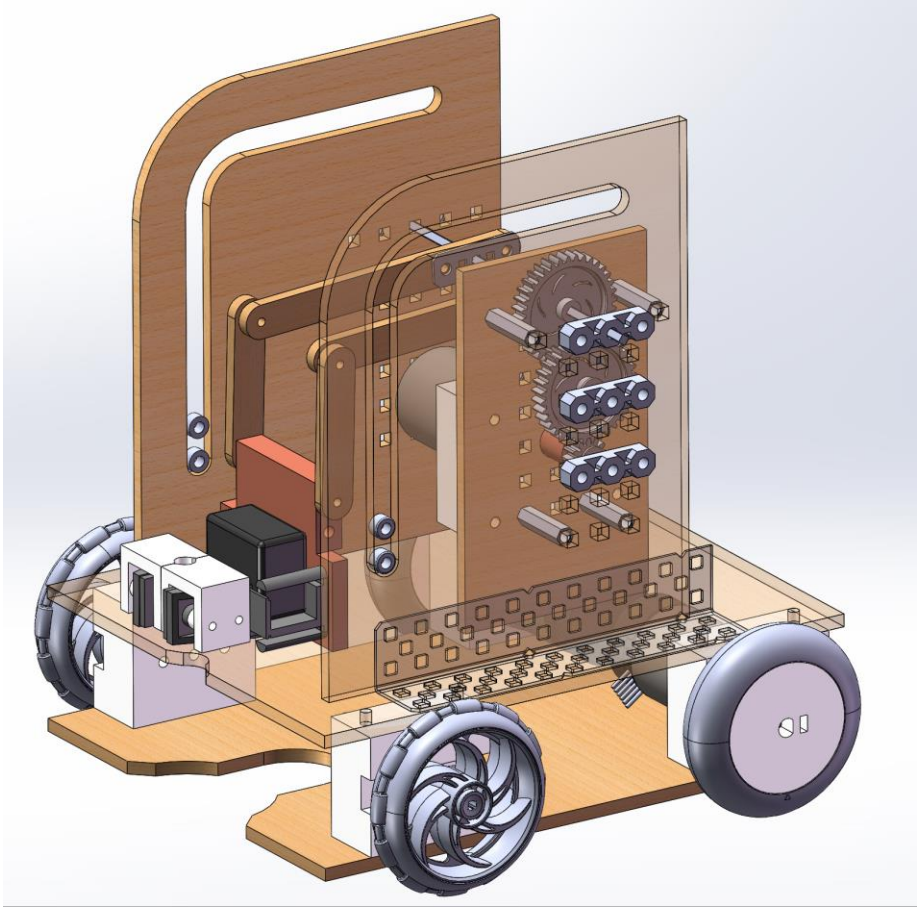
RBE 2001 Section A01 A ’17, Unified Robotics I

Final Project Report



Tabby Gibbs, Dan Oates, Luc Park, Michael Sidler

10/13/17

Abstract

This report summarizes our team's work on the RBE 2001 Unified Robotics I final project. This course centers around learning about actuation, and our final project required us to build a robot that was capable of picking up 'reactor rods' from the vertical position and storing them in 'storage tubes' in the horizontal position. This required us to create a robot featuring a four bar lifter, Bluetooth communication, and autonomous navigation. Our robot was successfully able to navigate the playing field, pick up the reactor rods, store them in appropriate storage tubes, obtain new reactor rods, and put them back in the reactor. We accomplished this by creating a minimalistic design constructed from laser cut acrylic and 3D printed parts.

Table of Contents

Table of Figures

Introduction

Our project was to design a robotic system to run the supply, storage, and resupply cycle in the nuclear facility pictured below using Bluetooth communication. The reactors are located on the ground on the left and right ends of the field. The funnel-shaped storage tubes, used to hold spent reactor rods, are located at the top of the wall in the upper portion of the diagram. The new reactor rods are stored in the supply tubes, located at the top of the bottom wall.

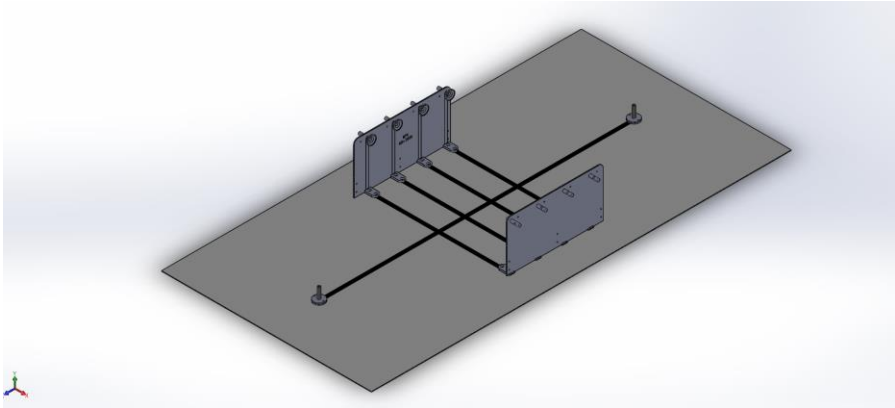


Figure 1: Field Rendering

The robot was charged with the task of running the reactor facility. First, the robot must remove the spent rod from reactor A. The robot will take this rod and communicate with the field via Bluetooth to determine which storage tubes are open, choose an open storage tube, and place the spent rod in the open tube. The robot will then turn around, use Bluetooth to find a supply tube with an available rod, retrieve the rod, and place it into the reactor. The robot will then drive to the other reactor and repeat this process. This loop continues as long as there are empty storage tubes and available storage tubes.

Methodology

Robot Design

The primary form of prototyping and design was using the virtual environment of SolidWorks. The first thing that was designed and modeled was the drive system. The first idea considered was a two-wheel drive approach where the front, free-spinning motors were simple omniwheels and the back wheels were the sticky wheels available in the parts buffet. The other idea considered was another two-wheel drive, except with this design there would be three wheels on each side. The front and back wheels would be free-spinning omniwheels, and the sticky middle wheels would be driven. The former option was chosen because it was decided it would be better to be able to have our line sensor in the middle of the robot and not be over the driven wheels.

After the height of the chassis was determined, the next part of the design process was the four bar to be used to move the grabbing mechanism from the height of the reactor to the high points of the storage and supply tubes. A carriage style four bar was chosen because it would make it easier to change the orientation of the tube from vertical to horizontal and back depending on where the tube

was being placed or retrieved from. The carriage track was designed to raise the tube three inches vertically to clear the reactor tube before being turned horizontally. The track was then placed in relation to the front of the robot such that when the front of the robot was against the tube walls, the servo gripper would have enough horizontal push forward to fully place the rod in the storage tube and enough backtrack to pull the rod out completely from the supply tube. The lengths of the four bar were determined using a guess and check method, with the main parameter being not to have a transmission angle less than 45 degrees. It was also decided to create an attachment piece for the servo gripper so it would be possible to more precisely grab the rod without it unintentionally rotating from jostling during transit.

During the design of the robot, necessarily sensors and mechanical guides were also considered and implemented. When the robot approaches the tubes and the reactor, mechanical guides would position the robot correctly. Limit switches are also used to signal to the Arduino when the robot is in place, which would trigger the arm to mobilize. For both positions, we rely on the bases of the mechanical alignment features of the field. For the reactor, the limit switch is placed on the top guide to push against the reactor tube. For the storage and supply tubes, the limit switch is positioned to trigger against the base of the alignment features.

(picture of limit switches and alignment part of base)

We also planned on using

- Design in SW
- Verify with math
- Manufacture

Code Organization

Implementing such a large and complex web of robotic state machines, control systems, and communication systems required over 1200 lines of code (excluding external libraries not specific to this project). Developing, testing, and debugging such a large amount of code efficiently made a multi-file structural approach necessary. Thus, we decided to group related sections of code together in separate files using C++ namespaces. A namespace allows for the grouping of related constants, variables, objects, and functions without the risk of identifier (name) collisions between files. For example, both the MotorL and MotorR namespaces (for the drive motors) have a DcMotor object named “motor” and an initialization function named “setup”. The specific instance of “motor” or “setup” can be accessed using the double colon operator, as shown in the code snippet below.

```
// Initializes ReactorBot (call in setup).
void robotSetup() {

    // Namespace initializations
    MotorL::setup();
    MotorR::setup();
    Arm::setup();
    Gripper::setup();
    GyroDrive::setup();
    LineFollower::setup();
    Bluetooth::setup();
    IndicatorLed::setup();
```

Figure 2: Initialization of namespaces

This allowable naming overlap between namespaces allows for quick, logical, and uniform code structuring. For example, as shown above, we gave every namespace initialization function the same name ("setup"). Thus, we never wasted time trying to determine the name of each individual initializer. Instead, if we wanted to use a namespace, we simply called "setup" on that namespace and were guaranteed for it to be the function we intended.

Namespaces also proved incredibly useful in organizing code by location and functionality. For example, the file "Arm.h" contains the namespace "Arm", which contains the following:

- Constants for Arduino pin configurations
- A DC motor object for interfacing with the arm drive motor
- A function for getting the arm angle from a potentiometer reading
- A PID controller object for controlling the arm
- A function which moves the arm to a given set-point with PID control
- Constants for several arm angle set-points needed for the robot's behavior

On average, each namespace used in the robot was less than 70 lines of code. Thus, if there was a problem with the arm, we could find the exact location in over 1200 lines of code where the fix was needed within a few seconds. This made field debugging easy and field-test turnaround times very fast. It will also make it much easier for anyone reading the code to traverse the functional tree and find the code they are curious about.

Analysis

Four Bar Force Analysis

Once the design was finished in Solid Works, the mechanics of the four-bar had to be tested. These mechanics were tested in Mathcad before the robot was constructed by implementing simple free body diagrams and equations.

For the sake of brevity, and since all the free body diagrams have identical equations, the four-bar at 45° is the one that will be examined for the analysis.

It is important to add that all distances are in inches, all forces are in lbf, and all torques are in in*lbf.

W_{grip}=Weight of the grip, W_2=Weight of the second link, W_1=Weight of the first link

x_A=Position of point A in the x direction in relation to A, y_A= Position of point A in the y direction in relation to A

x_B= Position of point B in the x direction in relation to A, y_B= Position of point B in the y direction in relation to A

x_C= Position of point C in the x direction in relation to A, y_C= Position of point C in the y direction in relation to A

x_D= Position of point D in the x direction in relation to A, y_D= Position of point D in the y direction in relation to A

x_E= Position of point E in the x direction in relation to A, y_E= Position of point E in the y direction in relation to A

x_F= Position of point F in the x direction in relation to A, y_F= Position of point F in the y direction in relation to A

x_G= Position of point G in the x direction in relation to A, y_G= Position of point G in the y direction in relation to A

x_H= Position of point H in the x direction in relation to A, y_H= Position of point H in the y direction in relation to A

A_x=Force of A in the x direction, A_y=Force of A in the y direction

B_x=Force of B in the x direction, B_y=Force of B in the y direction

C_x=Force of C in the x direction, C_y=Force of C in the y direction

D_y=Force of D in the y direction

E_x=Force of E in the x direction, E_y=Force of E in the y direction

F_x=Force of F in the x direction, F_y=Force of F in the y direction

G_x=Force of G in the x direction, G_y=Force of G in the y direction

H_x=Force of A in the x direction

T=Torque at point A

All the used equations to find out the forces and torque to move the robot were summation of forces/moments.

Since there are so many unknowns, there had to be many equations, so the free body equations were constructed for three diagrams; the crank, the coupler, and the entire system. The equations themselves

are just determining which equation each force is for. The trickiest part is that since all these distances are based on A, for the moment of the coupler it is important to make the system in relation to C.

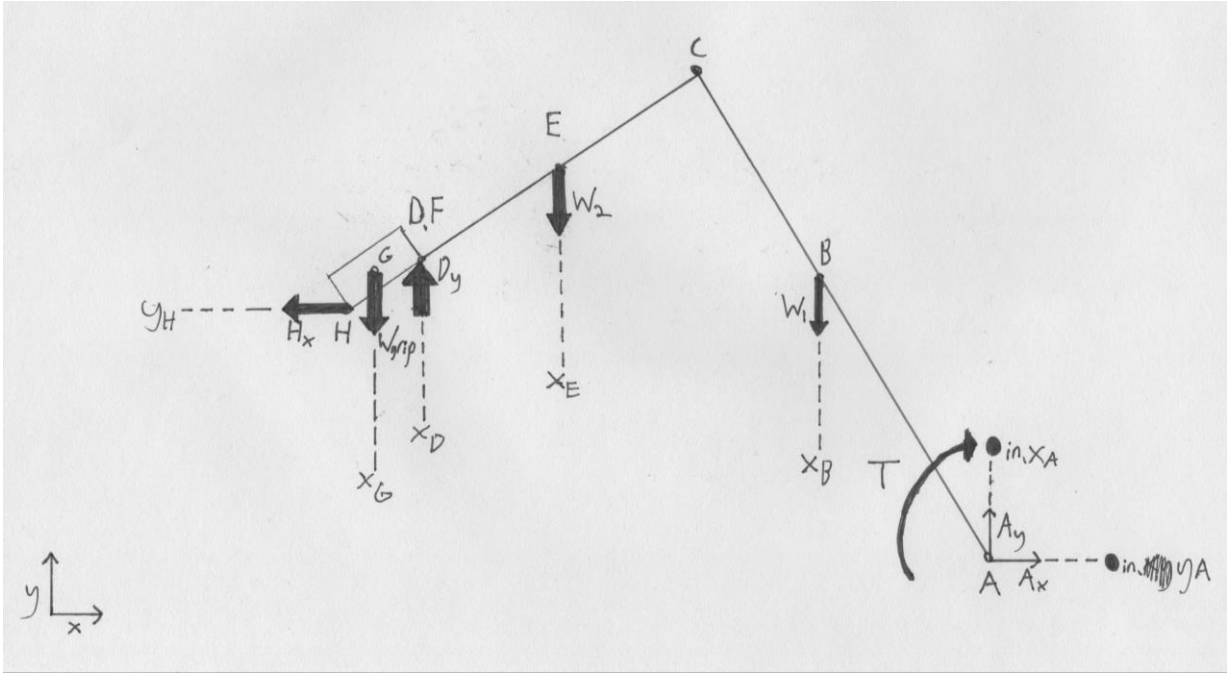


Figure 3: Full system FBD

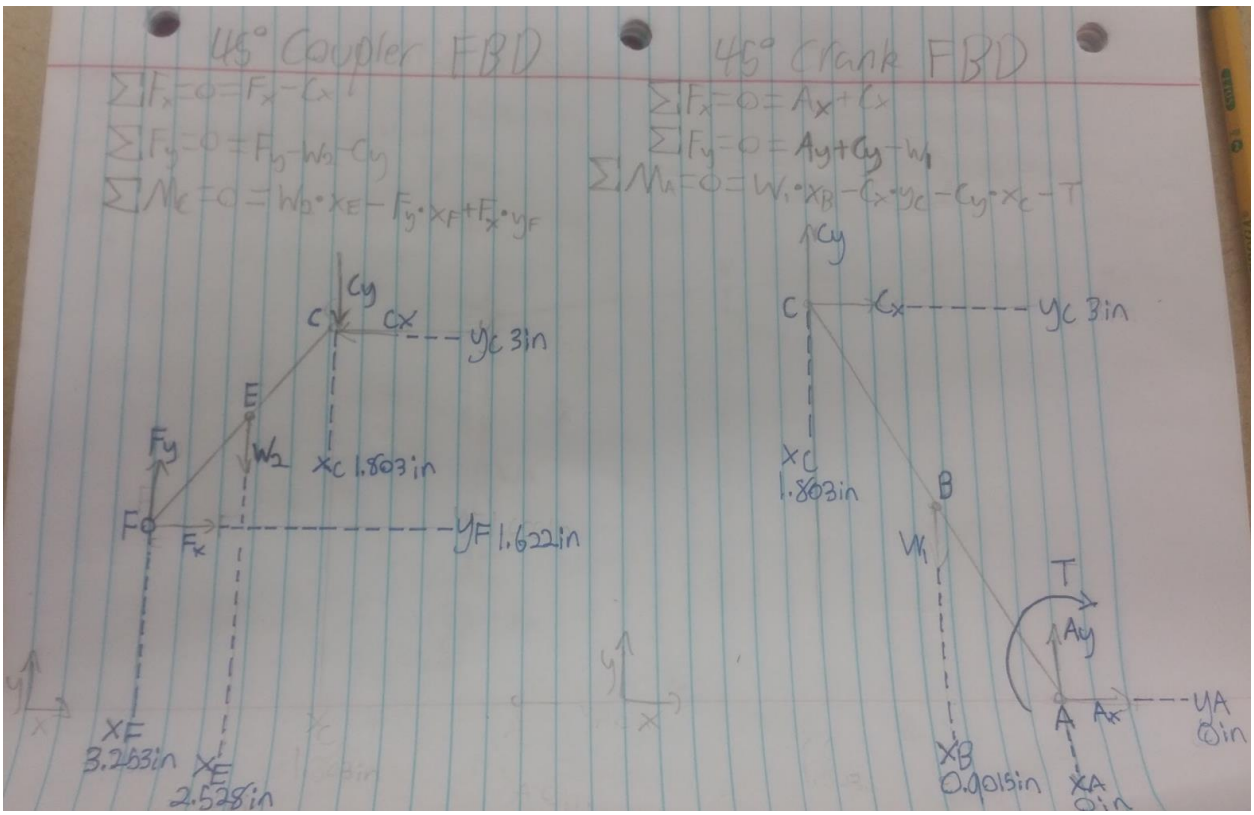


Figure 4: FBD of coupler and crank

Crank Equations:

Fx=0=Ax+Cx, Fy=0=Ay+Cy-W1, MA=0=W1(xB-xA)-Cx(yC-yA)-Cy(xC-xA)-T

Coupler Equations:

Fx=0=Fx-Cx, Fy=0=Fy-Cy-W2, MA=0=W2(xE-xC)-Fy(xF-xC)+Fx(yC-yF)

System Equations:

Fx=0=Ax-Hx, Fy=0=Ay+Dy-Wgrip-W2-W1, MA=0=Wgrip(xG-xA)+W2(xE-xA)+W1(xB-xA)-Dy(xD-xA)+Hx(yH-yA)-T

From these equations we found that the torques at the top, 45-degree position, and bottom were 0.163, 0.734, and 0.038 respectively.

Once the torque was found, the velocity had to be calculated.

G_{train}=The gear ratio of the robot

M_z=The torque from the FBD, T_m=Torque of the motor, M_s=Speed of the motor

η=Efficiency of the gear train

ω=Angular velocity

R_1=Length of link one

V=velocity

Velocity Equations:

Mz=Tm*efficiency*Geartrain

v=angular velocity*R(length of link 1)

From these equations we found the velocity to be about 8.162 in/sec at all positions.

These equations are much easier to solve than the previous ones. If the output torque, the one calculated from the FBD, is equal to the torque of the motor multiplied by the Gear train and the efficiency, then the torque of the motor can be found. Plugging that into a graph of the 393 motors gives the Speed of the motor, which is divided by the gear train and converted into radians per seconds. Using the equation that velocity is equal to angular velocity multiplied by distance gives the value of the velocity.

Four Bar Gearing

It was decided that an adequate speed for the full arm motion was about one second for the arm to move from extreme to extreme, which is about 130 degrees. The Pololu motor in our lab kit has an output RPM of 200, which translates to 1200 degrees per second. With a target output of 130 degrees per second, the optimal gear ratio would be 9.23:1. The chosen transmission was two stages,

each with a twelve tooth to thirty-six tooth gear, providing a 9:1 speed reduction. The chosen transmission provides us with an output torque of 8 ft-lbs.

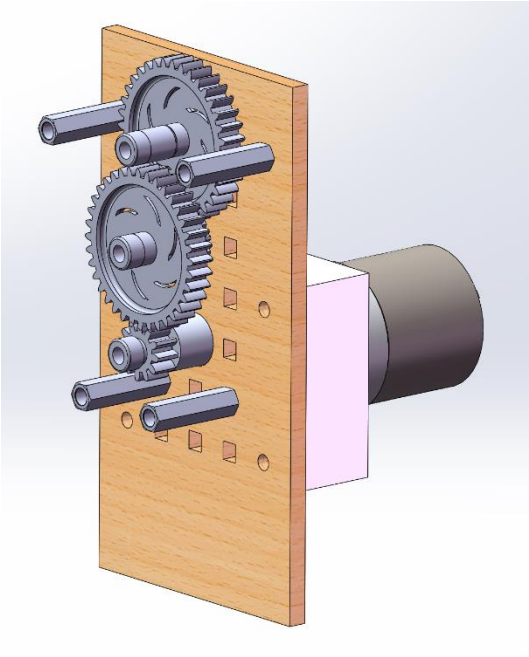


Figure 5: Gear Reduction

Drive/turning analysis

Using a direct drive for the drive wheels and the motor datasheet, we determined that the robot would turn at about 1 rotation per second and drive at a speed of 28 inches per second. Using the mass of the robot and a coefficient of friction of 1, we determined that the robot would be unable to stall the drive motors by driving into a wall.

Power Analysis

We also completed a power analysis to determine the required battery for our robot. The figure below shows the nominal and peak current draw and power required for each component

Component	Supply Vcc (V)	Current (A)		Power (W)	
		Nominal	Peak	Nominal	Peak
Arm Drive Motor	12.4	0.34	5	4.216	62
Left Drive Motor	12.4	0.3	5	3.72	62
Right Drive Motor	12.4	0.3	5	3.72	62
Gripper Servo	5	0.012	1	0.06	5
Line Sensor	5	0.08	0.08	0.4	0.4
Arduino Mega	5	0.1	0.1	0.5	0.5
Bluetooth Module	5	0.04	0.04	0.2	0.2
9DOF IMU	5	0.013	0.013	0.065	0.065
	Totals	**1.2**	**16.2**	**12.9**	**192.2**

Figure 6: Power requirements table

From this we determined that a 12V 2.2Ah 20-30C LiPo battery that we already had would be plenty to power our robot. The continuous current rating is the capacity multiplied by the C rating, which gives us a continuous current draw capability of 44A. This is plenty to cover the theoretical peak of our robot at 16.2A. The battery life of our robot at nominal current draw will be approximately 1.8 hours, which is more than enough to cover our 10-minute demonstration.

Control Systems

Use of PID Control

The success of our robot in field navigation was first contingent on robust, reliable control systems. PID control was selected as it is well-known, easy to implement, and responds well to variations. In order to improve abstraction and minimize repeated code, a “PidController” class was developed. On construction, it takes the following inputs:

- Proportional, integral, and derivative gains (kP, kI, kD)
- Output response cutoffs (minimum, maximum)
- Reset time (optional, default is 3.4 * 10^38 seconds, essentially infinite)

PidController objects automatically measure the time difference between calls to the “update” method, which inputs an error, performs one iteration of the controller, and returns the updated response. If the time difference between calls is greater than the reset time, the controller automatically resets (i.e. sets the integral and velocity terms to zero) to avoid windup. In many cases, this eliminates the need to manually reset controllers in the state machine. For example, the arm PID controller has a reset time of 0.1 seconds. Once the arm has lowered to pick up a rod from a reactor, a state change is initiated and the “update” method is no longer called. After the 1 second grip time, another state change triggers the arm moving to its upright position, and the controller (having passed 0.1 seconds) automatically resets, preventing the negative integral term accumulated during the previous state from slowing the arm’s positive raising response.

The PidController class also has a method “isStabilized”, which inputs a maximum error and maximum error derivative, and returns a boolean indicating if the system has stabilized at the set-point within these allowed ranges. This method can be (and is) used as an indicator for when a PID response is complete, and usually triggers a state change to the next desired action.

Gyroscopic Steering Control

The namespace “GyroDrive” (in “GyroDrive.h”) contains an interface for the robot’s Bno055 9DOF IMU, and two PidController objects for using it to control the robot’s drive system. The IMU contains an on-board accelerometer, gyroscope, and magnetometer, and performs on-board sensor fusion algorithms to determine absolute heading, pitch, roll, and acceleration. We use it exclusively for absolute heading and yaw velocity.

The first PID controller performs absolute heading control by reading the heading from the IMU and applying a differential voltage to the motors (positive voltage for one, negative for the other). This is important for steering towards different locations on the field and remaining aligned to the field for line following. Implementing absolute angle steering in PID proved to be a difficult challenge due to the modulo nature of heading. The IMU returns heading modulo 2pi, increasing in clock-wise rotation. Thus, there is a discontinuous toggle point where a minute change in heading will toggle the output between 0 and 2pi. This makes the traditional “target minus current” error computation invalid. With this method, if the robot is turned towards 23/24*pi and is tasked to turn to 1/24*pi, it will turn almost a full rotation counterclockwise to reach the target rather than making a small clockwise turn. Even worse, if it overshoots by more than 1/24*pi, it may hit the toggle point, read its heading as 2*pi, and attempt the full counterclockwise turn a second time. In order to eliminate this issue, an error transformation was applied to move the toggle point of the error to +/-pi, or one half-rotation away from the set-point, as shown graphically below:

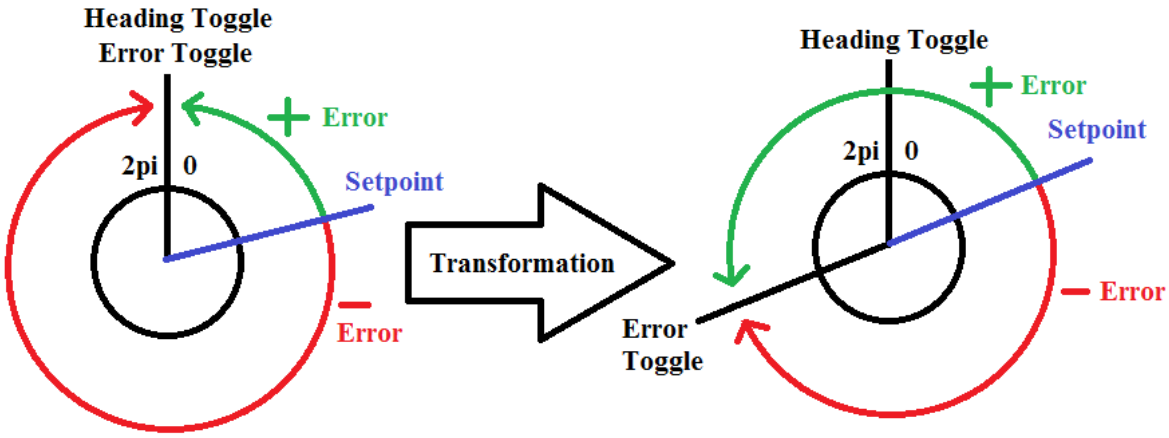


Figure 7: Gyroscopic Heading Error Transformation

Because the robot will never attempt to turn more than one half-rotation between field orientations, this system prevents the error from ever encountering a toggle point, thus avoiding undesirable behavior. It also guarantees that the robot will choose the closest direction to rotate from its current orientation.

Implementing this transformation in C++ required four conditionals based on the range of the set-point and the range of the current heading. The implementation is shown below, where:

- "h" is the heading set-point
- "hc" is the current heading
- "err" is the computed error

```
// PID turns robot to given absolute heading (rad)
// If stabilized, returns true and brakes motors
bool setAngle(float h) {
    float err;
    float hc = heading();
    if(h <= PI) {
        if(hc <= h + PI) err = h - hc;
        else err = h + TWO_PI - hc;
    } else {
        if(hc <= h - PI) err = h - TWO_PI - hc;
        else err = h - hc;
    }
```

Figure 8: Gyroscopic heading error calculation

We also implemented PID control for fixed angular velocity via motor voltage control, which is used in the line following. We found that proportional control alone was ineffective, as the motor voltage drops to 0 once at the target angular speed, causing the motors to stop and eventually oscillate. Instead, integration was primarily used. That is, the motors increase their differential voltage (thus increasing their free-spinning RPM) until the target angular velocity is achieved (very intuitive for DC motors). Applying a small amount of proportional control helped change the motor voltage faster at the beginning, and no software dampening was required as the internal motor and turning friction provided plenty of physical dampening to the system.

Proportional-Controlled Line Following

The namespace "LineFollower" in "LineFollower.h" contains an interface for the robot's line sensor and a PID controller for line-following. We used a QTR-8 analog sensor array for line tracking. This sensor provides eight 10-bit analog "darkness" readings in a linear array perpendicular to the line being followed. Using an algorithm similar to the discrete "center of mass" calculation in mechanics, we implemented a function which uses all 8 sensor readings to give a continuous estimation of the robot's displacement off the line, i.e. "center of dark". Positive readings indicate the robot is to the right of the line, negative readings to the left.

This continuous value is then fed as the error in a proportional controller which outputs a desired angular velocity. This angular velocity is then fed to the angular velocity controller in "GyroDrive", providing two layers of control. The result is the robot turning left when to the right of the line, and turning right when to the left of the line. While this sounds like a traditional line follower, the continuous proportional response of the turning allows the robot to quickly converge onto the line and then remain centered on it indefinitely. No integration is needed in this controller because the angular

velocity should be 0 when the robot is centered on the line – the difference in motor voltage due to motor biases is handled by the angular velocity controller one layer down.

Having two layers of control drastically improved the line follower's smoothness and convergence time. The original controller mapped line distance directly to differential motor voltage (1 layer). Even when thoroughly tuned, the response was either too jittery or too slow to respond to error, causing it to pass the line when intersecting it at an angle. Even when it did line follow successfully, the steady state error (i.e. distance off-center) was large enough to be clearly visible to us during early testing.

Field Navigation

Navigating the field was achieved through a combination of limit switches, line tracking, and a simple 2-dimensional mapping system. We decided to give each unique point on the field a discrete x-y coordinate which represented its position relative to other field positions. Thus, the robot could be given a target position, its current position, and decide at a high-level the turns and driving distances needed to navigate there. The field map is shown below:

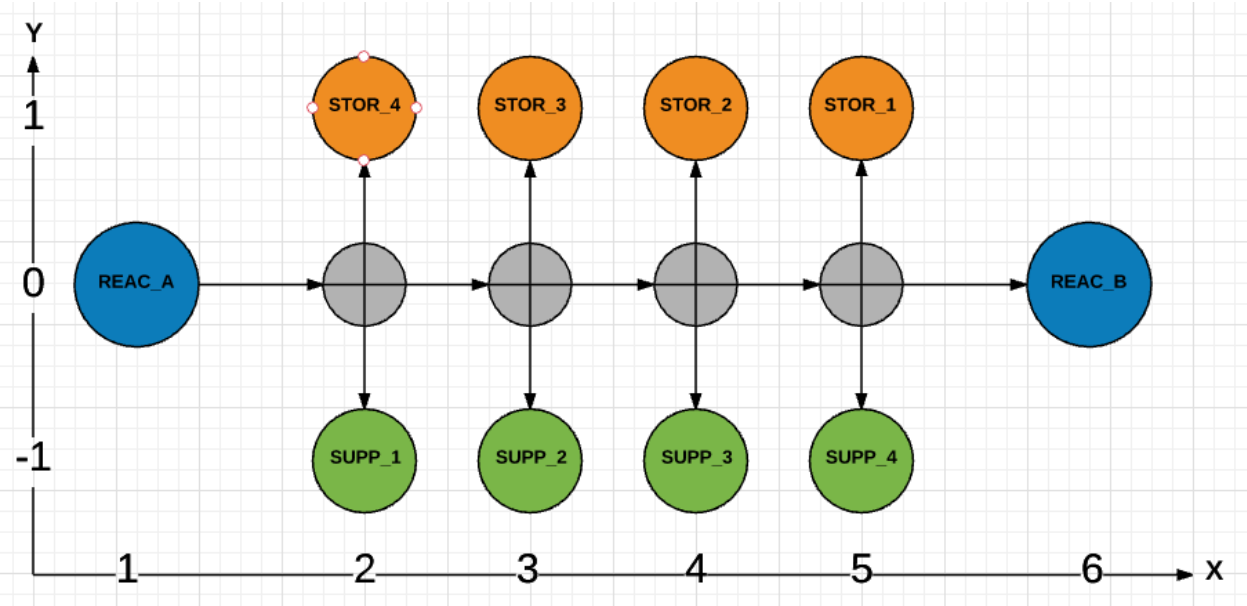


Figure 9: Field Map overlaid with X-Y Coordinate System

In order to improve abstraction and readability, a "FieldPosition" class with integer (x, y) fields was created in "FieldPosition.h". This file also enumerates all the critical field positions as constants: reactors, storage tubes, and supply tubes. The storage and supply tubes are enumerated as arrays of four FieldPosition objects in order to simplify the target position selection algorithm implemented in the state machine.

The robot assumes an initial field position of (2, 0) at startup and uses sensors to track its position as it navigates the field. When navigating in the x-direction, the line sensor uses absolute darkness thresholds and a rising edge detector to determine when it has crossed a 4-way line intersection on the field (see "LineFollower.h"). It then decides to increment or decrement its x-position depending on its heading (facing left or right). The robot also uses the reactor limit switch as a position

indicator when moving in the x-direction, and the tube limit switch when navigating in the y-direction (see “StateMachine.h”).

State Machine Design

The entire state machine of the robot is described in “StateMachine.h”. Despite the apparent complexity of the task of refueling two reactors, the use of multiple state variables significantly simplified and reduced the number of required states. The robot uses three state variables:

1. reactor: the current reactor being refueled (A or B)
2. task: the current task being performed for that reactor
3. state: the lowest-level where robot actions are performed

C++ enumerations were used to improve readability of the state machine and avoid null states. For each reactor, the “task” iterates through 4 stages, each of which involves navigating to a location, doing something with the arm and gripper, and re-orienting to mid-field:

1. TASK_EMPTY_REACTOR: Go to reactor and collect spent fuel rod
2. TASK_FILL_STORAGE: Go to nearest available storage tube and deposit spent rod
3. TASK_GET_SUPPLY: Go to nearest available supply tube and get new fuel rod
4. TASK_FILL_REACTOR: Return to reactor and deposit new rod

For each task, the low-level “state” variable iterates through a less linear ordering of up to 20 stages depending on the current reactor and task. This state machine is too complex to represent with a state transition diagram on paper, so instead its general structure will be shown in the table on the next page with exceptions and nuances discussed further on.

Order	**Name (STATE_...)**	**Description**
1	DECIDE_X	Decide target heading to rotate to target x-coordinate
2	TURNTO_X	Gyro steer towards target heading (PID loop)
3	GOTO_X	Line follow to target x-coordinate
4	DECIDE_Y	Decide target heading to rotate to target y-coordinate
5	TURNTO_Y	Gyro steer towards target heading (PID loop)
6	GOTO_Y	Line follow until tube limit switch contact
7	DECIDE_ARM	Decide proper arm target angle depending on task
8	ARM_FORWARD	Rotate arm to target angle (PID loop)

9	DECIDE_GRIPPER	Begin gripper open or close action depending on task
10	MOVE_GRIPPER	Wait a fixed period of time for gripper to open or close
11	ARM_REVERSE	Rotate arm to far back angle (PID loop)
12	BACK_TO_LINE	Gyro drive in reverse until line intersection contact (PID loop)
13	DECIDE_TASK	Choose next task, target position, and reactor

Figure 10: Special States Summary

Additional states are required based on the task and the location of the robot. For example, when at a reactor and about to refuel, the robot stops to lower the arm partway in order to avoid slippage in the four-bar while depositing in the reactor. As another example, when preparing for 90-degree turns in the middle of the field, the robot briefly enters "inching" states where it drives forward until the VTC is approximately over the line intersection. This must be done because the line tracking sensor is approximately 3 inches in front of the VTC, so an instantaneous turn would set the robot off the line leading to a storage or supply tube. All of these special states are described in the following table.

Name (STATE_)	**Description**
BEGIN	Set all initial state variables and target position (initializer state for state machine)
PREP_DEPOSIT_1	Lower arm half-way to prep for reactor deposit (performing deposit in 1 motion was unreliable in field testing due to slippage in the four-bar)
PREP_DEPOSIT_2	Raise arm slightly to prep for reactor deposit (Initial half-way lowering always lowered the arm slightly too far due to play in the four-bar)
APPROACH_REACTOR	Line follow at reduced speed until reactor limit switch contact to avoid field damage
INCH_X	Drive until VTC is on field line intersection
INCH_Y	Drive until VTC is on field line intersection

PICK_STORAGE	Stop the robot and repeatedly check field for available storage tube (priority given to that closest to the current reactor)
PICK_SUPPLY	Stop the robot and repeatedly check field for available supply tube (priority given to that closest to the current reactor).

Figure 11: Special States Summary

The state transition algorithm is designed so that this robot refuels reactors indefinitely, automatically alternating between reactor A and reactor B, and waiting for available storage and supply tubes when necessary. Thus, it is capable of complete autonomy. For more details on the state transition logic not discussed in this section, see "StateMachine.h" (in appendices).

Results

Overall, we were very pleased with the outcome of the robot. We began brainstorming and designing the robot very early on, which gave us ample amounts of time to test and refine our design. Our Solidworks model was completed before we began construction of the robot, so we were certain that the final design would properly align with the reactor and storage tubes. Our robot used a very minimalist design which decreased complexity and possible points of failure. Using laser cut acrylic pieces and 3D printed plastic allowed us to quickly make parts for the robot and cut down on part totals and assembly time.

Discussion

Despite our best efforts, we still ran into some complications. First was the use of the Pololu motors with the lab kit motor drivers. The drivers were not properly rated to ensure that they could handle the motors under all conditions, and one of ours was destroyed under normal operating conditions. This led us to purchase more appropriate drivers for the motor. The second and most significant complication that we encountered was a software glitch related to timers. We learned that various hardware timers in the Arduino were mapped to specific PWM pins. Additionally, the servo library uses the timers in a specific order. Our problem related to us unknowingly using timer 5 in the servo library while also trying to use the PWM pins controlled by timer 5. We could have solved the issue by either editing the servo library to use a different timer, or by changing the PWM pins controlling the motor drivers. We chose to take the latter approach.

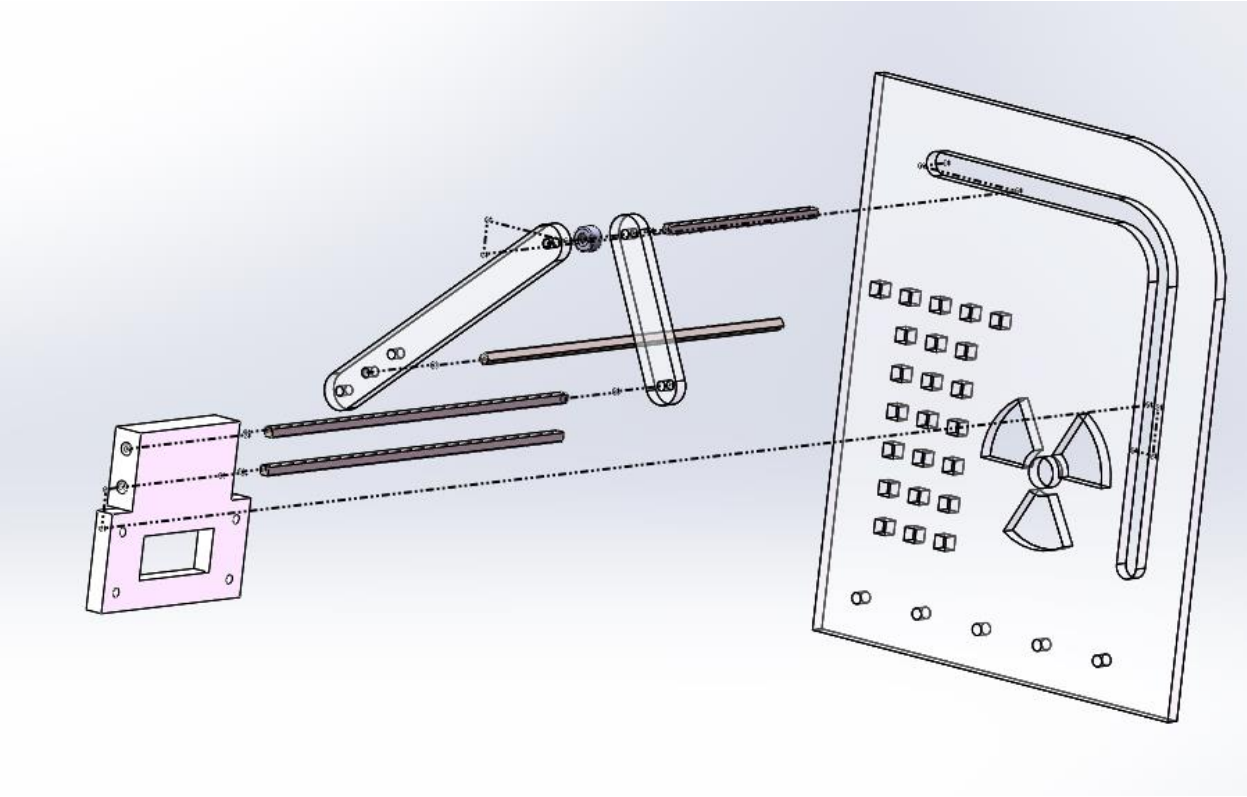
Conclusion

There is not much that we would change if we were to do this project again. One thing we would do is to put more mathematical optimization into the four bar. We learned how to do the calculations for the carriage four bar after the CDR and by that point we had already built our robot. We had already done a gearing for speed that would give us plenty of torque, but it would have been nice to optimize our transmission angles to build an even better robot.

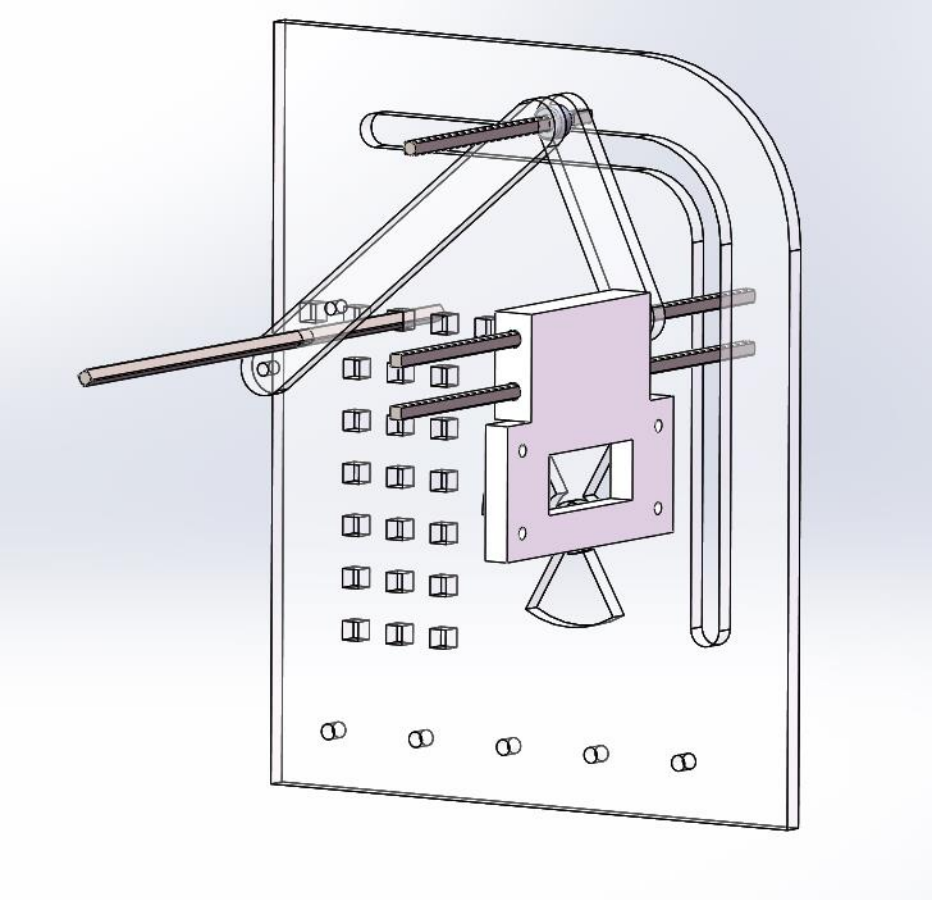
Appendix A: Work Contributions

Name	Lab Work	Project Work
Lucius Park	25%	25%
Michael Sidler	25%	25%
Dan Oates	25%	25%
Tabitha Gibbs	25%	25%

Appendix B: Four Bar Exploded View



Exploded Four Bar. Please note there was no way we could find to have our exploded line view sketches connect our circular points to floating points in the carriage track.



Assembled Four Bar

Appendix C: Bill of Materials

Part Number	Part Name	Qty.	Material	Unit Weight (lb)	Extended Weight (lb)	Unit Cost ($)	Extended Cost ($)
1	Bumper	1	Acrylic (Medium-high impact)	0.44	0.44	0.3199896938	0.3199896938
2	Omni Wheel Mount	2	ABS	0.12	0.24	1.636363636	3.272727273
3	Drive Motor Mount	2	ABS	0.07	0.14	0.9545454545	1.909090909
4	Pololu Motor	3	Various	1.3	3.9	45	135
5	New Wheel Hub	2	ABS	0.1	0.2	1.363636364	2.727272727
6	Sticky Wheel Tire	2	Polyurethane 11671	0.06	0.12	1.5	3

7	Top Chassis Plate	1	Acrylic (Medium-high impact)	0.48	0.48	0.3464455346	0.3464455346
8	Angle Bracket	2	Aluminum 6061 Alloy	0.04	0.08	1.3491	2.6982
9	Left Carriage Plate	1	Acrylic (Medium-high impact)	0.3	0.3	0.2198354392	0.2198354392
10	Spacer Thin	9	Delrin 2700	0.001	0.009	0.06	0.54
11	Servo Gripper Mount	1	ABS	0.19	0.19	2.590909091	2.590909091
12	Bearing Flat	9	Delrin 2700	0.007	0.063	0.07	0.63
13	Transmission Plate	1	Acrylic (Medium-high impact)	0.11	0.11	0.08125722538	0.08125722538
14	36 Tooth Gear	2	PE High density	0.01	0.02	1.25	2.5
15	12 Tooth Gear	2	PE High density	0.005	0.01	1.25	2.5
16	Shaft Adapter	1	Aluminum 6061 Alloy	0.076	0.076	0.6	0.6
17	Stand-Off 1 in	6	Aluminum 6061 Alloy	0.06	0.36	0.56	3.36
18	Arm Motor Mount	1	ABS	0.1	0.1	1.363636364	1.363636364
19	Plastic Washer	1	Teflon	0.001	0.001	0.05	0.05
20	Spacer Thick	3	Delrin 2700	0.003	0.009	0.06	0.18
21	Screw 632 .5in	7	Alloy Steel	0.007	0.049	0.099	0.693
22	Four Bar Link 1	2	Acrylic (Medium-high impact)	0.01	0.02	0.006928910692	0.01385782138
23	Four Bar Link 2	2	Acrylic (Medium-high impact)	0.02	0.04	0.01259801944	0.02519603888
24	Shaft 6in	3	Alloy Steel	0.03	0.09	1.87375	5.62125
25	Bar Lock Plate	4	Alloy Steel	0.01	0.04	0.13	0.52
26	Right Carriage Plate	1	Acrylic (Medium-high impact)	0.29	0.29	0.2129065285	0.2129065285
27	Shaft 3in	4	Alloy Steel	0.013	0.052	1.2475	4.99
28	Shaft 2in	1	Alloy Steel	0.01	0.01	1.2475	1.2475
29	Screw 832 .5in	48	Alloy Steel	0.007	0.336	0.0739	3.5472

30	Screw 632 .25	6	Alloy Steel	0.004	0.024	0.099	0.594
31	Screw 832 .625	10	Alloy Steel	0.008	0.08	0.0999	0.999
32	Nut 832 Hex	14	Alloy Steel	0.006	0.084	0.11	1.54
33	Shaft Collar	21	Alloy Steel	0.01	0.21	0.09	1.89
34	Spacer .5in	1	Teflon	0.007	0.007	0.1	0.1
35	Metal Washer	8	Aluminum 6061 Alloy	0.005	0.04	0.03	0.24
36	Omni Wheel	2	Various	0.12	0.24	12.46	24.92
37	Servo Gripper	1	Various	0.09	0.09	17.95	17.95
				Assembly Weight (lb)	8.55	Assembly Cost ($)	228.9932746

Note: The values in the bill of materials were originally calculated in SolidWorks. However, because our main assembly contained several subassemblies, it was easier to organize in an external Excel file.

Appendix D: Code

ReactorBot.ino

```
//*************************************************************/
// TITLE
//*************************************************************/

// ReactorBot.ino
// Executed code for ReactorBot RBE project.
// RBE-2001 A17 Team 7

#include "StateMachine.h"

//*************************************************************/
// MAIN FUNCTION DEFINITIONS
//*************************************************************/

// Runs once on Arduino reset.
void setup() {
      robotSetup();
}

// Runs repeatedly after setup.
void loop() {
```

```
        robotLoop();
}
```

StateMachine.h

```
//*************************************************************/
// TITLE
//*************************************************************/

// StateMachine.h
// File describing ReactorBot operating state machine.
// RBE-2001 A17 Team 7

#pragma once

//*************************************************************/
// DEPENDENCIES
//*************************************************************/

// Included Libraries
#include "Arduino.h"
#include "LimitSwitch.h"

// High Level Control
#include "FieldPosition.h"
#include "Bluetooth.h"

// Physical Object Namespaces
#include "MotorL.h"
#include "MotorR.h"
#include "Arm.h"
#include "Gripper.h"
#include "IndicatorLed.h"

// Drive Controllers
#include "GyroDrive.h"
#include "LineFollower.h"

//*************************************************************/
// INTERNAL ODOMETRY
//*************************************************************/

// Field location
FieldPosition currentPos(2, 0);
FieldPosition targetPos = REACTOR_A;

// Field orientation
float targetHeading = 0;
const float HEADING_U = PI * 0.0 / 2.0; // Up
const float HEADING_R = PI * 1.0 / 2.0; // Right
const float HEADING_D = PI * 2.0 / 2.0; // Down
const float HEADING_L = PI * 3.0 / 2.0; // Left
```

```
// Arm orientation
int targetArmAngle = 0;

// Motor angle to inch VTC onto line intersection
const float VTC_INCH_ANGLE = 2.465;

//*************************************************************/
// STATE MACHINE
//*************************************************************/

// Current reactor being refueled
enum reactor_t {
      A = 1,
      B = 6
} reactor;

// Task for current reactor
enum task_t {
      TASK_EMPTY_REACTOR,
      TASK_FILL_STORAGE,
      TASK_GET_SUPPLY,
      TASK_FILL_REACTOR,
} task;

// State within current task
enum state_t {
      STATE_BEGIN,
      STATE_DECIDE_X,
      STATE_TURNTO_X,
      STATE_GOTO_X,
      STATE_PREP_DEPOSIT_1,
      STATE_PREP_DEPOSIT_2,
      STATE_APPROACH_REACTOR,
      STATE_INCH_X,
      STATE_DECIDE_Y,
      STATE_TURNTO_Y,
      STATE_GOTO_Y,
      STATE_DECIDE_ARM,
      STATE_ARM_FORWARD,
      STATE_DECIDE_GRIPPER,
      STATE_MOVE_GRIPPER,
      STATE_ARM_REVERSE,
      STATE_BACK_TO_LINE,
      STATE_INCH_Y,
      STATE_SET_TASK,
      STATE_PICK_STORAGE,
      STATE_PICK_SUPPLY,
} state;
```

```
// Field radiation level
enum radiation_t {
      RAD_HIGH = 3,
      RAD_LOW = 2,
      RAD_NONE = 1,
} radiation;

//*************************************************************/
// LIMIT SWITCHES
//*************************************************************/

const uint8_t PIN_SWITCH_REACTOR = 25;
const uint8_t PIN_SWITCH_TUBE = 24;

LimitSwitch reactorSwitch(PIN_SWITCH_REACTOR);
LimitSwitch tubeSwitch(PIN_SWITCH_TUBE);

//*************************************************************/
// HELPER FUNCTION DEFINITIONS
//*************************************************************/

// Returns true if robot is at either reactor
bool atReactor() {
      return (currentPos == REACTOR_A)
            || (currentPos == REACTOR_B);
}

// Resets both drive encoders then transitions to given state.
void resetEncoders(state_t nextState) {
      MotorL::motor.zeroAngle();
      MotorR::motor.zeroAngle();
      state = nextState;
}

// Inches forward by fixed angle then transitions to given state.
void inchForward(state_t nextState) {
      LineFollower::drive();
      if((MotorL::motor.getAngle() +
            MotorR::motor.getAngle()) >= 2.0 * VTC_INCH_ANGLE)
      {
            state = nextState;
      }
}

//*************************************************************/
// MAIN FUNCTION DEFINITIONS
//*************************************************************/

// Initializes ReactorBot (call in setup).
void robotSetup() {
```

```
	// Namespace initializations
	MotorL::setup();
	MotorR::setup();
	Arm::setup();
	Gripper::setup();
	GyroDrive::setup();
	LineFollower::setup();
	Bluetooth::setup();
	IndicatorLed::setup();

	// Limit Switch initializations
	reactorSwitch.setup();
	tubeSwitch.setup();

	// Open gripper and raise arm to back position
	Gripper::open();
	while(!Gripper::ready());
	while(!Arm::setAngle(Arm::ANGLE_BACK));

	// State machine initialization
	state = STATE_BEGIN;
}

// Robot state machine loop (call in loop).
void robotLoop() {

	// Bluetooth communication
	Bluetooth::loop(radiation);

	// State Machine
	switch(state) {

		// Initialize state machine
		case STATE_BEGIN:
			reactor = A;
			task = TASK_EMPTY_REACTOR;
			state = STATE_DECIDE_X;
			radiation = RAD_NONE;
			targetPos = REACTOR_A;
			break;

		// Decide on x turning direction
		case STATE_DECIDE_X:
			if(targetPos.x == currentPos.x)
				state = STATE_DECIDE_Y;
			else {
				if(targetPos.x > currentPos.x)
					targetHeading = HEADING_R;
				else
```

```
                        targetHeading = HEADING_L;
                  state = STATE_TURNTO_X;
            }
            break;

      // Gyro turn to face target position x
      case STATE_TURNTO_X:
            if(GyroDrive::setAngle(targetHeading))
                  state = STATE_GOTO_X;
            break;

      // Line follow to target position x
      case STATE_GOTO_X:
            LineFollower::drive();
            if(LineFollower::hitIntersection()) {
                  if(GyroDrive::heading() < PI) currentPos.x++;
                  else currentPos.x--;
            }
            if(currentPos.x == targetPos.x) {
                  switch(task) {
                        case TASK_EMPTY_REACTOR:
                              state = STATE_APPROACH_REACTOR;
                              break;
                        case TASK_FILL_REACTOR:
                              MotorL::motor.brake();
                              MotorR::motor.brake();
                              state = STATE_PREP_DEPOSIT_1;
                              break;
                        default:
                              resetEncoders(STATE_INCH_X);
                              break;
                  }
            }
            break;

      // Lower arm halfway to deposit reactor rod
      case STATE_PREP_DEPOSIT_1:
            if(Arm::setAngle(Arm::ANGLE_PREP_1))
                  state = STATE_PREP_DEPOSIT_2;
            break;

      // Raise arm up a bit to avoid reactor collision
      case STATE_PREP_DEPOSIT_2:
            if(Arm::setAngle(Arm::ANGLE_PREP_2))
                  state = STATE_APPROACH_REACTOR;
            break;

      // Line follow until reactor limit switch contact
      case STATE_APPROACH_REACTOR:
            LineFollower::drive(2.0);
```

```
            if(reactorSwitch.pressed())
                  state = STATE_DECIDE_ARM;
            break;

      // Inch forward until robot VTC is on line intersection
      case STATE_INCH_X:
            inchForward(STATE_DECIDE_Y);
            break;

      // Decide on y turning direction
      case STATE_DECIDE_Y:
            if(targetPos.y == currentPos.y)
                  state = STATE_DECIDE_ARM;
            else {
                  if(targetPos.y > currentPos.y)
                        targetHeading = HEADING_U;
                  else
                        targetHeading = HEADING_D;
                  state = STATE_TURNTO_Y;
            }
            break;

      // Turn to face target position y
      case STATE_TURNTO_Y:
            if(GyroDrive::setAngle(targetHeading))
                  state = STATE_GOTO_Y;
            break;

      // Line follow until tube limit switch contact
      case STATE_GOTO_Y:
            LineFollower::drive();
            if(tubeSwitch.pressed()) {
                  switch(task) {
                        case TASK_FILL_STORAGE:
                              currentPos.y = +1;
                              break;
                        case TASK_GET_SUPPLY:
                              currentPos.y = -1;
                              break;
                        default: break;
                  }
                  state = STATE_DECIDE_ARM;
            }
            break;

      // Stop driving and choose arm forward position
      case STATE_DECIDE_ARM:
            MotorL::motor.brake();
            MotorR::motor.brake();
            switch(task) {
```

```
                case TASK_EMPTY_REACTOR:
                        targetArmAngle = Arm::ANGLE_PICKUP;
                        break;
                case TASK_FILL_REACTOR:
                        targetArmAngle = Arm::ANGLE_DROPOFF;
                        break;
                default:
                        targetArmAngle = Arm::ANGLE_TUBE;
        }
        state = STATE_ARM_FORWARD;
        break;

// Move arm to forward position
case STATE_ARM_FORWARD:
        if(Arm::setAngle(targetArmAngle))
                state = STATE_DECIDE_GRIPPER;
        break;

// Choose gripper action
case STATE_DECIDE_GRIPPER:
        switch(task) {
                case TASK_EMPTY_REACTOR:
                case TASK_GET_SUPPLY:
                        Gripper::close();
                        break;
                default:
                        Gripper::open();
                        break;
        }
        state = STATE_MOVE_GRIPPER;
        break;

// Wait for gripper to finish action
case STATE_MOVE_GRIPPER:
        if(Gripper::ready()) {
                switch(task) {
                        case TASK_GET_SUPPLY:
                                radiation = RAD_HIGH;
                                break;
                        case TASK_EMPTY_REACTOR:
                                radiation = RAD_LOW;
                                break;
                        default:
                                radiation = RAD_NONE;
                                break;
                }
                state = STATE_ARM_REVERSE;
        }
        break;
```

```
            // Move arm to back position
            case STATE_ARM_REVERSE:
                  if(Arm::setAngle(Arm::ANGLE_BACK))
                        state = STATE_BACK_TO_LINE;
                  break;

            // Gyro drive backwards to line intersection
            case STATE_BACK_TO_LINE:
                  GyroDrive::setVelocity(0.0, -4.0); // Drive backwards
                  if(LineFollower::hitIntersection()) {
                        currentPos.y = 0;
                        if(atReactor()) state = STATE_SET_TASK;
                        else resetEncoders(STATE_INCH_Y);
                  }
                  break;

            // Inch forward until robot VTC is on line intersection
            case STATE_INCH_Y:
                  inchForward(STATE_SET_TASK);
                  break;

            // Set next robot task, state, and target position
            case STATE_SET_TASK:
                  MotorL::motor.brake();
                  MotorR::motor.brake();
                  switch(task) {
                        case TASK_EMPTY_REACTOR:
                              task = TASK_FILL_STORAGE;
                              state = STATE_PICK_STORAGE;
                              break;
                        case TASK_FILL_STORAGE:
                              task = TASK_GET_SUPPLY;
                              state = STATE_PICK_SUPPLY;
                              break;
                        case TASK_GET_SUPPLY:
                              task = TASK_FILL_REACTOR;
                              switch(reactor) {
                                    case A: targetPos = REACTOR_A; break;
                                    case B: targetPos = REACTOR_B; break;
                              }
                              state = STATE_DECIDE_X;
                              break;
                        case TASK_FILL_REACTOR:
                              task = TASK_EMPTY_REACTOR;
                              switch(reactor) {
                                    case A:
                                          reactor = B;
                                          targetPos = REACTOR_B;
                                          break;
                                    case B:
```

```
                                    reactor = A;
                                    targetPos = REACTOR_A;
                                    break;
                              }
                              state = STATE_DECIDE_X;
                              break;
                  }
                  break;

            // Set target position to closest available storage tube
            case STATE_PICK_STORAGE:
                  switch(reactor) {
                        case A: // Storage 4 is closest
                              for(int i=4; i>=1; i--)
                                    if(Bluetooth::com.storageAvailable(i)) {
                                          targetPos = STORAGE[i-1];
                                          state = STATE_DECIDE_X;
                                          break;
                                    }
                              break;
                        case B: // Storage 1 is closest
                              for(int i=1; i<=4; i++)
                                    if(Bluetooth::com.storageAvailable(i)) {
                                          targetPos = STORAGE[i-1];
                                          state = STATE_DECIDE_X;
                                          break;
                                    }
                              break;
                  }
                  break;

            // Set target position to closest available supply tube
            case STATE_PICK_SUPPLY:
                  switch(reactor) {
                        case A: // Supply 1 is closest
                              for(int i=1; i<=4; i++)
                                    if(Bluetooth::com.supplyAvailable(i)) {
                                          targetPos = SUPPLY[i-1];
                                          state = STATE_DECIDE_X;
                                          break;
                                    }
                              break;
                        case B: // Supply 4 is closest
                              for(int i=4; i>=1; i--)
                                    if(Bluetooth::com.supplyAvailable(i)) {
                                          targetPos = SUPPLY[i-1];
                                          state = STATE_DECIDE_X;
                                          break;
                                    }
                              break;
```

```
                }
                break;
        }

        // Update radiation indicator LED
        switch(radiation) {
                case RAD_HIGH: IndicatorLed::setHigh(); break;
                case RAD_LOW:  IndicatorLed::setLow();  break;
                case RAD_NONE: IndicatorLed::setNone(); break;
        }
}
```

FieldPosition.h

```
//*************************************************************/
// TITLE
//*************************************************************/

// FieldPosition.h
// Class for keeping field positions for ReactorBot.
// RBE-2001 A17 Team 7

#pragma once

//*************************************************************/
// CLASS DECLARATION
//*************************************************************/

class FieldPosition {
public:
      FieldPosition(int x, int y) {
            this->x = x;
            this->y = y;
      }
      bool operator==(const FieldPosition& fp) {
            return ((x == fp.x) && (y == fp.y));
      }
      int x = 0;
      int y = 0;
};

//*************************************************************/
// OBJECT DECLARATIONS
//*************************************************************/

// Reactors
const FieldPosition REACTOR_A(+1, +0);
const FieldPosition REACTOR_B(+6, +0);

// Storage containers
const FieldPosition STORAGE[4]{
      FieldPosition(+5, +1),
      FieldPosition(+4, +1),
      FieldPosition(+3, +1),
      FieldPosition(+2, +1),
};

// Supply containers
const FieldPosition SUPPLY[4]{
      FieldPosition(+2, -1),
      FieldPosition(+3, -1),
```

```
        FieldPosition(+4, -1),
        FieldPosition(+5, -1),
};
```

Bluetooth.h

```
//*************************************************************/
// TITLE
//*************************************************************/

// Bluetooth.h
// Namespace for ReactorBot Bluetooth communication.
// RBE-2001 A17 Team 7

#pragma once
#include "ReactorComms.h"
#include "MotorL.h"
#include "MotorR.h"
#include "Arm.h"
#include "GyroDrive.h"
#include "LineFollower.h"
#include "Timer.h"

//*************************************************************/
// NAMESPACE DEFINITION
//*************************************************************/

namespace Bluetooth {

	Timer timer; // Counts time between heartbeats
	ReactorComms com(Serial3); // Reactor communication object

	// Initializes Bluetooth and heartbeat (call in setup).
	void setup() {
		com.init();
		timer.tic();
	}

	// Performs ReactorBot Bluetooth actions (call in loop).
	void loop(int radLevel) {

		// Check Bluetooth messages
		com.update();

		// Enable or disable drive
		if(com.getRobotEnabled()) {
			MotorL::motor.enable();
			MotorR::motor.enable();
			Arm::motor.enable();
		} else {
			MotorL::motor.disable();
			MotorR::motor.disable();
```

```
                Arm::motor.disable();
                GyroDrive::resetPids();
                LineFollower::resetPids();
                Arm::resetPids();
            }

            // Send heartbeat and radiation alerts
            if(timer.hasElapsed(1.0)) {
                timer.tic();
                com.sendHeartBeat();
                switch(radLevel) {
                    case 3:
                        com.sendRadAlert(RADIATION_HI); break;
                    case 2:
                        com.sendRadAlert(RADIATION_LO); break;
                    default: break;
                }
            }
        }
}
```

GyroDrive.h

```
//*************************************************************/
// TITLE
//*************************************************************/

// GyroDrive.h
// Namespace for ReactorBot gyroscopic drive control system.
// RBE-2001 A17 Team 7

#pragma once
#include "Bno055.h"
#include "PidController.h"
#include "MotorL.h"
#include "MotorR.h"

//*************************************************************/
// NAMESPACE DEFINITION
//*************************************************************/

namespace GyroDrive {

	Bno055 imu(trb); // IMU with dot on Top Right Back of chip
	double h0 = 0.0; // Absolute IMU heading offset

	//!b Initializes IMU (call in setup)
	void setup() {
		imu.begin();
		h0 = imu.heading();
	}

	//!b Returns robot heading (rad)
	float heading() {
		return imu.heading() - h0;
	}

	// PID controllers will reset if not used for this time
	const float PID_RESET_TIME = 0.1;

	// Robot Heading PID Controller
	// Input: Heading (rad)
	// Output: Differential motor voltage (V)
	const float ANGLE_VMAX = 8.0;
	const float ANGLE_KP = 3.0;
	const float ANGLE_KI = 1.0;
	const float ANGLE_KD = 0.0;
	PidController anglePid(
		ANGLE_KP,
```

```
        ANGLE_KI,
        ANGLE_KD,
        -ANGLE_VMAX,
        +ANGLE_VMAX,
        PID_RESET_TIME);

// PID turns robot to given absolute heading (rad)
// If stabilized, returns true and brakes motors
bool setAngle(float h) {
        float err;
        float hc = heading();
        if(h <= PI) {
                if(hc <= h + PI) err = h - hc;
                else err = h + TWO_PI - hc;
        } else {
                if(hc <= h - PI) err = h - TWO_PI - hc;
                else err = h - hc;
        }
        float vdd = anglePid.update(err);
        MotorL::motor.setVoltage(+vdd);
        MotorR::motor.setVoltage(-vdd);
        if(anglePid.isStabilized(0.05, 0.01)) {
                MotorL::motor.brake();
                MotorR::motor.brake();
                return true;
        } else
                return false;
}

// Angular Velocity PID Controller
// Input: Robot angular velocity (rad/s)
// Output: Differential motor voltage (V)
const float VEL_VMAX = 12.0;
const float VEL_KP = 1.5;
const float VEL_KI = 30.0;
const float VEL_KD = 0.0;
PidController velPid(
        VEL_KP,
        VEL_KI,
        VEL_KD,
        -VEL_VMAX,
        +VEL_VMAX,
        PID_RESET_TIME);

// PID sets robot angular velocity and linear drive voltage
// w is target angular velocity (rad/s)
// v is straight line drive voltage (V)
void setVelocity(float w, float v = 0) {
        float vdd = velPid.update(w - imu.gZ());
        MotorL::motor.setVoltage(v - vdd);
```

```
            MotorR::motor.setVoltage(v + vdd);
      }

      // Resets all PID controllers in namespace
      void resetPids() {
            anglePid.reset();
            velPid.reset();
      }
}
```

LineFollower.h

```
//*************************************************************/
// TITLE
//*************************************************************/

// LineFollower.h
// Namespace for ReactorBot line follower.
// RBE-2001 A17 Team 7

#pragma once
#include "Qtr8.h"
#include "GyroDrive.h"

//*************************************************************/
// NAMESPACE DEFINITION
//*************************************************************/

namespace LineFollower {

	// Line Follower Parameters
	const float DRIVE_VOLTAGE = 4.0; // Default (V)
	const float ANGULAR_SPEED = 1.0; // Maximum (rad/s)

	// QTR-8 Analog Line Sensor
	const int THRESHOLD_WHITE = 100; // 10-bit ADC value
	const int THRESHOLD_BLACK = 700; // 10-bit ADC value
	const uint8_t PINS[8] = { A0, A1, A2, A3, A4, A5, A6, A7 };
	Qtr8 sensor(PINS);

	// Initializes light sensor (call in setup)
	void setup() {
		sensor.setup();
		sensor.setWhiteThreshold(THRESHOLD_WHITE);
		sensor.setBlackThreshold(THRESHOLD_BLACK);
	}

	// Line Follower PID Controller
	// Input: Line displacement (cm)
	// Output: Angular velocity (rad/s)
	const float KP = 1.0;
	const float KI = 0.0;
	const float KD = 0.0;
	PidController pid(KP, KI, KD,
		-ANGULAR_SPEED,
		+ANGULAR_SPEED);

	// Line follows forward with given drive voltage
```

```
	// Default drive voltage is DRIVE_VOLTAGE parameter
	void drive(float v = DRIVE_VOLTAGE) {
		float w = pid.update(sensor.linePos());
		GyroDrive::setVelocity(w, v);
	}

	// Memory for racking line intersections
	bool blackBefore = false;

	// Returns true on black line intersection (rising edge)
	bool hitIntersection() {
		bool blackNow = sensor.onBlack();
		bool intersect = (blackNow && !blackBefore);
		blackBefore = blackNow;
		return intersect;
	}

	// Resets all PID controllers in namespace
	void resetPids() {
		pid.reset();
	}
}
```

MotorL.h

```
//*************************************************************/
// TITLE
//*************************************************************/

// MotorL.h
// Namespace for ReactorBot left drive motor.
// RBE-2001 A17 Team 7

#pragma once
#include "DcMotor.h"

//*************************************************************/
// NAMESPACE DEFINITION
//*************************************************************/

namespace MotorL {

	// Arduino Pin Settings
	const uint8_t PIN_ENABLE = 31;
	const uint8_t PIN_FORWARD = 4;
	const uint8_t PIN_REVERSE = 5;
	const uint8_t PIN_ENCODER_A = 18;
	const uint8_t PIN_ENCODER_B = 19;

	// Motor Physical Properties
	const float TERMINAL_VOLTAGE = 12.0;
	const float ENCODER_CPR = 3200.0;

	// Motor object
	DcMotor motor(
		TERMINAL_VOLTAGE,
		PIN_ENABLE,
		PIN_FORWARD,
		PIN_REVERSE,
		PIN_ENCODER_A,
		PIN_ENCODER_B,
		ENCODER_CPR);

	// Encoder ISRs
	void interruptA() { motor.interruptA(); }
	void interruptB() { motor.interruptB(); }

	// Motor initialization (call in setup)
	void setup() {
		motor.setup();
		motor.enable();
```

```
            attachInterrupt(
                  motor.getInterruptA(),
                  interruptA,
                  CHANGE);
            attachInterrupt(
                  motor.getInterruptB(),
                  interruptB,
                  CHANGE);
      }
}
```

MotorR.h

```
//*************************************************************/
// TITLE
//*************************************************************/

// MotorR.h
// Namespace for ReactorBot right drive motor.
// RBE-2001 A17 Team 7

#pragma once
#include "DcMotor.h"

//*************************************************************/
// NAMESPACE DEFINITION
//*************************************************************/

namespace MotorR {

	// Arduino Pin Settings
	const uint8_t PIN_ENABLE = 6;
	const uint8_t PIN_FORWARD = 8;
	const uint8_t PIN_REVERSE = 7;
	const uint8_t PIN_ENCODER_A = 2;
	const uint8_t PIN_ENCODER_B = 3;

	// Motor Physical Properties
	const float TERMINAL_VOLTAGE = 12.0;
	const float ENCODER_CPR = 3200.0;

	// Motor object
	DcMotor motor(
		TERMINAL_VOLTAGE,
		PIN_ENABLE,
		PIN_FORWARD,
		PIN_REVERSE,
		PIN_ENCODER_A,
		PIN_ENCODER_B,
		ENCODER_CPR);

	// Encoder ISRs
	void interruptA() { motor.interruptA(); }
	void interruptB() { motor.interruptB(); }

	// Motor initialization (call in setup)
	void setup() {
		motor.setup();
		motor.enable();
```

```
            attachInterrupt(
                  motor.getInterruptA(),
                  interruptA,
                  CHANGE);
            attachInterrupt(
                  motor.getInterruptB(),
                  interruptB,
                  CHANGE);
      }
}
```

Arm.h

```
//*************************************************************/
// TITLE
//*************************************************************/

// Arm.h
// Namespace for ReactorBot arm.
// RBE-2001 A17 Team 7

#pragma once
#include "DcMotor.h"
#include "PidController.h"

//*************************************************************/
// NAMESPACE DEFINITION
//*************************************************************/

namespace Arm {

	// Arduino Pin Settings
	const uint8_t PIN_ENABLE = 11;
	const uint8_t PIN_FORWARD = 12;
	const uint8_t PIN_REVERSE = 13;
	const uint8_t PIN_ANGLE_POT = A15;

	// Motor Physical Properties
	const float TERMINAL_VOLTAGE = 12.0;

	// Motor object
	DcMotor motor(
		TERMINAL_VOLTAGE,
		PIN_ENABLE,
		PIN_FORWARD,
		PIN_REVERSE,
		0, 0, 1); // Ignored settings

	// Motor initialization (call in setup)
	void setup() {
		motor.setup();
		motor.enable();
		pinMode(PIN_ANGLE_POT, INPUT);
	}

	// Returns arm angle (10-bit ADC)
	int getAngle() {
		return analogRead(PIN_ANGLE_POT);
	}
```

```
	// Arm PID Setpoints
	const int ANGLE_BACK = 543;
	const int ANGLE_TUBE = 344;
	const int ANGLE_PREP_1 = 294;
	const int ANGLE_PREP_2 = 305;
	const int ANGLE_PICKUP = 69;
	const int ANGLE_DROPOFF = 110;

	// Arm Angle PID Controller
	// Input: Potentiometer reading (10-bit ADC)
	// Output: Motor voltage (V)
	const float PID_KP = 0.015;
	const float PID_KI = 0.011;
	const float PID_KD = 0.0;
	const float RESET_TIME = 0.1;
	PidController pid(
		PID_KP,
		PID_KI,
		PID_KD,
		-TERMINAL_VOLTAGE,
		+TERMINAL_VOLTAGE,
		RESET_TIME);

	// PID rotates arm to given setoint (1 iteration)
	// Returns true and brakes motor if arm is stable at setpoint
	bool setAngle(int setPoint) {
		motor.setVoltage(pid.update(setPoint - getAngle()));
		if(pid.isStabilized(5.0, 1.0)) {
			motor.brake();
			return true;
		} else
			return false;
	}

	// Resets all PID controllers in namespace
	void resetPids() {
		pid.reset();
	}
}
```

Gripper.h

```
//*************************************************************/
// TITLE
//*************************************************************/

// Gripper.h
// Namespace for ReactorBot gripper servo.
// RBE-2001 A17 Team 7

#pragma once
#include "Servo.h"
#include "Timer.h"

//*************************************************************/
// NAMESPACE DEFINITION
//*************************************************************/

namespace Gripper {

	// Constants
	const uint8_t PIN_SERVO = 10; // Servo PWM pin
	const float GRIP_TIME = 1.0;  // Time of grip (sec)
	const int ANGLE_OPEN = 60;    // Servo control angle
	const int ANGLE_CLOSED = 144; // Servo control angle

	Servo gripper;
	Timer timer;

	// Initializes gripper (call in setup).
	void setup() {
		gripper.attach(PIN_SERVO);
	}

	// Opens gripper.
	void open() {
		gripper.write(ANGLE_OPEN);
		timer.tic();
	}

	// Closes gripper.
	void close() {
		gripper.write(ANGLE_CLOSED);
		timer.tic();
	}

	// Returns true when the gripper is done gripping.
	bool ready() {
```

```
            return timer.hasElapsed(GRIP_TIME);
        }
}
```

IndicatorLed.h

```
//*****************************************************************/
// TITLE
//*****************************************************************/

// IndicatorLed.h
// Namespace for ReactorBot indicator LED.
// RBE-2001 A17 Team 7

#pragma once
#include "Led.h"

//*****************************************************************/
// NAMESPACE DEFINITION
//*****************************************************************/

namespace IndicatorLed {

	// RGB LED
	const uint8_t PIN_R = 26;
	const uint8_t PIN_G = 27;
	const uint8_t PIN_B = 28;
	Led rLed(PIN_R);
	Led gLed(PIN_G);
	Led bLed(PIN_B);

	// Initializes LED (call in setup)
	void setup() {
		rLed.init();
		gLed.init();
		bLed.init();
	}

	// Sets LED to indicate high radiation.
	void setHigh() {
		rLed.on();
		gLed.off();
		bLed.off();
	}

	// Sets LED to indicate low radiation.
	void setLow() {
		rLed.off();
		gLed.on();
		bLed.off();
	}
```

```
	// Sets LED tp indicate no radiation.
	void setNone() {
		rLed.off();
		gLed.off();
		bLed.on();
	}
}
```

PidController.h

```
//*************************************************************/
// TITLE
//*************************************************************/

//!t PidController.h
//!b Class for implementing discrete-time PID controllers.
//!a Dan Oates (WPI Class of 2020)

//!d This PID controller operates on the formula:
//!d
//!d u[n] = kp*e[n] + ki*sum(e[n]*dt[n]) + kd*(e[n]-e[n-1])/dt[n]
//!d
//!d where u[n] is the response, e[n] is the error, and dt[n] is
//!d the time difference between updates. The response u[n] is
//!d also constrained by minimum and maximum parameters.

//!et
#pragma once
#include "Timer.h"

//*************************************************************/
// CLASS DECLARATION
//*************************************************************/

class PidController {
      public:
            PidController(
                  double,
                  double,
                  double,
                  double,
                  double,
                  double = 3.4 * pow(10, 38));

            double update(double);
            void reset();
            bool isStabilized(double, double);

      private:
            Timer deltaTime;
            double kp, ki, kd; // gains
            double min = -3.4 * pow(10, 38); // min response
            double max = +3.4 * pow(10, 38); // max response
            double tRst = 3.4 * pow(10, 38); // reset time
            double e  = 0.0; // current error
            double eL = 0.0; // last error
            double eI = 0.0; // error integral
```

```
            double eD = 0.0; // error derivative
            bool initialized = false;
```

PidController.cpp

```
//*************************************************************/
// TITLE
//*************************************************************/

//!t PidController.cpp
//!a Dan Oates (WPI Class of 2020)
//!et
#include "PidController.h"

//*************************************************************/
// CONSTRUCTOR DEFINITIONS
//*************************************************************/

//!b Constructs PID controller with response limits.
//!i Proportional gain
//!i Integral gain
//!i Derivative gain
//!i Minimum response constraint
//!i Maximum response constraint
//!i Reset time (seconds between updates)
PidController::PidController(
      double kp,
      double ki,
      double kd,
      double min,
      double max,
      double tRst)
{
      this->kp = kp;
      this->ki = ki;
      this->kd = kd;
      this->min = min;
      this->max = max;
      this->tRst = tRst;
}

//*************************************************************/
// PUBLIC METHOD DEFINITIONS
//*************************************************************/

//!b Updates PID with error e[n] and returns u[n].
//!d On first call, integral and derivative computations are
//!d not made or included. Following, dt[n] is defined as the
//!d time difference between calls to this method.
//!d Additionally, if tRst seconds pass between calls to this
//!d method, an automatic reset occurs.
//!i error e[n] (see library details)
```

```
double PidController::update(double e) {
	double dt = deltaTime.toc();
	deltaTime.tic();
	if(dt >= tRst) reset();

	this->e = e;
	if(initialized) {
		eI += (e * dt);
		eD = (e - eL)/dt;
	}
	eL = e;

	double u = (kp * e) + (ki * eI) + (kd * eD);
	u = constrain(u, min, max);

	initialized = true;
	return u;
}

//!b Resets all PID memory values to 0.
void PidController::reset() {
	e = 0.0;
	eL = 0.0;
	eI = 0.0;
	eD = 0.0;
	initialized = false;
}

//!b Returns true if error is near 0 and stable.
//!d Controller is considered stabilized if the magnitude of the
//!d error and error derivative are below the given thresholds.
//!i Maximum passable error
//!i Maximum passable error derivative
bool PidController::isStabilized(double eMax, double dMax) {
	return
		initialized &&
		(abs(e) <= eMax) &&
		(abs(eD) <= dMax);
}
```

Qtr8.h

```
//*************************************************************/
// TITLE
//*************************************************************/

//!t Qtr8.h
//!b Class for operating QTR-8 line sensor array
//!a Dan Oates (WPI Class of 2020)

//!d While it contains some general light sensing functions,
//!d this class is primarily designed for line-tracking and
//!d line following purposes. Raw and average analog values can
//!d be read from each sensor, and thresholds can be set for
//!d on-line, off-line, and line intersection detection, as well
//!d as continuous displacement off of a line center relative to
//!d the sensor body.

//!et
#pragma once
#include "Arduino.h"

//*************************************************************/
// CLASS DECLARATION
//*************************************************************/

class Qtr8 {
public:
      Qtr8(const uint8_t*);
      void setup();

      int getRaw(int);
      int getAvg();

      void setBlackThreshold(int);
      void setWhiteThreshold(int);
      bool onBlack();
      bool onWhite();

      float linePos();
private:
      static int* pos;

      int blackThreshold = 700;
      int whiteThreshold = 100;
      uint8_t* pins;
};
```

Qtr8.cpp

```cpp
//*************************************************************/
// TITLE
//*************************************************************/

//!t Qtr8.cpp
//!a Dan Oates (WPI Class of 2020)
//!et
#include "Qtr8.h"

int* Qtr8::pos = new int[8]{7, 5, 3, 1, -1, -3, -5, -7};

//*************************************************************/
// PUBLIC METHOD DEFINITIONS
//*************************************************************/

//!b Constructs Qtr8 with with analog pins.
//!d pins[i] = analog pin of Qtr8 cell i+1 for i = 0:7
Qtr8::Qtr8(const uint8_t* pins) {
	this->pins = pins;
}

//!b Initializes analog pins as inputs (call before use).
void Qtr8::setup() {
	for(int i=0; i<8; i++)
		pinMode(pins[i], INPUT);
}

//!b Returns raw analog value from given cell.
//!i Cell to read from (valid 1-8)
int Qtr8::getRaw(int cell) {
	return analogRead(pins[cell-1]);
}

//!b Returns average of all raw analog sensor values
int Qtr8::getAvg() {
	int sum = 0;
	for(int i=1; i<=8; i++) sum += getRaw(i);
	return sum >> 3;
}

//!b Sets minimum analog threshold for all-black detection.
//!i Threshold value (valid 0-1023)
void Qtr8::setBlackThreshold(int t) {
	blackThreshold = t;
}
```

```
//!b Sets maximum analog threshold for all-white detection.
//!i Threshold value (valid 0-1023)
void Qtr8::setWhiteThreshold(int t) {
      whiteThreshold = t;
}

//!b Returns true if line sensor is on a horizontal black line.
bool Qtr8::onBlack() {
      return getAvg() > blackThreshold;
}

//!b Returns true if line sensor is not on a line at all.
bool Qtr8::onWhite() {
      return getAvg() < whiteThreshold;
}

//!b Returns center of line relative to sensor in cm.
//!d Assumes sensor is oriented perpendicular to the line and
//!d located somewhere on the line.
float Qtr8::linePos() {
      int pSum = 0;
      int aSum = 0;
      for(int i=0; i<7; i++) {
            int reading = analogRead(pins[i]);
            pSum += reading*pos[i];
            aSum += reading;
      }
      return (float)pSum / (float)aSum * 0.4828571;
}
```

DcMotor.h

```
//*************************************************************/
// TITLE
//*************************************************************/

//!t DcMotor.h
//!b Class for operating DC motors with quadrature encoders.
//!a Dan Oates (WPI Class of 2020)

//!d A digital pin is used to enable and disable the motor.
//!d Digital PWM pins are used to drive the motor forward and
//!d reverse. Optionally, encoder outputs can be attached to
//!d digital interrupt pins for full-resolution angle tracking.

//!et
#pragma once
#include "Arduino.h"

//*************************************************************/
// CLASS DECLARATION
//*************************************************************/

class DcMotor {
public:
      DcMotor(float,
                  uint8_t,
                  uint8_t,
                  uint8_t,
                  uint8_t,
                  uint8_t,
                  float);
      void setup();
      void enable();
      void disable();

      void setVoltage(float);
      void brake();

      void zeroAngle();
      float getAngle();

      int getInterruptA();
      int getInterruptB();
      void interruptA();
      void interruptB();
private:
      uint8_t pinEn, pinFw, pinRv;
```

```
      uint8_t pinEa, pinEb;

      long count = 0;
      float voltScale = 0;
      float tickScale = 0;
};
```

DcMotor.cpp

```
//*************************************************************/
// TITLE
//*************************************************************/

//!t DcMotor.cpp
//!a Dan Oates (WPI Class of 2020)
//!et
#include "DcMotor.h"

//*************************************************************/
// PUBLIC METHOD DEFINITIONS
//*************************************************************/

//!b Constructs DC motor with given Arduino pins.
//!i Terminal supply voltage (V)
//!i Motor enable pin (digital output)
//!i Motor forward pin (PWM out)
//!i Motor reverse pin (PWM out)
//!i Encoder pin A (interrupt)
//!i Encoder pin B (interrupt)
//!i Encoder counts per revolution
DcMotor::DcMotor(
      float voltage,
      uint8_t pinEn,
      uint8_t pinFw,
      uint8_t pinRv,
      uint8_t pinEa,
      uint8_t pinEb,
      float cpRev)
{
      this->pinEn = pinEn;
      this->pinFw = pinFw;
      this->pinRv = pinRv;
      this->pinEa = pinEa;
      this->pinEb = pinEb;

      voltScale = 255.0 / voltage;
      tickScale = TWO_PI / cpRev;
}

//!b Initializes digital IO and disables motor.
void DcMotor::setup() {
      pinMode(pinEn, OUTPUT);
      pinMode(pinFw, OUTPUT);
      pinMode(pinRv, OUTPUT);
      pinMode(pinEa, INPUT);
```

```
	pinMode(pinEb, INPUT);

	digitalWrite(pinEn, LOW);
	digitalWrite(pinFw, LOW);
	digitalWrite(pinRv, LOW);
}

//!b Enables motor motion.
void DcMotor::enable() {
	digitalWrite(pinEn, HIGH);
}

//!b Disables motor motion.
void DcMotor::disable() {
	digitalWrite(pinEn, LOW);
}

//!b Drives motor at given voltage using PWM.
//!d Voltage has valid range of +Vcc to -Vcc
void DcMotor::setVoltage(float voltage) {
	int analogValue = voltage * voltScale;
	if(analogValue > 0) {
		analogWrite(pinFw, analogValue);
		digitalWrite(pinRv, LOW);
	} else {
		analogWrite(pinRv, -analogValue);
		digitalWrite(pinFw, LOW);
	}
}

//!b Brakes motor
void DcMotor::brake() {
	setVoltage(0);
}

//!b Zeros the encoder angle.
void DcMotor::zeroAngle() {
	count = 0;
}

//!b Returns angle displacement of motor shaft in radians.
float DcMotor::getAngle() {
	return (float)count * tickScale;
}

//!b Returns interrupt ID associated with encoder A.
int DcMotor::getInterruptA() {
	return digitalPinToInterrupt(pinEa);
}
```

```
//!b Returns interrupt ID associated with encoder B.
int DcMotor::getInterruptB() {
      return digitalPinToInterrupt(pinEb);
}

//!b Executes interrupt subroutine for encoder A.
void DcMotor::interruptA() {
      if(digitalRead(pinEa)) {
            if(digitalRead(pinEb)) count++;
            else count--;
      } else {
            if(digitalRead(pinEb)) count--;
            else count++;
      }
}

//!b Executes interrupt subroutine for encoder B.
void DcMotor::interruptB() {
      if(digitalRead(pinEb)) {
            if(digitalRead(pinEa)) count--;
            else count++;
      } else {
            if(digitalRead(pinEa)) count++;
            else count--;
      }
}
```

ReactorComms.h

```
//*************************************************************/
// TITLE
//*************************************************************/

//!t ReactorComms.h
//!b Class for RBE-2001 final project Bluetooth communication.
//!a Dan Oates (WPI Class of 2020)

//!d This class utilizes the HC-05 Bluetooth serial bridge
//!d module through any hardware serial port available on
//!d any Arduino. The serial port is specified on object
//!d construction.
//!et

#pragma once
#include "Arduino.h"

//*************************************************************/
// CONSTANT DEFINITIONS
//*************************************************************/

const bool RADIATION_HI = true;
const bool RADIATION_LO = false;

//*************************************************************/
// CLASS DECLARATION
//*************************************************************/

class ReactorComms {
public:
      ReactorComms(HardwareSerial& serial);
      void init();

      void update();
      bool getRobotEnabled();
      bool storageAvailable(int);
      bool supplyAvailable(int);

      void sendHeartBeat();
      void sendRadAlert(bool);
private:
      HardwareSerial* serial;

      bool robotEnabled = false;
      byte storData = 0x00;
      byte fuelData = 0x00;
```

```
    byte checkSum = 0xFF;
    byte read();
    void write(byte);
};
```

ReactorComms.cpp

```
//*************************************************************/
// TITLE
//*************************************************************/

//!t ReactorComms.cpp
//!a Dan Oates (Team 7, Class of 2020)
//!et
#include "ReactorComms.h"

//*************************************************************/
// CONSTRUCTOR DEFINITIONS
//*************************************************************/

//!b Constructs ReactorComms through given hardware serial port.
ReactorComms::ReactorComms(HardwareSerial& serial) {
      this->serial = &serial;
}

//*************************************************************/
// PUBLIC METHOD DEFINITIONS
//*************************************************************/

//!b Initializes serial communication with the HC-05.
void ReactorComms::init() {
      serial->begin(115200);
}

//!b Processes new message packets from reactor control module.
//!d Ignores packets if:
//!d - They are not from the reactor control module
//!d - They are intended for another robot
//!d - The have incorrect checksums
void ReactorComms::update() {
      while(serial->available()) {
            if(serial->read() == 0x5F) { // Search for start

                  // Read full message
                  checkSum = 0xFF;
                  byte len = read() + 1;
                  byte msg[len];
                  msg[0] = 0x5F;
                  msg[1] = len;
                  for(int i=2; i<len; i++) msg[i] = read();

                  // Check read conditions
                  if(checkSum == 0x00 // Checksum passed
                    && msg[3] == 0x00 // Source is field
```

```
                  //&& msg[4] == 0x07 // Destination is Team 7
                ){
                      // Check message type
                      switch(msg[2]) {
                            case 0x01: // Storage tube availability
                                  storData = msg[5];
                                  break;
                            case 0x02: // Supply tube availability
                                  fuelData = msg[5];
                                  break;
                            case 0x04: // Stop movement
                                  robotEnabled = false;
                                  break;
                            case 0x05: // Resume movement
                                  robotEnabled = true;
                                  break;
                      }
                }
          }
      }
}

//!b Returns true if the robot is enabled by reactor control.
bool ReactorComms::getRobotEnabled() {
      return robotEnabled;
}

//!b Returns true if given storage tube is empty.
//!i Storage tube ID (valid 1-4).
bool ReactorComms::storageAvailable(int id) {
      switch(id) {
            case 1: return (storData & B00000001) != B00000001;
            case 2: return (storData & B00000010) != B00000010;
            case 3: return (storData & B00000100) != B00000100;
            case 4: return (storData & B00001000) != B00001000;
            default: return false;
      }
}

//!b Returns true if given supply tube is full.
//!i Supply tube ID (valid 1-4)
bool ReactorComms::supplyAvailable(int id) {
      switch(id) {
            case 1: return (fuelData & B00000001) == B00000001;
            case 2: return (fuelData & B00000010) == B00000010;
            case 3: return (fuelData & B00000100) == B00000100;
            case 4: return (fuelData & B00001000) == B00001000;
            default: return false;
      }
}
```

```
//!b Sends one heart-beat message to reactor control.
void ReactorComms::sendHeartBeat() {
      checkSum = 0xFF;
      write(0x5F); // Start delimeter
      write(0x05); // 5-byte message
      write(0x07); // Heart beat type
      write(0x07); // From robot 7
      write(0x00); // To reactor control
      write(checkSum);
}

//!b Sends radiation alert to reactor control.
//!i True for new rod, false for spent rod
void ReactorComms::sendRadAlert(bool high) {
      checkSum = 0xFF;
      write(0x5F); // Start delimeter
      write(0x06); // 6-byte message
      write(0x03); // Radiation alert type
      write(0x07); // From robot 7
      write(0x00); // To reactor control
      if(high)
            write(0xFF); // New fuel rod
      else
            write(0x2C); // Spent fuel rod
      write(checkSum);
}

//*************************************************************/
// PRIVATE METHOD DEFINITIONS
//*************************************************************/

//!d Reads 1 byte from the serial buffer and
//!d decrements it from the checksum.
byte ReactorComms::read() {
      while(!serial->available());
      byte b = serial->read();
      checkSum -= b;
      return b;
}

//!d Writes 1 byte to the serial buffer and
//!d decrements it from the checksum.
void ReactorComms::write(byte b) {
      serial->write(b);
      checkSum -= b;
}
```