MTE 100/121 Course Project Report

UNIVERSITY OF
Waterloo

Department of Mechanical and Mechatronics Engineering

A Report Prepared For:

The University of Waterloo

Prepared By:

Andrew Guo (20992753), Akil Pathiranage (21030408), Alison Thompson (21005124), and Francis Bui (21030408)

150 University Ave W.

Waterloo, Ontario, N2N 2N2

December 6, 2022

Summary

The final robot created by this team can consistently reach various “delivery locations” autonomously to mimic a real-world delivery scenario. In order to meet this challenge, a custom differential swerve drive train is used, this allows the robot to move in any direction without the need for turning the chassis. This was chosen to improve maneuverability in tight spaces.

Major constraints were the short amount of time to complete the project, the physical and computational limitations of the required use of the LEGO EV3, and the poor performance and limited functionality of the required programming language RobotC. Success criteria can be summarized as: the robot can consistently reach the given delivery locations, the swerve modules operate as expected, and all constraints were overcome.

Programming restraints were overcome via thorough planning, exploration of alternative solutions, and repeated trial and error. The main code process is broken down and managed by individual files, structs, subtasks, and functions to maintain an easy to read and highly modular system while also completing several complex tasks in a highly efficient manner.

Due to the selected fast-paced and iterative design strategy, the full swerve drive was created of modular parts and subassemblies that were replaceable and quick to manufacture. At the center of the design is the differential gearbox which enables omnidirectional movement. The result is an extremely compact driving platform, that is more robust than traditional drivetrains.

Overall, the completed project met all criteria and constraints. Hardware recommendations include spending more time iterating mechanical design, finding higher resolution printers to 3D print gears. Software recommendations include completing the development of a teleoperated mode as well as 2D path plotting software.

Acknowledgements

The initial design of the robot was based on a design by FIRST Tech Challenge Team 11115 – Gluten Free [1]. The CAD models of the EV3 and other LEGO components were found on GrabCAD via account “dk” [2].

Table of Contents

Table of Figures

Table of Tables

1.0 - Scope

This project started as an open-ended assignment with the primary requirement to solve a problem of our choosing using a LEGO Mindstorms EV3 kit. The team chose to tackle the challenge of autonomous delivery and the final robot, affectionately named Magnemite, autonomously reaches a series of set locations, within a radius of 5 cm, to represent potential delivery locations. This is accomplished mechanically using a custom differential swerve drivetrain, a drivetrain with two “swerve modules” that allow each of the two powered wheels to both spin as a traditional wheel would to translate the whole robot forward and backward, as well as allows for controlled change of the angle that each wheel is pointing, giving it the additional ability to drive sideways or diagonally. This is discussed in more detail in the **3.0 - Mechanical Design** and Implementation section.

Magnemite uses 2 ultrasonic sensors, a gyroscope, and the 4 built in motor encoders for collecting data that guides the successful completion of the delivery routes. It interacts with its environment by driving set amounts of time in pre-programmed directions to reach the various “delivery locations” along a route. There are four motors on Magnemite and each powers a single differential gear. The direction of travel and the angle of the robot also reacts should its path be interrupted. As discussed further in **4.0 - Software Design** and Implementation, the ultrasonic sensors are used to detect objects within a range of 10cm. When an obstacle is detected, the robot enters its shutdown procedure.

The robot’s shutdown procedure is also activated after a timeout period corresponding to each route. When either of these conditions are true, the robot rotates the wheels to 45 degrees inwards, immediately stopping momentum from carrying it any further. More details can be found in the **4.0 - Software Design** and Implementation section.

The initial report indicated that the purpose of the project was to allow an EV3 to drive in any horizontal direction without the need to rotate the entire robot. After reviewing with TA’s, the scope was changed to tackle the specific task of autonomous delivery so that should the proposed complex design not pan out in time, a complete a project that meets the minimum requirements of the problem was still achievable. At this meeting a goal of reaching the delivery locations perfectly each time was established, however as the robot came together and code was tested it became clear that that goal was too ambitious given the time restraint. The decision was then made to adjust the goal to reach each location within 5cm in any direction which was far more realistic.

2.0 - Constraints and Criteria

This project required the following general requirements to be met: at least 3 motors are used, at least 4 inputs are used, the timer is used, and the robot's code is written in RobotC. For the formal presentation the following broad success criteria were added to measure if the identified problem of autonomous delivery had been met: the robot drives a given distance reliably and the robot performs the deliveries entirely autonomously.

The following constraints were also identified during the formal presentation: time, the EV3, and using RobotC. The time constraint arose due to the complexity of the mechanical system the team aimed to build and the complexity of the software required to properly operate this design. The EV3 itself, as well as limitations caused by using RobotC, contributed to the overall complexity of the final software and magnified the challenges posed by the time constraint. These constraints remained unchanged throughout the project, though the challenge posed by time did become amplified the closer it got to the deadline.

For the robot demo, the success criteria was revisited to be both more specific and align with what the final robot was able to accomplish. In particular, the decision was made to allow for 5cm of leeway around each target to account for mechanical and software imperfections that there simply was not time to iterate out.

The final success criteria are:

- Robot reaches all path checkpoints consistently (within 5cm)
- Robot performs the "deliveries" entirely autonomously
- The challenges posed by time have been overcome (robot operational before demo day)
- The challenges posed by the mechanical limitations of the EV3 have been overcome (project was completed without relying on other motors or controller)
- The challenges posed by coding in RobotC have been overcome (project was completed using RobotC)

- Swerve modules can rotate (left/right)
- Swerve modules can drive (forward/reverse)
- At least 3 motors have been used
- At least 4 inputs have been used
- Timer has been used

3.0 - Mechanical Design and Implementation

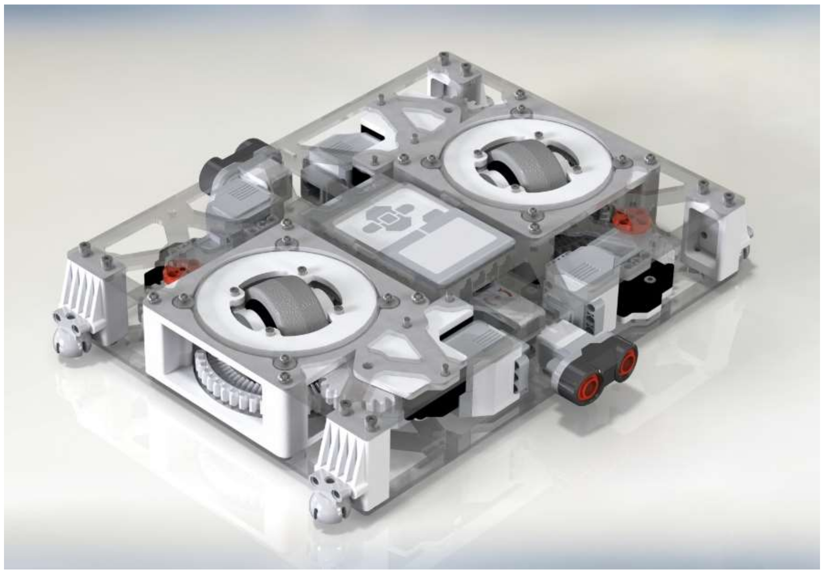

Figure 1 - Isometric View of the Full Assembly

For the challenge of reaching delivery locations autonomously, differential swerve drive was selected due to its versatility as a driving platform and efficiency in reaching various locations. Normally to navigate to various locations traditional “tank-drive” robots must perform a combination of turns. However, given the use case of autonomous delivery and potential warehouse applications, space to turn maybe difficult or even impossible. By using an omnidirectional driving platform like swerve, the robot can simultaneously turn and drive the wheels individually which allows it to navigate tighter spaces than traditional drivetrains.

Project Magnemite is a 2-module differential swerve drive, a full isometric view is shown in Figure 1 above. This means there are 2 “swerve-modules” that each house a wheel powered by a differential mechanism. The differential mechanism enables the robot’s unique ability to both drive and turn its wheels at the same time. The differential can also be isolated to just turning the wheels or driving the wheels, allowing the robot to stop at a location, adjust the wheel angle, and then simply drive on the given angle to the next location.

To better understand the rest of this report it is best to clarify that, “turning the wheel” means rotating it from the above position, creating an angular offset from the direct heading of the robot, whereas “driving the wheel” means rotating the wheel itself, causing a displacement from the origin of the robot’s position.

A common alternative to differential swerve is co-axial swerve. In both versions of swerve drive, 2 motors are used per module. The major difference lies in the fact that unlike coaxial, differential swerve can allow for both motors to contribute to turning and driving, whereas coaxial swerve limits one motor for driving and the other for turning. Considering this project’s limited constraints and resources, differential swerve was selected over coaxial due to the advantages of utilizing both motors for turning as well driving, thereby getting the most power we could for each function.

3.1 - Swerve Modules

Each swerve module is composed of several main components, the differential gearbox, custom wheel assembly, and swerve housing as shown in the exploded view of the swerve module in Figure 2 below. One goal for the swerve modules was to make them modular and easy to switch/replace during the testing and iteration phases of the design cycle. The modules were also designed to be symmetrical and square which aids in assembly.

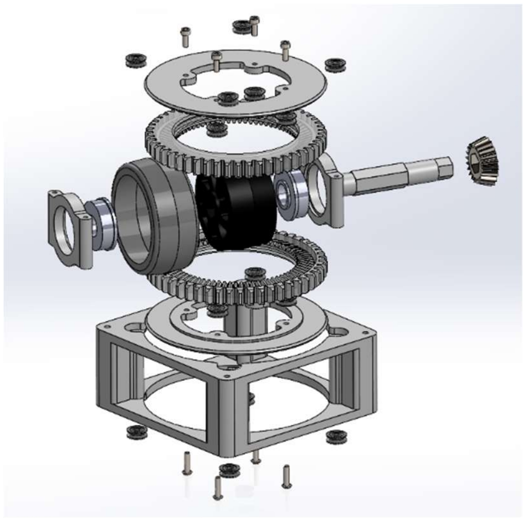

Figure 2 - Exploded View of Swerve Module

3.1.1 - Differential Gearbox

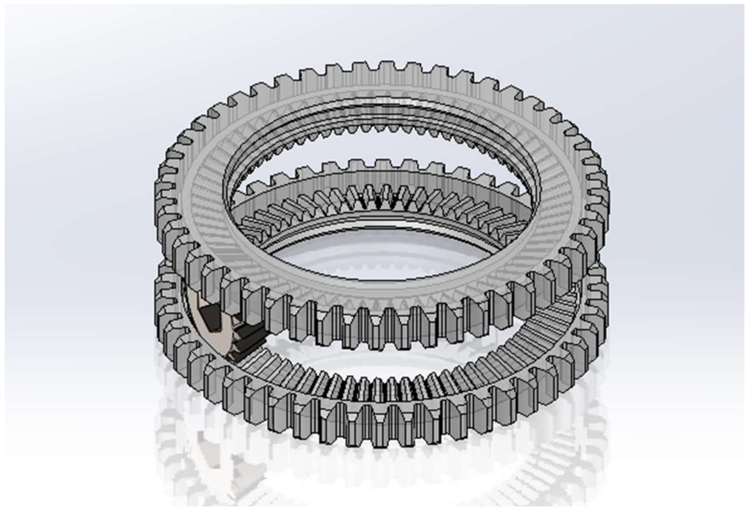

Figure 3 - Core of Swerve Module

In the core of the swerve module sits 2 differential gears and one bevel pinion gear, as seen in Figure 3 above. The differential gears have an internal bevel gear and an outer spur gear. Each differential gear is powered individually by one LEGO EV3 large motor with a 25-tooth spur gear. Depending on the speed and direction of each differential gear, the pinion gear will exhibit different behaviour as shown in Table 1 below.

Table 1 - Gear Speeds and Pinion Behaviour

Top Differential Gear Speed	Bottom Differential Gear Speed	Pinion Behaviour
100% Clockwise	100% Counterclockwise	100% drive No rotation around central axis
100% Clockwise	100% Clockwise	No drive 100% rotation around central axis
100% Counterclockwise	50% Clockwise	25% drive 75% rotation around central axis

There are 2 stages of reductions in the gearbox. The first is from the motor pinion (25-tooth) to the outer spur gear of the differential gear (50-tooth). This achieves an initial 2:1 reduction. Then from the bevel gear of the differential gear (70-tooth) to the pinion bevel gear (20-tooth), there is a 1:3.5 gear ratio. The total gear ratio of the differential is: 1:1.75. The LEGO EV3 Large motors can run at a maximum of 170rpm, meaning the pinion can spin at a maximum of 298rpm.The differential gears and pinions were printed in resin in order to ensure a high resolution and ensure appropriate tolerances.

3.1.2 - Wheel Assembly

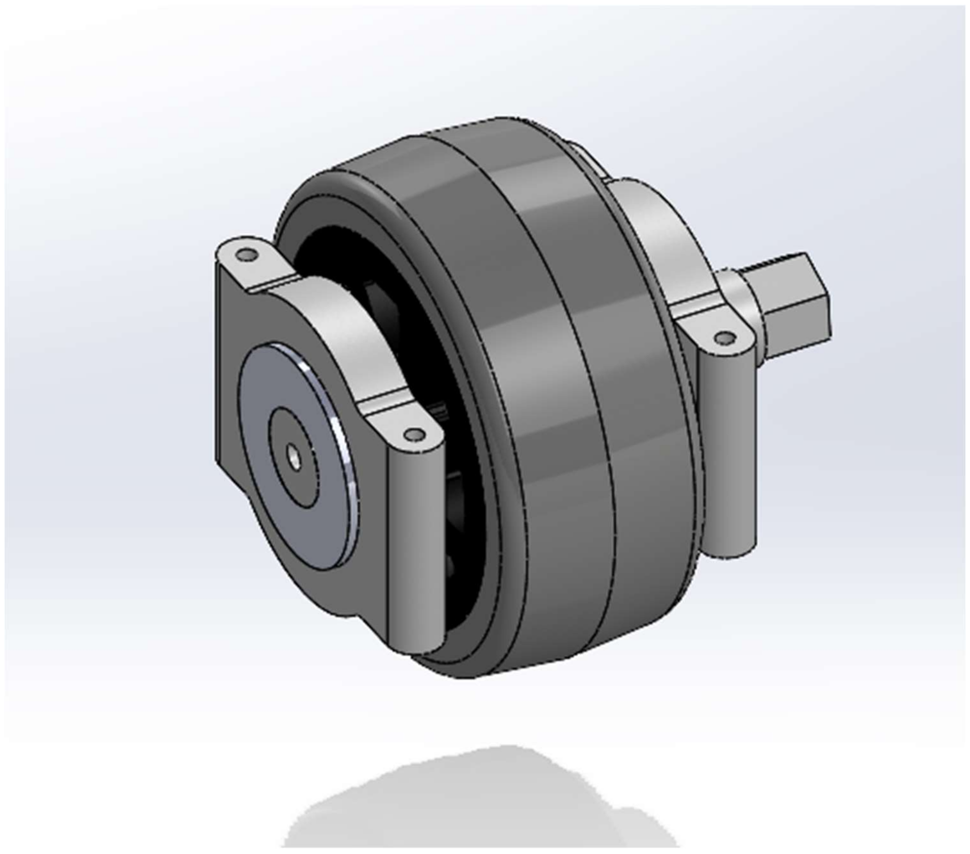

Figure 4 - Isometric View of the Wheel Assembly

The wheel assembly consists of a custom wheel grip, hub, shaft, stock bearings and their accompanying custom bearing blocks as seen above in Figure 4. Custom wheels were chosen to ensure the maximum wheel diameter (2.625in) was met which allows the wheel to fit into the differential gearbox assembly. Additionally, the wheel grip and hub were tapered to the center to maintain alignment during driving through increased surface speed in the center.

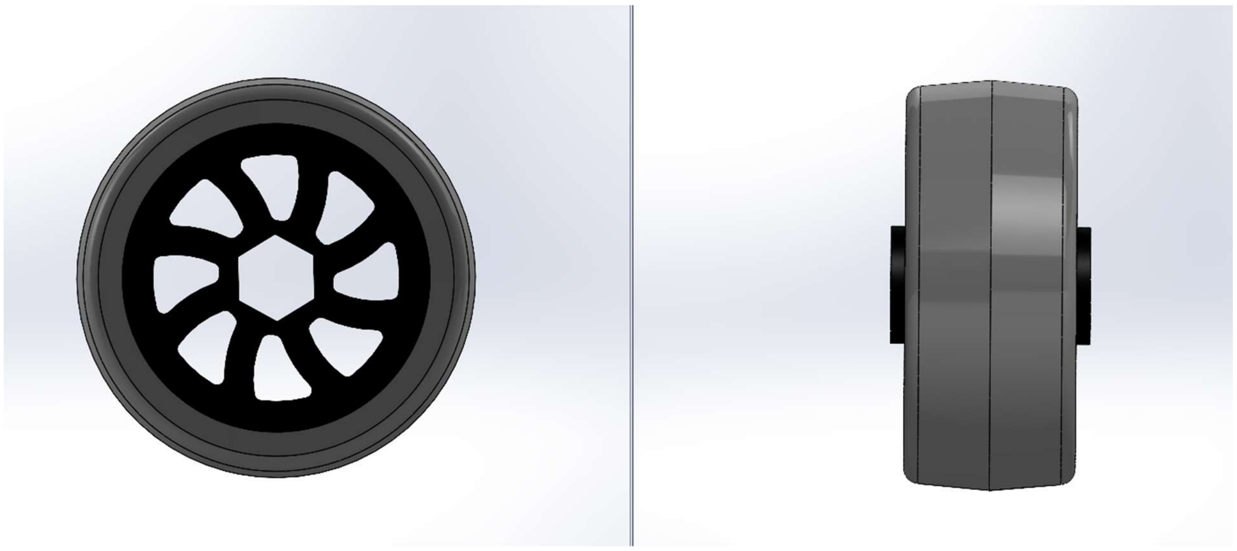

Figure 5 - Side and Front View of the Wheel Hub and Grip

The wheel hubs were printed out of ABS and the wheel grips were printed out of TPU. The assembly consisting of these parts is shown in Figure 5 above. After testing the smooth configuration of the TPU wheel grips, custom treads were melted into the outer surface using a soldering iron to increase traction between the wheel and the floor. The full assembly containing the wheel, v-groove bearings, bearings, and bearing blocks is shown in Figure 6 below.

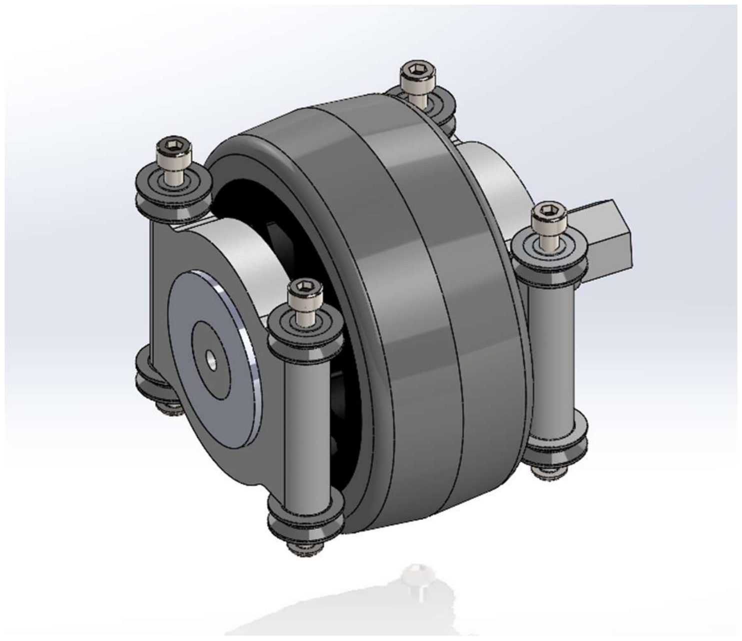

Figure 6 - Isometric View of Wheel Assembly with V-Groove Bearings

The bearing blocks support the main 6001-2RS ball bearing for the wheel as well as 4 V623ZZ v-groove bearings. These v-groove bearings mesh with an inner v-groove on the differential gears, highlighted in Figure 7 below, supporting the differential gearbox by reducing the load on the pinion and allowing the differential gears to move independently of the wheel and entire module.

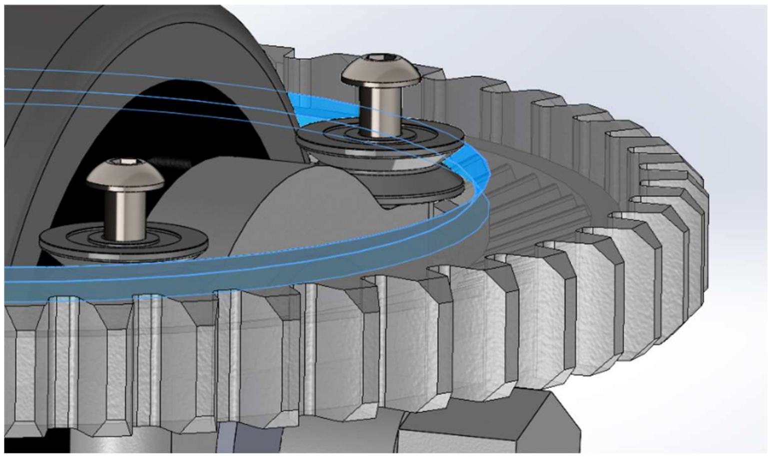

Figure 7 - Differential Gear V-Groove

These bearings are held in place by M3 socket head cap screws on the top of the robot and M3 button head screws on the bottom to increase ground clearance. These screws also hold the rail guards in place which have an exterior v-groove that meshes with the second set of v-groove bearings in the motor housing. This outer set of v-groove bearings allows the wheel assembly to rotate within the housing while supporting it vertically. The full swerve-module assembly is shown below in Figure 8.

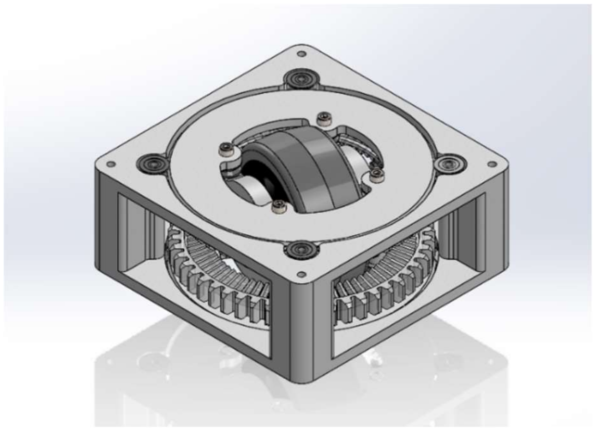

Figure 8 - Isometric View of Swerve Module

3.2 - Motor Housing

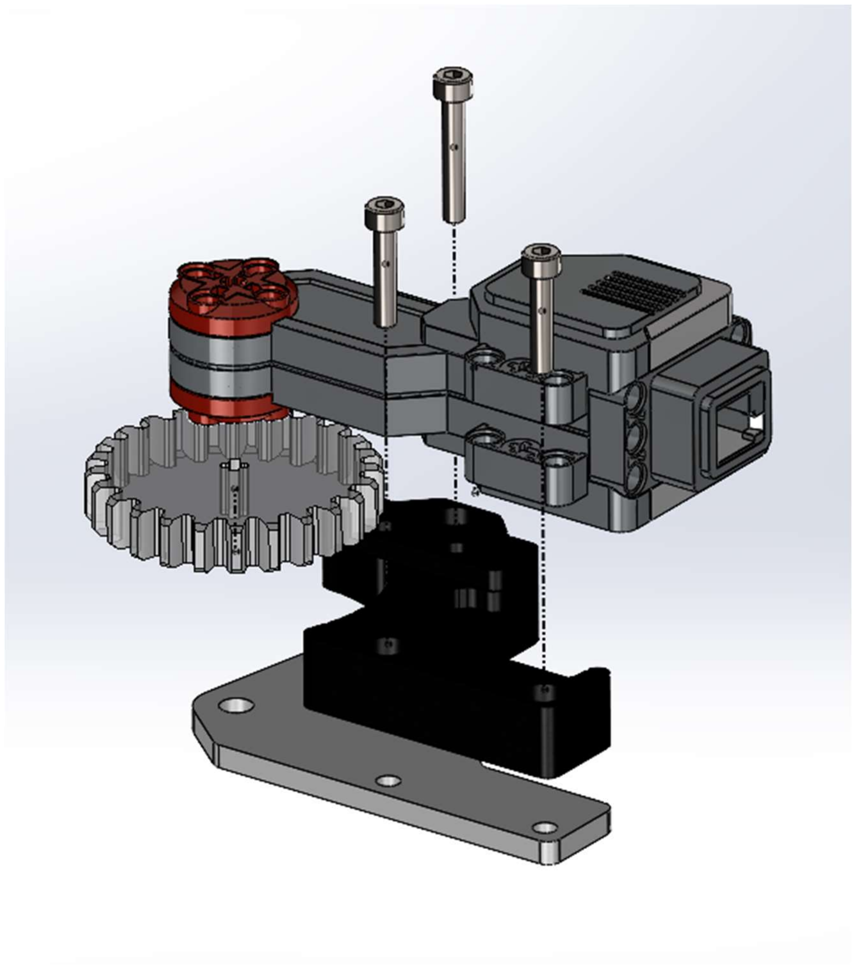

Figure 9 - Exploded View of Motor Assembly

The motor housing mounts the motor and pinion to the chassis as seen above in Figure 9. The motor connects to the 3D printed motor mounts using LEGO EV3 pins. The 25-tooth spur gear is sandwiched between the motor and the bottom plate of the motor mount. The result is a modular and compact assembly that can be easily replaceable and mounted to the chassis.

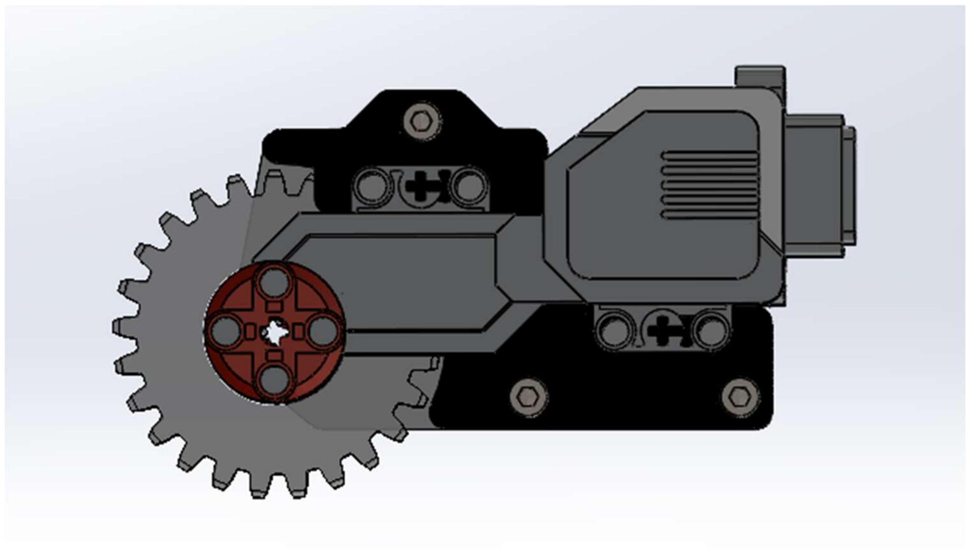

Figure 10 - Top View of Motor Assembly

M4 socket head cap screws and nuts are used to mount this assembly to the chassis as seen added to the motor mount assembly above in Figure 10 and added to the chassis assembly below in Figure 11. An important feature of the design was to invert them on opposite sides of the swerve module, so only one version needs to be designed and manufactured instead of both left and right versions.

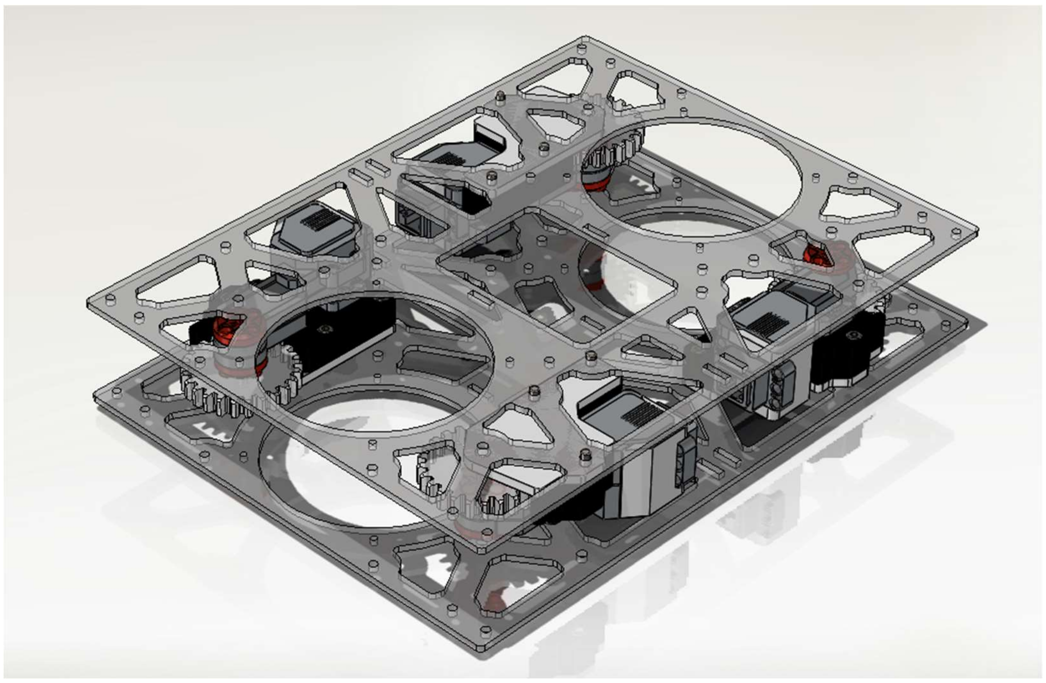

Figure 11 - Motor Assembly and Chassis

3.3 - Ball Caster Mounts

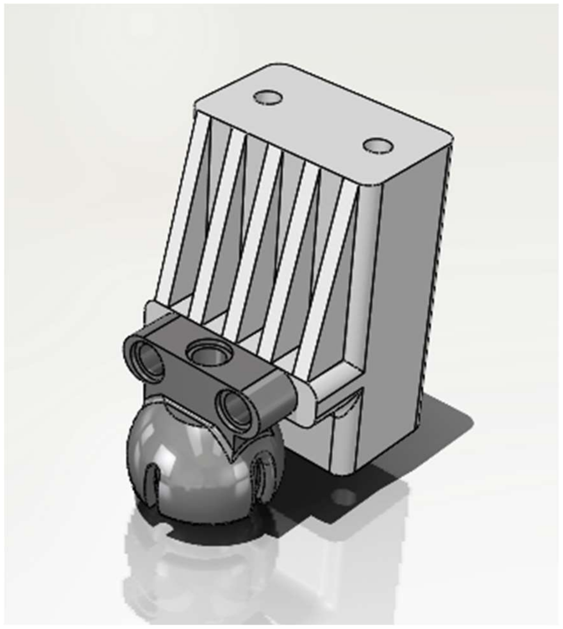

Figure 12 - Ball Caster Mounts

Initially the plan was to create custom corner omni-wheels in order to stabilize the swerve drive during operation. However, after setbacks during the manufacturing process, LEGO EV3 ball casters were selected instead as it reduced the amount of 3D printing that needed to be done and simplified the overall design significantly. The ball caster mounts, shown mated to the ball caster in Figure 12 above, also double as a standoff supporting the top and bottom plates of the chassis. The marble in the ball caster is raised 0.1375in compared to the main swerve drive wheels, as shown in Figure 13 below, giving the drivetrain an overall center-drop to aid in turning and to account for spots of varying height on the driving surface. The ball caster mount is attached to the chassis using M4 screws.

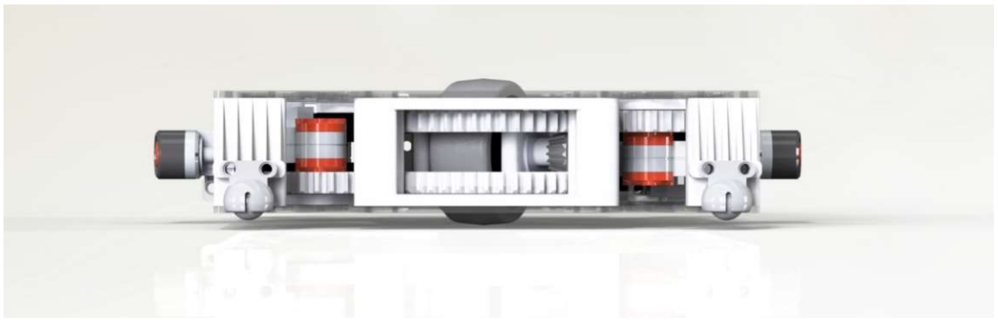

Figure 13 - Visualization of Center Drop

3.4 - Chassis & Sensor Mounts

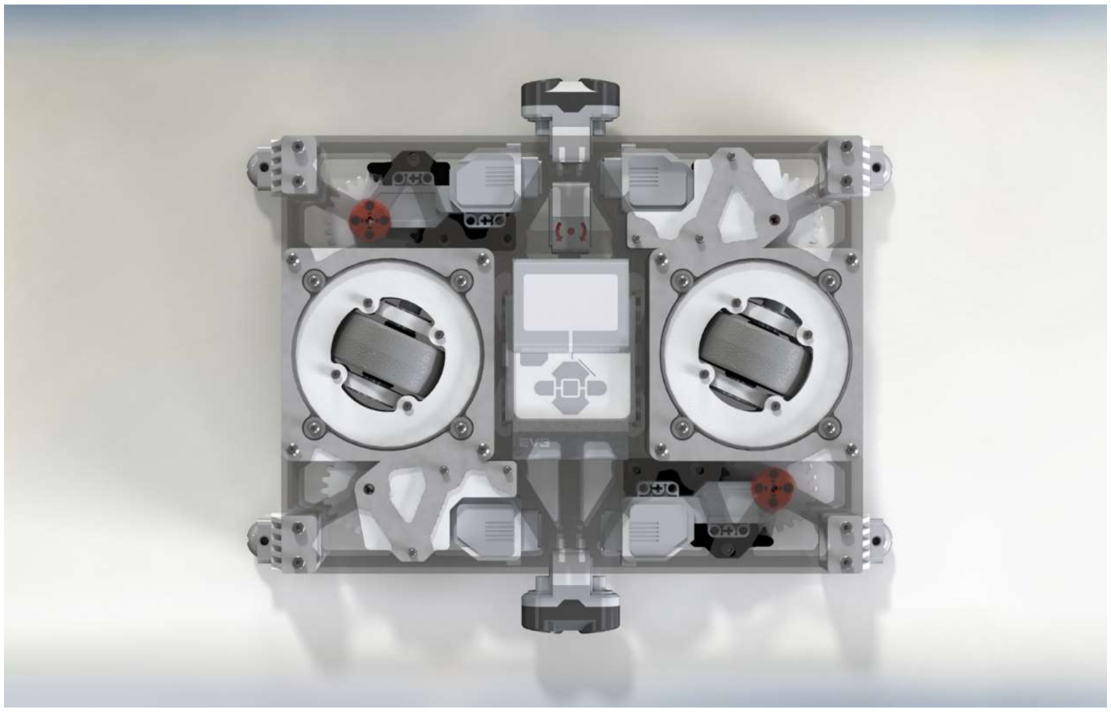

Figure 14 - Chassis with Sensors Mounted

The LEGO EV3 sensors, LEGO EV3 computer, both swerve modules and the ball caster mounts are arranged into 32cm x 24cm x 6cm box, as seen assembled above in Figure 14. The chassis plates and sensor mounts were laser cut out of 1/8" acrylic. Since weight was a concern for the project, weight reductions were cut out of the main plates to reduce the overall weight. During assembly, lubrication was added to all the gears and V Grooves which greatly reduced the friction of the gearboxes.

4.0 - Software Design and Implementation

4.1 - Software Breakdown

The code for the robot is broken down into multiple tasks, handling multiple structs simulating object orientation in RobotC. Multiple tasks are needed to create PID loops for speed, angle and offset. This is done by declaring PIDController structs, SwerveModule structs and the Robot Struct globally, allowing tasks to be run with different parameters. These tasks were run “simultaneously” through the task scheduler in the background while the main task was run.

4.1.1 – Main Task

The main task as seen in **Appendix F - Subsystems/Drive.c**, is broken down in Figure 15.

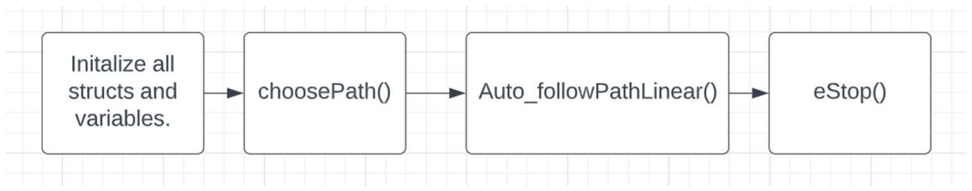

Figure 15 - Main Task Flowchart

4.1.2 – Choosing a Path

Beginning with the choosePath() function, this function takes in no parameters and has no return. It displays the paths to choose from on the EV3 and checks which button is pressed, then calls Auto_followPathLinear() with the selected path’s configuration. The breakdown can be seen in the flowchart in Figure 16 below. This function was written by Alison Thompson.

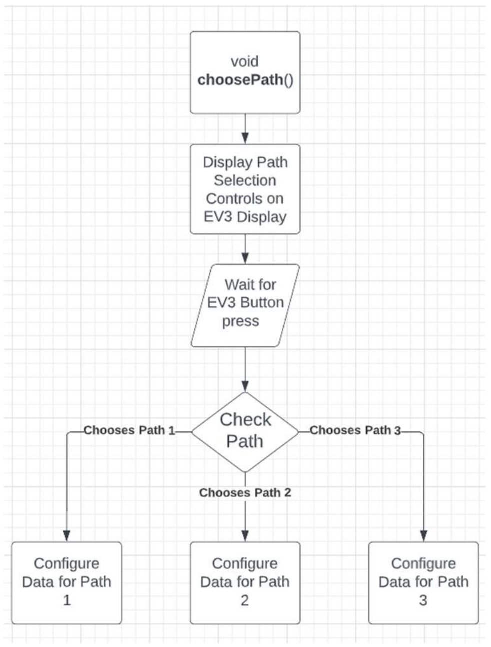
void choosePath()
Display Path Selection Controls on EV3 Display
Wait for EV3 Button press
Check Path
Chooses Path 1
Chooses Path 2
Chooses Path 3
Configure Data for Path 1
Configure Data for Path 2
Configure Data for Path 3

Figure 16 - Choose Path Function Flowchart

4.1.3 – Autonomous Path Following

Auto_followPathLinear(), located in Appendix F - Subsystems/Drive.c, returns nothing and takes in five parameters. It takes in four parallel float arrays repesting data for each segment of the path. First is the direction to drive the wheels in, rpm to drive at, the time to drive for and the rotation of the robot. Then it also takes in an integer representing the length of the arrays. The function begins by iterating through the arrays, turning the wheels in the correct direction and setting the wheel speed to the speed from the array. Next, it waits until it reaches the time it is supposed to drive for or if it detects an obstacle in its way by calling getPathStatus(). If the robot detects an obstacle in the way then it breaks out of for loop and proceeds to call eStop(). This function was written by Francis Bui and a visualisation can be found in Figure 17 below.

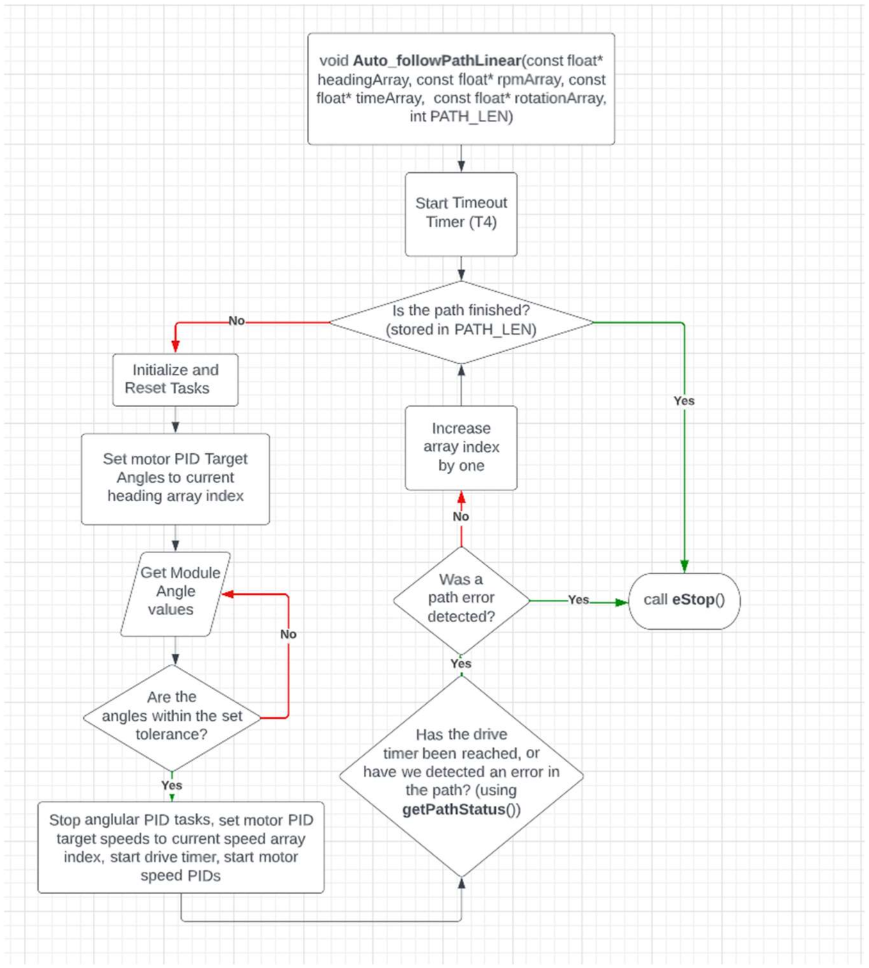
void Auto_followPathLinear(const float* headingArray, const float* rpmArray, const float* timeArray, const float* rotationArray, int PATH_LEN)
Start Timeout Timer (T4)
Is the path finished? (stored in PATH_LEN)
No
Yes
Initialize and Reset Tasks
Set motor PID Target Angles to current heading array index
Get Module Angle values
No
Are the angles within the set tolerance?
Yes
Stop anglular PID tasks, set motor PID target speeds to current speed array index, start drive timer, start motor speed PIDs
Increase array index by one
No
Was a path error detected?
Yes
Yes
call eStop()
Has the drive timer been reached, or have we detected an error in the path? (using getPathStatus())

Figure 17 - Path Following Flowchart

4.1.4 - PID Tasks

There are nine PID tasks used in the robot code, one of which is the offset PID. Shown below is the motor speed PID task, however the angle PID task functions similarly except that its error is calculated with a target angle as seen in **Appendix D - util/PID.c**. The offset PID calculates error by looking at the heading measured from the gyro and aims to get an heading of 0. The PID tasks begin by calculating the error, then passing its PID controller and the error calculated into the PID_calculateDrive() function. Using the value returned from PID_calculateDrive(), it sets the motor to that speed. In the case of the offset controller, it changes the offset variable's value, which is then added to the speeds of the motors when doing a speed PID task. Each PID task has a wait statement in it which gives each task a refresh rate. The process is illustrated in Figure 18 below. This task was written by Akil Pathiranage.

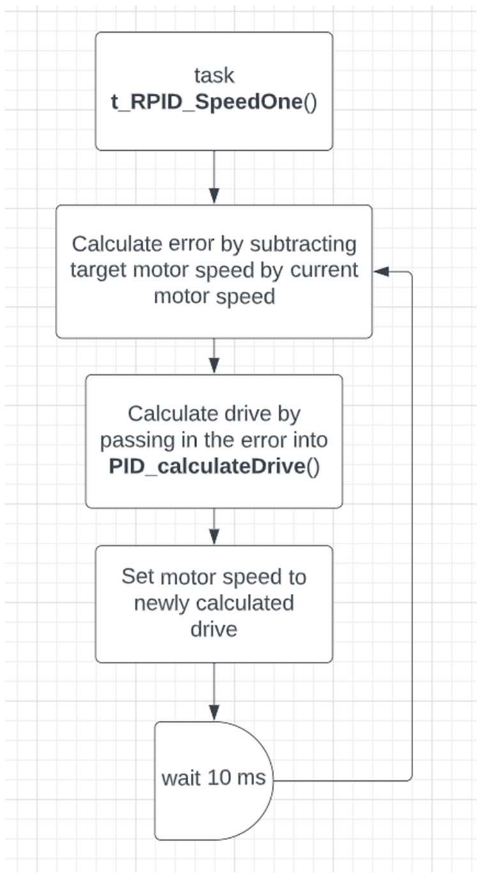
task
t_RPID_SpeedOne()
Calculate error by subtracting target motor speed by current motor speed
Calculate drive by passing in the error into **PID_calculateDrive**()
Set motor speed to newly calculated drive
wait 10 ms

Figure 18 - PID Task Flowchart

4.1.5 - PID Calculations

The PID_calculateDrive() function takes in a pointer to a PID controller and a float representing the error. It then begins by calculating the integral value, derivative value and calculates a drive by multiplying the PID gains by the error, integral and derivative. Before returning the value, it clamps it within a max and min range set by the PIDController struct, full calculations are available in **Appendix D - util/PID.c**. This function was written by Akil Pathiranage and can be understood via the flowchart in Figure 19.

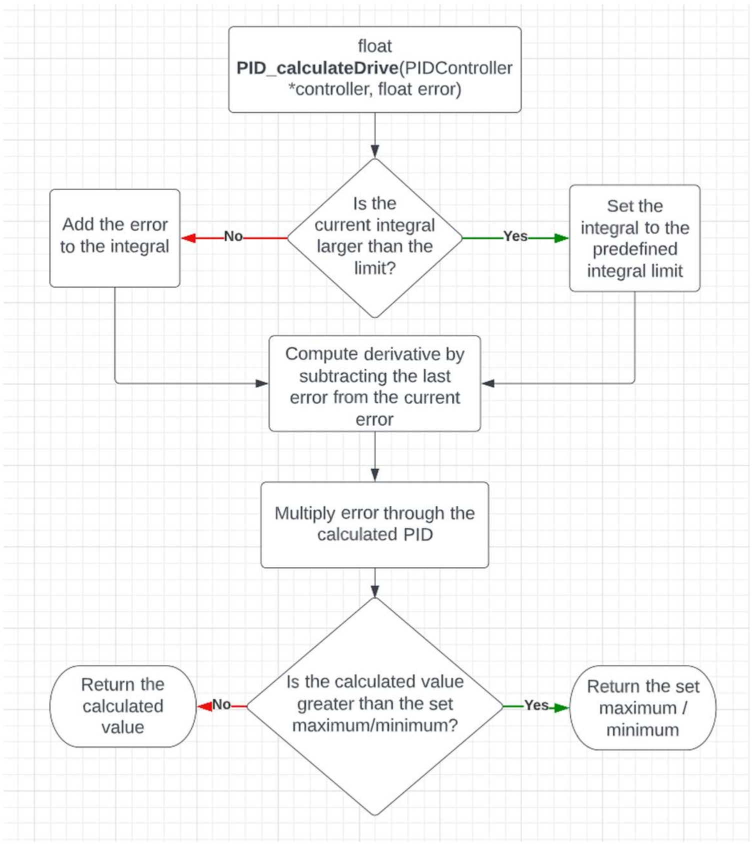
float
PID_calculateDrive(PIDController *controller, float error)
Is the current integral larger than the limit?
No
Add the error to the integral
Yes
Set the integral to the predefined integral limit
Compute derivative by subtracting the last error from the current error
Multiply error through the calculated PID
Is the calculated value greater than the set maximum/minimum?
No
Return the calculated value
Yes
Return the set maximum / minimum

Figure 19 - PID Calculate Drive Flowchart

4.1.6 - Path Verification

The getPathStatus() function, as seen in Appendix E - auto/PathWatch.c, returns a boolean value and takes in no parameters. It checks the ultrasonic reading and the timeout timer. If the ultrasonic reading is lower than the threshold of 10 cm or the timer is greater than the general timeout, 35 seconds, it returns false. Otherwise, it returns true. The decision structure can be seen in Figure 20 below. This function was written by Francis Bui.

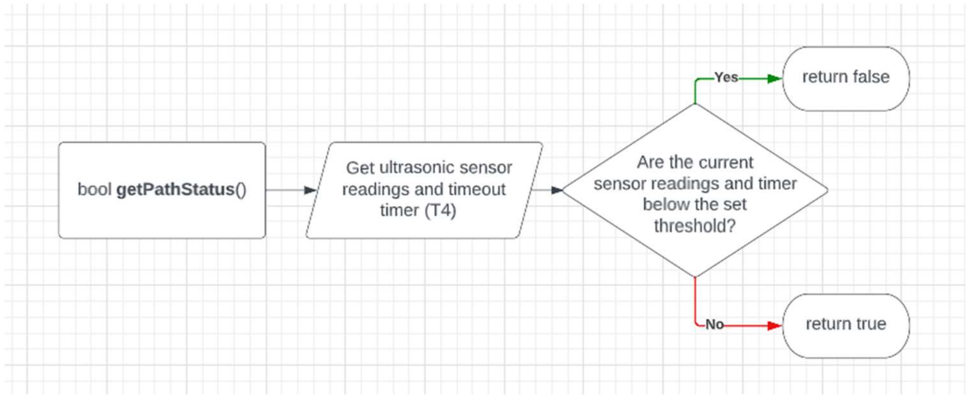
bool **getPathStatus**()
Get ultrasonic sensor readings and timeout timer (T4)
Are the current sensor readings and timer below the set threshold?
Yes
return false
No
return true

Figure 20 - Path Status Check Function Flowchart

4.1.7 - Emergency Stop

The eStop() function is called when the path is completed, or it exits path following. It begins by stopping all PID tasks, then sets the module angles to locking positions, sets all motors to brake mode and stops all tasks including main. A step-by-step flowchart can be seen in Figure 21. This function is written by Andrew Guo.

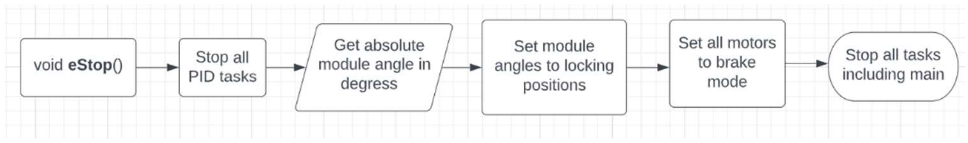
void eStop()
Stop all PID tasks
Get absolute module angle in degress
Set module angles to locking positions
Set all motors to brake mode
Stop all tasks including main

Figure 21 - Emergency Stop Function Flowchart

4.2 - Task Checklist

The checklist for the demo was modified from previous checklists to include tolerances for reaching checkpoints and turning angles. This is needed due to the sensor drift and imprecision when driving. The main parts of the checklist remained the same, check all paths are functional, wheels turn to correct angles, robot correctly reaches each checkpoint and properly locks at the end of each path or when an obstacle is found. Table 2 shows the various tasks within each operation procedure.

Table 2 - Operation Procedure

Procedure	**Tasks**
Start Up	• UI Correctly selects paths • All paths are functional
Normal Operation	• Reaches all path checkpoints consistently (within 5cm) • Faces correct direction when turning (within 2 degrees) • Completes path entirely (assuming no shutdown case) • If the robot detects an obstacle within 10cm it activates the shutdown procedure to avoid collision
Shutdown	• Wheels rotate to lock position • Motors stop • EV3 program stops

4.3 - Data Storage

Structs are used with the PID controllers, Swerve Modules and Robot to organize parts of the robot into objects. In general, constants are stored in the Constants.c file – as seen in **Appendix A - util/Constants.c**. Paths are organised into multiple parallel arrays. This is done so these can be passed as parameters into the Auto_followPathLinear() function. All paths are declared as constant float arrays in the PathTransformer.c file as seen in **Appendix B - auto/PathTransformer.c**.

4.4 - Testing

The main method of testing different function is to use the displayString method to display values to the EV3 screen. These values are compared to the expected value to determine if the function is working properly.

The biggest part of testing is of the robot for this robot is tuning the PID loops. This is done by streaming data from the robot to the computer via Bluetooth. RobotC has a built-in graph that can be used to track data over time. This graphing feature was used to visualise the PID loops and help determine changes in the P, I and D values. Using the graphing tool, the gains were adjusted to minimise overshoot and oscillation around target values. Various versions of these PID graphs can be seen in Figures 22 – 24 below.

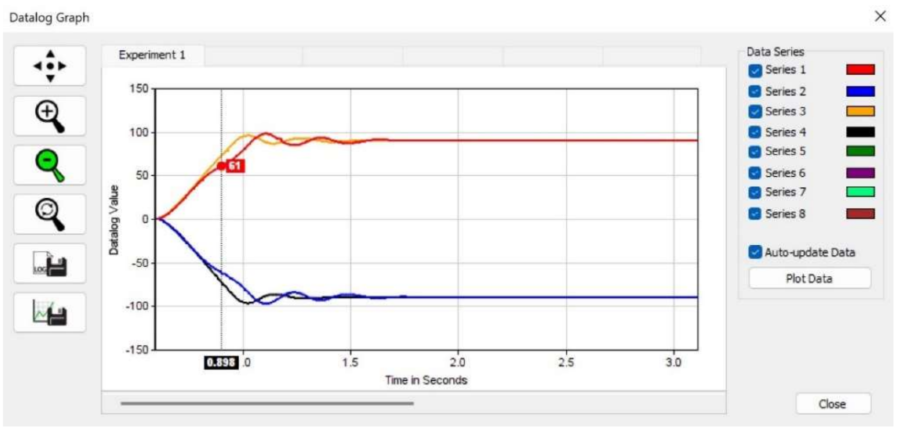

Figure 22 - Angular PID Tuning Graph (90° Turn)

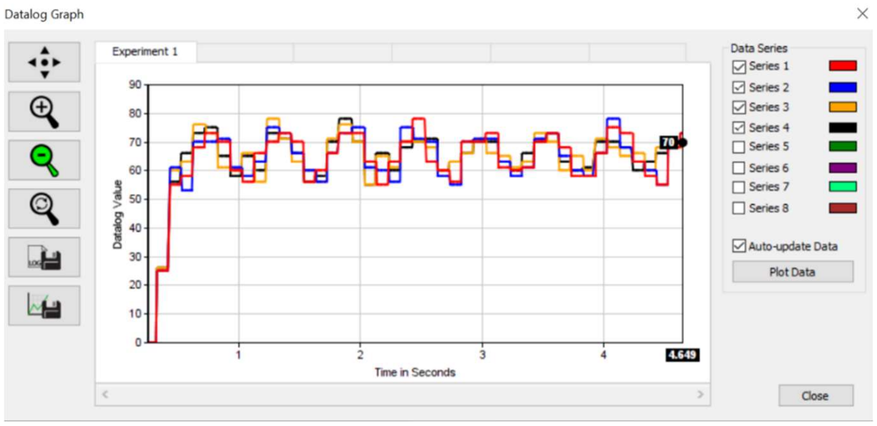

Figure 23 - Early Motor Speed PID Tuning

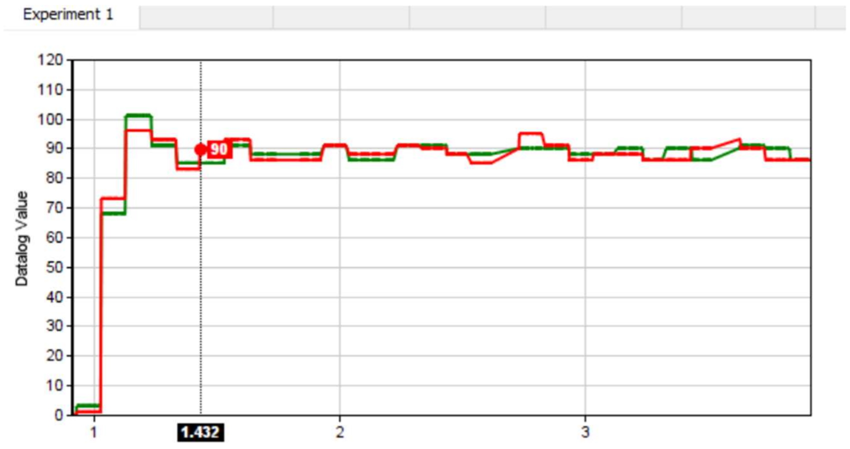

Figure 24 - Final Motor Speed PID Tuning

4.5 - Resolved Issues

The largest issue that occurred was jolting of the left module and speed fluctuations due to the mechanical inconsistencies with each swerve module. In order to solve this issue, an offset PID was added. This PID aims to keep the robot measured robot heading from the gyro at 0. This offset is added to the drive of the left module's speed PIDs and subtracted from drive of the right module's speed PIDs. This offset allows the robot to correct itself when it jolts and also counteract drift over longer distances.

There were also issues with the speed of the swerve modules. Due to the mechanical issues, the modules could not be run at max speed as there is too much speed fluctuation. The top speed recorded from the modules is 146 rpm, however in longer paths the speed is kept under 120 rpm to improve consistency and accuracy.

5.0 - Verification

The constraints for the project were fully met, although they were modified to accommodate for the new barriers discovered during the design process. In particular, the constraint of reaching a point with complete precision was altered to reaching a point with 5cm of tolerance.

This constraint was changed after testing the autonomous path following software, discovering the inaccuracies, and repeatedly making software and hardware revisions to accommodate for the drift over the course of the path. This included reprinting differential gears, full robot teardowns, adding more PID loops, tuning for an offset, and various other solutions. The path following was significantly approved after these revisions but due to the time constraints, the tuning was only capable of bringing the accuracy of the path following to a 5cm radius around a given delivery point.

6.0 - Project Plan

The primary division of tasks was done based on the skill and interest of each team member. This led to the formation of two "subteams", software and mechanical. Francis and Akil would focus on software, while Andrew and Alison made up the "mechanical team". Though each person had a focus on one side of the project, all group members were included in both aspects.

As soon as the project was assigned, a weekly schedule consisting of 3 in-person meetings each week on top of the provided class time was created as outlined in Table 3 below.

Table 3 - Initial Project Plan

Week #	Sunday	Tuesday	Thursday	Deliverables Due
1	N/A	Ideas/Brainstorm	Write initial proposal	Proposal

2	Initial design + prints start	Design iterations + code start	Design iterations + code continue	N/A
3	Formal presentation work + final chassis changes	Formal presentation work + code continue	Major parts printed	Presentation
4	Code flow chart + gearbox tests	Gearbox tests + chassis assembly	Test code on chassis	Software Meeting
5	Test + revise code, identify mechanical changes needed	Assemble new revisions + test new revisions	Prepare for demo	Project Demo

While all deliverables were complete on time, many of the internal deadlines originally outlined were pushed back due to issues with hardware. By week 4 all of the parts were printed and the robot was assembled, however there proved to be imperfections in the differential gears causing a delay while they could be reprinted. Once the differential gears were replaced and the robot was reassembled, the right swerve module would drift when it was meant to remain aligned with the heading of the robot. This caused further delays while new rail guards were printed and synthetic grease was delivered. When the robot was again reassembled, now a week behind schedule in terms of software testing, it became clear that there was another hardware issue causing the robot to drift to the right. Troubleshooting this issue burned through another few days where the robot was taken apart, reassembled, and tested, each time replacing various components to identify the issue. In the end, the drifting issue was counteracted in code the night before the demo and much of the rest of final code testing was also completed that night.

7.0 - Conclusions

The goal of this project was to create a robot that would solve the problem of autonomous delivery by consistently reaching pre-set locations. The final robot met all the defined criteria, performing the paths consistently while under all constraints.

The most vital mechanical feature on this robot is the two differential swerve modules that allow the robot to move in every direction. The entire robot was also designed to be light, reducing the load on the low power EV3 motors to relieve some of the constraint given by having to use EV3 hardware.

The most important programming task is the capability to fluently control the individual swerve modules both in rotation and drive power by using PID tasks. This motor control was precise enough to consistently follow a predesignated path and accomplish the task of autonomous delivery according to our guidelines.

8.0 - Recommendations

In terms of mechanical recommendations, machining and manufacturing could have been done on more precise or higher resolution 3D printers. One of the modules had a lot more friction that the other, causing the robot to drive one way when driving. Additionally, more iterations could have been made on the swerve drive and chassis, as assembly and mounting the modules was difficult and tedious at times.

Regarding software, if more time was available to be allocated towards Project Magnemite, it would be spent on the development of 2D path plotting software and manual wireless operation. This would add even more flexibility to the software, allowing for rapid manual debugging with real-time inputs – and make autonomous path creation a smoother and faster process.

If this project were to be used in the industry it would fair quite well. The software team's robot programming experience stems from work with industry professionals. Thanks to this experience, a core part of the time spent programming was ensuring that the code held up to industry standards. This includes a focus on modularity, version control, and code readability. More time could potentially be spent on creating UI/UX tools to speed up development.

References

[1] ElectricWizzz, "FIRST Tech Challenge Differential Swerve Drive," 2019. Retrived from https://www.youtube.com/watch?v=aETaRclTDDo&ab_channel=ElectricWizzz

[2] dk, "LEGO Mindstorms EV3," 2016. Retrived from https://grabcad.com/library/lego-mindstorms-ev3-1

Appendix A - util/Constants.c

```
const float GEAR_RADIUS = 52.3875; // Differential Gear OD is 4.125in, 
radius is 52.3875mm

const float WHEEL_RADIUS = 33.3375;

const float DIFF_TO_WHEEL = 3.50;

const float NET_GEAR_RATIO = 1.75;

const float TRACK_WIDTH = 10.0; //radius * cos(pi/4);

const float WHEEL_BASE = 10.0; //radius * sin(pi/4)

const float ENCODER_TO_ANGLE = 0.5;

const float ANGULAR_SPEED = 100.0;

const float POWER_RATE = 96.49; // rate of motor power to top speed.

const float ANGLE_TOL = 2.0;

const float DIST_TOL = 5.0;

const float MAX_SPEED = 0.0;

// MANUAL / TELEOP CONSTANTS

const float JOYSTICK_SCALAR = 1.27;

const float POWER_SCALAR = 1.70;

// AUTO CONSTANTS

const int LOCK_ANGLE = 45;

const int HEADING_TOL = 5;

// MOTOR INDEX

const tMotor TOP_LEFT_MOTOR = motorA;

const tMotor BOT_LEFT_MOTOR = motorB;

const tMotor TOP_RIGHT_MOTOR = motorC;

const tMotor BOT_RIGHT_MOTOR = motorD;

// SENSOR INDEX

const tSensors GYRO_PORT = S2;

const tSensors ACCEL_PORT = S1;

const tSensors F_ULTRASONIC_PORT = S3;

const tSensors B_ULTRASONIC_PORT = S4;
```

```
// PID arrays formatted as
// arr = {p, i ,d ,integral limit, max, min}
const float L_SPEED_ONE[6] = {0.350, 0.0, 0.00, 50000.0, 100, -100};
const float L_SPEED_TWO[6] = {0.350, 0.0, 0.00, 50000.0, 100, -100};
const float R_SPEED_ONE[6] = {0.350, 0.0, 0.0, 50000.0, 100, -100};
const float R_SPEED_TWO[6] = {0.350, 0.0, 0.0, 50000.0, 100, -100};
//=====================================================================
const float L_ANGLE_ONE[6] = {1.985, 0, 0, 0, 100, -100};
const float L_ANGLE_TWO[6] = {1.985, 0, 0, 0, 100, -100};
const float R_ANGLE_ONE[6] = {1.985, 0, 0, 0, 100, -100};
const float R_ANGLE_TWO[6] = {1.985, 0, 0, 0, 100, -100};

//======================================================================
const float DRIVE_STRAIGHT[6] = {0, 0, 0, 0, 100, -100};
//======================================================================
onst float ROTATE[6] = {0,0,0,0,100,-100};
#define true 1
#define false 0

// Paths are stored:
//{distance, absolute heading, rpm, time}
```

Appendix B - auto/PathTransformer.c

```
const int PATH_ONE_LEN = 4;
const float PATH_ONE_RPM[PATH_ONE_LEN] = {120,120,120,120};
const float PATH_ONE_HEADING[PATH_ONE_LEN] = {0,90,90,90};
const float PATH_ONE_ROTATION[PATH_ONE_LEN] = {0,0,0,0};
const float PATH_ONE_TIME[PATH_ONE_LEN] = {2000,2000,2000,2000};



const int PATH_TWO_LEN = 4;
const float PATH_TWO_RPM[PATH_TWO_LEN] = {120,70,100,120};

const float PATH_TWO_HEADING[PATH_TWO_LEN] = {45,-45,-90,90};

const float PATH_TWO_ROTATION[PATH_TWO_LEN] = {0,0,0,0};

const float PATH_TWO_TIME[PATH_TWO_LEN] = {2000,1500,1000,2000};


const int PATH_THREE_LEN = 1;

const float PATH_THREE_RPM[PATH_THREE_LEN] = {140};

const float PATH_THREE_HEADING[PATH_THREE_LEN] = {0};

const float PATH_THREE_ROTATION[PATH_THREE_LEN] = {0};

const float PATH_THREE_TIME[PATH_THREE_LEN] = {7000};
```

Appendix C - subsystems/Robot.c

```
typedef struct Robot

{

    tSensors gyroIndex;

    tSensors accelIndex;

    tSensors frontUltrasonicIndex;

    tSensors backUltrasonicIndex;


} Robot;


void Robot_resetGyro(Robot *robot);

void Robot_resetAccel(Robot *robot);


void Robot_initRobot(Robot *robot, tSensors gyroIndexIn, tSensors
accelIndexIn, tSensors frontUltrasonicIndexIn, tSensors
backUltrasonicIndexIn)

{

    robot -> gyroIndex = gyroIndexIn;

    robot -> accelIndex = accelIndexIn;

    robot -> frontUltrasonicIndex = frontUltrasonicIndexIn;
    robot -> backUltrasonicIndex = backUltrasonicIndexIn;
    Robot_resetGyro(robot);
    //Robot_resetAccel(robot);
}

/**
* Returns robot rotation in degrees
* based on last setpoint
* @param Robot pointer to robot struct
*/
float Robot_getRotation(Robot *robot)
{
    return getGyroDegrees(robot -> gyroIndex);
}

void Robot_resetGyro(Robot *robot)
{
    resetGyro(robot -> gyroIndex);
}

void Robot_resetAccel(Robot *robot)
{
sensorReset(robot -> accelIndex);
}

float Robot_getFrontDistance(Robot *robot)
{
    return getUSDistance(robot -> frontUltrasonicIndex);
}
```

```
float Robot_getBackDistance(Robot *robot)
{
    return getUSDistance(robot -> backUltrasonicIndex);
}
```

Appendix D - util/PID.c

```
typedef struct PIDController{
float pidIntegral;
float pidDerivative;
float pidDrive;
float pidLastError;

float maxOutput;
float minOutput;

float kP;
float kI;
float kD;

float integralLimit;
float target; // variable is used by drive straight controller
} PIDController;


void PID_initPIDConstants(PIDController *controller, float p, float i,
float d, float integralLimit)
{
controller -> kP = p;
controller -> kI = i;
controller -> kD = d;

controller -> integralLimit = integralLimit;

}


void PID_initOutputRange(PIDController *controller, float max, float
min)

{

controller -> maxOutput = max;

controller -> minOutput = min;

}



void PID_reset(PIDController *controller)

{

controller -> pidLastError = 0;

controller -> pidIntegral = 0;

controller -> pidDerivative = 0;

}


float PID_calculateDrive(PIDController *controller, float error)

{

float drive;


if(controller -> kI != 0 && controller -> pidIntegral < controller ->
integralLimit)

controller -> pidIntegral += error;

if(controller -> pidIntegral > controller -> integralLimit)

controller -> pidIntegral = controller -> integralLimit;
```

```
  float derivative = error - (controller -> pidLastError);
  controller -> pidLastError = error;

  drive = (controller -> kP)*error + (controller -> kI)*(controller ->
pidIntegral) + (controller -> kD)*derivative;
  if(drive > controller -> maxOutput)
      drive = controller -> maxOutput;

  if (drive < controller -> minOutput)
      drive = controller -> minOutput;

  return drive;
}

#include "PID.c"
#include "Constants.c"

typedef struct SwerveModule{
int motorOneIndex;
int motorTwoIndex;
int motorPorts[2];

float targetAngle;
float targetDriveSpeed;

float absoluteAngle[2];
float targetMotorSpeeds[2];
float targetMotorAngles[2];
float targetAngularSpeed;
float motorOneSpeed;
float motorTwoSpeed;
float currentDriveSpeed;
float currentAngularSpeed;
float currentAngle;


PIDController ctrlSpeedOne;
PIDController ctrlSpeedTwo;

PIDController ctrlAngleOne;
PIDController ctrlAngleTwo;
} SwerveModule;

/*
Reset encoders for swerve
*/
void Swerve_resetEncoders(SwerveModule *swerve)
{
resetMotorEncoder(swerve -> motorPorts[0]);
resetMotorEncoder(swerve -> motorPorts[1]);
}

/*
Initialize motors for swerve
*/
void Swerve_initModule(SwerveModule *swerve, int motorOneIndexIn, int
                motorTwoIndexIn)
{
```

```
swerve -> motorPorts[0] = motorOneIndexIn;

swerve -> motorPorts[1] = motorTwoIndexIn;


Swerve_resetEncoders(swerve);

}


/**

* Sets target speed for PID loop in rpms

*/

void Swerve_setMotorTargetSpeed(SwerveModule *swerve, int motIdx,
float target)

{

swerve -> targetMotorSpeeds[motIdx] = target * 2;

}


/**

* Sets target angle for specific motor in degrees

*/

void Swerve_setMotorTargetAngle(SwerveModule *swerve, int motIdx,
float target)

{

swerve -> targetMotorAngles[motIdx] = target;

swerve -> absoluteAngle[motIdx] = (swerve -> absoluteAngle[motIdx]) +
                             target;

}


void Swerve_setDriveSpeed(SwerveModule *swerve, float motorOneSpeed,
float motorTwoSpeed) // set drive speed out of 100%

{

setMotorSpeed(swerve -> motorPorts[0], motorOneSpeed);

setMotorSpeed(swerve -> motorPorts[1], motorTwoSpeed);
```

```
}

/**
* Returns speed in milimeteres per second
*/
float Swerve_getSpeed(SwerveModule *swerve)
{
swerve -> currentDriveSpeed = ((2 * PI * WHEEL_RADIUS *
                                getMotorRPM(swerve -> motorPorts[0]) *
                                NET_GEAR_RATIO / 60) +
                                 (2 * PI * WHEEL_RADIUS *
                                getMotorRPM(swerve -> motorPorts[1]) *
                                NET_GEAR_RATIO / 60));
return swerve -> currentDriveSpeed;
}

/**
* Returns motor speed in rpm
*/
float Swerve_getMotorSpeed(SwerveModule *swerve, int motIdx)
// Returns meters per second of differential gear
{
// swerve -> motorOneSpeed = (PI * GEAR_RADIUS * getMotorRPM(swerve ->
                                motorOneIndex) / 60);
// return swerve -> motorOneSpeed;
return getMotorRPM(swerve -> motorPorts[motIdx]);
}

/**
* Returns wheel angles relative to motor
*/
```

```
float Swerve_getMotorAngle(SwerveModule *swerve, int motIdx)

{

//implement angle tracking

return getMotorEncoder(swerve -> motorPorts[motIdx]) *
                      ENCODER_TO_ANGLE;

}


float Swerve_getAbsoluteAngle(SwerveModule *swerve, int motIdx)

{

return swerve -> absoluteAngle[motIdx];

}


float Swerve_getDriveSpeed(SwerveModule *swerve)

{

swerve -> currentDriveSpeed = ((Swerve_getMotorSpeed(swerve,0) -
Swerve_getMotorSpeed(swerve,1)) / GEAR_RADIUS) * DIFF_TO_WHEEL *
WHEEL_RADIUS;


return swerve -> currentDriveSpeed;

}

#include "../subsystems/Robot.c"
```

Appendix E - auto/PathWatch.c

```
Robot Magnemite;

const int MAX_RUNTIME = 35000;

const int SAFETY_TOL = 10;

bool getPathStatus()

{

    if (Robot_getFrontDistance(&Magnemite) < SAFETY_TOL ||
Robot_getBackDistance(&Magnemite) < SAFETY_TOL || time1[T4] >
MAX_RUNTIME)

    {
        return false;
    }
    return true;
}
```

Appendix F - Subsystems/Drive.c

```
#include "../util/SwerveModule.c"
#include "../auto/PathTransformer.c"
#include "../auto/PathWatch.c"
#include "../teleop/OI.c"

#pragma config(Sensor, S2,     ,               sensorEV3_Gyro)
#pragma config(Sensor, S3,     ,               sensorEV3_Ultrasonic)
#pragma config(Sensor, S4,     ,               sensorEV3_Ultrasonic)
//*!!Code automatically generated by 'ROBOTC' configuration wizard
!!*//

SwerveModule leftModule;
SwerveModule rightModule;

PIDController driveStraightController;
PIDController rotateRobotController;
PIDController offsetController;

void Drive_setOpposingSync();
void Drive_setLinearSync();

void initializePIDSpeed();
void initializePIDAngle();
void initializePIDStraight();
void initializePIDRotate();
void resetPIDSpeedControllers();
void resetPIDAngleControllers();
void resetPIDStraight();
void resetPIDRotate();



void startSpeedPIDTasks();
void stopSpeedPIDTasks();
void startAnglePIDTasks();
void stopAnglePIDTasks();
void stopDriveStraightPID();
void startDriveStraightPID();
void stopRotateRobotPID();
void startRotateRobotPID();
void eStop();
void selectPath();
void logMotorData();


void Manual_teleop(bool closedLoop);
void Auto_followPathLinear(const float* headingArray, const float*
rpmArray,
const float* timeArray, const float* rotationArray, const int
PATH_LEN);


void Auto_followPathCurve(const float* rpmAlpha, const float* rpmBeta,
const float* timeArray, const float* rotationArray, const int
PATH_LEN);


static float offset;


typedef enum DriveStates
```

```
{
MANUAL,
AUTO,
IDLE,
} DriveStates;

task t_syncDriveControllerOne()
{
while(true){
float err =-leftModule.targetMotorSpeeds[0] -
Swerve_getMotorSpeed(&leftModule, 0);
float drive = PID_calculateDrive(&(leftModule.ctrlSpeedOne), err);
Drive_straightOpposingSync(drive - offset);
wait1Msec(90);
}
}

task t_OffsetController()
{
while(true){
//0 - rotation
float err = -Robot_getRotation(&Magnemite);
offset = PID_calculateDrive(&offsetController, err);
wait1Msec(5);
}
}

task t_RPID_SpeedOne()
{
while(true)
```

```
{

float err = rightModule.targetMotorSpeeds[0] -
Swerve_getMotorSpeed(&rightModule, 0);

float drive = PID_calculateDrive(&(rightModule.ctrlSpeedOne), err);

setMotorSpeed(rightModule.motorPorts[0], drive - offset);

wait1Msec(10);

}

}


task t_RPID_SpeedTwo()

{

while(true)

{

float err = rightModule.targetMotorSpeeds[1] -
Swerve_getMotorSpeed(&rightModule, 1);

float drive = PID_calculateDrive(&(rightModule.ctrlSpeedTwo), err);

setMotorSpeed(rightModule.motorPorts[1], drive - offset);

wait1Msec(10);

}

}


task t_LPID_SpeedOne()

{

while(true)

{

float err = leftModule.targetMotorSpeeds[0] -
Swerve_getMotorSpeed(&leftModule, 0);

float drive = PID_calculateDrive(&(leftModule.ctrlSpeedOne), err);

setMotorSpeed(leftModule.motorPorts[0], drive + offset);

wait1Msec(90);
}
}

task t_LPID_SpeedTwo()
{
while(true)
{
float err = leftModule.targetMotorSpeeds[1] -
Swerve_getMotorSpeed(&leftModule, 1);
float drive = PID_calculateDrive(&(leftModule.ctrlSpeedTwo), err);
setMotorSpeed(leftModule.motorPorts[1], drive + offset);
wait1Msec(90);
}
}

task t_LPID_AngleOne()
{
resetMotorEncoder(leftModule.motorPorts[0]);
while(true)
{
float err = leftModule.targetMotorAngles[0] -
Swerve_getMotorAngle(&leftModule, 0);
float drive = PID_calculateDrive(&(leftModule.ctrlAngleOne),err);
setMotorSpeed(leftModule.motorPorts[0], drive);
}
}
task t_LPID_AngleTwo()
{
resetMotorEncoder(leftModule.motorPorts[1]);
while(true)
```

```
{
float err = leftModule.targetMotorAngles[1] -
Swerve_getMotorAngle(&leftModule, 1);
float drive = PID_calculateDrive(&(leftModule.ctrlAngleTwo),err);
setMotorSpeed(leftModule.motorPorts[1], drive);
}
}

task t_RPID_AngleOne()
{
resetMotorEncoder(rightModule.motorPorts[0]);
while(true)
{
float err = rightModule.targetMotorAngles[0] -
Swerve_getMotorAngle(&rightModule, 0);
float drive = PID_calculateDrive(&(rightModule.ctrlAngleOne),err);
setMotorSpeed(rightModule.motorPorts[0], drive);
}
}
task t_RPID_AngleTwo()
{
resetMotorEncoder(rightModule.motorPorts[1]);
while(true)
{
float err = rightModule.targetMotorAngles[1] -
Swerve_getMotorAngle(&rightModule, 1);
float drive = PID_calculateDrive(&(rightModule.ctrlAngleTwo),err);
setMotorSpeed(rightModule.motorPorts[1], drive);
}
}
```

```
task t_DriveStraightPID()

{

Robot_resetGyro(&Magnemite);

while(true)

{

float err = driveStraightController.target -
Robot_getRotation(&Magnemite);

float drive = PID_calculateDrive(&driveStraightController, err);


Swerve_setDriveSpeed(&leftModule, 50 + drive, 50 + drive);

Swerve_setDriveSpeed(&rightModule, 50 - drive, 50 - drive);

}

}


task t_RotateRobotPID()

{

Robot_resetGyro(&Magnemite);

while(true)

{

float err = rotateRobotController.target -
Robot_getRotation(&Magnemite);

float drive = PID_calculateDrive(&rotateRobotController, err);


Swerve_setDriveSpeed(&leftModule, drive, drive);

Swerve_setDriveSpeed(&rightModule, -drive, -drive);

}

}


task main()
```

```
{
DriveStates DriveState = AUTO;

datalogClear();

Robot_initRobot(&Magnemite, GYRO_PORT, ACCEL_PORT, F_ULTRASONIC_PORT,
B_ULTRASONIC_PORT);
Swerve_initModule(&leftModule, TOP_LEFT_MOTOR, BOT_LEFT_MOTOR);
Swerve_initModule(&rightModule, TOP_RIGHT_MOTOR, BOT_RIGHT_MOTOR);
initializePIDSpeed();
initializePIDAngle();
initializePIDStraight();

PID_initPIDConstants(&offsetController, 0.4, 0.001, 0, 30)
PID_initOutputRange(&offsetController, 50, -50);
PID_reset(&offsetController);

//Swerve_setMotorTargetSpeed(&leftModule, 0, 120);
//Swerve_setMotorTargetSpeed(&rightModule, 0, 120);
//Swerve_setMotorTargetSpeed(&rightModule, 1, 120);
//Swerve_setMotorTargetSpeed(&leftModule, 1, 120);
//startSpeedPIDTasks();
//startTask(t_OffsetController);
//while(true)
//{

//}
selectPath();
}
```

```
void Auto_followPathLinear(const float* headingArray, const float*
rpmArray,
const float* timeArray,  const float* rotationArray, int PATH_LEN)
{
time1[T4] = 0;

  for(int i = 0; i < PATH_LEN; i++)
  {
    stopSpeedPIDTasks();
    stopTask(t_OffsetController);
stopAnglePIDTasks();
resetPIDAngleControllers();
resetPIDSpeedControllers();
    Swerve_setDriveSpeed(&leftModule, 0, 0);
    Swerve_setDriveSpeed(&rightModule, 0, 0);
Swerve_resetEncoders(&leftModule);
Swerve_resetEncoders(&rightModule);

time1[T3] = 0;
while(time1[T3] < 300){}

Swerve_setMotorTargetAngle(&leftModule, 0, headingArray[i]);
Swerve_setMotorTargetAngle(&leftModule, 1, -headingArray[i]);
Swerve_setMotorTargetAngle(&rightModule, 0, headingArray[i]);
Swerve_setMotorTargetAngle(&rightModule, 1, -headingArray[i]);

startAnglePIDTasks();
const float tol = 1.1;
while (
fabs(headingArray[i] - Swerve_getMotorAngle(&rightModule, 0)) >  tol
||
fabs(-headingArray[i] - Swerve_getMotorAngle(&rightModule, 1)) >  tol
||
fabs(headingArray[i] - Swerve_getMotorAngle(&leftModule, 0)) >  tol ||
fabs(-headingArray[i] - Swerve_getMotorAngle(&leftModule, 1)) >  tol)
{
}

stopAnglePIDTasks();

Swerve_setMotorTargetSpeed(&leftModule, 0, rpmArray[i]);
Swerve_setMotorTargetSpeed(&leftModule, 1, rpmArray[i]);
Swerve_setMotorTargetSpeed(&rightModule, 0, rpmArray[i]);
Swerve_setMotorTargetSpeed(&rightModule, 1, rpmArray[i]);

startOffsetPID();
startSpeedPIDTasks();

time1[T3] = 0;

while  (time1[T3] < timeArray[i] && getPathStatus() == true /*&&
Robot_getRotation(&Magnemite) < rotationArray[i] - HEADING_TOL &&
Robot_getRotation(&Magnemite) > rotationArray + HEADING_TOL*/){}

if (!getPathStatus())
break;
        //while (Swerve_getDist(&rightModule) != distanceArray[i] ||
Swerve_getDist(&leftModule) != distanceArray[i]){}
}
eStop();
```

```
}


void Auto_followPathCurve(const float* rpmAlpha, const float* rpmBeta,
const float* timeArray, const float* rotationArray, const int*
PATH_LEN)
{
time1[T4] = 0;


    for(int i = 0; i < PATH_LEN; i++)
    {
        stopSpeedPIDTasks();
      resetPIDSpeedControllers();
      Swerve_setDriveSpeed(&leftModule, 0, 0);
      Swerve_setDriveSpeed(&rightModule, 0, 0);
      Swerve_resetEncoders(&leftModule);
      Swerve_resetEncoders(&rightModule);


      time1[T3] = 0;
      while(time1[T3] < 300){}



      Swerve_resetEncoders(&leftModule);
      Swerve_resetEncoders(&rightModule);


      Swerve_setMotorTargetSpeed(&leftModule, 0, rpmAlpha[i]);
      Swerve_setMotorTargetSpeed(&leftModule, 1, rpmBeta[i]);
      Swerve_setMotorTargetSpeed(&rightModule, 0, rpmAlpha[i]);
      Swerve_setMotorTargetSpeed(&rightModule, 1, rpmBeta[i]);


      startSpeedPIDTasks();
      time1[T3] = 0;

      while(time1[T3] < timeArray[i] && getPathStatus() == true){}

      if (getPathStatus() == false || Robot_getRotation(&Magnemite) <=
      rotationArray[i] - HEADING_TOL || Robot_getRotation(&Magnemite)
      >= rotationArray[i] + HEADING_TOL)
      break;
 eStop();
}

void Drive_straightOpposingSync(float motorSpeed)
{
setMotorSync(motor[leftModule.motorPorts[0]],
motor[rightModule.motorPorts[0]], motorSpeed);
setMotorSync(motor[leftModule.motorPorts[1]],
motor[rightModule.motorPorts[1]], -motorSpeed);
}

void Drive_straightLinearSync(float motorSpeed)
{
setMotorSync(motor[leftModule.motorPorts[0]],
motor[rightModule.motorPorts[1]], motorSpeed);
setMotorSync(motor[leftModule.motorPorts[1]],
motor[rightModule.motorPorts[0]], motorSpeed);
}

void selectPath()
{
displayString(1,"Press left for path 1");
displayString(2,"Press up for path 2");
displayString(3,"Press right for path 3");
int buttonPressed = 0;
while(buttonPressed==0)
{
if (getButtonPress(buttonLeft))
{
buttonPressed = 1;
  }
  else if (getButtonPress(buttonUp))
  {
     buttonPressed = 2;
  }
  else if (getButtonPress(buttonRight))
  {
     buttonPressed = 3;
  }
}
switch(buttonPressed)
{
case 1:
Auto_followPathLinear(PATH_ONE_HEADING, PATH_ONE_RPM, PATH_ONE_TIME,
PATH_ONE_ROTATION, PATH_ONE_LEN);
break;
case 2:
Auto_followPathLinear(PATH_TWO_HEADING, PATH_TWO_RPM, PATH_TWO_TIME,
PATH_TWO_ROTATION, PATH_TWO_LEN);
break;
case 3:
```

```
Auto_followPathLinear(PATH_THREE_HEADING, PATH_THREE_RPM,
PATH_THREE_TIME, PATH_THREE_ROTATION, PATH_THREE_LEN);

break;

}

}


void Manual_teleop(bool closedLoop)

{

stopSpeedPIDTasks();

stopAnglePIDTasks();

resetPIDAngleControllers();

resetPIDSpeedControllers();


startSpeedPIDTasks();


while (true)

{


short* joystickInput = getJoystickInput();

short* motorPowers = getMotorPowers(joystickInput[0],
joystickInput[1]);


if (closedLoop == true)

{

Swerve_setMotorTargetSpeed(&rightModule, 0, motorPowers[0] *
POWER_SCALAR);

Swerve_setMotorTargetSpeed(&rightModule, 1, motorPowers[1] *
POWER_SCALAR);

Swerve_setMotorTargetSpeed(&leftModule, 0, motorPowers[0] *
POWER_SCALAR);
Swerve_setMotorTargetSpeed(&leftModule, 1, motorPowers[1] *
POWER_SCALAR);

}


else

{

Swerve_setDriveSpeed(&leftModule, motorPowers[0] * POWER_SCALAR,
motorPowers[1] * POWER_SCALAR);

Swerve_setDriveSpeed(&rightModule, motorPowers[0] * POWER_SCALAR,
motorPowers[1] * POWER_SCALAR);

}


}

}


void startSpeedPIDTasks()

{

startTask(t_RPID_SpeedOne);

startTask(t_RPID_SpeedTwo);

startTask(t_LPID_SpeedOne);

startTask(t_LPID_SpeedTwo);

}


void stopSpeedPIDTasks()

{

stopTask(t_LPID_SpeedOne);

    stopTask(t_LPID_SpeedTwo);

stopTask(t_RPID_SpeedOne);

    stopTask(t_RPID_SpeedTwo);

}
```

```
void startAnglePIDTasks()
{
resetPIDAngleControllers();
Swerve_resetEncoders(&leftModule);
Swerve_resetEncoders(&rightModule);
startTask(t_LPID_AngleOne);
startTask(t_LPID_AngleTwo);
startTask(t_RPID_AngleOne);
startTask(t_RPID_AngleTwo);
}

void stopAnglePIDTasks()
{
stopTask(t_LPID_AngleOne);
stopTask(t_LPID_AngleTwo);
stopTask(t_RPID_AngleOne);
stopTask(t_RPID_AngleTwo);
}

void startOffsetPID(){
PID_reset(&offsetController);
Robot_resetGyro(&Magnemite);
startTask(t_OffsetController);
}

void stopOffsetPID()
{
stopTask(t_OffsetController);
}

void resetPIDSpeedControllers()
{
PID_reset(&(leftModule.ctrlSpeedOne));
PID_reset(&(leftModule.ctrlSpeedTwo));
PID_reset(&(rightModule.ctrlSpeedOne));
PID_reset(&(rightModule.ctrlSpeedTwo));
}

void resetPIDAngleControllers()
{
PID_reset(&(leftModule.ctrlAngleOne));
PID_reset(&(leftModule.ctrlAngleTwo));
PID_reset(&(rightModule.ctrlAngleOne));
PID_reset(&(rightModule.ctrlAngleTwo));
}

void resetPIDStraight()
{
PID_reset(&driveStraightController);
}

void resetPIDRotate()
{
PID_reset(&rotateRobotController);
}

void initializePIDSpeed()
```

```
{
PID_initPIDConstants(&(leftModule.ctrlSpeedOne), L_SPEED_ONE[0],
L_SPEED_ONE[1], L_SPEED_ONE[2], L_SPEED_ONE[3]);
PID_initOutputRange(&(leftModule.ctrlSpeedOne), L_SPEED_ONE[4],
L_SPEED_ONE[5]);
PID_reset(&(leftModule.ctrlSpeedOne));


PID_initPIDConstants(&(leftModule.ctrlSpeedTwo), L_SPEED_TWO[0],
L_SPEED_TWO[1], L_SPEED_TWO[2], L_SPEED_TWO[3]);
PID_initOutputRange(&(leftModule.ctrlSpeedTwo), L_SPEED_TWO[4],
L_SPEED_TWO[5]);
PID_reset(&(leftModule.ctrlSpeedTwo));


PID_initPIDConstants(&(rightModule.ctrlSpeedOne), R_SPEED_ONE[0],
R_SPEED_ONE[1], R_SPEED_ONE[2], R_SPEED_ONE[3]);
PID_initOutputRange(&(rightModule.ctrlSpeedOne), R_SPEED_ONE[4],
R_SPEED_ONE[5]);
PID_reset(&(rightModule.ctrlSpeedOne));


PID_initPIDConstants(&(rightModule.ctrlSpeedTwo), R_SPEED_TWO[0],
R_SPEED_TWO[1], R_SPEED_TWO[2], R_SPEED_TWO[3]);
PID_initOutputRange(&(rightModule.ctrlSpeedTwo), R_SPEED_TWO[4],
R_SPEED_TWO[5]);
PID_reset(&(rightModule.ctrlSpeedTwo));
}


void initializePIDAngle()
{
PID_initPIDConstants(&(leftModule.ctrlAngleOne), L_ANGLE_ONE[0],
L_ANGLE_ONE[1], L_ANGLE_ONE[2], L_ANGLE_ONE[3]);
PID_initOutputRange(&(leftModule.ctrlAngleOne), L_ANGLE_ONE[4],
L_ANGLE_ONE[5]);
PID_reset(&(leftModule.ctrlAngleOne));
```

```
PID_initPIDConstants(&(leftModule.ctrlAngleTwo), L_ANGLE_TWO[0],
L_ANGLE_TWO[1], L_ANGLE_TWO[2], L_ANGLE_TWO[3]);

PID_initOutputRange(&(leftModule.ctrlAngleTwo), L_ANGLE_TWO[4],
L_ANGLE_TWO[5]);

PID_reset(&(leftModule.ctrlAngleTwo));


PID_initPIDConstants(&(rightModule.ctrlAngleOne), R_ANGLE_ONE[0],
R_ANGLE_ONE[1], R_ANGLE_ONE[2], R_ANGLE_ONE[3]);

PID_initOutputRange(&(rightModule.ctrlAngleOne), R_ANGLE_ONE[4],
R_ANGLE_ONE[5]);

PID_reset(&(rightModule.ctrlAngleOne));


PID_initPIDConstants(&(rightModule.ctrlAngleTwo), R_ANGLE_TWO[0],
R_ANGLE_TWO[1], R_ANGLE_TWO[2], R_ANGLE_TWO[3]);

PID_initOutputRange(&(rightModule.ctrlAngleTwo), R_ANGLE_TWO[4],
R_ANGLE_TWO[5]);

PID_reset(&(rightModule.ctrlAngleTwo));

}


void initializePIDStraight()

{

PID_initPIDConstants(&driveStraightController, DRIVE_STRAIGHT[0],
DRIVE_STRAIGHT[1], DRIVE_STRAIGHT[2], DRIVE_STRAIGHT[3]);

PID_initOutputRange(&driveStraightController, DRIVE_STRAIGHT[4],
DRIVE_STRAIGHT[5]);

PID_reset(&driveStraightController);

driveStraightController.target = 0;

}


void initializePIDRotate()

{
```

```
PID_initPIDConstants(&rotateRobotController, ROTATE[0], ROTATE[1],
ROTATE[2], ROTATE[3]);
PID_initOutputRange(&rotateRobotController, ROTATE[4], ROTATE[5]);
PID_reset(&rotateRobotController);

rotateRobotController.target = 0;
}

void logMotorData()
{
datalogAddValueWithTimeStamp(0, Swerve_getMotorSpeed(&leftModule, 0));
datalogAddValueWithTimeStamp(1, Swerve_getMotorSpeed(&leftModule, 1));
datalogAddValueWithTimeStamp(2, Swerve_getMotorSpeed(&rightModule,
0));
datalogAddValueWithTimeStamp(3, Swerve_getMotorSpeed(&rightModule,
1));
datalogAddValueWithTimeStamp(4, Swerve_getMotorAngle(&leftModule, 0));
datalogAddValueWithTimeStamp(5, Swerve_getMotorAngle(&leftModule, 1));
datalogAddValueWithTimeStamp(6, Swerve_getMotorAngle(&rightModule,
0));
datalogAddValueWithTimeStamp(7, Swerve_getMotorAngle(&rightModule,
1));
}

void eStop()
{
float absAngle;

    stopSpeedPIDTasks();
stopAnglePIDTasks();
resetPIDAngleControllers();
```

```
resetPIDSpeedControllers();

Swerve_setDriveSpeed(&leftModule, 0, 0);
Swerve_setDriveSpeed(&rightModule, 0, 0);
Swerve_resetEncoders(&leftModule);
Swerve_resetEncoders(&rightModule);

absAngle = fabs(Swerve_getAbsoluteAngle(&leftModule, 0));

float absDifference = absAngle - LOCK_ANGLE;
absAngle = absAngle - absDifference;

Swerve_setMotorTargetAngle(&leftModule, 0, absAngle);
Swerve_setMotorTargetAngle(&leftModule, 1, -absAngle);
Swerve_setMotorTargetAngle(&rightModule, 0, -absAngle);
Swerve_setMotorTargetAngle(&rightModule, 1, absAngle);

startAnglePIDTasks();

time1[T3] = 0;
while(time1[T3] < 3000){}

stopAnglePIDTasks();
resetPIDAngleControllers();

Swerve_resetEncoders(&leftModule);
Swerve_resetEncoders(&rightModule);

Swerve_setDriveSpeed(&leftModule, 0, 0);
Swerve_setDriveSpeed(&rightModule, 0, 0);

stopAllTasks();

setMotorBrakeMode(leftModule.motorOneIndex, motorBrake);
setMotorBrakeMode(leftModule.motorTwoIndex, motorBrake);
setMotorBrakeMode(rightModule.motorOneIndex, motorBrake);
setMotorBrakeMode(rightModule.motorTwoIndex, motorBrake);

}
```

Appendix G – util/SwerveModule.c

```
#include "PID.c"
#include "Kinematics.c"
#include "Constants.c"

typedef struct SwerveModule{
	int motorOneIndex;
	int motorTwoIndex;
	int motorPorts[2];

	float targetAngle;
	float targetDriveSpeed;

	float absoluteAngle[2];
	float targetMotorSpeeds[2];
	float targetMotorAngles[2];
	float targetAngularSpeed;
	float motorOneSpeed;
	float motorTwoSpeed;
	float currentDriveSpeed;
	float currentAngularSpeed;
	float currentAngle;


	PIDController ctrlSpeedOne;
	PIDController ctrlSpeedTwo;

	PIDController ctrlAngleOne;
	PIDController ctrlAngleTwo;
} SwerveModule;

/*
Reset encoders for swerve
*/
void Swerve_resetEncoders(SwerveModule *swerve)
{
	resetMotorEncoder(swerve -> motorPorts[0]);
	resetMotorEncoder(swerve -> motorPorts[1]);
}

/*
Initialize motors for swerve
*/
```

```
void Swerve_initModule(SwerveModule *swerve, int motorOneIndexIn, int
motorTwoIndexIn)
{
      swerve -> motorPorts[0] = motorOneIndexIn;
      swerve -> motorPorts[1] = motorTwoIndexIn;


      Swerve_resetEncoders(swerve);
}


/**
 * Sets target speed for PID loop in rpms
*/
void Swerve_setMotorTargetSpeed(SwerveModule *swerve, int motIdx,
float target)
{
      swerve -> targetMotorSpeeds[motIdx] = target * 2;
}


/**
 * Sets target angle for specific motor in degrees
*/
void Swerve_setMotorTargetAngle(SwerveModule *swerve, int motIdx,
float target)
{
      swerve -> targetMotorAngles[motIdx] = target;
      swerve -> absoluteAngle[motIdx] = (swerve ->
absoluteAngle[motIdx]) + target;
}
```

```
void Swerve_setDriveSpeed(SwerveModule *swerve, float motorOneSpeed,
float motorTwoSpeed) // set drive speed out of 100%
{
      setMotorSpeed(swerve -> motorPorts[0], motorOneSpeed);
      setMotorSpeed(swerve -> motorPorts[1], motorTwoSpeed);
}


/**
 * Returns speed in milimeteres per second
*/
float Swerve_getSpeed(SwerveModule *swerve) // Returns current swerve
module speed in meters per second
{
      swerve -> currentDriveSpeed = ((2 * PI * WHEEL_RADIUS *
getMotorRPM(swerve -> motorPorts[0]) * NET_GEAR_RATIO / 60) +
                                        (2 * PI * WHEEL_RADIUS *
getMotorRPM(swerve -> motorPorts[1]) * NET_GEAR_RATIO / 60));
      return swerve -> currentDriveSpeed;
}


/**
 * Returns motor speed in rpm
*/
float Swerve_getMotorSpeed(SwerveModule *swerve, int motIdx) //
Returns meters per second of differential gear
{
      return getMotorRPM(swerve -> motorPorts[motIdx]);
}
/**
 * Returns wheel angles relative to motor
```

```
*/
float Swerve_getMotorAngle(SwerveModule *swerve, int motIdx)
{
      //implement angle tracking
      return getMotorEncoder(swerve -> motorPorts[motIdx]) *
ENCODER_TO_ANGLE;
}
float Swerve_getAbsoluteAngle(SwerveModule *swerve, int motIdx)
{
      return swerve -> absoluteAngle[motIdx];
}
float Swerve_getDriveSpeed(SwerveModule *swerve)
{
      swerve -> currentDriveSpeed = ((Swerve_getMotorSpeed(swerve,0) -
Swerve_getMotorSpeed(swerve,1)) / GEAR_RADIUS) * DIFF_TO_WHEEL *
WHEEL_RADIUS;


      return swerve -> currentDriveSpeed;
}
```