM5210 Description Pin Assignments - Diodes Incorporated*. https://www.diodes.com/assets/Datasheets/ZXBM5210.pdf.

[6] "What We Offer." *Illinois MakerLab*, https://makerlab.illinois.edu/pricingservices/#printing.

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

Appendix A | Requirements and Verification Tables

Switch Module:

Requirement	Verification
Switch must be activated when each face is aligned 90 degrees	Rotate face 2 full rotations in either direction, ensure switch is closed when face is within 5 degrees of alignment and is open when the face is misaligned.
*Motors must have enough torque to turn the cube (est. 0.5 N*M)*	Adjust spring tension until current required to drive cube is within recommended current limit of motors (250 mA)
Motor driver must supply enough current to the motors without overloading	Measure current consumption of motors while turning cube, should not exceed 1A

Power Subsystem:

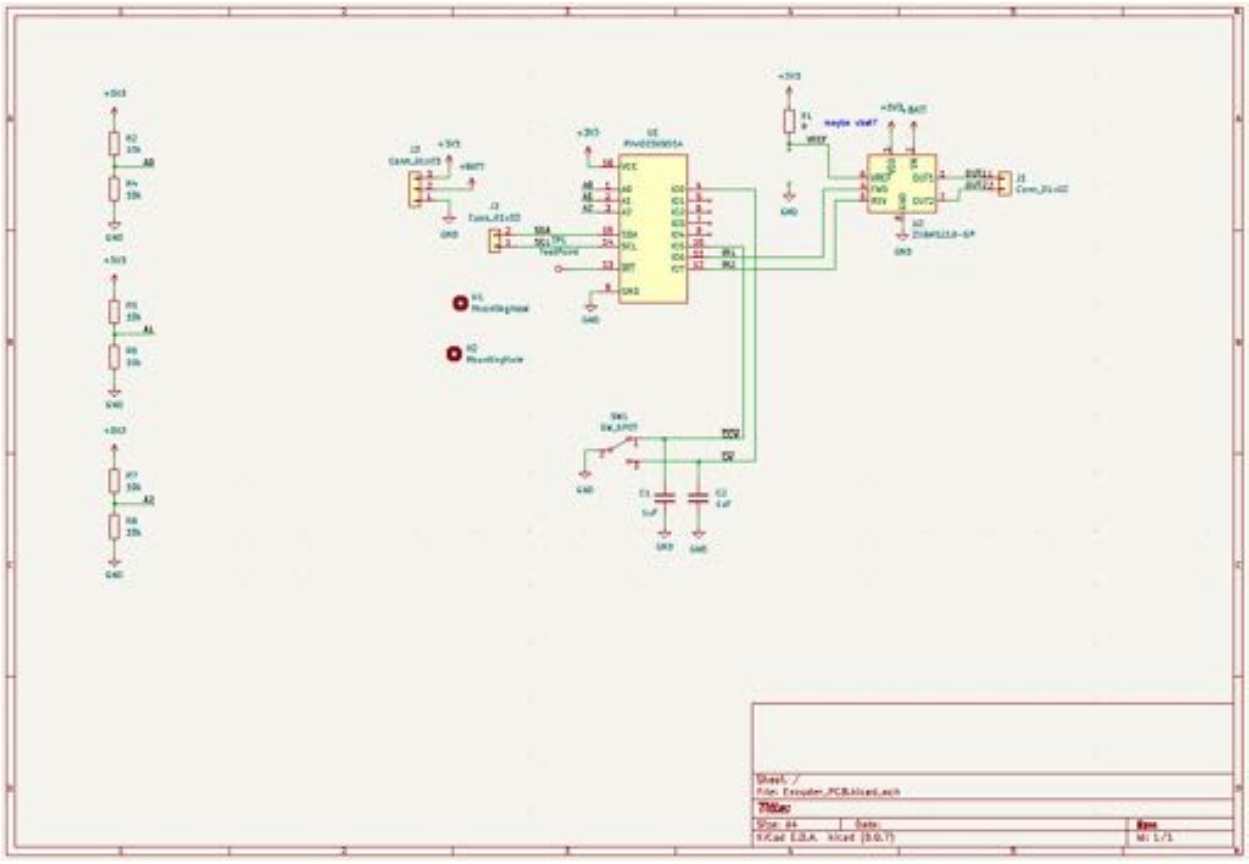
Requirement	Verification
Batteries must supply enough current for the motors and control system	Measure the current consumed by the motors and control system using a bench power supply, ensure rated battery current is above that.
Battery voltage must be within 6.0 to 8.0v with 1A output	Measure voltage of battery while drawing 1A.
Voltage Regulator must be able to supply 3.0v to 3.6v with enough current to the ICs	Measure the control system only and ensure the rated current from the regulators is above that.

Software Subsystem:

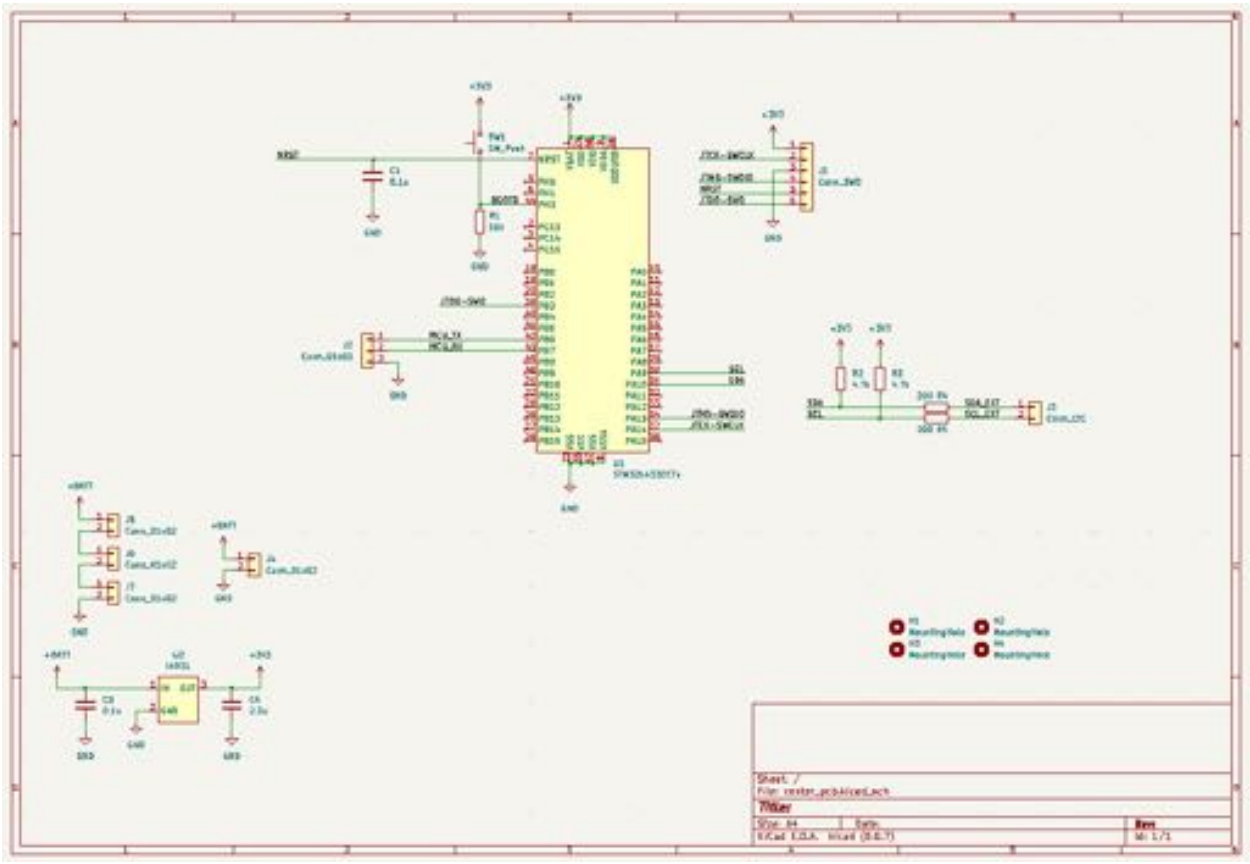
Requirement	**Verification**
Must be able to distinguish between a full face rotation from all other partial rotations	Will check the current state of the cube of the internal software system to see if it matched the outer cube appearance during a simulated partial turn
Must be able to keep track of states of full cube	Will check the current state of the cube in memory against the physical state of the cube after numerous rotations, and resets.
Must be able to identify user control rotations	Will check the current flag of the user interaction to see if it identifies as user control rotation
Must be able to scramble cube randomly	Match cube face rotations to random number generated rotation string
Should be able to solve the cube from any state it is in	From the unsolved rubix state, trigger the solve algorithm and check that the end state results in solved faces.

Appendix B | Circuit Schematics

Switch Module Schematic:

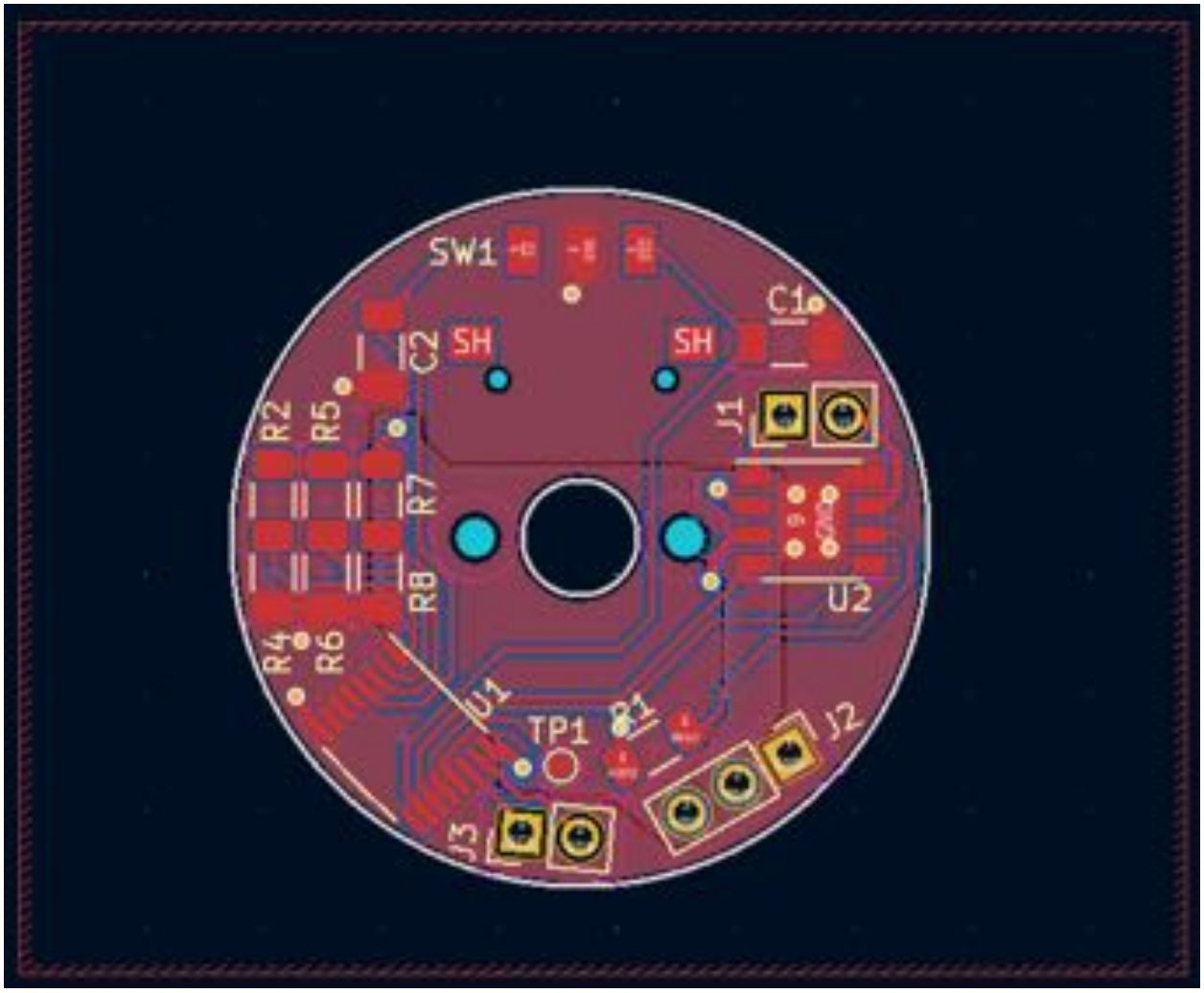


Control Module Schematic:

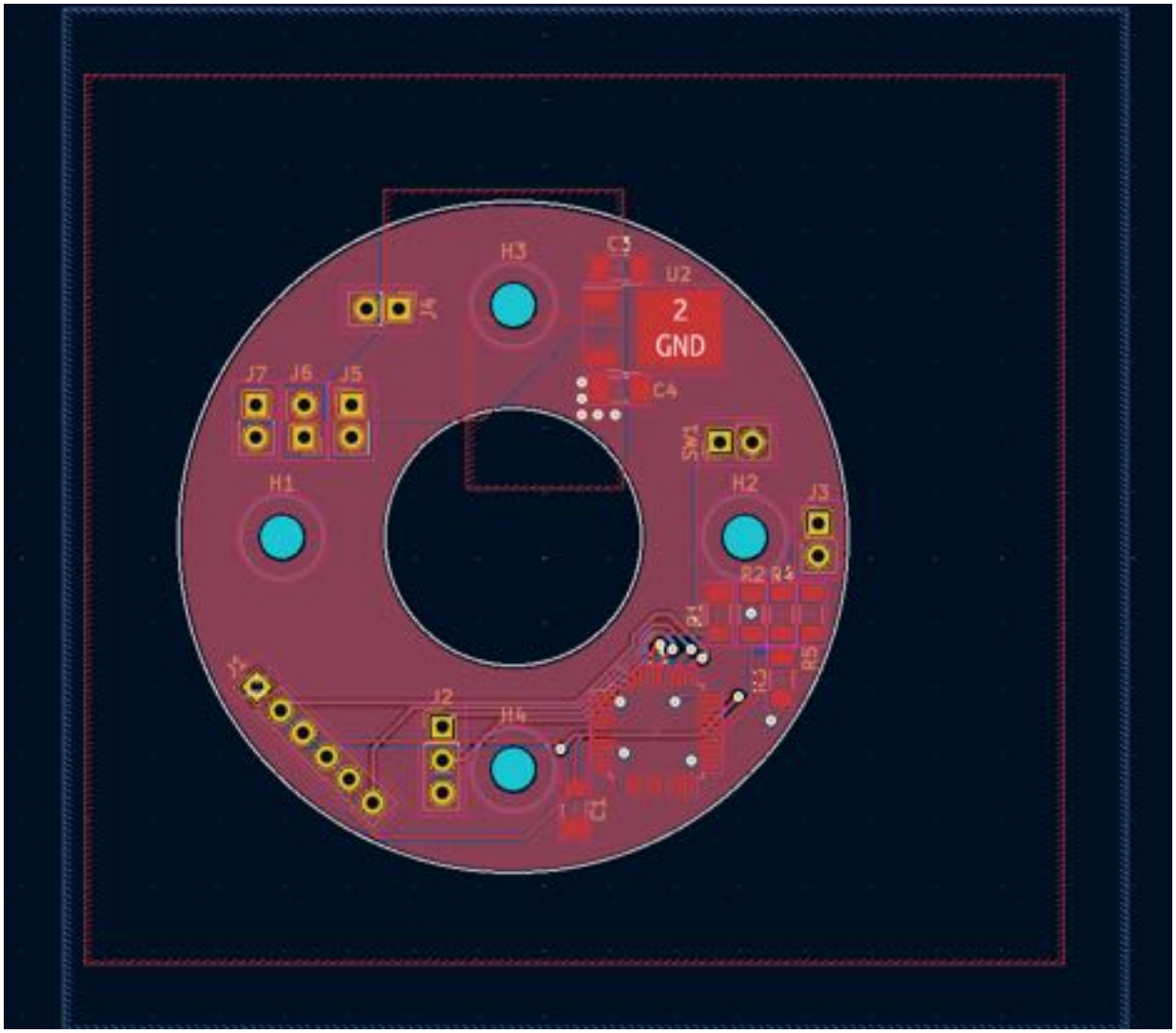


Appendix C | PCB Schematics

Switch Module PCB:



Control Module PCB:



Appendix D | Code

Link to Project Gitlab:
https://gitlab.engr.illinois.edu/walteru2/ece-445-senior-design

```
extern "C"
{
#ifdef __GNUC__
#define PUTCHAR_PROTOTYPE int __io_putchar(int ch)
#else
#define PUTCHAR_PROTOTYPE int fputc(int ch, FILE *f)
#endif

 PUTCHAR_PROTOTYPE
 {
   HAL_UART_Transmit(&huart2, (uint8_t *)&ch, 1, HAL_MAX_DELAY);
   return ch;
 }
}
class Movement
{
public:
 int motor;
 std::string direction;
 int time;
};

int motors[6] = {56, 57, 58, 59, 60, 61};
int validMoveTime[6] = {0, 0, 0, 0, 0, 0};
int randmotor[5] = {0, 1, 2, 4, 5};
std::vector<Movement> history;
HAL_StatusTypeDef ret;
uint8_t data[6];
uint8_t last_data[6];

uint8_t cmd[] = {3, 0x3f};
uint8_t read_cmd;
/* USER CODE END 0 */

void recordMoves(int motor, std::string direction)
{
 Movement move;
 move.motor = motor;
 move.direction = direction;
 move.time = HAL_GetTick();
 validMoveTime[motor] = move.time;
 history.push_back(move);
}

// controls motor movement and sends signals to pin
```

```
void motorControl(int motor, std::string state)
{

 if (state == "forward")
 {
   printf("---- STARTED TURNING BACKWARDS----\r\n");

   uint8_t addr = 0;
   int flag = 0;
   int firstCommand = 0;

   ret = HAL_I2C_Master_Transmit(&hi2c1, (motors[motor] << 1) + 0, &addr, 1, 1000);
   if (ret != HAL_OK)
   {
     printf("Theres been a Transmit HAL error!!!!!!!\r\n");
     HAL_I2C_Master_Transmit(&hi2c1, (motors[motor] << 1) + 0, &addr, 1, 1000);
   }
   while (1)
   {
     // Telling to read from input register
     ret = HAL_I2C_Master_Receive(&hi2c1, (motors[motor] << 1) + 0, &data[motor], 1, 1000);
     printf("Motor: %d,turning Backward, Current Data %x Last Data %x \n\r", motor,
data[motor], last_data[motor]);
     if (ret != HAL_OK)
     {
       printf("Theres been a Recieve HAL error!!!!!!!\r\n");
     }
     printf(" inverted data: %x \r\n", (~data[motor]));

     if (~data[motor] & L_MASK)
     {
       printf("Detected Backward Turn \r\n");
       if (flag == 1)
       {
         printf("We are done spinning\r\n");
         break;
       }
     }
     else if (~data[motor] & R_MASK)
     {
       printf("detected Forward Turn\r\n");
       if (flag == 1)
       {
         printf("We are done spinning\r\n");
         break;
       }
     }
     else
     {
       flag = 1;
     }
```

```
      cmd[0] = 1;            // this is say to preform a write
      cmd[1] = REV & (~FWD); // this is saying to set bits 2,0 to 1,0 respectively
      // moves the motor backward
      ret = HAL_I2C_Master_Transmit(&hi2c1, (motors[motor] << 1) + 0, cmd, 2, 1000);
      if (ret != HAL_OK)
      {
        printf("Theres been a Transmit HAL error!!!!!!!\r\n");
      }
      firstCommand = 1;
      ret = HAL_I2C_Master_Transmit(&hi2c1, (motors[motor] << 1) + 0, &addr, 1, 1000);
      if (ret != HAL_OK)
      {
        printf("Theres been a Transmit HAL error!!!!!!!\r\n");
      }

    }
    // After the break statement we break the motor
    cmd[0] = 1;
    cmd[1] = 0xff;
    // break mode
    ret = HAL_I2C_Master_Transmit(&hi2c1, (motors[motor] << 1) + 0, cmd, 2, 1000);
    if (ret != HAL_OK)
    {
      printf("Theres been a Transmit HAL error!!!!!!!\r\n");
    }

    printf("---- STOPPED TURNING ----\r\n");
  }
  else if (state == "backward")
  {
    printf("---- STARTED TURNING BACKWARDS----\r\n");

    uint8_t addr = 0;
    int flag = 0;
    int firstCommand = 0;

    ret = HAL_I2C_Master_Transmit(&hi2c1, (motors[motor] << 1) + 0, &addr, 1, 1000);
    if (ret != HAL_OK)
    {
      printf("Theres been a Transmit HAL error!!!!!!!\r\n");
    }

    while (1)
    {
      // Telling to read from input register
      ret = HAL_I2C_Master_Receive(&hi2c1, (motors[motor] << 1) + 0, &data[motor], 1, 1000);
      printf("Motor: %d,turning Backward, Current Data %x Last Data %x \n\r", motor,
data[motor], last_data[motor]);

      if (ret != HAL_OK)
      {
        printf("Theres been a Receive HAL error!!!!!!!\r\n");
```

```
    }
    printf(" inverted data: %x \r\n", (~data[motor]));

    if (~data[motor] & L_MASK)
    {
      printf("Detected Backward Turn \r\n");
      if (flag == 1)
      {
        printf("We are done spinning\r\n");
        break;
      }
    }
    else if (~data[motor] & R_MASK)
    {
      printf("detected Forward Turn\r\n");
      if (flag == 1)
      {
        printf("We are done spinning\r\n");
        break;
      }
    }
    else
    {
      flag = 1;
    }

    cmd[0] = 1;            // this is say to preform a write
    cmd[1] = (~REV) & FWD; // this is saying to set bits 2,0 to 1,0 respectively
    // moves the motor backward
    ret = HAL_I2C_Master_Transmit(&hi2c1, (motors[motor] << 1) + 0, cmd, 2, 1000);
    if (ret != HAL_OK)
    {
      printf("Theres been a Transmit HAL error!!!!!!!\r\n");
    }
    firstCommand = 1;
    ret = HAL_I2C_Master_Transmit(&hi2c1, (motors[motor] << 1) + 0, &addr, 1, 1000);
    if (ret != HAL_OK)
    {
      printf("Theres been a Transmit HAL error!!!!!!!\r\n");
    }
  }
  // After the break statement we break the motor
  cmd[0] = 1;
  cmd[1] = 0xff;
  // break mode
  ret = HAL_I2C_Master_Transmit(&hi2c1, (motors[motor] << 1) + 0, cmd, 2, 1000);
  if (ret != HAL_OK)
  {
    printf("Theres been a Transmit HAL error!!!!!!!\r\n");
  }
  printf("---- STOPPED TURNING ----\r\n");
}
```

```
 else if (state == "brake")
 {
   // send signal to pin for motor
   cmd[0] = 1;
   cmd[1] = 0xff;
   // break mode
   HAL_Delay(50);
   HAL_I2C_Master_Transmit(&hi2c1, (motors[motor] << 1) + 0, cmd, 2, 1000);
   HAL_Delay(50);
 }
 else if (state == "coast")
 {
   // send signal to pin for motor
   cmd[0] = 1;
   cmd[1] = 0xf0;
   // coast mode
   HAL_Delay(50);
   HAL_I2C_Master_Transmit(&hi2c1, (motors[motor] << 1) + 0, cmd, 2, 1000);
   HAL_Delay(50);
 }
}

void allControl(std::string state)
{
 motorControl(0, state);
 motorControl(1, state);
 motorControl(2, state);
 motorControl(3, state);
 motorControl(4, state);
 motorControl(5, state);
}

void randomize()
{
 //  set all motors to break
 printf("---------- RANDOMIZE STARTING ----------\n\r");
 allControl("break");
 // while loop to 20 moves
 int count = 0;
 while (count < 20)
 {
   // rand call for which motor to control
   int motor = rand() % 5;
   // rand call determines which direction to turn motor
   int direction = rand() % 2;


   if (direction == 0)
   {
     motorControl(randmotor[motor], "backward");
     recordMoves((randmotor[motor]), "backward");
   }
```

```
    else
    {
      motorControl(randmotor[motor], "forward");
      recordMoves(randmotor[motor], "forward");
    }
    count = count + 1;
  }
  // setting all motors to coast
  printf("---------- RANDOMIZE ENDING ----------\n\r");
  allControl("coast");

   for (int x = 0; x < 6; x++)
  {
    data[x] = 0;
    // Telling to read from input register
    HAL_I2C_Master_Transmit(&hi2c1, (motors[x] << 1) + 0, &data[0], 1, 1000);

    /* Get the intial state of all the switches for the motors*/
    HAL_I2C_Master_Receive(&hi2c1, (motors[x] << 1) + 0, &last_data[x], 1, 1000);
  }
}

void solve()
{
  // Setting all motors to break
  printf("solve\r\n");
  allControl("brake");

  while (!history.empty())
  {
    Movement temp = history.back();
    printf("Current record is motor %d in the %s direction\r\n", temp.motor,
temp.direction.c_str());
    if (temp.direction == "backward")
    {
      printf("Solve is moving motor %d forwards\n\r", temp.motor);
      motorControl(temp.motor, "forward");
    }
    else
    {
      printf("Solve is moving motor %d backwards\n\r", temp.motor);
      motorControl(temp.motor, "backward");
    }
    // popping history
    history.pop_back();
  }
  printf("Solve sequence is done\n\r");

  allControl("coast");
   for (int x = 0; x < 6; x++)
  {
    data[x] = 0;
```

```
   // Telling to read from input register
   HAL_I2C_Master_Transmit(&hi2c1, (motors[x] << 1) + 0, &data[0], 1, 1000);

   /* Get the intial state of all the switches for the motors*/
   HAL_I2C_Master_Receive(&hi2c1, (motors[x] << 1) + 0, &last_data[x], 1, 1000);
 }
}

void checkInterface()
{
 if (history.size() >= 2)
 {
   printf("Checking Interface\r\n");
   if (history[history.size() - 1].motor == SOLVE_MOTOR && history[history.size() - 2].motor
== SOLVE_MOTOR)
   {
     if (history[history.size() - 1].direction != history[history.size() - 2].direction)
     {
       int timeDif = history[history.size() - 1].time != history[history.size() - 2].time;
       if (timeDif <= INT_TIME)
       {
         // Interface moves were recorded so they must be popped
         history.pop_back();
         history.pop_back();

         // Delay so motors don't immediately impede movement
         HAL_Delay(400);
         solve();
       }
     }
   }
   if (history[history.size() - 1].motor == RAND_MOTOR && history[history.size() - 2].motor
== RAND_MOTOR)
   {
     if (history[history.size() - 1].direction != history[history.size() - 2].direction)
     {
       int timeDif = history[history.size() - 1].time != history[history.size() - 2].time;
       if (timeDif <= INT_TIME)
       {
         history.pop_back();
         history.pop_back();
         // Delay so motors don't immediately impede movement
         HAL_Delay(400);
         randomize();
       }
     }
   }
 }
}

void printHistory()
{
```

```
 int size = history.size();
 printf("------ History Start ------\r\n");
 for (int i = 0; i < size; i++)
 {
   printf("Motor %d moved %s\n\r", history[i].motor, history[i].direction.c_str());
 }
 printf("------ History End ------\r\n");
}
int main(void)
{
 /* USER CODE BEGIN 1 */

 /* USER CODE END 1 */

 /* MCU Configuration--------------------------------------------------------*/

 /* Reset of all peripherals, Initializes the Flash interface and the Systick. */
 HAL_Init();

 /* USER CODE BEGIN Init */

 /* USER CODE END Init */

 /* Configure the system clock */
 SystemClock_Config();

 /* USER CODE BEGIN SysInit */

 /* USER CODE END SysInit */

 /* Initialize all configured peripherals */
 MX_GPIO_Init();
 MX_USART2_UART_Init();
 MX_I2C1_Init();
 /* USER CODE BEGIN 2 */
 srand(HAL_GetTick());
 printf("---- Program Starting ----\n\r");
 // setting which pins will be input/output
 for (int x = 0; x < 6; x++)
 {
   HAL_I2C_Master_Transmit(&hi2c1, (motors[x] << 1) + 0, cmd, 2, 1000);

   HAL_Delay(100);

   // reading the pin configuration that was just set
   read_cmd = 0;
   HAL_I2C_Master_Receive(&hi2c1, (motors[x] << 1), &read_cmd, 1, 1000);
   printf("Read config: %x\n\r", read_cmd);

   data[x] = 0;
   // Telling to read from input register
   HAL_I2C_Master_Transmit(&hi2c1, (motors[x] << 1) + 0, &data[0], 1, 1000);
```

```
}
/* USER CODE END 2 */

/* USER CODE BEGIN WHILE */
allControl("coast");
for (int x = 0; x < 6; x++)
{
  data[x] = 0;
  // Telling to read from input register
  HAL_I2C_Master_Transmit(&hi2c1, (motors[x] << 1) + 0, &data[0], 1, 1000);

  /* Get the intial state of all the switches for the motors*/
  HAL_I2C_Master_Receive(&hi2c1, (motors[x] << 1) + 0, &last_data[x], 1, 1000);
}

while (1)
{
  // checking all I2C controlling all motors
  for (int j = 0; j < 6; j++)
  {
    HAL_I2C_Master_Receive(&hi2c1, (motors[j] << 1) + 0, &data[j], 1, 1000);
    if (data[j] != last_data[j])
    {
      if (HAL_GetTick() - validMoveTime[j] > DEBOUNCE_TIME)
      {
        // check state relative to the last state
        if (~data[j] & L_MASK)
        {
          printf("L\r\n");
          recordMoves(j, "backward");
          printf("backward\r\n");
          printHistory();
          checkInterface();
        }
        else if (~data[j] & R_MASK)
        {
          printf("R\r\n");
          recordMoves(j, "forward");
          printf("forward\r\n");
          printHistory();
          checkInterface();
        }
        else
        {
          printf("rotating\r\n");
        }

        uint8_t addr = 0;
        // telling the I2C which register we want to read from
        // reading from the input register
        HAL_I2C_Master_Transmit(&hi2c1, (motors[j] << 1) + 0, &addr, 1, 1000);
```

```
        printf("Received Data: %x\r\n", data[j]);
        last_data[j] = data[j];
      }
    }

    /* USER CODE END WHILE */

    /* USER CODE BEGIN 3 */
  }
  /* USER CODE END 3 */
}
}
```