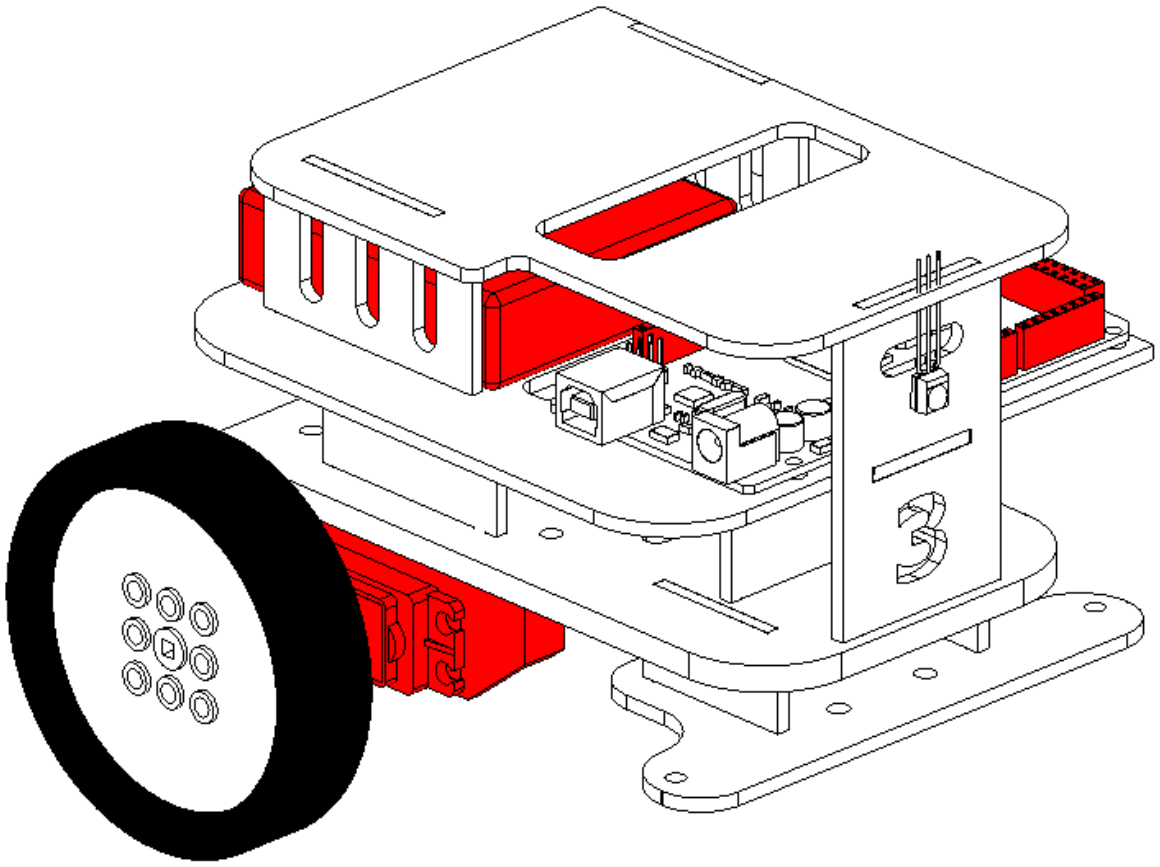
University of Delaware, Department of Mechanical Engineering

TRANSFORMABLE, ROBOTIC, INDEPENDENT DEVICE FOR EXTINGUISHING AND NAVIGATING TERRAIN

Mechatronics Team 3 Final Project Report

May 2024



Team Members

Andrew Smith, Noah Etienne, Dylan Taylor, and Venkata Sai Phanindra Mandadapu

Design of a Transformable Robotic Independent Device for Extinguishing and Navigating Terrain (TRIDENT) to navigate an unknown environment and detect and extinguish fires using on-board sensors, navigation control, and procedural pathfinding algorithms. The three wheeled, multilayered design is aimed at completing a course circuit the fastest while minimizing the vehicle envelope and weight. The vertical configuration of components allows for cheap, in-house laser cut manufacturing and modularity in subsystem design. Infrared sensors on the left, front, and right side allow for continuous, concurrent building and fire detection, while QTI sensors adjust vehicle positioning during travel. The custom software utilizes the A algorithm for optimal pathfinding capabilities around known obstacles, while another algorithm acts in conjunction to scan all faces of each building for fires. This comprehensive report covers the design and evaluation of the prototype prepared during the spring 2024 semester.*

Table of Contents

Introduction

Project Description

The objective of this project was to design and manufacture an autonomous "fire-fighting" vehicle that would operate in an unknown simulated scenario. "Fires" are detected based on an IR signal being emitted from up to one of the four faces of each building in an 8 x 8 grid environment. The vehicle must search the full 8 x 8 grid for buildings and put out any "fires" that may or may not exist by lowering a "ladder" arm onto the building. This action interrupts a break beam sensor on top of the building and turns off the IR signal, effectively putting out the fire. The final result of this project was a vehicle called the TRIDENT (Transformable Robotic Independent Device for Extinguishing and Navigating Terrain). Along the course of the semester, three major milestones were used to make sure the design of the vehicle was progressing at a sufficient rate and that each major subsystem was functioning properly (Table 1).

Table 1. Milestones in the project evaluation used. All descriptions are taken from the Final Project Overview page.

Milestone	Date	Description
1	3/22	*"Vehicles must be able to drive in a straight line and count the number of single or double line crossings."*
2	4/12	*"Vehicles must receive driving commands from a base station and demonstrate out-of-bounds indicator."*
3	5/3	*"Vehicles must detect the IR signal and successfully lower and then raise their ladder."*

Constraints

The following constraints are used to confirm the validity of the design as determined by the project description (Table 2). Several of the design iterations mentioned in later sections refer to these constraints as they form the backbone of the physical prototype besides basic autonomous functionality desired. An evaluation of the final design, vehicle dimensions, materials used, and budget are discussed later in this report as well. Apart from satisfying these constraints of the project, the ultimate goal was to achieve full autonomy, meaning the performance of the aforementioned tasks with minimal user intervention–this performance will be evaluated in reference to the complete strengths and weaknesses of the design.

Table 2. Project constraints including all target metrics and achieved values.

Constraint	Metric	Achieved	Pass/Fail
Project Milestones	Pass all project milestones by discretion of professor.	Milestone 1: 100% Milestone 2: 100% Milestone 3: 100%	Pass
Battery Supply	No more than four AA batteries and one 9V battery used.	Four AA batteries One 9V battery	Pass
Microcontroller	Stock circuit components used without modifications.	Pass	Pass
	Protect microcontroller PCB from environmental hazards.	Partial covering of three faces	
Weight	Maximum mass of 900g.	434.5g	Pass
Dimensions	Maximum envelope size of 20cm x 20cm at start of trial.	Width: 19.0cm Length: 18.6cm Height: 33.0cm	Pass
	No unretrievable projectiles used in design.	Pass	
Materials	Only approved materials used in the vehicle design.	Circuitry: stock Framing: acrylic Adhesive: hot glue	Pass
Budget	Maximum cost of purchased parts of $300.	Total: $9.99	Pass

Design Overview

Key Features

The final design of the TRIDENT vehicle is unique and innovative. The vehicle has two different modes of function dictated by two different codes. One mode allows the vehicle to be remote controlled by the user via a small radio transmitter with two joysticks to control left and right wheel speeds independently. In this mode, when the vehicle senses the IR signal indicating “fire” in front of it, it engages the ladder automatically for assisted functionality. The other mode is designed for complete automation–it is designed to allow the vehicle to traverse the 8 x 8 grid and scan every block on its own without user input. Again, in this mode the code is designed to engage the ladder automatically if the vehicle senses a “fire” in front of it.

The final design of the vehicle has at least three distinct characteristics that comprise its functionality and innovation. The first major characteristic of the vehicle design the body is a multi-layered, modular design which allows for easier part replacement or conversion. Each layer of the vehicle structure holds specific, distinguished aspects of the design that can easily be removed and

adjusted without taking apart disrupting the design as a whole. It is this capability that makes the device essentially transformable, as parts can be replaced for different functionalities. The code that runs the vehicle operations is also modular in that most of the program is written in terms of separate functions, which allows for ease of editing and versatility. Refer to Appendix A for the full drawing packet of the final assembly.

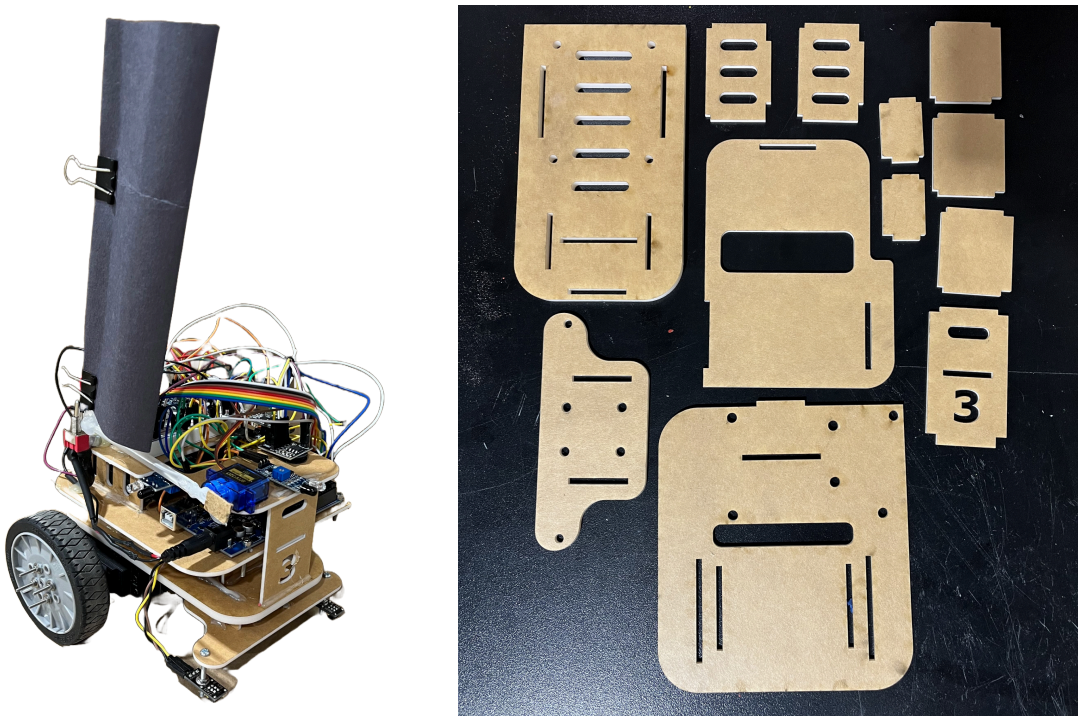


Figure 1: (Left) Photo of the final, assembled TRIDENT design, illustrating the layers. (Right) Individual laser cut elements used in each layer and the vertical construction.

The second major and innovative characteristic of the design is the three-directional sensing capability. The design has IR sensors glued to the top layer of the design facing forward and to either side of the vehicle. This innovative design choice allows the vehicle to sense in all relevant directions during movement, effectively shortening the necessary travel path of the vehicle by passively scanning more area (three blocks) in the grid by default compared to a single front sensor design that can only sense one block at a time. This should boost efficiency in the vehicle movement path, therefore shortening overall navigation time.

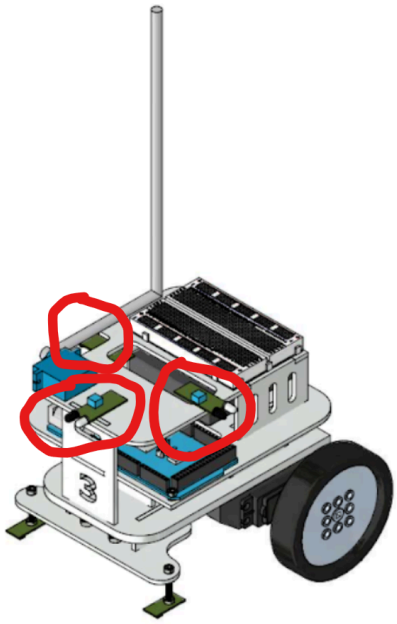


Figure 2: CAD model of the TRIDENT featuring the three IR sensors on the vehicle (circled in red).

The third and final major innovation of this design is arguably the most advanced and technical innovation of the design, and rather than manifesting physically, it appears in the code that runs the automation of the vehicle. The automation version of the code tells the vehicle to search through a predetermined set of blocks in the grid. As the vehicle senses buildings, it adds new coordinates to this search in order to sweep the perimeters of each new-found building. The vehicle knows where to go based on a custom implementation of the A* path finding algorithm. This function automates the pathfinding of the TRIDENT by outputting a series of directional commands based on the start and end coordinates of where the vehicle is and where it intends to go, and it generates the shortest and most efficient path to take while avoiding any known, previously searched buildings. The design innovates by making heavy use of this function, using it to dictate every movement throughout its journey.

Some other smaller elements of design innovations in the TRIDENT vehicle include its three wheel design and ladder assembly. Rather than the standard four wheeled convention, at the sacrifice of a little stability, the three wheeled design reduces the weight, power consumption, and 2D footprint by implementing one turning wheel in the front rather than two. The loss in stability is very minimal, and was corrected by the implementation of an automated course correction code that accurately determines directional movement using a PID-controlled gyroscope. The ladder needed a new design because the initial design was simply a servo motor that would lower a straw a designated number of degrees. The issue was that if the distance from the building or the number of degrees the ladder should move was off, the design would encounter one of two problems: (1) the ladder would not come down far enough to touch the building and trip the break beam sensors or (2) the ladder would come down too far, either straining the servo motor or moving the building. In order to address this issue, a ladder with built-in tolerance was constructed. The innovation was as simple as a rolled up sheet of paper attached to the edge of the ladder as shown in Figure 3. This paper would contact the building and flex as it caved until the ladder reached its designated position. The straw arm of the ladder was programmed to move a number of degrees that would bring it relatively close to the building but not actually touch it. The rolled up paper

edge would contact the building, tripping the sensors without exerting too much force on the structure itself or straining the servo motor. See Appendix B for more information on the design process and previous iterations leading to these innovations.

Figure 3: Photo of the TRIDENT ladder assembly, featuring a flexible, black paper arm clipped onto a plastic straw.

Operating Mechanics

The TRIDENT operates using a few concurrently operating algorithms which allow for the entire grid to be searched for buildings, each side of each building to be checked for potential fires, and a stored set of data to prevent redundancy in building searches. These algorithms make use of the general modularity of the code and therefore each call several functions for vehicle movements, sensor readings, parsing instructions, updating the 8 x 8 grid, etc.

The principle movement mechanic behind the design is a custom implementation of the A* search algorithm which tells the vehicle exactly which movement instructions to take based on a given starting and end location (Figure 4). This means that given a known grid of building and out-of-bounds locations–data which is continuously updated throughout the course of the program using the three onboard IR sensors–the vehicle can determine the shortest path to navigate around these obstacles (i.e., formerly searched buildings) and reach the next desired point in the search path. A* operates using a Manhattan heuristic model (*h* value) which allows the algorithm to apply an educated guess of the best move towards the target location [1]. At each iteration, the algorithm chooses a cell with the lowest total cost (*f* parameter) which itself is a combination of the heuristic/predicted cost and the actual movement cost (*g* value). When the target is reached, and all cells have been explored to locally optimize the *f*

scores, the path can then be backtracked and fed into the vehicle as a set of movement instructions (i.e., turning directions and calls to move forward one tile).

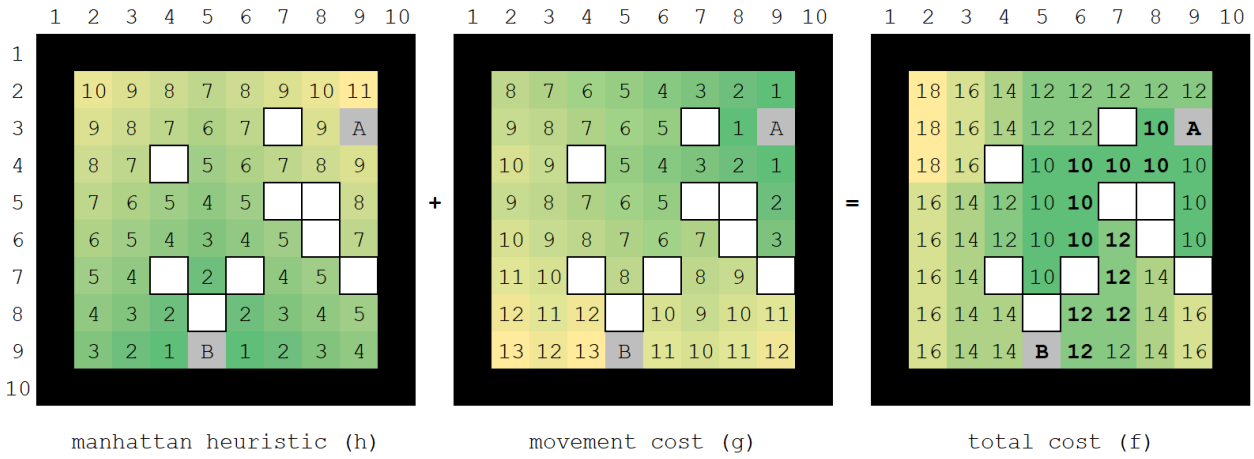


Figure 4. A* algorithm applied to a sample grid traversing from point A to point B with buildings marked as white boxes and out-of-bounds locations as black boxes. The final path is bolded on the right.

As the environment is simplified into a grid pattern, the movement mechanics are designed such that the TRIDENT performs discrete steps to adjacent tiles using “move forward” and “turn angle” commands as discussed in Appendix C. The operation allows for an initial speed calibration to determine the starting angle of the vehicle as well as the distance for tile-to-tile movements in the beginning of the trial. For a hardware and electronics overview of the design that allows these mechanics to work, see Appendix D as well. Note that in the current implementation, the QTI sensors are only needed for initial vehicle speed calibration and after that, all fire detection, building proximity detection, steering control, etc. is handled on-board using the gyroscope of the accelerometer and the IR sensors alone.

A complete overview of the system operation can be summarized in the flowchart below (Figure 5). The setup function initializes all variables, creates the predefined pinned connections, connects all motors and sensors, and calibrates the gyroscope based on the current direction of the vehicle. After the initial vehicle calibration (see Appendix C), two different processes are performed: (1) the vehicle travels a predefined search path, specified by a set of grid points (using A*), and (2) if a building is detected, the vehicle checks all four grid points around the building for a fire (also using A*). Both these processes are easily modified for better search patterns, for example if the side IR sensors are utilized, the search path entered can be set to skip every other row, and if multiple buildings are detected while the TRIDENT is searching the neighbors around a building, these points can also be added to the neighbor search path. Refer to Appendix C for the search path used in the final design.

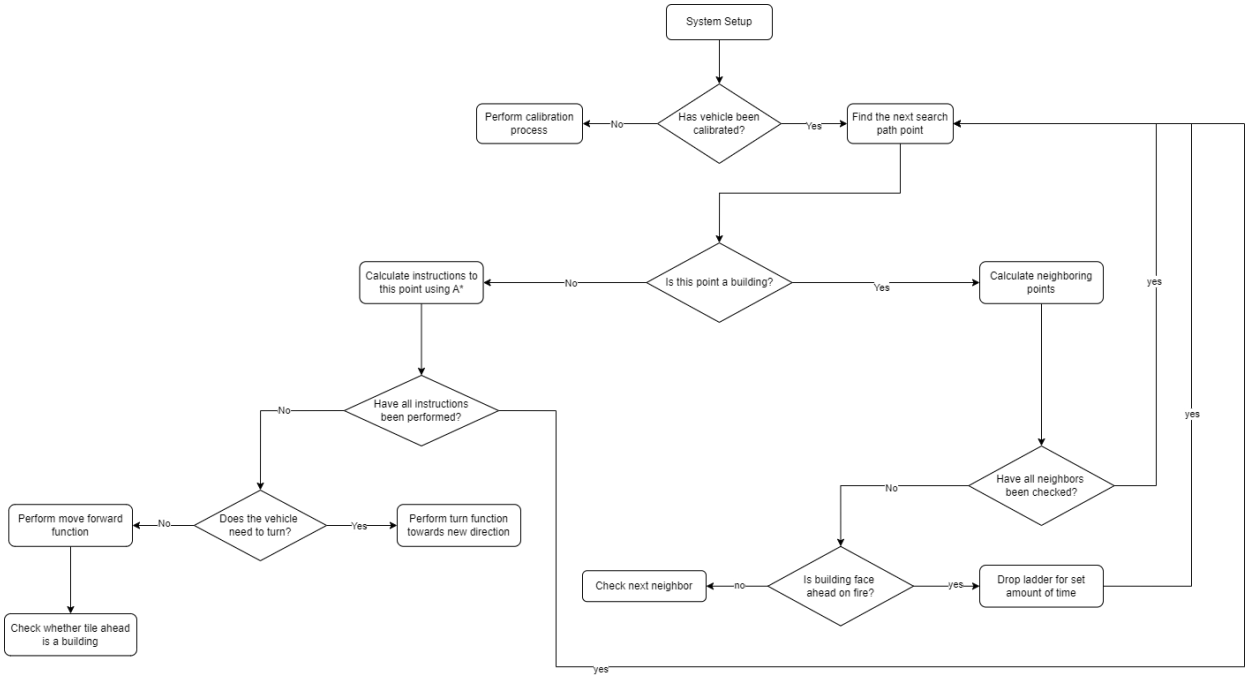


Figure 5. System-level flowchart of the main program functionality. This process repeats throughout the trial and does not include minor calls to functions used to implement these mechanics.

Design Details

The final design of the trident is shown in Figure 6 below. Here, the key innovations of the physical design can be observed with the three wheels, layered structure, and side facing IR sensors to act both as fire (IR) detection sensors and proximity sensors for close building engagement.

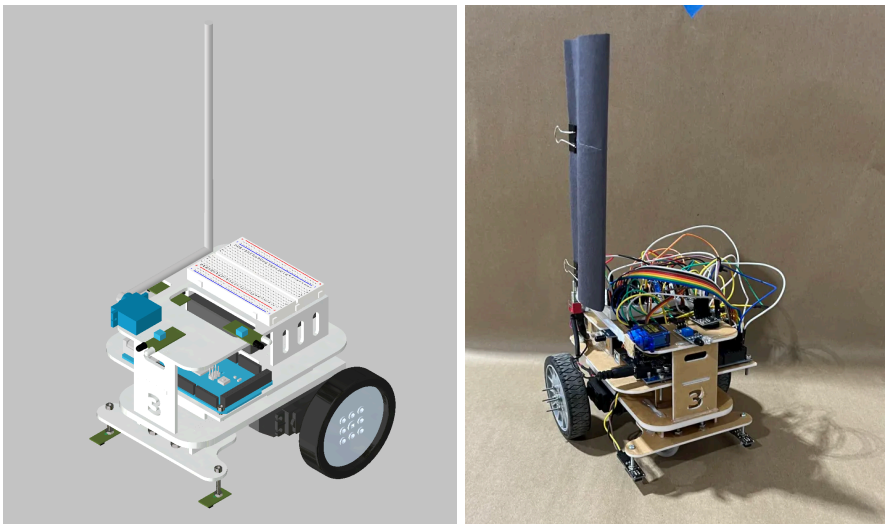


Figure 6. Final design for the TRIDENT. (Left) 3D assembly model generated using Autodesk Inventor. (Right) Fully assembled physical model.

In addition to the innovations behind the custom written software of the vehicle, being able to mount, wire, track, process, and display a wide range of data continuously and compactly was vital to the

success of the design. The design uses extensive hardware for these capabilities and includes three IR sensors, two QTI sensors, and an accelerometer with an attached gyroscope in addition to the battery power, motors, display controls, remote receiving, and various other minor electronic components. The complex nature of this electrical system requires detailed schematics like that shown in the Figure below. Refer to Appendix D for more information on the electrical design.

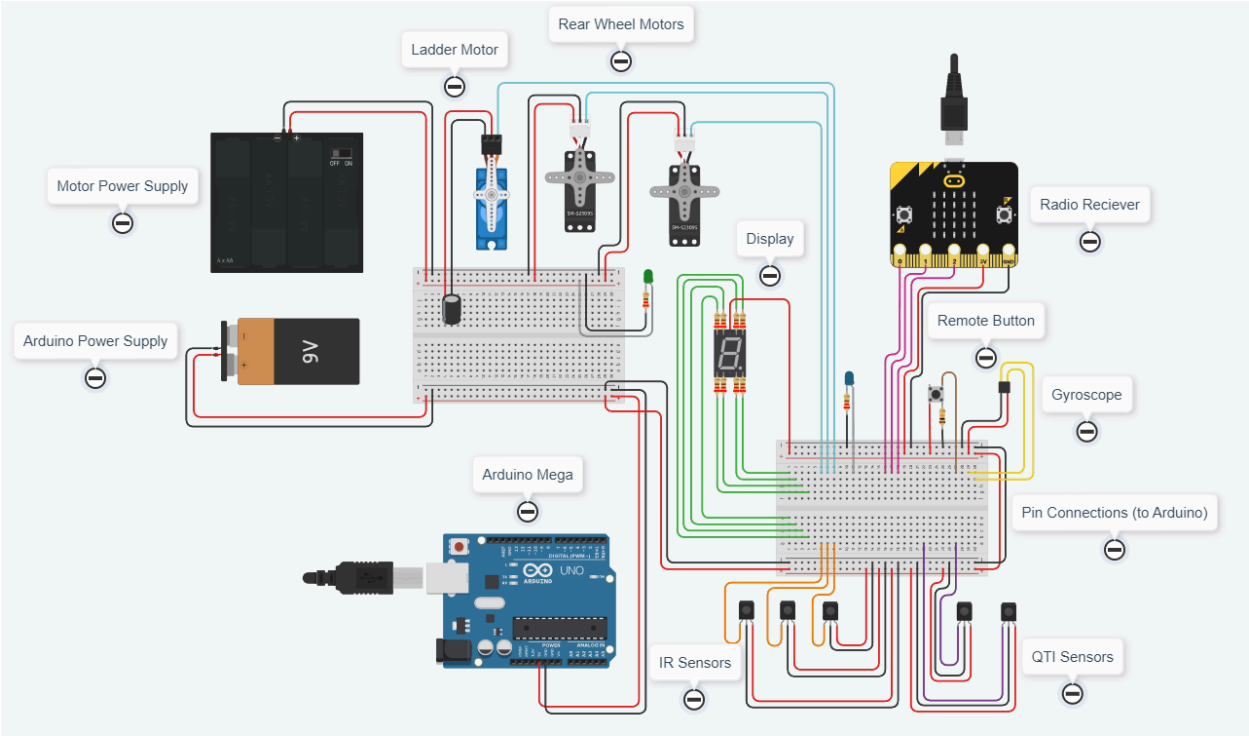


Figure 7. Circuit schematics of the TRIDENT electrical system. All pin connections are shown on the breadboard right of the microcontroller in this diagram.

The design was also made fairly inexpensively as of the $300 budget allocated for external ordering, just $3.33 of wholesale cost was used with a total product costing $187.03 (Figure 8). Note that this final cost includes the cost of the remote controller but does not include any previous iterations. With these iterations included however, the cost has minimal changes as most electrical components were used in each stage of the project, such as the microcontrollers, sensors, minor electronics, motors, etc. which take up the majority of the total cost. Refer to Appendix E for a further breakdown of the bill of materials for this project.

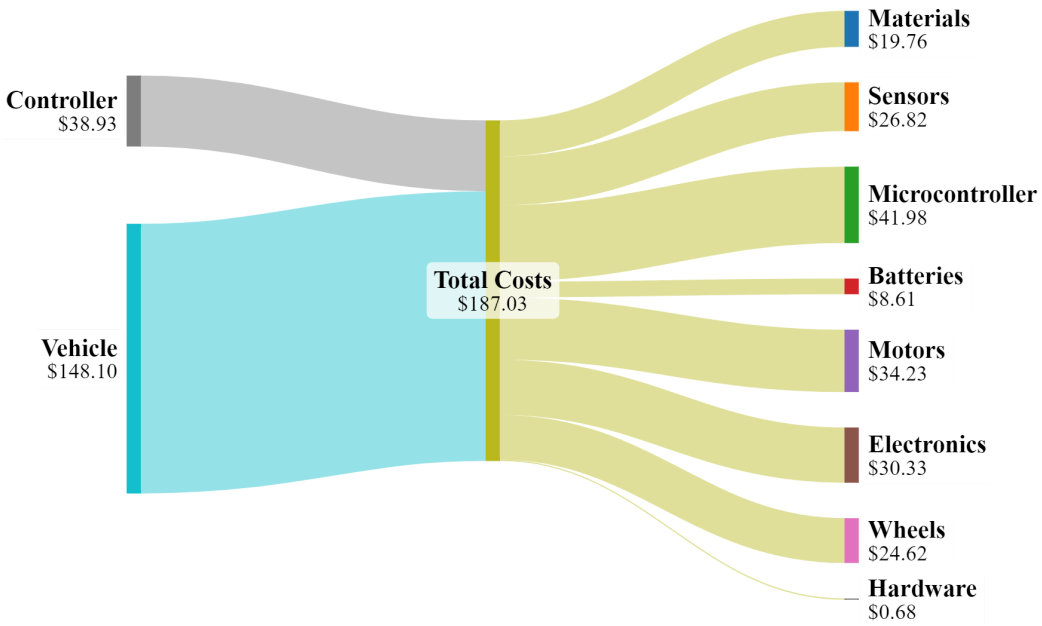


Figure 8. Sankey chart of the total cost breakdown of the final design, including costs per category and costs per major design component.

Evaluation

Strength and Weaknesses

The TRIDENT has many strengths and weaknesses inherent to the design. Among its strengths, its navigation tactics stand out the most–the use of the A* algorithm for autonomous movement enforced that the vehicle take the most efficient and shortest path to travel the dictated path of travel while also sweeping any buildings encountered. Refer to Figure 5 and Appendix F for the full complexity of this implementation. The TRIDENT's physical design also excelled physically, as it was compact, lightweight , and fits well within the size and weight constraints set for the project (see Table 2). This success was owed in part to the modular, layered design as the stacked components reduced the overall footprint and allowed easy modifications for future iterations. The software was also designed to be modular for similar reasons and divided the autonomous processes into separate functions that could be tested and modified without disrupting the code as a whole. Another strength of the vehicle's design lies in its ladder–the ladder was flexible, which allowed it to hit the buildings without straining the servo motor or moving the buildings, and yet rigid, still breaking the beam of the sensors

The vehicle's design was even further optimized by implementing simple user controls, such as having a battery pack controlled by a switch and a separate switch to power on the Arduino for a simple "boot-up" process. This user friendliness also extended to the 7-segment panel that displayed numbers according to what step of the process the vehicle was in and for other debugging purposes such as showing the letter "F" when the vehicle detected a fire. One of the vehicle's biggest strengths was its electronic control components for automatic movement. The accuracy of the vehicle's path was enforced by a gyroscope that used PID control to keep the vehicle on a path that was less than 0.2 degrees off of the desired direction. This kept the vehicle on a relatively straight path and allowed for pinpoint accuracy in turning a specified number of degrees.

While the TRIDENT had many strengths, it also had many weaknesses. To start, it is visibly very messy with its wiring which can allow wires to come loose during movement and create faulty readings from the sensors. While this was minimized by tying wires down, future iterations necessitate perfboard management. The compactness of the design is turned negative when it becomes harder to troubleshoot and reach components that need maintenance due to a lack of space and organization of wires. Another weakness of the design lies in its battery drainage–the wheel motors drain the four x AA batteries incredibly fast when being used, and the Arduino barely drains its 9V battery any slower. There are a lot of components to be powered by these batteries, and in the future it may be worth using the budget to purchase rechargeable batteries. Another weakness of the TRIDENT's design is in its wheel design. Three wheel design was used to minimize weight and reduce errors that could have stemmed from misaligned axles, however, in pursuing this design, the directional accuracy was also diminished because the front wheel was on a 360 degree, unstable swivel and therefore had no specific direction to steer the vehicle in. This wheel placement also made turning in place difficult. Had the powered wheel been placed at the midpoint of the vehicle length-wise, the vehicle could spin in place by having the motors run at equal speeds in opposite directions of one another (see Appendix C). The TRIDENT also significantly suffers due to its IR sensor issues. The sensors lack the proper sensitivity to reliably sense building walls at further ranges, making the three sensor design fail during movement. The sensors must either be adjusted or replaced to allow for better building detection. The QTI sensors also suffer in some ways, as they are non-adaptive. The readings obtained from the QTI sensors are often sporadic and unreliable. When putting the vehicle on a new surface other than the designated 8 x 8 grid, the sensors fail to give useful information even when their threshold values are adjusted. The final weakness worth mentioning in the TRIDENT's design is in its code. While the modularity of the functions helps, it is still a lengthy code of over 1500 lines (see Appendix F), making it difficult to troubleshoot and debug. Future iterations may break the code into smaller files or just shorten it altogether in some way.

Open Single Constraint Argument

The constraints of this project are well-defined and therefore highly define the design of the vehicle. Conformity to these rules and constraints is done to ensure the vehicle functionality and repeatability. The unfortunate byproduct of this conformity is a lack of innovation and creativity–a modified rule that could allow for more innovation is the relaxation of the fire detection rule. If, instead of having to detect that a building is on fire, the vehicle is allowed to engage the ladder on every building it discovers, several new avenues would open up for the ladder mechanism. For example, if all the vehicle had to do was sense a building, the ladder mechanism could have been a cross or a more complex mechanism that could trigger the sensors on top of the buildings. Another rule modification that may yield more creative, successful, and innovative projects is a change to the size constraints of the vehicle. If the vehicle could have been wider, it could have been easier to keep the vehicle on a straight path by using QTI sensors to enforce the boundaries of the vehicle's motion similar to an object bouncing between two walls.

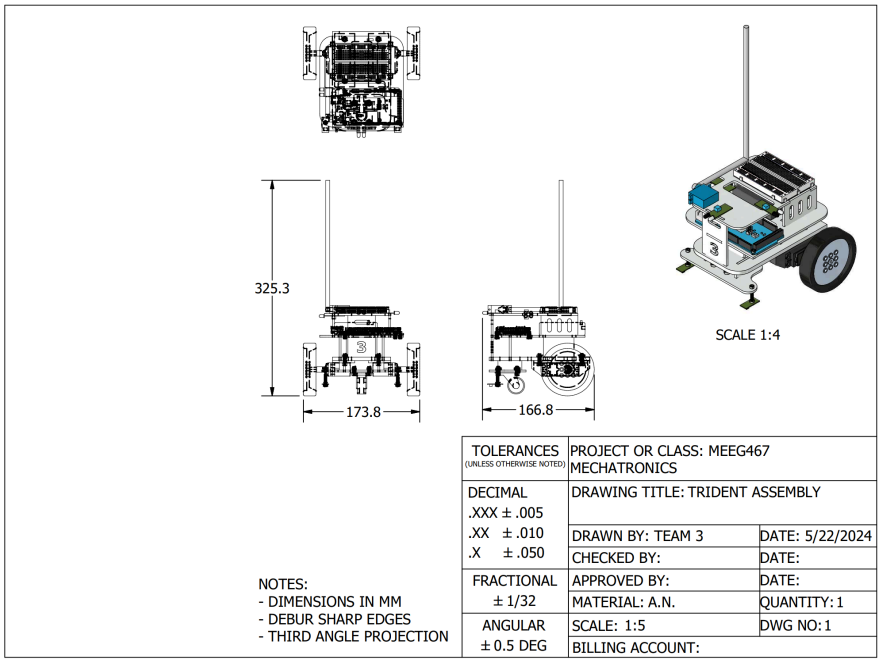
References

[1] "A* search algorithm," GeeksforGeeks, https://www.geeksforgeeks.org/a-search-algorithm/ (accessed May 22, 2024).

Appendix

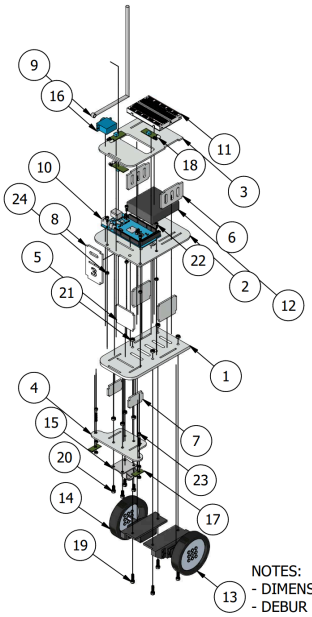
Appendix A: Drawing Packet

This appendix contains the complete drawing packet used for manufacturing the final TRIDENT iteration as well as details used in part modifications. In the figures below, assembly, exploded assembly, and laser cut 2D drawings are used to show the construction of the design and the parts composing each layer.



TOLERANCES (UNLESS OTHERWISE NOTED)	PROJECT OR CLASS: MEEG467 MECHATRONICS	
DECIMAL .XXX ± .005 .XX ± .010 .X ± .050	DRAWING TITLE: TRIDENT ASSEMBLY	
	DRAWN BY: TEAM 3	DATE: 5/22/2024
	CHECKED BY:	DATE:
FRACTIONAL ± 1/32	APPROVED BY:	DATE:
	MATERIAL: A.N.	QUANTITY: 1
ANGULAR ± 0.5 DEG	SCALE: 1:5	DWG NO: 1
	BILLING ACCOUNT:	

NOTES:
- DIMENSIONS IN MM
- DEBUR SHARP EDGES
- THIRD ANGLE PROJECTION



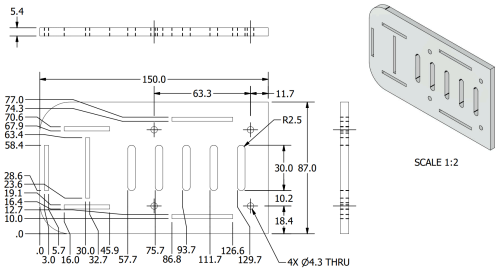
PARTS LIST

ITEM NO.	PART NAME	DESCRIPTION	QTY
1	BASE PLATFORM	ACRYLIC	1
2	INTERMEDIATE PLATFORM	ACRYLIC	1
3	TOP PLATFORM	ACRYLIC	1
4	SENSOR PLATFORM	ACRYLIC	1
5	PRIMARY SPACER	ACRYLIC	3
6	SECONDARY SPACER	ACRYLIC	2
7	SENSOR SPACER	ACRYLIC	2
8	FRONT PLATE	ACRYLIC	1
9	LADDER	POLYPROPYLENE	1
10	MICROCONTROLLER BOARD	ELEGOO MEGA R3 2560	1
11	BREADBOARD	MCMASTER-CARR PART NO. 1388N13	1
12	BATTERY PACK	DAIERTEK AA BATTERY BHOLDER	1
13	LEFT WHEEL ASSEMBLY	ABS	1
14	RIGHT WHEEL ASSEMBLY	ABS	1
15	CASTER	STANDARD CASTER	1
16	MICRO SERVO	SG90 MICRO SERVO	1
17	QTI Sensor	PARALLAX SKU 555-27401	2
18	IR SENSOR	IR INFRARED SENSOR	3
19	91292A038	MCMASTER-CARR PART NO. 91292A038	4
20	91292A116	MCMASTER-CARR PART NO. 91292A116	4
21	91828A231	MCMASTER-CARR PART NO. 91828A231	8
22	91292A112	MCMASTER-CARR PART NO. 91292A112	4
23	91292A020	MCMASTER-CARR PART NO. 91292A020	2
24	91828A211	MCMASTER-CARR PART NO. 91828A211	8

TOLERANCES (UNLESS OTHERWISE NOTED)	PROJECT OR CLASS: MEEG467 MECHATRONICS	
DECIMAL .XXX ± .005 .XX ± .010 .X ± .050	DRAWING TITLE: TRIDENT EXPLODED ASSEMBLY	
	DRAWN BY: TEAM 3	DATE: 5/22/2024
	CHECKED BY:	DATE:
FRACTIONAL ± 1/32	APPROVED BY:	DATE:
	MATERIAL: A.N.	QUANTITY: 1
ANGULAR ± 0.5 DEG	SCALE: 1:6	DWG NO: 2
	BILLING ACCOUNT:	

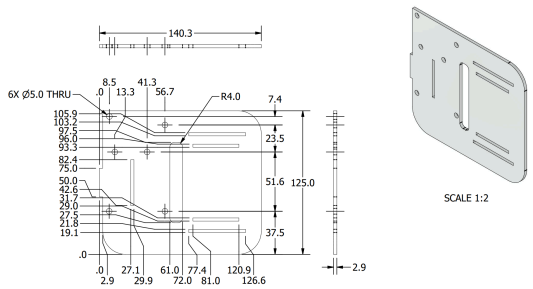
NOTES:
- DIMENSIONS IN MM
- DEBUR SHARP EDGES
- THIRD ANGLE PROJECTION

Figure A1. Assembly drawings for final TRIDENT design. (Top) Full assembly drawing. (Bottom) Exploded assembly drawing with parts list. Drawings were generated using Autodesk Inventor.



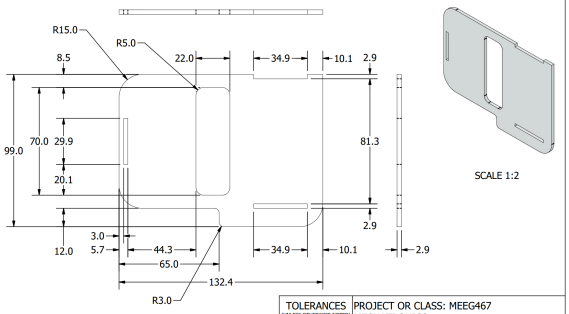
TOLERANCES (UNLESS OTHERWISE NOTED)	PROJECT OR CLASS: MEEG467 MECHATRONICS	
DECIMAL .XXX ± .005 .XX ± .010 .X ± .050	DRAWING TITLE: BASE PLATFORM	
	DRAWN BY: TEAM 3	DATE: 5/22/2024
	CHECKED BY:	DATE:
FRACTIONAL ± 1/32	APPROVED BY:	DATE:
	MATERIAL: ACRYLIC	QUANTITY: 1
ANGULAR ± 0.5 DEG	SCALE: 2:3	DWG NO: 3
	BILLING ACCOUNT:	

NOTES:
- DIMENSIONS IN MM
- DEBUR SHARP EDGES
- THIRD ANGLE PROJECTION



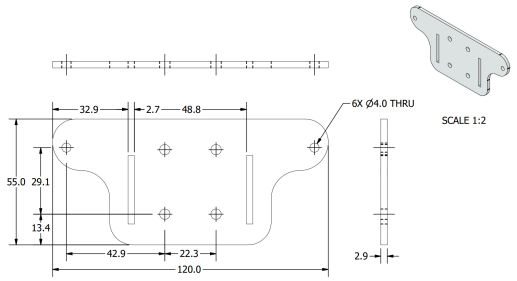
TOLERANCES (UNLESS OTHERWISE NOTED)	PROJECT OR CLASS: MEEG467 MECHATRONICS	
DECIMAL .XXX ± .005 .XX ± .010 .X ± .050	DRAWING TITLE: INTERMEDIATE PLATFORM	
	DRAWN BY: TEAM 3	DATE: 5/22/2024
	CHECKED BY:	DATE:
FRACTIONAL ± 1/32	APPROVED BY:	DATE:
	MATERIAL: ACRYLIC	QUANTITY: 1
ANGULAR ± 0.5 DEG	SCALE: 1:2	DWG NO: 4
	BILLING ACCOUNT:	

NOTES:
- DIMENSIONS IN MM
- DEBUR SHARP EDGES
- THIRD ANGLE PROJECTION



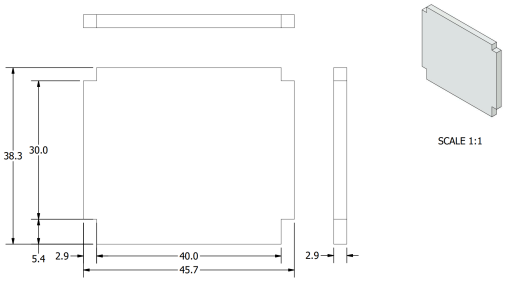
TOLERANCES (UNLESS OTHERWISE NOTED)	PROJECT OR CLASS: MEEG467 MECHATRONICS	
DECIMAL .XXX ± .005 .XX ± .010 .X ± .050	DRAWING TITLE: TOP PLATFORM	
	DRAWN BY: TEAM 3	DATE: 5/22/2024
	CHECKED BY:	DATE:
FRACTIONAL ± 1/32	APPROVED BY:	DATE:
	MATERIAL: A.N.	QUANTITY: 1
ANGULAR ± 0.5 DEG	SCALE: 2:3	DWG NO: 5
	BILLING ACCOUNT:	

NOTES:
- DIMENSIONS IN MM
- DEBUR SHARP EDGES
- THIRD ANGLE PROJECTION



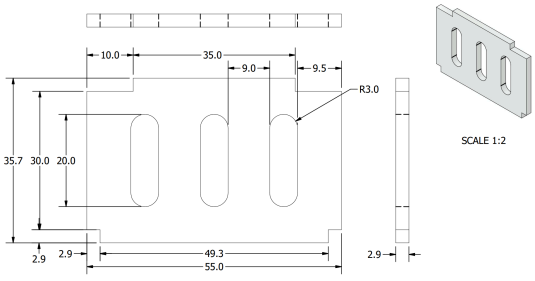
TOLERANCES (UNLESS OTHERWISE NOTED)	PROJECT OR CLASS: MEEG467 MECHATRONICS	
DECIMAL .XXX ± .005 .XX ± .010 .X ± .050	DRAWING TITLE: SENSOR PLATFORM	
	DRAWN BY: TEAM 3	DATE: 5/22/2024
	CHECKED BY:	DATE:
FRACTIONAL ± 1/32	APPROVED BY:	DATE:
	MATERIAL: A.N.	QUANTITY: 1
ANGULAR ± 0.5 DEG	SCALE: 1:1	DWG NO: 6
	BILLING ACCOUNT:	

NOTES:
- DIMENSIONS IN MM
- DEBUR SHARP EDGES
- THIRD ANGLE PROJECTION



TOLERANCES (UNLESS OTHERWISE NOTED)	PROJECT OR CLASS: MEEG467 MECHATRONICS	
DECIMAL .XXX ± .005 .XX ± .010 .X ± .050	DRAWING TITLE: PRIMARY SPACER	
	DRAWN BY: TEAM 3	DATE: 5/22/2024
	CHECKED BY:	DATE:
FRACTIONAL ± 1/32	APPROVED BY:	DATE:
	MATERIAL: ACRYLIC	QUANTITY: 3
ANGULAR ± 0.5 DEG	SCALE: 2:1	DWG NO: 7
	BILLING ACCOUNT:	

NOTES:
- DIMENSIONS IN MM
- DEBUR SHARP EDGES
- THIRD ANGLE PROJECTION



TOLERANCES (UNLESS OTHERWISE NOTED)	PROJECT OR CLASS: MEEG467 MECHATRONICS	
DECIMAL .XXX ± .005 .XX ± .010 .X ± .050	DRAWING TITLE: SECONDARY SPACER	
	DRAWN BY: TEAM 3	DATE: 5/22/2024
	CHECKED BY:	DATE:
FRACTIONAL ± 1/32	APPROVED BY:	DATE:
	MATERIAL: ACRYLIC	QUANTITY: 2
ANGULAR ± 0.5 DEG	SCALE: 2:1	DWG NO: 8
	BILLING ACCOUNT:	

NOTES:
- DIMENSIONS IN MM
- DEBUR SHARP EDGES
- THIRD ANGLE PROJECTION

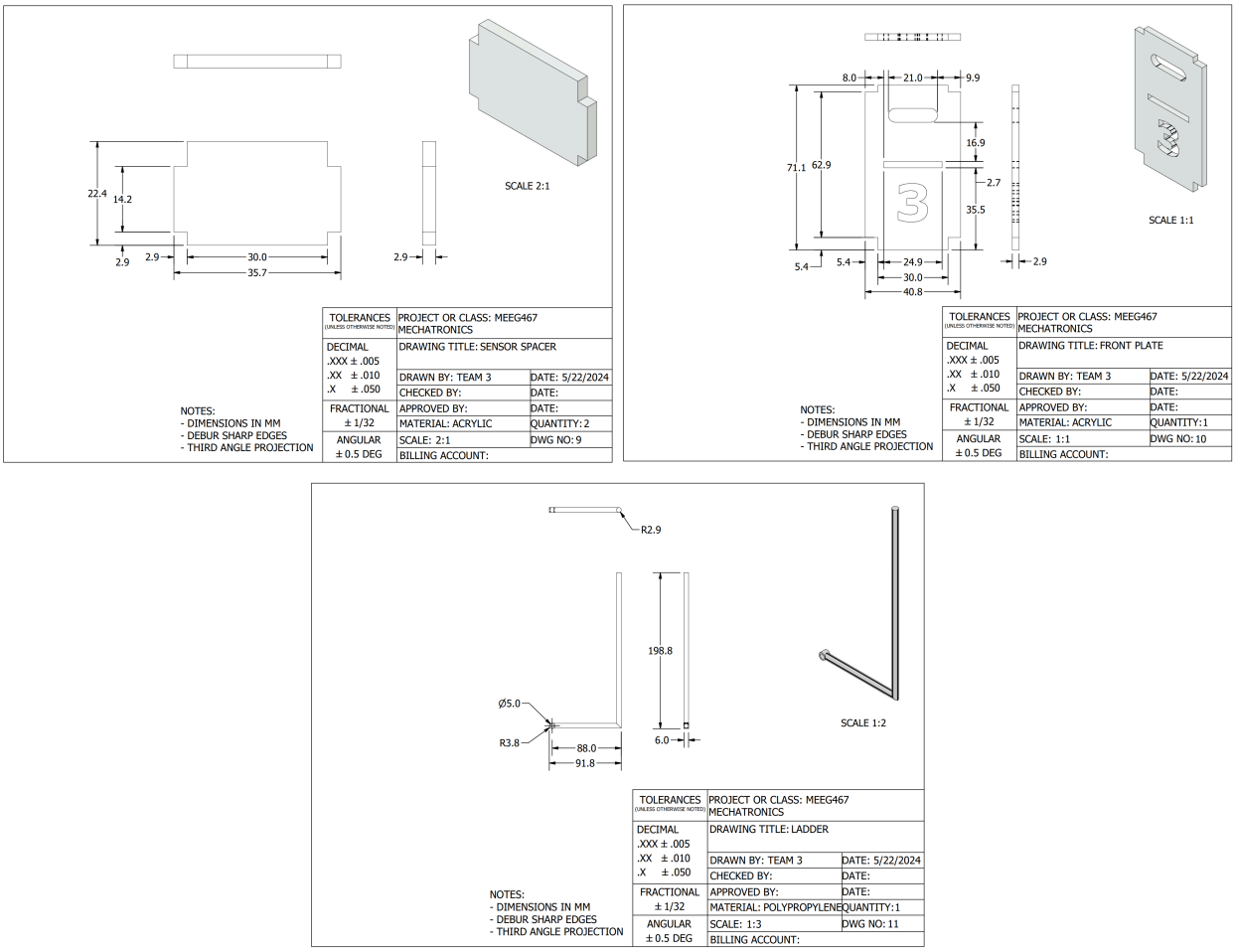


Figure A2. Individual part drawings for items 1-9 of the parts list seen in Figure A.1. Each part was designed and manufactured by the team. Standard hardware and purchased parts are not shown. Drawings were generated using Autodesk Inventor.

Appendix B: Design Iterations

This appendix contains an overview of the design process from initial prototypes early in the semester (March 2024) to the final physical prototype used in the showcase (May 2024).

The design of the vehicle was gradually modified throughout the semester given the changing performance requirements of each milestone and the limitations of the current version discovered in the process (see Table 1). The initial design featured a rear-wheel drive with two axially connected wheels in the front, a wooden base for easy prototyping, a spacer block for two front-facing QTI sensors, and a fairly compact electrical system on top of the vehicle (Figure B1). This early prototype provided a solid foundation for later development as the stacked battery pack and breadboard layout allowed the weight distribution to be closer to the rear axle while preserving room for wire management. A layered approach was realized in this design where having a separate base plate for mounting all drive components (i.e., motor and wheel assemblies), a second tier for electrical systems, and a bottom assembly for close-up QTI detection of the lines was very beneficial in rapid development. However, the increased size, weight, and loss of turning control of the vehicle due to the front axle favored a three-wheeled design, and the need for easy maintenance and troubleshooting favored a more vertically centered design with space for circuit and hardware components.

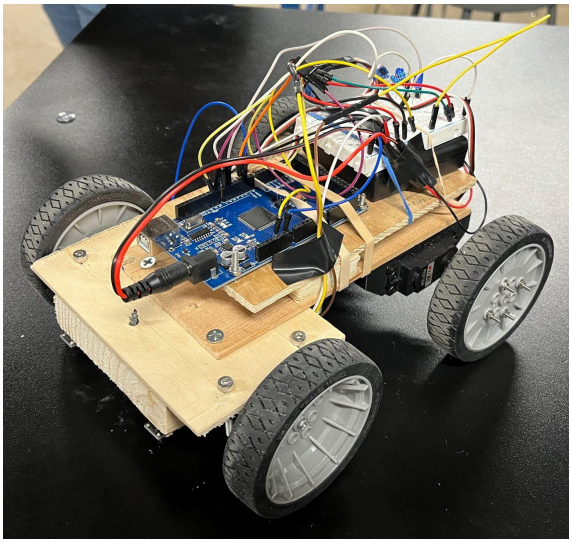


Figure B1. Initial design prototype used for straight driving functionality and basic line detection.

The major hardware components remained the same through Milestone 2, with the addition of a remote control receiver component on the uppermost layer of the vehicle and the front axle being replaced with a single swivel wheel (Figure B2). Manufacturing was favored for cheap, fast, in-house processes like laser cutting, so the design was created using ¼” acrylic layers with the modular approach started in Milestone 1 (see Figure B1). Ample room was given for the electrical components as the addition of many indicator LEDs, motor drivers, remote control receiver, and QTI sensors necessitated easy troubleshooting and repair. The three-wheeled, multi-layered approach also allowed for a smaller envelope area as the major limiting size for the vehicle lay in the electrical assembly itself. With precision manufacturing, the vehicle had much more reliable performance which allowed the majority of the design changes to lie in the vehicle software. Custom, 3D printed wheel inserts were also created to increase the

reliability of the design, although the vehicle still struggled with varying wheel speeds which required extensive calibration when in autonomous functionality mode.

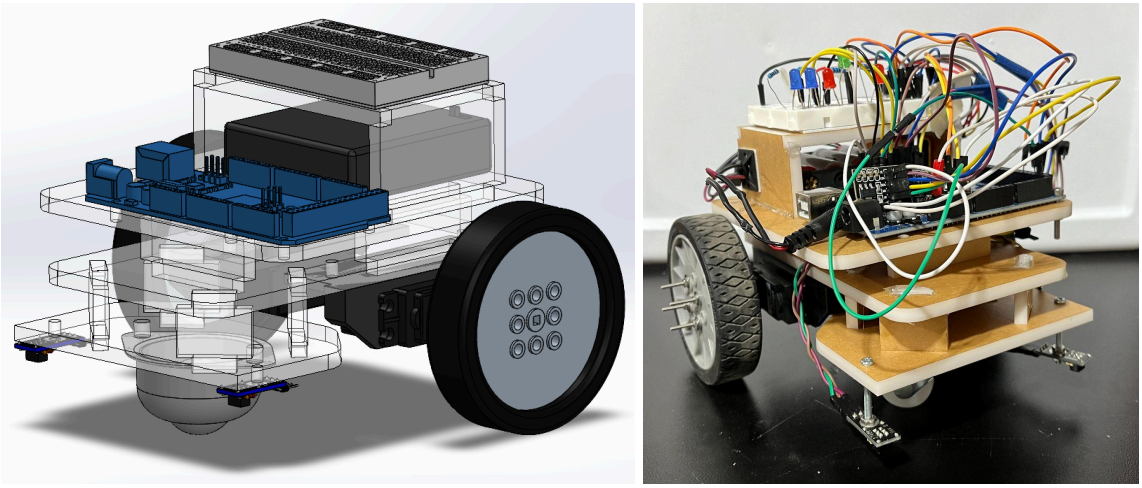


Figure B2. Second vehicle iteration for remote control functionality. (Left) CAD model. (Right) Physical prototype.

The software changes from Milestone 1 to 2 were fairly minor, however there were three big functionality changes that carried to further Milestones. The biggest improvement to the system was in modular programming, allowing separate functions to be written concurrently for easier debugging and repeated recall purposes, such as with line detection, LED indication, radio receiving, radio receiver setup, QTI sensor parsing, and wheel movement. This allowed for the line detection code to be rewritten to explicitly return the number of lines previously passed over by counting the number of changes in light levels over a set time period. In the Milestone 1 software, separate LEDs were used for single and double line detection which would erroneously return single line values until a second line was detected. Lastly, the modular approach in the code allowed for separate radio setup and radio receive functions to be written which allowed for faster and automatic setup of the remote control receiver and simplified reading of transmitted values from the vehicle side. These combined to allow for fast signal processing so the vehicle could operate in real time at a moderate BAUD rate.

The final physical prototype of the design was created to address several major issues in performance and constraint satisfaction: (1) protection of the microcontroller from environmental hazards, (2) minimizing the mass of the vehicle, (3) allowing for additional hardware implementation, and (4) facilitating manufacturing and maintenance of the vehicle (Figure B3). The microcontroller was protected by adding a front plate as well as a roof layer to the design which doubles to allow for more room to mount the IR sensors, Servo motor for the ladder design, and the radio receiver module. The ¼” acrylic proved to be too heavy for the design and given the low loads on most of the major structural elements above the base layer, ⅛” acrylic would be sufficient. The mass was also decreased by introducing slots and trimming any extraneous material off the vehicle–this doubled to allow for easier wiring given the number of added electrical components. Note also that for debugging purposes, the indicator LEDs were replaced with a seven segment display which was used for testing line counting, position display, QTI sensing, radio receiving, etc. Minimal software changes were made for this Milestone as the major changes were in the physical prototype with added code to support the IR signal detection and ladder control based on remote control commands.

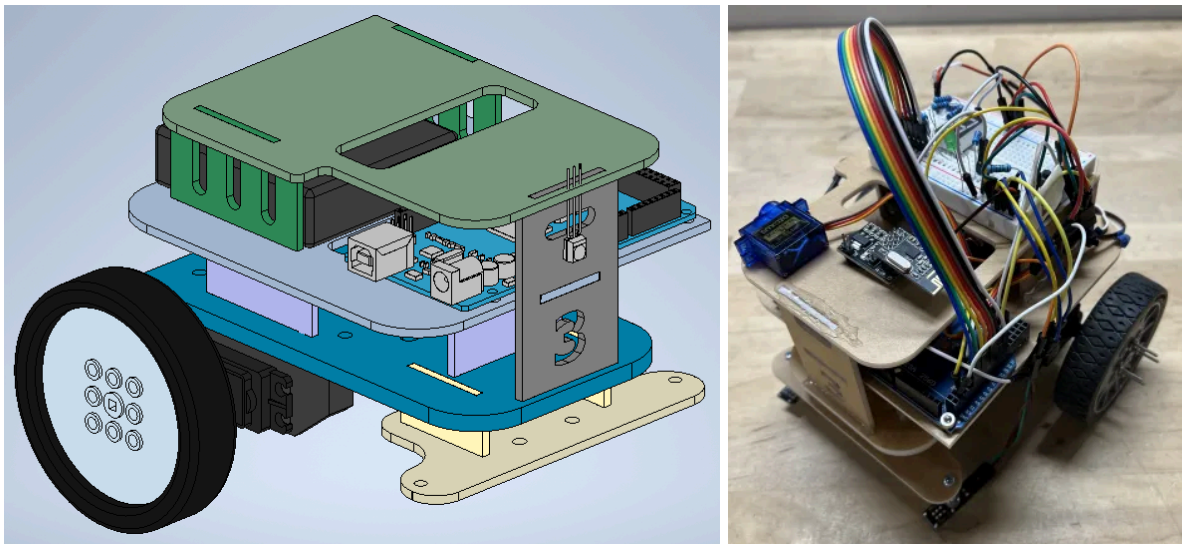


Figure B3. Final design prototype for autonomous functionality. (Left) Initial CAD model. (Right) Milestone 3 physical construction.

Appendix C: Movement Functions

This appendix discusses the functionality of the TRIDENT to traverse the grid while tracking the position and direction at all times. In doing so, it employs two main movement functions, the "move forward" command and the "turn angle" command.

With either function, the vehicle needs to have an accurate sense of direction at all times which means being able to travel in a straight line for an extended distance (an estimated search path length of 24 feet) and turning to the exact orthogonal directions desired. An MPU-6050 gyroscope is used to determine the angular rate of rotation of a three-axis gyroscope at all times, and, through integrating these values over time using standard Arduino libraries, allows an angular measurement of at most +/- 0.2 degrees. Note that slower vehicle rotation speeds and general slower movement allow more accuracy; this value was the minimum threshold utilized in performance. To make sure the vehicle travels in a straight line, the gyroscope angle measurement (the "true" angular direction of the vehicle) is compared to the desired angular direction. As this integration only allows for relative angular directions, the starting direction is set as $\theta = 0$ deg with angles increasing counter-clockwise–e.g., if the vehicle started facing "East", then a left turn would have a desired angular direction of $\theta = 90$ deg. A discrete PID controller can then be utilized to compare the current gyroscope measurement to the desired heading (0, 90, 180, or 270 degrees relative to the start) with

$$\Delta V = K_p e(t) + K_i \Sigma e(t) + K_d [e(t) - e(t-1)] \qquad \text{[C1]}$$

where ΔV is the change in the wheel speed; K_p, K_i, and K_d are the gains of the controller; $e(t)$ is the current error of the system, and $e(t-1)$ is the error in the last frame measured. For this controller, tuned gains of $[K_p, K_i, K_d] = [8E\text{-}6, 1E\text{-}7, 1E\text{-}2]$ were used for a fast response time with minimal steady-state error.

The first stage of the vehicle's trial involves a calibration cycle of tuning the wheel speeds and step distances based on the layout of the track (Figure C1). The vehicle first travels forward and straightens itself out using the PID controller discussed in Equation C1– (a) in the Figure below. When it reaches the first line, as determined by a drop in the QTI sensor signals at the front of the vehicle, it will straighten itself out until both sensors are contacting the line (b, c)–note that the line placement may be slightly askew from the gyroscopic readings so this allows for more accurate distance measurements. The vehicle will then move forward until both sensors are off the line and record this time value (d). With the PID controller, the vehicle will travel in a straight line until it reaches the next line and record the time value of this encounter (e). As this distance inside edge line-to-line distance is assumed constant and can be measured, as well as the average tile-to-tile spacing, distance from the midpoint of the vehicle to the rear axles, etc., the amount of time needed to travel a distance d_i is given by

$$t_i = t_0 \frac{d_i}{d_0} \qquad \text{[C2]}$$

Note that in this equation, the vehicle is assumed to always be traveling at a constant speed and thus a command can be given to tell the vehicle to stop moving after the time calculated in Equation C2 has elapsed for the desired distance.

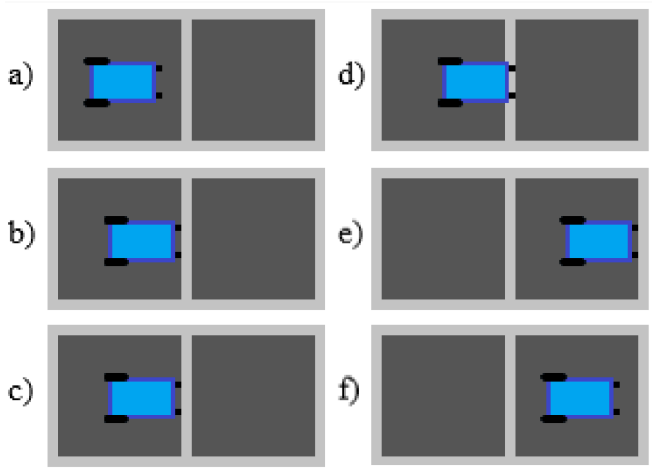


Figure C1. Initial vehicle speed calibration performed at the start of the trial. Steps (a) through (f) are performed in order before any "move forward" or "turn angle" commands are given.

With the vehicle speed calibrated, the TRIDENT is instructed to travel backwards to the center of the tile to begin the search path (f). When a "move forward" command is given, Equation C2 will be used to determine how long to allow the vehicle to move before turning off the motors and receiving the next command.

While the gyroscope enables precise turning with minimal error in angular direction, a separate movement mechanic is needed to allow the vehicle to consistently turn (Figure C2). The vehicle is centered in each tile with the rear axle placed slightly away from the geometric center of the tile (a). To get the vehicle to turn about the center, the rear axle needs to be moved forward to the tile center (b) before a turn is initiated (c); this utilizes the vehicle speed calibration values discussed in Figure C1. Once the vehicle is turned, a recentering process can begin before the vehicle is given any "move forward" commands again (d).

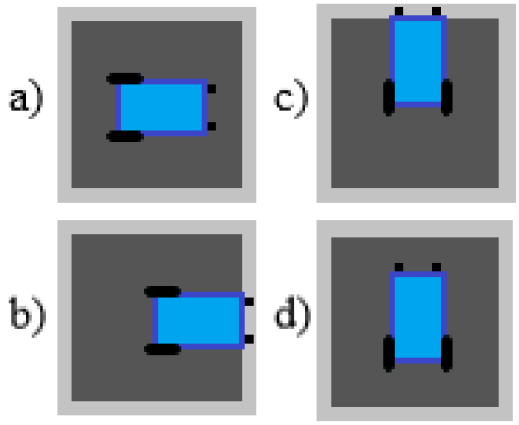


Figure C2. Turning process (a-d) to allow the TRIDENT to remain centered in the tile after a turn command is issued.

In the final implementation of the design, it was decided for simplicity that only the front sensor would be used for building detection as faulty readings on the side IR sensors prevented full operation of this innovation. With this, the search path utilized in the final iteration is shown in Figure C3 below. This path is constructed to allow the vehicle to check every building on the board, ignoring the outermost rows

and columns as these did not have any buildings in them. Note that as this is a predefined path as an array of points for the vehicle to follow, this can be easily changed for different desired search paths. For example, Figure C3 shows a previous iteration of a simplified search path with all three IR sensors utilized for minimum distance. Other search paths considered include a spiral path, either heading towards the center or away for early building detection and prevention of redundancy in traveled points (e.g., allowing the vehicle to search for other buildings while traveling around known buildings in fewest steps).

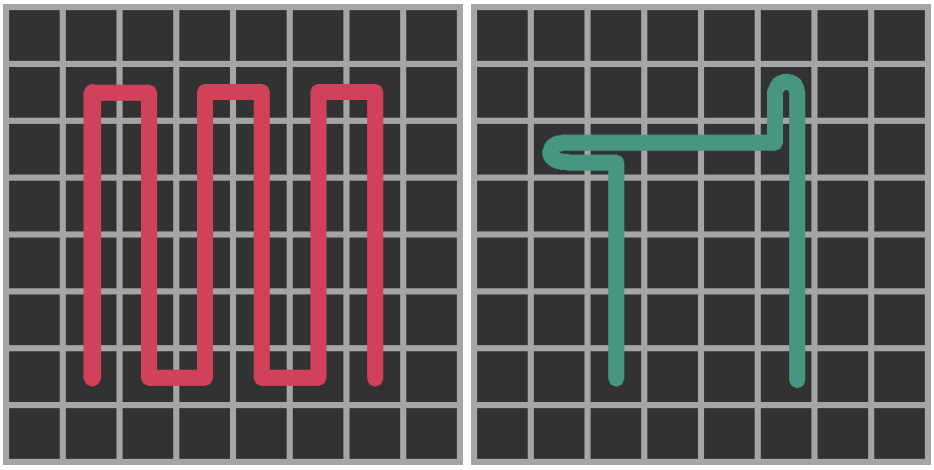


Figure C3. (Left) Search path considered in the final iteration of the TRIDENT, utilizing just one on-board IR sensor (L = 36). (Right) Early iteration of the search path utilizing all three IR sensors (L = 16).

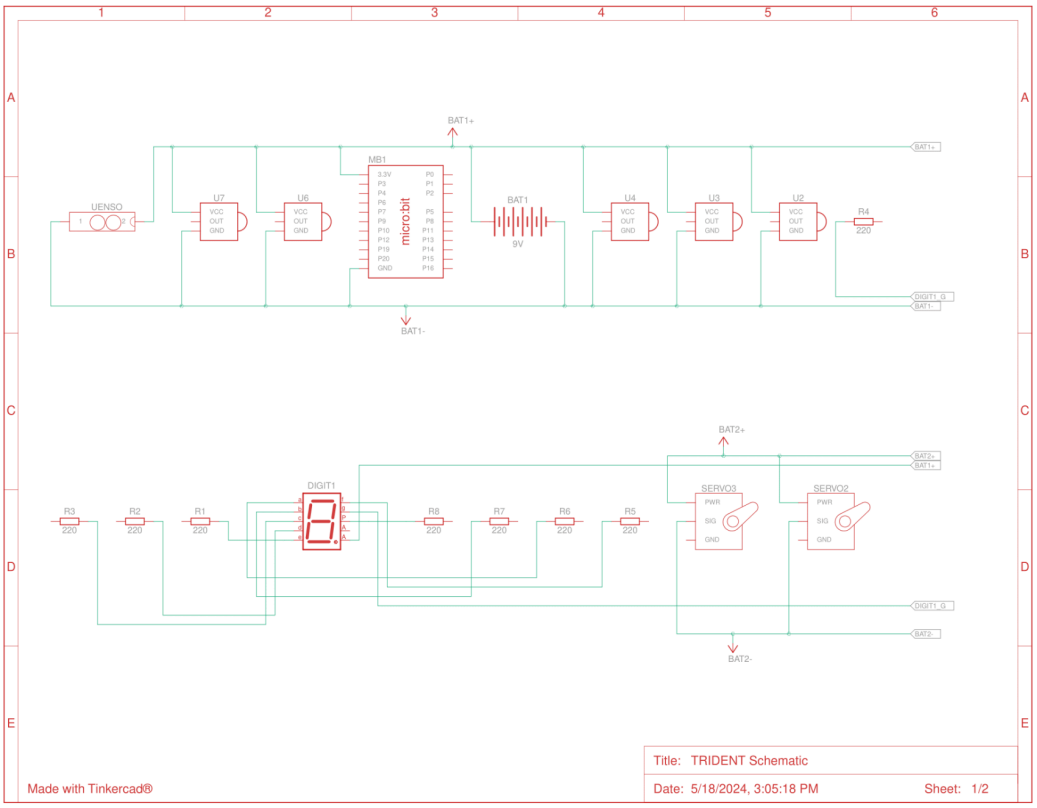
Appendix D: Electrical System

This appendix contains an overview of the hardware utilized in the design as well as detailed views of the wiring schematics used to implement this hardware into the compact space of the vehicle body. A complete list of the electronic components used is broken down by subsystem and described in the table below–refer to Figure 7 for more information on the design and power input to each component.

Table D1. Electronic components used across the various design subsystems.

Part	Qty.	System	Description
Arduino	1	Controller	Arduino Mega microcontroller board used for all logical control and central programming capabilities
Radio	1	Controller	Radio receiver module to receive signals from base station and operate vehicle movement remotely
Remote Button	1	Controller	Toggling button to switch between remote operating and autonomous operating modes
Battery Pack	1	Power	Four x AA batteries with a switch to power the Micro Servo and Continuous Rotation Servos
9V Battery	1	Power	9V battery to power the Arduino and all minor electrical components (e.g., sensors, LED's, radio module)
Micro Servo	1	Ladder	Motor used for raise and lowering the ladder assembly
Capacitor	1	Ladder	100 μF capacitor for stabilizing Micro Servo power supply
Continuous Rotation Servo	2	Drive	Motors used on the rear wheels for moving the vehicle
IR Sensor	3	Sense	Detects fire transmission from buildings as well as non-fire signals from building blank faces
QTI Sensor	2	Sense	Detects light levels underneath vehicle to determining whether vehicle is over a line or plain mat
Display Panel	1	Display	Seven segment display used for debugging and active monitoring of system when disconnected from base station
LED	2	Display	LED indicator lights used for indicating power source as well as IR signal detection
Gyroscope	1	Sense	Accelerometer with built-in gyroscope for determining the vehicle yaw angle

The electrical schematics showing these component interactions is shown below in Figure D1.



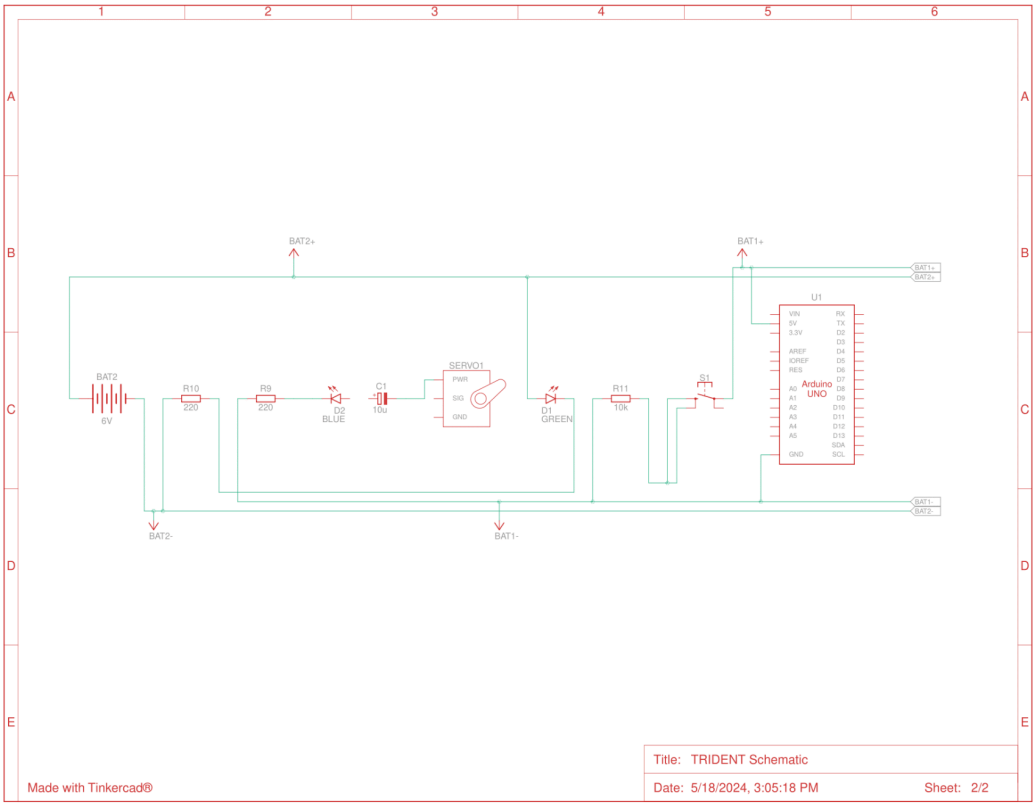


Figure D1. Electrical schematics for the TRIDENT design, featuring both powered and pin connections.

Appendix E: Bill of Materials and Budget

This appendix details the cost of all materials used in building the TRIDENT in a formatted bill of materials and tabulated budget. In Table E1 below, a bill of materials of the final design is calculated based on wholesale prices of components and estimated costs for all in-house manufactured components, such as the laser cut acrylic structural elements.

Table E1. TRIDENT Bill of Materials based on calculated price per unit of each item.

Item	Quantity	Unit Price	Total Price
¼” Acrylic (quantity measured in in^2)	20.23	$0.10	$2.02
⅛” Acrylic (quantity measured in in^2)	78.94	$0.10	$7.89
Plywood (quantity measured in in^2)	76.57	$0.05	$3.83
ABS (quantity measured in in^3)	1.852	$3.25	$6.02
3 Axis Accelerometer Gyroscope	1	$3.33	$3.33
Arduino Mega 2560	2	$20.99	$41.98
9V Battery	1	$2.41	$2.41
AmazonBasics 8-Pack AA Batteries	4	$0.85	$3.40
Servo Motor	1	$2.33	$2.33
QTI Sensor	2	$9.99	$19.98
4 AA Battery Holder	1	$2.80	$2.80
Wireless Transceiver Module	2	$1.40	$2.80
Breadboard	2	$7.25	$14.50
Regulator	2	$1.50	$3.00
Continuous Rotation Servo FS5106R	2	$15.95	$31.90
IR Infrared Sensor for Arduino	3	$1.17	$3.51
Wheel Add-On Kit	2	$3.13	$6.26
Low-Profile Swivel Caster	1	$18.36	$18.36
7 Segment 0.56" Red LED Display	1	$0.70	$0.70
M3 Small Metric Screws	30	$0.02	$0.68
Wires	50	$0.06	$2.91
LEDs	2	$0.40	$0.80
220 Ohm Resistors	10	$0.05	$0.55
Capacitor	1	$1.11	$1.11
Joystick	2	$1.98	$3.96
Total			$187.03

In the table below, a breakdown of all purchased parts is also given to verify compliance with the constraints listed in Table 2. As shown below, of the $300 budget allocated for this project, only $3.33 had to be sourced externally, with all other design components being borrowed from Spencer Lab of the University of Delaware. The project had a want of minimizing cost throughout the semester to best utilize the materials available and overall lead to a cost efficient design.

Table E2. TRIDENT budget used out of $300 based on price per unit of each item.

Item	Quantity	Unit Price	Total Price
3 Axis Accelerometer Gyroscope	1	$3.33	$3.33
Total			$3.33

This appendix contains the complete program contained and performed on the Arduino Mega 2560 microcontroller for the final iteration of the TRIDENT. As the design had two operating modes, the main, autonomous functionality and the remote-controlled operation, both sets of code are given in this appendix. Note that each code is the product of semester-long testing and iterations and as such reflect multiple design changes and need for generalized, modular approaches which often require lengthy code implementations.

Code F1. Final iteration of the main autonomous functionality code of the TRIDENT.

```
C/C++
// TITLE: Autonomous Functionality Code
// AUTHOR: Andrew Smith, Noah Etienne, Dylan Taylor, and Venkata Sai Phanindra
Mandadapu
// UPDATED: May 22nd, 2024
//
// VERSION HISTORY:
// - (5/11) Initial version, forward march program and turning protocol
// - (5/13) Added fire detection function for IR sensors
// - (5/13) Improved forward motion functionality
// - (5/16) Replaced Servo motor for ladder control
// - (5/17) Added gyroscope and accurate turning control
// - (5/17) Added PID control for straight course correction
// - (5/19) Tuned PID values and added backwards control
// - (5/19) Implemented a set search path and method to read instructions
// - (5/19) Used gyroscope and PID control for continuous course correction
// - (5/20) Added building sweep protocol and path point
//
// NOTES:
// - Fix long term error propogation in the search path
// - Allow building detection during sweep protocol
// - Implement side IR sensors
// - Write edge cases for double buildings, OOB-adjacent buildings, etc.



//
================================================================================
============================================================
// INITIALIZE PARAMETERS (Define all constants and variables used in code)
//
================================================================================
============================================================
```

```
// System variables
unsigned long time;                // current system time in ms
const int start_delay_time = 3000; // delay time before motor movement (in ms)
const int step_delay_time = 500;   // time to wait after moving one tile
forward (in ms)

// Define wheel motors
#include <Servo.h>
Servo leftMotor;
Servo rightMotor;

// Define LED pins
const int dsp_LEDs[] = {47, 45, 43, 41, 39, 37, 35}; // connection LEDs for all
segments of display
const int onLED = 33;                              // "ON" indicator -> dot
on 7-segment display
const int rmtLED = onLED;                          // remote control
functionality indicator (same on ON LED except it blinks)
const int fireLED = 46;                            // LED to sync with the
fire IR emission


// Define sensors
int snsL, snsR;              // raw QTI sensor reading values
int stsL, stsR;              // readings of QTI sensors -> 0 = grey, 1 = line
int prev_stsL, prev_stsR;    // previous readings " "
const int left_QTI_pin =  3; // left QTI sensor connection pin
const int right_QTI_pin = 2; // right " "

// Line detection settings
int num_lines = 0;                  // number of lines passed over last
const int threshold[2] = {70, 60};  // light threshold to use for line
detection -> one for each sensor {L, R}
int line_dtc_time;                  // amount of time to collect all line
detection signals (in ms) -> scales with vehicle speed
const int line_dtc_time_max = 1000; // " " MAXIMUM value -> based on maximum
speed of both wheels
unsigned long LED_start_time;       // starting time for LED display
const int LED_on_time = 600;        // amount of time to turn LED indicator
lights on (in ms)
```

```
bool oob = false;                   // whether vehicle has reached
out-of-bounds or not
unsigned long oob_start_time;       // starting time for out-of-bounds line
detection
int oob_dtc_time;                   // amount of time for continuous line
detection to indicate out-of-bounds line (in ms)
const int oob_dtc_time_max = 350;   // " " MAXIMUM value -> based on maximum
speed of both wheels


// Motion control parameters
const int motor_pin_L = 30;             // attachment pin of motor
const int motor_pin_R = 31;             // " "
float r_spd, l_spd;                     // speed of each wheel [-1, 1] -> % of
max speed -> FORWARD direction
float r_spd_rev, l_spd_rev;             // " " -> REVERSE direction
const float r_spd_initial =  0.152;     // set initial wheel speeds ("straight"
value) -> FORWARD direction
const float l_spd_initial =  0.130;     // " "
const float r_spd_initial_rev = -0.130; // set initial wheel speeds ("straight"
value) -> REVERSE direction
const float l_spd_initial_rev = -0.165; // " "
float oob_spd_threshold = 0.07;         // threshold to prevent moving out of
bounds
float max_spd = 0.2;                    // maximum wheel speed, used to
throttle control


// Set up remote control
#include <SPI.h>
#include "RF24.h"
#define CE_PIN 7
#define CSN_PIN 8
RF24 radio(CE_PIN, CSN_PIN);
uint8_t address[][6] = { "1Node", "2Node" };
bool radioNumber = 1;  // 0 uses address[0] to transmit, 1 uses address[1] to
transmit
bool role = false;     // true = TX role, false = RX role
int payload[3];
int i;
float p0;
float p1;

// Remote control parameters
```

```
const int rmt_btn_pin = 48;       // pin for remote control button
bool rmt_on = false;              // whether or not remote control
functionality is on
const float rmt_blnk_freq = 1.0;  // frequency to blink "on" LED when remote is
on (in Hz)
const int rmt_btn_threshold = 10; // any signal below this value will count as
the button being pressed in


// Define ladder servo
#include <Servo.h>
Servo ladder;
const int ladder_pin = 49;    // pin for ladder servo motor
const int ladder_pos_0 = 180; // default position (upright)
const int ladder_pos_1 = 120; // engaged position (horizontal)
bool ladder_down = false;     // whether to engage ladder mechanism

// Set up IR detection sensor
#include <IRremote.hpp>
const int IR_RECEIVE_PIN[3] = {42,24,40};    // IR sensor connection pins
(fire/building detection) -> 0 = left, 1 = front, 2 = right
bool prox_sns[3] = {false,false,false};      // reading from each IR sensor
bool prev_prox_sns[3] = {false,false,false}; // previous readings " "

// Fire detection variables
const int fire_dtc_time_max = 1000;
// amount of time to wait before determining whether (1) building is on fire or
(2) building is near
long fire_dtc_time[3] = {-fire_dtc_time_max, -fire_dtc_time_max,
-fire_dtc_time_max}; // time of first signal detection of each IR sensor -> set
as high value initially
bool fire_timer_exp[3] = {true, true, true};
// whether fire detection timer has expired (for each sensor)
int building_dtc[3] = {0,0,0};
// values read from fire detection function -> 0 = no signal, 1 = building
(blank face), 2 = fire detected
bool building_in_front = false;
// whether a building is in the next search path position (skips path points
that are blocked)

// Forward Step Calibration
bool tile_calibrate = true;   // setting to calibrate distance measurement
bool tile_recalibrate = true; // " "
unsigned long t1; // time from line-to-line during calibration
```

```
unsigned long t2; // time to recenter vehicle after calibration
unsigned long tn; // time to move 1 tile (step time)
unsigned long tc; // time to move vehicle to next line (used for recalibration)
const float d1 = 28.5; // distance from line-to-line (inside distance) (cm)
const float d2 = 6.4;  // distance to recenter vehicle after calibration (cm)
const float dn = 30.0; // distance from line-to-line (tile dimension, cm)
const float dc = 5.0;  // distance from sensor at tile center to line (used for
recalibration)

// Gyroscope calibration
#include "Wire.h"
#include <MPU6050_light.h>
MPU6050 mpu(Wire);
float ang;                    // current yaw heading of vehicle
const float ang_error = 0.2; // accepted deviation from desired angle for
turning (degrees)
const float d_wc = 5.0;      // distance from wheel axle to center of vehicle
body (cm)

// Straight line calibration
const float straight_Kp = 0.0000080; // PID parameters for straight line
movement calibration
const float straight_Ki = 0.0000001; // " "
const float straight_Kd = 0.0100000; // " "


// A* Path Finding Parameters
// Initialize default grid
const int ROW = 10; // size of grid (8x8 grid with a 1 tile border)
const int COL = 10; // " "
int grid[ROW][COL] =
  { { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
              { 0, 1, 1, 1, 1, 1, 1, 1, 1, 0 },
              { 0, 1, 1, 1, 1, 1, 1, 1, 1, 0 },
              { 0, 1, 1, 1, 1, 1, 1, 1, 1, 0 },
              { 0, 1, 1, 1, 1, 1, 1, 1, 1, 0 },
              { 0, 1, 1, 1, 1, 1, 1, 1, 1, 0 },
              { 0, 1, 1, 1, 1, 1, 1, 1, 1, 0 },
              { 0, 1, 1, 1, 1, 1, 1, 1, 1, 0 },
    { 0, 1, 1, 1, 1, 1, 1, 1, 1, 0 },
              { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 }
};
/* Grid above is numbered ***upside down***:
            South
```

```
      [0,0] [1,0] [2,0] ...
West  [0,1] [1,1] [2,1] ...  East
      [0,2] [1,2] [2,2] ...
       ...   ...   ...  ...
           North
0 = building/out-of-bounds, 1 = free tile
*/
// Initial position
int pos[2] = {4,8}; // enter starting position (x,y) using (0,9) for South-West
corner
int end[2] = {6,3}; // end (x,y) coordinate " " -> TEMP used for verifying path
finding works
String dir = "E";   // enter vehicle starting direction -> N = North (+y), E =
East (+x), S = South (-y), W = West (-x)
// Define A* computation matrices
int h[ROW][COL]; // heuristic matrix -> uses manhattan distance from grid point
to end target
int g[ROW][COL]; // path matrix      -> cost of the path from start point to
grid point
int f[ROW][COL]; // cost matrix      -> sum of path and heuristic matrix values


// Define Search Path
int path_index = 0; // current step of path to take
int search_path[36][2] = {
  {7,7}, {7,6}, {7,5}, {7,4}, {7,3}, {7,2},
  {6,2}, {6,3}, {6,4}, {6,5}, {6,6}, {6,7},
  {5,7}, {5,6}, {5,5}, {5,4}, {5,3}, {5,2},
  {4,2}, {4,3}, {4,4}, {4,5}, {4,6}, {4,7},
  {3,7}, {3,6}, {3,5}, {3,4}, {3,3}, {3,2},
  {2,2}, {2,3}, {2,4}, {2,5}, {2,6}, {2,7},
};



//
================================================================================
================================================================
// SETUP FUNCTION (Performs once at beginning of code)
//
================================================================================
================================================================
```

```
void setup() {
  Serial.begin(9600);

  delay(start_delay_time);
  display_char("-");

  // Declare LED pins as outputs
  pinMode(onLED, OUTPUT);
  pinMode(rmtLED, OUTPUT);
  pinMode(fireLED, OUTPUT);
  for(i=0; i<8; i++){
    pinMode(dsp_LEDs[i], OUTPUT); // set all display pins as outputs
  }

  // Set motor attachments
  leftMotor.attach(motor_pin_L);  // left rear wheel
  rightMotor.attach(motor_pin_R); // right " "

  // Initiate speed control
  r_spd = r_spd_initial; // default motor speed (forwards direction)
  l_spd = l_spd_initial; // " "
  r_spd_rev = r_spd_initial_rev; // default motor speed (reverse direction)
  l_spd_rev = l_spd_initial_rev; // " "

  //radio_setup();

  // Initiate ladder
  ladder.attach(ladder_pin);  // attach servo motor for ladder
  ladder.write(ladder_pos_0); // default position

  // Initiate QTI sensor readings
  prev_stsL = 0;
  prev_stsR = 0;

  // Initiate IR sensors
  IrReceiver.begin(IR_RECEIVE_PIN[0], ENABLE_LED_FEEDBACK);
  IrReceiver.begin(IR_RECEIVE_PIN[1], ENABLE_LED_FEEDBACK);
  IrReceiver.begin(IR_RECEIVE_PIN[2], ENABLE_LED_FEEDBACK);

  // Print starting information
  print_grid(grid, ROW, COL, "Grid");
// display grid array as text characters (for debugging purposes)
  Serial.println("Start: (" + String(pos[0]) + ", " + String(pos[1]) + ")"); //
print start and end coordinates
```

```
  Serial.println("Direction: " + dir);
  //Serial.println("End  : (" + String(end[0]) + ", " + String(end[1]) + ")");
// " "
  Serial.print("Path: ");
  for(int i=0; i<16; i++) {
    Serial.print("(" + String(search_path[i][0]) + "," +
String(search_path[i][1]) + "), ");
  }
  Serial.println("");

  // Run A* algorithm
  String test_path;
  for(int i=0; i<16; i++) {
    if(i==0){
      test_path = aStar(pos[0], pos[1], search_path[i][0], search_path[i][1],
dir);
      Serial.println("");
      Serial.print("{" + String(pos[0]) + "," + String(pos[1]) + "}->{" +
String(search_path[i][0]) + "," + String(search_path[i][1]) + "}: ");
      Serial.println(test_path);
    }
    else if(i>=1){
      test_path = aStar(search_path[i-1][0], search_path[i-1][1],
search_path[i][0], search_path[i][1], dir);
      Serial.print("{" + String(search_path[i-1][0]) + "," +
String(search_path[i-1][1]) + "}->{" + String(search_path[i][0]) + "," +
String(search_path[i][1]) + "}: ");
      Serial.println(test_path);
    }
  }

  // Setup gyroscope
  display_char("--");
  Serial.println("");
  Serial.println("Setting up gyroscope...");
  gyro_setup();


  // Delay at beginning
  display_char("3");
  delay(start_delay_time / 4);
  display_char("2");
  delay(start_delay_time / 4);
  display_char("1");
```

```
  delay(start_delay_time / 4);
  display_char(" ");
  delay(start_delay_time / 4);
}



//
================================================================================
================================================================
// MAIN LOOP (Repeats each timestep)
//
================================================================================
================================================================



void loop() {

  // SYSTEM VARIABLES
  if( !rmt_on ) { // turn on "ON" LED (constant = on, blinking = remote on)
    digitalWrite(onLED, HIGH);
  }
  time = millis() - start_delay_time; // update current time


  // HARDWARE INPUT
  set_sensors();   // set sensor values
  toggle_remote(); // toggle remote control functionality

  // GENERAL MOVEMENT
  if(tile_calibrate){ // only perform at start of code
    time = millis() - start_delay_time; // update current time
    move_forward_calibrate(time); // initial calibration
    tile_calibrate = false;
  }
  else {
    time = millis() - start_delay_time; // update current time
    // Sweep building if present
    int px = search_path[path_index][0]; // next point in path search
    int py = search_path[path_index][1]; // " "
    if(grid[py][px] == 0) {
      // Get neighboring grid points of building
      /*
```

```
      int bp_n [2] = {px, py+1}; // point NORTH of the building
      int bp_e [2] = {px+1, py}; // point EAST of the building
      int bp_s [2] = {px, py-1}; // point SOUTH of the building
      int bp_w [2] = {px-1, py}; // point WEST of the building
      int bn [4] = {bp_n, bp_e, bp_s, bp_w}; // neighbors of building
      */
      int bn [4][2] = {{px, py+1}, {px+1, py}, {px, py-1}, {px-1, py}}; //
neighbors of building

      // Get starting point
      int si = 0; // starting index in building neighbor search list
      if( dir == "N") { si = 2; }
      else if( dir == "E") { si = 3; }
      else if( dir == "S") { si = 0; }
      else if( dir == "W") { si = 1; }

      // Turn right to start sweep
      turn_angle(270);

      // Check each point
      int checked_pts = 0; // number of building neighbors checked
      while(checked_pts < 4){
        // Check for fires
        left_wheel(0);  // turn off wheels while searching
        right_wheel(0); // " "
        time = millis() - start_delay_time; // update time
        unsigned long dtc_time_end = time + step_delay_time;
        while(time < dtc_time_end){ // pause for step_delay_time duration
          // Update time
          time = millis() - start_delay_time; // update time
          // Update sensor values
          for(int i=0; i<3; i++){
            building_dtc[i] = fire_detection(i);
          }
          // Turn on LEDs if signal is detected
          if(building_dtc[0]!=0 ||building_dtc[1]!=0 || building_dtc[2]!=0 ) {
            digitalWrite(fireLED, HIGH);
          } else {
            digitalWrite(fireLED, LOW);
          }
          // Turn vehicle to face fire and drop ladder
          if(building_dtc[0] == 2){
            turn_angle(90);
            ladder.write(ladder_pos_1); // set to load position
```

```
            display_char("F");          // manually set fire signal
            delay(1000);                // wait set time period
            ladder.write(ladder_pos_0); // return to default position
            path_index += 1; // skip path point
          }
        }

        // Move to neighbor point
        /*
        String temp_steps = aStar(pos[0], pos[1], bn[si][0], bn[si][1], dir);
        for(int i=0; i<temp_steps.length(); i++) { // go through each
instruction
          time = millis() - start_delay_time; // update current time
          String c = String(temp_steps.charAt(i)); // get specific instruction
          if(c == "L") {
            turn_angle(90);
          }
          else if(c == "R") {
            turn_angle(270);
          }
          else if(c == "U") {
            turn_angle(180);
          }
          else if(c == "S" || c == "F") {
            move_forward(1, time); // always move forward after turn direciton
is stated
          }
        }
        */
        move_forward(1, time); turn_angle(90); move_forward(1, time); // manual
control -> tell vehicle to go counter-clockwise around building

        // Go to next neighbor
        si += 1; if(si > 3){ si -= 4; } // get index of next neighbor to check
        checked_pts += 1;
      }



    }
    // Skip blocked path points
    while(grid[py][px] == 0){
      building_in_front = true; // used to skip path points
      path_index += 1;
```

```
      px = search_path[path_index][0];
      py = search_path[path_index][1];
    }
    // Calculate instructions to next path point
    String steps = aStar(pos[0], pos[1], search_path[path_index][0],
search_path[path_index][1], dir);
    //Serial.println(steps);
    //display_char(String(path_index));
    // Move vehicle
    for(int i=0; i<steps.length(); i++) { // go through each instruction
      time = millis() - start_delay_time; // update current time
      String c = String(steps.charAt(i)); // get specific instruction
      if(c == "L") {
        turn_angle(90);
      }
      else if(c == "R") {
        turn_angle(270);
      }
      else if(c == "U") {
        turn_angle(180);
      }
      else if(c == "S" || c == "F") {
        move_forward(1, time); // always move forward after turn direciton is
stated
      }
    }
    // Update path index
    if(building_in_front == false){ // path index was already skipped above
      path_index += 1; // increase index (go to next position in search path)
    }
    // Reset variables
    building_in_front = false;
    //digitalWrite(fireLED, LOW);
  }

  /*
  // BUILDING DETECTION
  // Determine detection type for each sensor
  for(int i=0; i<3; i++){
    building_dtc[i] = fire_detection(i);
  }
  // Turn on LED if any IR signal is detected
  set_sensors(); // update sensor values
  if(prox_sns[0] || prox_sns[1] || prox_sns[2] ) {
```

```
    digitalWrite(fireLED, HIGH);
  } else {
    digitalWrite(fireLED, LOW);
  }
  // Ladder control
  if(building_dtc[1] == 2){
    ladder.write(ladder_pos_1); // set to load position
    display_char("F");          // manually set fire signal
    delay(750);                 // wait set time period
    building_dtc[1] = 0;        // manually turn front sensor off
    ladder.write(ladder_pos_0); // return to default position
  } else {
    ladder.write(ladder_pos_0); // return to default position
  }


  // UPDATE MAP
  cartography(); // continuously add buildings to grid as they are found
  //print_grid(grid, ROW, COL, "Grid");
  */

  // DISPLAY DEBUG INFORMATION
  debug("null"); // print debug information -> [sensor, status, line, remote,
fire, calibration, position, gyroscope, null]
}



//
================================================================================
================================================================
// DEFINE GLOBAL FUNCTIONS - Part 1 (Basic vehicle inputs and outputs)
//
================================================================================
================================================================



long RCTime(int sensorIn){
  // Read QTI sensor at pin and output sensor signal strength
  long duration = 0;
  pinMode(sensorIn, OUTPUT);     // Make pin OUTPUT
  digitalWrite(sensorIn, HIGH);  // Pin HIGH (discharge capacitor)
  delay(1);                      // Wait 1ms
```

```
  pinMode(sensorIn, INPUT);      // Make pin INPUT
  digitalWrite(sensorIn, LOW);   // Turn off internal pullups
  while(digitalRead(sensorIn)){  // Wait for pin to go LOW
    duration++;
  }
  return duration;
}



int left_wheel(float spd) {
  // Set motor rotation rate based on input motor speed
  int spd_us = int( spd * 500.0 + 1500.0 ); // map microsecond PWS input for
motor
  leftMotor.writeMicroseconds(spd_us);      // move motor set amount
}
int right_wheel(float spd) {
  // See above
  int spd_us = int( -1*spd * 500.0 + 1500.0 ); // map microsecond PWS input for
motor -> use positive values for forward movement
  rightMotor.writeMicroseconds(spd_us);        // move motor set amount
}



void set_sensors() {
  // Read QTI sensors and determine whether line is detected
  snsL = RCTime(left_QTI_pin);  // left QTI sensor value
  snsR = RCTime(right_QTI_pin); // right " "
  if( snsL < threshold[0] ) { stsL = 1; } else { stsL = 0; } // left line
detection status -> 1 = line, 0 = grey
  if( snsR < threshold[1] ) { stsR = 1; } else { stsR = 0; } // " "
}



void gyro_setup() {
  // Function to setup gyroscope at start of code.
  Wire.begin();

  byte status = mpu.begin();
  Serial.print(F("MPU6050 status: "));
  Serial.println(status);
  while(status!=0){ } // stop everything if could not connect to MPU6050
```

```
  Serial.println(F("Calculating offsets, do not move MPU6050"));
  delay(1000);
  // mpu.upsideDownMounting = true; // uncomment this line if the MPU6050 is
mounted upside-down
  mpu.calcOffsets(); // gyro and accelero
  Serial.println("Gyroscope sucessfully set up.");
}


void read_gyro() {
  // Read angle from gyroscope data and set "ang" value accordingly.
  // OUTPUT: ang will be in range [0, 360] always.

  // Read data from gyroscope -> use built in library for integration
  mpu.update(); // update readings
  ang = mpu.getAngleZ(); // raw reading, in range [-inf, inf]

  // Convert to value from [0, 360]
  while(ang < 0.0) {
    ang = ang + 360.0;
  }
  while(ang > 360.0) {
    ang = ang - 360.0;
  }

}



void get_timing() {
  // Calculate the timing variables based on the speed of the right sensor.
  float w = 14.8;                     // width between wheels (in cm)
  float d = 1.5;                      // distance between right sensor and
right wheel
  float r = d / w;                    // ratio of distances, used for linear
scaling
  float wq = l_spd*r + r_spd*(1-r);   // effective speed at the sensor ->
ranges from [-1, 1] just like wheel speeds
  // Return scaled value
  if( wq <= oob_spd_threshold ) { // prevents divide by zero error
    wq = oob_spd_threshold;
  }
  float s = wq / max_spd; // ratio of sensor ground speed to maximum speed
```

```
  line_dtc_time = line_dtc_time_max / s; // linearly scale detection time ->
slower speeds = more detection time
  oob_dtc_time = oob_dtc_time_max / s;   // " "
}



bool curr_but = 1; // current button value -> initialize as OFF
bool prev_but = 1; // previous button value -> initialize as OFF
void toggle_remote() {
  // Turn remote control functionality on and off

  curr_but = digitalRead(rmt_btn_pin); // current button value

  if( curr_but == 0 && prev_but == 1 ) { // button is pressed
    if ( rmt_on == true ) { rmt_on = false; } // if on -> turn off
    else if ( rmt_on == false ) { rmt_on = true; } // if off -> turn on
  }

  // Blink "on" LED when remote is on
  float x = 2*(2*floorf(rmt_blnk_freq*time/1000) -
floorf(2*rmt_blnk_freq*time/1000) ) + 1;
  if( rmt_on ) {
    if( x > 0 ) {
      digitalWrite(rmtLED, HIGH);
    }
    else {
      digitalWrite(rmtLED, LOW);
   }
  }

  // Update previous value
  prev_but = curr_but; // update previous button value


  // Prevent boot-up signal errors
  if( time < 100 ) { // ignore signal in first 100ms
    rmt_on = false;
  }

}
```

```
unsigned long currentTime = 0;  // time at current line detection
int temp_num = 0;               // number of lines detected (temporary
variable)
bool reset_counter = false;     // used to reset temporary counter only once
int prev_sts = 0;               // previous line status (temporary variable)
int line_detection(int sts) {
  // Return the number of lines detected from the input sensor status
  //    0 -> plain mat, no lines
  //    1 -> single line
  //    2 -> double line
  //   -1 -> out-of-bounds line

  // Set detection time
  if (sts == 1) { // line IS being detected
    if( prev_sts == 0 && time > currentTime + line_dtc_time ) {
      oob = false;
      temp_num = 0;               // reset counter if this is the first
detection in a while
    }

    currentTime = time;           // save time of any line detection encounter
  }
  else { // line is NOT being detected
    oob_start_time = time;        // if line detection stops we are no longer
at out-of-bounds line, so don't let timer run out
  }

  // Output I: out-of-bounds
  if( time > oob_start_time + oob_dtc_time ) { // line has been continously
detected for set amount of time
    LED_start_time = currentTime + line_dtc_time; // set starting time for LED
display as the end of this timer
    return -1; // out-of-bounds indicator value
  }

  // Output II: line count
  if( (time > currentTime + line_dtc_time) && !oob) { // time has passed for
line detection
      LED_start_time = currentTime + line_dtc_time; // set starting time for
LED display as the end of this timer
      return temp_num;                            // return number of lines
detected
  }
  else { // still within line detection timer
```

```
    if ( (sts - prev_sts) == 1 && !oob) { // sensor went from grey to line
      temp_num += 1;
    }
  }

  prev_sts = sts; // update previous value

}



void line_display(int num) {
  // Use indicator light to show the line detection state

  if ( time > (LED_start_time + LED_on_time) ) { // indicator time has elapsed
    display_char(" "); // turn off all LEDs, return to grey indicator
    num_lines = 0;     // reset number of lines variable
  }
  else {
    if ( num == 0 ) { // no line
      display_char(" "); // turn off all LEDs, return to grey indicator
    }
    else if ( num == 1 ) { // single line
      display_char("1"); // display "1"
    }
    else if ( num == 2 ) { // double line
      display_char("2"); // display "2"
    }
    else if ( num == -1) { // out-of-bounds
      oob = true;
      display_char("---"); // display 3 dashes
    }
  }

}



void display_char(String s){
  // Recieve input character and display that character on the 7-segment
display
  if( s == "0") { // not currently used
    digitalWrite(dsp_LEDs[0], HIGH);
    digitalWrite(dsp_LEDs[1], HIGH);
```

```
  digitalWrite(dsp_LEDs[2], HIGH);
  digitalWrite(dsp_LEDs[3], LOW);
  digitalWrite(dsp_LEDs[4], HIGH);
  digitalWrite(dsp_LEDs[5], HIGH);
  digitalWrite(dsp_LEDs[6], HIGH);
}
else if( s == "1") { // single line
  digitalWrite(dsp_LEDs[0], LOW);
  digitalWrite(dsp_LEDs[1], LOW);
  digitalWrite(dsp_LEDs[2], HIGH);
  digitalWrite(dsp_LEDs[3], LOW);
  digitalWrite(dsp_LEDs[4], LOW);
  digitalWrite(dsp_LEDs[5], LOW);
  digitalWrite(dsp_LEDs[6], HIGH);
}
else if( s == "2") { // double line
  digitalWrite(dsp_LEDs[0], LOW);
  digitalWrite(dsp_LEDs[1], HIGH);
  digitalWrite(dsp_LEDs[2], HIGH);
  digitalWrite(dsp_LEDs[3], HIGH);
  digitalWrite(dsp_LEDs[4], HIGH);
  digitalWrite(dsp_LEDs[5], HIGH);
  digitalWrite(dsp_LEDs[6], LOW);
}
else if( s == "3") { // misc. numerical value
  digitalWrite(dsp_LEDs[0], LOW);
  digitalWrite(dsp_LEDs[1], HIGH);
  digitalWrite(dsp_LEDs[2], HIGH);
  digitalWrite(dsp_LEDs[3], HIGH);
  digitalWrite(dsp_LEDs[4], LOW);
  digitalWrite(dsp_LEDs[5], HIGH);
  digitalWrite(dsp_LEDs[6], HIGH);
}
else if( s == "4") { // misc. numerical value
  digitalWrite(dsp_LEDs[0], HIGH);
  digitalWrite(dsp_LEDs[1], LOW);
  digitalWrite(dsp_LEDs[2], HIGH);
  digitalWrite(dsp_LEDs[3], HIGH);
  digitalWrite(dsp_LEDs[4], LOW);
  digitalWrite(dsp_LEDs[5], LOW);
  digitalWrite(dsp_LEDs[6], HIGH);
}
else if( s == "5") { // misc. numerical value
  digitalWrite(dsp_LEDs[0], HIGH);
```

```
  digitalWrite(dsp_LEDs[1], HIGH);
  digitalWrite(dsp_LEDs[2], LOW);
  digitalWrite(dsp_LEDs[3], HIGH);
  digitalWrite(dsp_LEDs[4], LOW);
  digitalWrite(dsp_LEDs[5], HIGH);
  digitalWrite(dsp_LEDs[6], HIGH);
}
else if( s == "6") { // misc. numerical value
  digitalWrite(dsp_LEDs[0], HIGH);
  digitalWrite(dsp_LEDs[1], HIGH);
  digitalWrite(dsp_LEDs[2], LOW);
  digitalWrite(dsp_LEDs[3], HIGH);
  digitalWrite(dsp_LEDs[4], HIGH);
  digitalWrite(dsp_LEDs[5], HIGH);
  digitalWrite(dsp_LEDs[6], HIGH);
}
else if( s == "7") { // misc. numerical value
  digitalWrite(dsp_LEDs[0], LOW);
  digitalWrite(dsp_LEDs[1], HIGH);
  digitalWrite(dsp_LEDs[2], HIGH);
  digitalWrite(dsp_LEDs[3], LOW);
  digitalWrite(dsp_LEDs[4], LOW);
  digitalWrite(dsp_LEDs[5], LOW);
  digitalWrite(dsp_LEDs[6], HIGH);
}
else if( s == "8") { // misc. numerical value
  digitalWrite(dsp_LEDs[0], HIGH);
  digitalWrite(dsp_LEDs[1], HIGH);
  digitalWrite(dsp_LEDs[2], HIGH);
  digitalWrite(dsp_LEDs[3], HIGH);
  digitalWrite(dsp_LEDs[4], HIGH);
  digitalWrite(dsp_LEDs[5], HIGH);
  digitalWrite(dsp_LEDs[6], HIGH);
}
else if( s == "9") { // misc. numerical value
  digitalWrite(dsp_LEDs[0], HIGH);
  digitalWrite(dsp_LEDs[1], HIGH);
  digitalWrite(dsp_LEDs[2], HIGH);
  digitalWrite(dsp_LEDs[3], HIGH);
  digitalWrite(dsp_LEDs[4], LOW);
  digitalWrite(dsp_LEDs[5], LOW);
  digitalWrite(dsp_LEDs[6], HIGH);
}
else if( s == "-") { // no line (grey/mat)
```

```
  digitalWrite(dsp_LEDs[0], LOW);
  digitalWrite(dsp_LEDs[1], LOW);
  digitalWrite(dsp_LEDs[2], LOW);
  digitalWrite(dsp_LEDs[3], HIGH);
  digitalWrite(dsp_LEDs[4], LOW);
  digitalWrite(dsp_LEDs[5], LOW);
  digitalWrite(dsp_LEDs[6], LOW);
}
else if( s == "XX") { // left and right both 0
  digitalWrite(dsp_LEDs[0], LOW);
  digitalWrite(dsp_LEDs[1], LOW);
  digitalWrite(dsp_LEDs[2], LOW);
  digitalWrite(dsp_LEDs[3], LOW);
  digitalWrite(dsp_LEDs[4], HIGH);
  digitalWrite(dsp_LEDs[5], LOW);
  digitalWrite(dsp_LEDs[6], HIGH);
}
else if( s == "LR") { // left and right both 1
  digitalWrite(dsp_LEDs[0], HIGH);
  digitalWrite(dsp_LEDs[1], LOW);
  digitalWrite(dsp_LEDs[2], HIGH);
  digitalWrite(dsp_LEDs[3], LOW);
  digitalWrite(dsp_LEDs[4], LOW);
  digitalWrite(dsp_LEDs[5], LOW);
  digitalWrite(dsp_LEDs[6], LOW);
}
else if( s == "LX") { // left is 1, right is 0
  digitalWrite(dsp_LEDs[0], HIGH);
  digitalWrite(dsp_LEDs[1], LOW);
  digitalWrite(dsp_LEDs[2], LOW);
  digitalWrite(dsp_LEDs[3], LOW);
  digitalWrite(dsp_LEDs[4], LOW);
  digitalWrite(dsp_LEDs[5], LOW);
  digitalWrite(dsp_LEDs[6], HIGH);
}
else if( s == "XR") { // right is 1, left is 0
  digitalWrite(dsp_LEDs[0], LOW);
  digitalWrite(dsp_LEDs[1], LOW);
  digitalWrite(dsp_LEDs[2], HIGH);
  digitalWrite(dsp_LEDs[3], LOW);
  digitalWrite(dsp_LEDs[4], HIGH);
  digitalWrite(dsp_LEDs[5], LOW);
  digitalWrite(dsp_LEDs[6], LOW);
}
```

```
  else if( s == "F") { // F for fire
    digitalWrite(dsp_LEDs[0], HIGH);
    digitalWrite(dsp_LEDs[1], HIGH);
    digitalWrite(dsp_LEDs[2], LOW);
    digitalWrite(dsp_LEDs[3], HIGH);
    digitalWrite(dsp_LEDs[4], HIGH);
    digitalWrite(dsp_LEDs[5], LOW);
    digitalWrite(dsp_LEDs[6], LOW);
  }
  else if( s == "--") { // two dashed lines, initial calibration
    digitalWrite(dsp_LEDs[0], LOW);
    digitalWrite(dsp_LEDs[1], HIGH);
    digitalWrite(dsp_LEDs[2], LOW);
    digitalWrite(dsp_LEDs[3], LOW);
    digitalWrite(dsp_LEDs[4], LOW);
    digitalWrite(dsp_LEDs[5], HIGH);
    digitalWrite(dsp_LEDs[6], LOW);
  }
  else if( s == "---") { // three dashed lines, out-of-bounds
    digitalWrite(dsp_LEDs[0], LOW);
    digitalWrite(dsp_LEDs[1], HIGH);
    digitalWrite(dsp_LEDs[2], LOW);
    digitalWrite(dsp_LEDs[3], HIGH);
    digitalWrite(dsp_LEDs[4], LOW);
    digitalWrite(dsp_LEDs[5], HIGH);
    digitalWrite(dsp_LEDs[6], LOW);
  }
  else if( s == " ") { // turn off all LEDs
    digitalWrite(dsp_LEDs[0], LOW);
    digitalWrite(dsp_LEDs[1], LOW);
    digitalWrite(dsp_LEDs[2], LOW);
    digitalWrite(dsp_LEDs[3], LOW);
    digitalWrite(dsp_LEDs[4], LOW);
    digitalWrite(dsp_LEDs[5], LOW);
    digitalWrite(dsp_LEDs[6], LOW);
  }
}



void debug(String type) {
  // Print statements to check functionality of the code
  // type: "sensor" = print raw sensor readings
  //       "status" = print status readings
```

```
  //       "line" = print line detection variables
  //       "remote" = show on display direction of wheel motion
  //       "fire" = print whether digital read of IR sensor is high/low
  //       "position" = x and y coordinates of vehicle
  //       "calibrate" = forward movement calibration timing variables
  //       "gyroscope" = angle readings from the gyroscope

  if( type == "sensor" ) {
    Serial.println("QTI_L: " + String(snsL) + " , QTI_R: " + String(snsR) ); //
print raw sensor readings
    if(stsL == 1 && stsR == 1) { display_char("LR"); }
    if(stsL == 1 && stsR == 0) { display_char("LX"); }
    if(stsL == 0 && stsR == 1) { display_char("XR"); }
    if(stsL == 0 && stsR == 0) { display_char("XX"); }
  }

  else if( type == "status" ) {
    Serial.println("STS_L: " + String(stsL) + " , STS_R: " + String(stsR) ); //
print sensor status readings
    if(stsL == 1 && stsR == 1) { display_char("LR"); }
    if(stsL == 1 && stsR == 0) { display_char("LX"); }
    if(stsL == 0 && stsR == 1) { display_char("XR"); }
    if(stsL == 0 && stsR == 0) { display_char("XX"); }
  }

  else if( type == "line" ) {
    Serial.println("Num Lines = " + String(num_lines) + ", LineDtcTime = " +
String(line_dtc_time) + ", OOBDtcTime = " + String(oob_dtc_time));
  }

  else if( type == "remote" ) { // turn on LEDs on display based on value of
joystick transmission sent
    Serial.println("Payload values: " + String(payload[0]) + ", " +
String(payload[1]) + ", " + String(payload[2]));
    float pL_temp = 2.0*payload[0]/1023.0 - 1.0; // value of left joystick ->
range from -1 to 1
    float pR_temp = 2.0*payload[1]/1023.0 - 1.0; // " " right joystick
    if(pL_temp > 0 && pR_temp > 0) { display_char("LR"); }
    if(pL_temp > 0 && pR_temp < 0) { display_char("LX"); }
    if(pL_temp < 0 && pR_temp > 0) { display_char("XR"); }
    if(pL_temp < 0 && pR_temp < 0) { display_char("XX"); }
  }

  else if( type == "fire" ) {
```

```
    //Serial.println("Fire Detection (Front) -> Building: " +
String(building_dtc[1]) + ", IR Signal: " + String(prox_sns[1]) + ", IR Signal
(prev.): " + String(prev_prox_sns[1]) + ", Signal Expire: " +
String(fire_timer_exp[1]) + ", Detection Time: " + String(fire_dtc_time[1]) );
    if(building_dtc[0] == 1 || building_dtc[1] == 1 || building_dtc[2] == 1) {
// any sensor detects building (blank face)
      display_char("1");
    }
    else if(building_dtc[0] == 2 || building_dtc[1] == 2 || building_dtc[2] ==
2) { // any sensor detects fire
      display_char("F"); // Fire
      delay(500);
    }
    else {
      display_char("0");
    }
  }

  else if( type == "position" ) {
    Serial.println("(x,y) : (" + String(pos[0]) + ", " + String(pos[1]) + ")"
);
    display_char(String(pos[1])); // display y-position
  }

  else if( type == "calibrate") {
    //Serial.println("[t1,tn] = [" + ", " + String(t1) + ", " + String(tn) +
"]");
  }

  else if( type == "gyroscope") {
    mpu.update(); // update readings
    float temp_x_ang = mpu.getAngleX();
    float temp_y_ang = mpu.getAngleY();
    read_gyro(); // set yaw angle using modulus function -> ang = [0, 360]
    Serial.println("Roll: " + String(temp_x_ang) + "\t Pitch:  " +
String(temp_y_ang) + "\t Yaw: " + String(ang) );
  }

  else {
    ; // do nothing
  }

}
```

```
//
/**********************************************************************
************************************************************
// DEFINE GLOBAL FUNCTIONS - Part 2 (Remote control functionality)
//
/**********************************************************************
************************************************************



void radio_setup(){

    // initialize the transceiver on the SPI bus
  if (!radio.begin()) {
    Serial.println(F("radio hardware is not responding!!"));
    while (1) {}  // hold in infinite loop
  }

  // print example's introductory prompt
  Serial.println(F("Connection created"));

  // Set the PA Level low to try preventing power supply related problems
  // because these examples are likely run with nodes in close proximity to
  // each other.
  radio.setPALevel(RF24_PA_LOW);  // RF24_PA_MAX is default.

  // save on transmission time by setting the radio to only transmit the
  // number of bytes we need to transmit a float
  radio.setPayloadSize(sizeof(payload));  // float datatype occupies 4 bytes
  //radio.setPayloadSize(sizeof(rightPayload));

  // set the TX address of the RX node into the TX pipe
  radio.openWritingPipe(address[radioNumber]);  // always uses pipe 0

  // set the RX address of the TX node into a RX pipe
  radio.openReadingPipe(1, address[!radioNumber]);  // using pipe 1

  // additional setup specific to the node's role
  if (role) {
    radio.stopListening();  // put radio in TX mode
  } else {
    radio.startListening();  // put radio in RX mode
```

```
  }
}



void radio_receive(){

  // Automatically set vehicle to be in "listening" mode
  uint8_t pipe;
  if (radio.available(&pipe)) {              // is there a payload? get the
pipe number that recieved it
    uint8_t bytes = radio.getPayloadSize();  // get the size of the payload
    radio.read(&payload, bytes);             // fetch payload from FIFO
  radio.startListening();
  }

}



//
================================================================================
================================================================
// DEFINE GLOBAL FUNCTIONS - Part 3 (Motion Control Elements)
//
================================================================================
================================================================



float straight_integral; // keep adding to this to compute steady state error
float last_error;        // track last error for derivative calculation
void straight_PID(float input_ang, float Kp, float Ki, float Kd, String type) {
  // Performs PID function on vehicle yaw alignment each frame called.
  // INPUT: desired angle (0 to 360), PID gains, and type ("forward" or
"reverse")
  // OUTPUT: sets l_spd and r_spd values OR l_spd_rev and r_spd_rev values

  // 1 CALCULATE ERROR
  read_gyro();
  float d_ang; // error of angle or amount of degrees to turn left -> (+) =
turn left, (-) = turn right
  if( (input_ang - ang) < 180 && (input_ang - ang) > 0 ) {
    d_ang = input_ang - ang;
```

```
  }
  else if ( (input_ang - ang) > 180 && (input_ang - ang) > 0 ) {
    d_ang = (input_ang - 360) - ang;
  }
  else if ( (ang - input_ang) < 180 && (ang - input_ang) > 0 ) {
    d_ang = input_ang - ang;
  }
  else if ( (ang - input_ang) > 180 && (ang - input_ang) > 0 ) {
    d_ang = input_ang - (ang-360);
  }

  // 2 CALCULATE INTERGRAL
  straight_integral += d_ang;

  // 3 ADD ALL TERMS
  float PID_value = d_ang * Kp + straight_integral * Ki + (d_ang - last_error)
* Kd; // sum of each term
  if(type == "forward"){
    l_spd -= PID_value; // PID controller
    r_spd += PID_value; // " "
  }
  else if(type == "reverse"){
    l_spd_rev -= PID_value; // PID controller
    r_spd_rev += PID_value; // " "
  }

  // 5 UPDATE VARIABLES
  last_error = d_ang;

  // 6 DEBUG
  //Serial.println("l_spd: " + String(l_spd) + ", r_spd: " + String(r_spd) + ",
d_ang: " + String(d_ang) );
  String d_ang_digit = String(String(int(abs(d_ang * 1.0))).charAt(0)); //
display first character of d_ang (scale up to make it visible)
  display_char(d_ang_digit);

}


void move_forward_calibrate(unsigned long curr_time){
  // Calibrates the motion of the vehicle for moving n tiles. Input calibration
mode and time of function call.
  // Notes: Only gets performed once at start of code to (1) position vehicle
to center of tile and (2) calculate required delay times
```

```
  unsigned long move_time = curr_time; // time spent actually moving forward

  // 1 MOVE TO FIRST LINE
  //Serial.println("Calibration: Moving to first line");
  display_char("1");
  set_sensors(); // update sensor values
  while( stsL==0 && stsR==0 ) { // line is not being detected
    // Update sensors
    set_sensors(); // update sensor values
    // Move forward
    left_wheel(l_spd);
    right_wheel(r_spd);
    // Straighten vehicle
    straight_PID(0, straight_Kp, straight_Ki, straight_Kd, "forward");
  }
  straight_integral = 0; // reset PID variables
  last_error = 0;        // " "
  left_wheel(0);  // turns wheels off
  right_wheel(0); // " "
  delay(step_delay_time);


  // 2 ADJUST DIRECTION
  display_char("2");
  float temp_l_spd = l_spd; // revert back to original values after direciton
is adjusted
  float temp_r_spd = r_spd; // " "
  while( ang > ang_error && ang < (360-ang_error) ) { // angle outside
allowable "straight" region
    // Determine turn direction
    if( ang < 180 ) { // turn to the right
      l_spd = temp_l_spd;
      r_spd = 0;
    }
    else if (ang > 180) { // turn to the left
      l_spd = 0;
      r_spd = temp_r_spd;
    }
    // Move vehicle
    left_wheel(l_spd);
    right_wheel(r_spd);
    // Update yaw direction
    read_gyro();
```

```
  }
  // reset wheel speeds
  l_spd = temp_l_spd;
  r_spd = temp_r_spd;
  //Serial.println("Done.");
  left_wheel(0);  // turns wheels off
  right_wheel(0); // " "
  delay(step_delay_time);
  /*
  display_char("2");
  float temp_l_spd = l_spd; // revert back to original values after direciton
is adjusted
  float temp_r_spd = r_spd; // " "
  while( stsL != stsR ) { // only one sensor detects line
    set_sensors(); // update sensor values
    // Determine turn direction
    if( stsR == 1 ) { // turn to the right
      l_spd = temp_l_spd;
      r_spd = 0;
    }
    else if ( stsL == 1) { // turn to the left
      l_spd = 0;
      r_spd = temp_r_spd;
    }
    // Move vehicle
    left_wheel(l_spd);
    right_wheel(r_spd);
  }
  // 2b Reset wheel speeds
  l_spd = temp_l_spd;
  r_spd = temp_r_spd;
  // Pause
  left_wheel(0);  // turns wheels off
  right_wheel(0); // " "
  delay(step_delay_time);
  // 2c Setup gyroscope now that the vehicle is "straightened"
  gyro_setup();
  */


  // 3 GET OFF LINE
  display_char("3");
  set_sensors(); // update sensor values
  while( stsL==1 || stsR==1 ){ // get off current line
```

```
  set_sensors(); // update sensor values
  left_wheel(l_spd);
  right_wheel(r_spd);
}
left_wheel(0);  // turns wheels off
right_wheel(0); // " "
delay(step_delay_time);


// 4 MOVE TO SECOND LINE
//Serial.println("Calibration: Moving to second line");
display_char("4");
unsigned long temp_t1 = millis();
while( stsL==0 && stsR==0 )  { // while on grey or while on first line still
  // Read sensor values
  set_sensors(); // update sensor values
  left_wheel(l_spd);
  right_wheel(r_spd);
  straight_PID(0, straight_Kp, straight_Ki, straight_Kd, "forward");
}
// Calculate timing parameters
t1 = millis() - temp_t1; // set line-to-line time
tn = t1 * dn/d1;         // set tile step time
t2 = t1 * d2/d1;         // set recentering time
//Serial.println("Done.");
left_wheel(0);  // turns wheels off
right_wheel(0); // " "
delay(step_delay_time);


// 5 RECENTER VEHICLE
display_char("5");
time = millis() - start_delay_time;
unsigned long temp_t2 = time;
while( time < temp_t2 + t2 ) { // loop until recenter time exceeds
  // Update time
  time = millis() - start_delay_time;
  // Course correction
  straight_PID(0, straight_Kp, straight_Ki, straight_Kd, "reverse");
  // Move vehicle
  left_wheel(l_spd_rev);  // move backwards to recenter
  right_wheel(r_spd_rev); // " "
}
// Pause
```

```
  left_wheel(0);  // turns wheels off
  right_wheel(0); // " "
  delay(step_delay_time);


  // Update position (2 steps ahead at end of calibration)
  if(dir == "N") {
    pos[1] += 1;
  }
  else if(dir == "E"){
    pos[0] += 1;
  }
  else if(dir == "S"){
    pos[1] -= 1;
  }
  else if(dir == "W"){
    pos[0] -= 1;
  }

}



void move_forward(int n, unsigned long curr_time) {
  // Move vehicle forward n squares. Works in collaboration with calibration
function (see above).
  // Input: number of tiles (n) and current time of program (curr_time)

  // MOVE FORWARD
  // Get initial state data
  read_gyro(); // update yaw angle
  int ang_desired;
  if( dir == "N" ) { // vehicle starts EAST
    ang_desired = 270;
  }
  else if( dir == "E" ) {
    ang_desired = 0;
  }
  else if( dir == "S" ) {
    ang_desired = 90;
  }
  else if( dir == "W" ) {
    ang_desired = 180;
  }
```

```
  // Loop until time limit is reached
  time = millis() - start_delay_time;
  unsigned long temp_t1 = time;
  while(time < temp_t1 + tn*n ) { // move for set duration
    // Update time
    time = millis() - start_delay_time;

    // 2b Course Correction (PID)
    straight_PID(ang_desired, straight_Kp, straight_Ki, straight_Kd,
"forward");

    /*
    // 2c Recalibration
    set_sensors(); // update sensors
    if( stsL == 1 || stsR == 1) { // line has been reached
      if(tile_recalibrate) {
        time = millis() - start_delay_time; // current time
        tc = time - temp_t1;                // set reclibration value
        tn = tc * dn/dc;                    // update tile-to-tile time based
on updated speed
        tile_recalibrate = false;           // prevent constant recalibration
      }
    }
    */

    // 2d Move Wheels
    left_wheel(l_spd);
    right_wheel(r_spd);
  }


  // Update position
  if(dir == "N") {
    pos[1] += 1;
  }
  else if(dir == "E"){
    pos[0] += 1;
  }
  else if(dir == "S"){
    pos[1] -= 1;
  }
  else if(dir == "W"){
    pos[0] -= 1;
```

```
}

/*
// Pause
left_wheel(0);
right_wheel(0);
delay(step_delay_time);
*/

// Reset calibration variable for next cycle
tile_recalibrate = true;

// Search for buildings around square
left_wheel(0);  // turn off wheels while searching
right_wheel(0); // " "
time = millis() - start_delay_time; // update time
unsigned long dtc_time_end = time + step_delay_time;
while(time < dtc_time_end){ // pause for step_delay_time duration
  // Update time
  time = millis() - start_delay_time; // update time
  // Update sensor values
  for(int i=0; i<3; i++){
    building_dtc[i] = fire_detection(i);
    cartography(); // update grid based on findings
  }
  // Turn on LEDs if signal is detected
  if(building_dtc[0]!=0 ||building_dtc[1]!=0 || building_dtc[2]!=0 ) {
    digitalWrite(fireLED, HIGH);
  } else {
    digitalWrite(fireLED, LOW);
  }
}

/*
// Ladder control
if(building_dtc[1] == 2){
  ladder.write(ladder_pos_1); // set to load position
  display_char("F");          // manually set fire signal
  delay(750);                 // wait set time period
  //building_dtc[1] = 0;        // manually turn front sensor off
  ladder.write(ladder_pos_0); // return to default position
} else {
  ladder.write(ladder_pos_0); // return to default position
}
```

```
  */

}



void turn_angle(int input_ang){
  // Turn the vehicle in place a specified angle.
  // Input: 90 = counter-clockwise quarter-turn (left), 270 = clockwise
quarter-turn (right), 180 = U-turn, X = counter-clockwise turn of X degrees

  // Recenter vehicle -> move axles to center of tile
  float temp_t_rc = tn * d_wc / dn;
  left_wheel(l_spd);
  right_wheel(r_spd);
  delay(temp_t_rc);

  // Get angle values
  float ideal_ang; // ideal angle of the vehicle (0, 90, 180, or 270 degrees)
  if( dir == "N" ) { // vehicle starts EAST
    ideal_ang = 270;
  }
  else if( dir == "E" ) {
    ideal_ang = 0;
  }
  else if( dir == "S" ) {
    ideal_ang = 90;
  }
  else if( dir == "W" ) {
    ideal_ang = 180;
  }
  float desired_ang = ideal_ang + input_ang; // get new angle direction
(direction to turn towards)
  if(desired_ang > 360) {              // make sure target is in same range as
readings
    desired_ang -= 360;
  }

  // Get current wheel speeds
  float temp_l_spd = l_spd;
  float temp_r_spd = r_spd;
  float turn_spd_pct = 0.9; // turn slowly to prevent error propogation

  // Set turn direction
```

```
  if( input_ang > 180 ) { // turn right (clockwise)
    l_spd = temp_l_spd * turn_spd_pct;
    r_spd = -temp_r_spd * turn_spd_pct;
  } else {                // turn left (counter-clockwise)
    l_spd = -temp_l_spd * turn_spd_pct;
    r_spd = temp_r_spd * turn_spd_pct;
  }

  // Turn vehicle
  while( abs(ang-desired_ang) > ang_error ) { // keep turning until vehicle is
within acceptable range of desired angle
    // Move wheels
    left_wheel(l_spd);
    right_wheel(r_spd);
    // Update yaw
    read_gyro();
  }

  // Pause
  left_wheel(0);
  right_wheel(0);
  delay(step_delay_time);

  // Revert wheel speeds
  l_spd = temp_l_spd; // use speeds before turn
  r_spd = temp_r_spd; // " "

  // Recenter vehicle -> move axles back to original position
  left_wheel(l_spd_rev);
  right_wheel(r_spd_rev);
  delay(temp_t_rc);

  // Pause
  left_wheel(0);
  right_wheel(0);
  delay(step_delay_time);

  // Update position
  if( dir == "N" ) {
    if( input_ang == 90 ) { dir = "E"; }
    else if( input_ang == 270 ) { dir = "W"; }
    else if( input_ang == 180 ) { dir = "S"; }
  }
  else if( dir == "E" ) {
```

```
    if( input_ang == 90 ) { dir = "S"; }
    else if( input_ang == 270 ) { dir = "N"; }
    else if( input_ang == 180 ) { dir = "W"; }
  }
  else if( dir == "S" ) {
    if( input_ang == 90 ) { dir = "W"; }
    else if( input_ang == 270 ) { dir = "E"; }
    else if( input_ang == 180 ) { dir = "N"; }
  }
  else if( dir == "W" ) {
    if( input_ang == 90 ) { dir = "N"; }
    else if( input_ang == 270 ) { dir = "S"; }
    else if( input_ang == 180 ) { dir = "E"; }
  }

}



//
================================================================================
================================================================
// DEFINE GLOBAL FUNCTIONS - Part 4 (Building detection)
//
================================================================================
================================================================



int fire_detection(int i) {
  // Reads IR sensor signals and determines whether a building or a fire is
detected.
  // Input: sensor to read values from -> 0 = left, 1 = front, 2 = right
  // Output: detection type -> 0 = no detection, 1 = building (blank face), 2 =
fire (window face)

  // Update time
  time = millis() - start_delay_time;


  // Read sensor value
  prox_sns[i] = (digitalRead(IR_RECEIVE_PIN[i]) == LOW); // true = signal
detected, false = no detection (detects around 4cm away)
```

```
  // Determine whether timer has expired
  fire_timer_exp[i] = (time > fire_dtc_time[i] + fire_dtc_time_max); // true =
timer has expired, false = timer is still going

  if(prox_sns[i] && !prev_prox_sns[i] && fire_timer_exp[i]) { // signal is
detected for first time (timer has not expired)
    fire_dtc_time[i] = time;        // set timer value
  }

  if(!fire_timer_exp[i]) { // timer has not yet expired
    prev_prox_sns[i] = prox_sns[i]; // update previous value
    if(!prox_sns[i]) {
      return 2; // signal is lost during timer, fire has been detected
    }
    return 1; // assume blank face of building until fire is detected
  } else{
    prev_prox_sns[i] = prox_sns[i]; // update previous value
    return 0; // timer has expired
  }

  // Default output
  prev_prox_sns[i] = prox_sns[i]; // update previous value
  return 0;                       // assume no signal detected otherwise

}



void cartography() {
  // Updates grid matrix based on readings from IR sensors.

  if(dir == "N") { // North heading
    if(fire_detection(0)==1 || fire_detection(0)==2){ // left sensor
      grid[pos[1]][pos[0]-1] = 0; // add building
    }
    if(fire_detection(1)==1 || fire_detection(1)==2){ // front sensor
      grid[pos[1]+1][pos[0]] = 0; // " "
    }
    if(fire_detection(2)==1 || fire_detection(2)==2){ // right sensor
      grid[pos[1]][pos[0]+1] = 0; // " "
    }
  }
  else if(dir == "E") { // East heading
    if(fire_detection(0)==1 || fire_detection(0)==2){ // left sensor
```

```
      grid[pos[1]+1][pos[0]] = 0; // add building
    }
    if(fire_detection(1)==1 || fire_detection(1)==2){ // front sensor
      grid[pos[1]][pos[0]+1] = 0; // " "
    }
    if(fire_detection(2)==1 || fire_detection(2)==2){ // right sensor
      grid[pos[1]-1][pos[0]] = 0; // " "
    }
  }
  else if(dir == "S") { // South heading
    if(fire_detection(0)==1 || fire_detection(0)==2){ // left sensor
      grid[pos[1]][pos[0]+1] = 0; // add building
    }
    if(fire_detection(1)==1 || fire_detection(1)==2){ // front sensor
      grid[pos[1]-1][pos[0]] = 0; // " "
    }
    if(fire_detection(2)==1 || fire_detection(2)==2){ // right sensor
      grid[pos[1]][pos[0]-1] = 0; // " "
    }
  }
  else if(dir == "W") { // West heading
    if(fire_detection(0)==1 || fire_detection(0)==2){ // left sensor
      grid[pos[1]-1][pos[0]] = 0; // add building
    }
    if(fire_detection(1)==1 || fire_detection(1)==2){ // front sensor
      grid[pos[1]][pos[0]-1] = 0; // " "
    }
    if(fire_detection(2)==1 || fire_detection(2)==2){ // right sensor
      grid[pos[1]+1][pos[0]] = 0; // " "
    }
  }
}



//
//////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////
// DEFINE GLOBAL FUNCTIONS - Part 5 (Path finding)
//
//////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////
```

```
int grid_to_index(int x, int y) {
  // Converts grid coordinate into a node name (used for A* algorithm).
  return x+y*COL;
}

int temp_pos[2]; // temporary coordinate (x,y) for function output
void index_to_grid(int i) {
  // Converts index position to grid coordinates (x,y)
  // Note: "temp_pos" variable is used instead of a function output.
  temp_pos[0] = i%COL;         // update temp variable (x value)
  temp_pos[1] = (i-i%COL)/COL; // " " (y value)
}



void print_grid(int arr[ROW][COL], int r, int c, String type){
  // Print an array of values as text characters in the Serial for debugging
purposes.
  // Input: 2D array "arr", number of rows, number of columns, type [Grid,
Cost, Path]

  // print header
  Serial.println();
  Serial.println(type + ": ");

  // loop over each element of 2D array
  //for(int i=r-1; i>=0; i--){ // print from top to bottom -> reverse order!!!
  for(int i=0; i<r; i++){ // print from bottom to top
    for(int j=0; j<c; j++){
      int a = arr[i][j]; // value to print
      if ( type == "Grid" ) {
        if( a == 0 ) { Serial.print("X "); } // convert integers to grid
information
        if( a == 1 ) { Serial.print("0 "); } // " "
      }
      if ( type == "Cost" ) {
        if(a < 10) { Serial.print("00"); }        // print leading zeroes
(format as 3 digit number)
        if(a>=10 && a<100) { Serial.print("0"); } // " "
        Serial.print(a + String(" "));            // display number
      }
      if ( type == "Path" ) {
        ; // print path information ?
```

```
      }
    }
    Serial.println();
  }

}



String get_directions(int p1, int p2, String d) {
  // Instructions for moving vehicle, used during path reconstruction of A*
algorithm.
  // Input: first position index (p1), second position index (p2), and
direction at first position (d)
  // Output: first character = new direction heading [N = north, S = south, E =
east, W = west], second character = turn direction [R = right, L = left, U =
turn around, S = straight/no-turn]
  // Notes: forward movement is implied after each instruction

  String s = ""; // string to output

  // get coordinate values
  index_to_grid(p1);                          // p1 = (x1, y1)
  int x1 = temp_pos[0]; int y1 = temp_pos[1]; // " "
  index_to_grid(p2);                          // p2 = (x2, y2)
  int x2 = temp_pos[0]; int y2 = temp_pos[1]; // " "

  // Check all starting directions
  // North
  if( d == "N" ) { // Towards user (increasing y)
    if( x1 == x2 ) {
      if( y1 < y2 ) { s = "NS";  }
      if( y1 > y2 ) { s = "SU";  }
    }
    else if( x1 < x2 ) { s = "EL"; }
    else if( x1 > x2) {  s = "WR";  }
  }
  // East
  else if( d == "E" ) { // Right (increasing x)
    if( y1 == y2 ) {
      if( x1 < x2 ) { s = "ES"; }
      if( x1 > x2 ) { s = "WU"; }
    }
    else if( y1 < y2 ) { s = "NR"; }
```

```
    else if( y1 > y2 ) { s = "SL"; }
  }
  // South
  else if( d == "S" ) { // Away from user (decreasing y) -> starting direction
    if( x1 == x2 ) {
      if( y1 < y2 ) { s = "NU";  }
      if( y1 > y2 ) { s = "SS";  }
    }
    else if( x1 < x2 ) { s = "ER"; }
    else if( x1 > x2) {  s = "WL"; }
  }
  // West
  else if( d == "W" ) { // Left (decreasing y)
    if( y1 == y2 ) {
      if( x1 < x2 ) { s = "EU"; }
      if( x1 > x2 ) { s = "WS"; }
    }
    else if( y1 < y2 ) { s = "NL"; }
    else if( y1 > y2 ) { s = "SR"; }
  }

  return s; // default value to return, throws an error in system

}



String aStar(int startx, int starty, int endx, int endy, String dir){
  // Performs A* search algorithm using the current grid.
  // Input: grid of values (0 = wall, 1 = free space), start position (x, y),
end position, and starting direction
  // Output: list of instructions on how to move vehicle -> [F = forward 1
tile, R = turn 90deg CW, L = turn 90 deg CCW]
  // See reference [1]
https://medium.com/@nicholas.w.swift/easy-a-star-pathfinding-7e6689c7f7b2

  // === 1 ===
  // Initialize A* matrices
  for(int i=0; i<ROW; i++){
    for(int j=0; j<COL; j++){
      h[i][j] = abs(j - endx) + abs(i - endy); // compute manhattan heuristic
      if( i == starty && j == startx ) { // starting node
        g[i][j] = 0;       // set path distance as 0 for starting node
        f[i][j] = h[i][j]; // starting cost function is just heuristic value
```

```
      } else { // all other nodes
        g[i][j] = 999;     // assign path matrix as "infinite" to start
        f[i][j] = 999;     // assign cost function also as infinite
      }
    }
  }

  // === 2 ===
  // Initialize variables used
  // positioning
  int x, y;                 // position of current node checked
  // parent references
  int parents[ROW*COL];     // parents[i] = k -> node i has a parent at node k
  for(int i=0; i<ROW*COL; i++) {
    parents[i] = -1; // set as -1 to start
  }
  // open and closed lists
  int open_list_size = 0;   // size of lists, used for adding nodes to search
query
  int closed_list_size = 0; // " "
  // initialize open and closed lists (whether node indices are in list or not)
  bool open_list[ROW*COL];   // value at index = 1 if node index is in list,
elsewise it is 0
  bool closed_list[ROW*COL]; // " "
  // set all values as false to start (lists are both empty)
  for(int i=0; i<ROW*COL; i++) {
    open_list[i] = false;
    closed_list[i] = false;
  }

  // === 3 ===
  // Add starting node to open list
  open_list[grid_to_index(startx, starty)] = true; // set starting position
node to be in open list
  open_list_size += 1;


  // === 4 ===
  // Perform search algoritm
  while( open_list_size > 0 ) { // loop through nodes until all nodes have been
visited and expanded
    // === 5 ===
    // Get the current node (node with minimum f value from priority queue)
    int temp_min_f = 999; // minimum f value
```

```
    int temp_min_node;    // node index of above minimum
    int temp_h_at_min;    // h value of above minimum (used for tiebreakers)
    for(int i=0; i<ROW*COL; i++) {
      if( open_list[i] ) { // only look at nodes in open list
        index_to_grid(i);                          // get position of node in
list
        int xa = temp_pos[0]; int ya = temp_pos[1]; // " " (store as temporary
variables)
        int f_val = f[ya][xa]; // score of current node to check (prevents
repeated array calls)
        int h_val = h[ya][xa]; // " "
        if( f_val < temp_min_f ) { // choose value of lowest f score
          temp_min_f = f_val;    // update minimum
          temp_h_at_min = h_val; // " ""
          temp_min_node = i;     // update node
        }
        else if ( f_val == temp_min_f && h_val < temp_h_at_min ) { // break
ties if present
          temp_min_f = f_val;    // update minimum
          temp_h_at_min = h_val; // " ""
          temp_min_node = i;     // update node
        }
      }
    }
    // save current cell position
    index_to_grid(temp_min_node);     // get position of current node
    x = temp_pos[0]; y = temp_pos[1]; // " "

    // === 6 ===
    // Move current node from OPEN list to CLOSED list
    open_list[temp_min_node] = false;  open_list_size -= 1;   // remove from
open list
    closed_list[temp_min_node] = true; closed_list_size += 1; // add to closed
list

    //Serial.println("Current node: " + String(temp_min_node)); // print
current progress (debugging)

    // === 7 ===
    // End condition (target reached)
    if( x == endx && y == endy ) { // current node = end node
      // === 7a ===
      // Set up reconstruction algorithm
      int start_id = grid_to_index(startx, starty); // retrieve start/end node
```

```
      int end_id = grid_to_index(endx, endy);       // " "
      int path[ROW*COL];                            // return path (list of
indices)
      for(int i=0; i<ROW*COL; i++) { path[i]=-1; }  // initialize to be -1
(default value)
      int i = end_id; // current node in path -> start at end to work backwards
      int j = 1;      // counter variable

      // === 7b ===
      // Reconstruct path based on parent node information
      path[0] = end_id; // first index should be end value
      while( i != start_id ) {
        path[j] = parents[i]; // path follows heirarchy of parent nodes
        i = parents[i];       // new i = parent node of former i
        j += 1;               // increase counter
      }
      // print path (for debugging)
      for(int k=j-1; k>=0; k--){
        //Serial.print(String(path[k]) + ", ");
      }

      // === 7c ===
      // Go backwards through path to get instructions
      String guide = ""; // directions for vehicle to follow -> F = forward, R
= 90deg CW turn, L = 90 degree CCW turn
      String d = dir;       // current heading of vehicle in path -> start in
indicated start direction
      for(int k=j-1; k>=1; k--) { // go backwards through path starting at
first node
        String dir_output = get_directions(path[k], path[k-1], d); // output of
directions function -> use current position, next position, and current
direction to get instructions of travel
        d = dir_output[0];      // first character is new direction heading ->
update heading
        guide += dir_output[1]; // second character is turn data (i.e., R
(right), L (left), or S (straight))
        guide += "F";           // forward direction is implied with each step
      }
      //Serial.println(guide); // print guide information (for debugging)

      return guide; // return list of instructions
    }

    // === 8a ===
```

```
    // Generate children of current node
    int children[4]; // list of node indices that neighbor current node
    // North
    int ci = grid_to_index(x, y+1);                       // child index (temp
variable)
    if( grid[y+1][x] == 1 && closed_list[ci] == false ) { // check if child
node is not a barrier AND it hasnt be explored before
      children[0] = ci;
    } else {
      children[0] = -1;                                   // use default value
otherwise (skips child)
    }
    // South
    ci = grid_to_index(x, y-1);                           // see above
    if( grid[y-1][x] == 1 && closed_list[ci] == false ) { // " "
      children[1] = ci;
    } else {
      children[1] = -1;                                   // " "
    }
    // East
    ci = grid_to_index(x+1, y);                           // see above
    if( grid[y][x+1] == 1 && closed_list[ci] == false ) { // " "
      children[2] = ci;
    } else {
      children[2] = -1;                                   // " "
    }
    // West
    ci = grid_to_index(x-1, y);                           // see above
    if( grid[y][x-1] == 1 && closed_list[ci] == false ) { // " "
      children[3] = ci;
    } else {
      children[3] = -1;                                   // " "
    }

    // === 8b ===
    // set children as successors of parents
    for(int i=0; i<4; i++) {
      int n = children[i]; // child node
      if( n != -1) {
        parents[n] = temp_min_node; // set parent node
      }
    }

    /*
```

```
    // print current progress (debugging)
    String c_print = ""; // print children for
    c_print += "Parent: " + String(temp_min_node);
    c_print += ", Children: [";
    c_print += String(children[0]) + ", " + String(children[1]) + ", " +
String(children[2]) + ", " + String(children[3]);
    c_print += "]";
    Serial.println(c_print);
    */

    // === 9 ===
    // Check each child cell
    for(int k=0; k<4; k++) {
      if( children[k] > -1) { // make sure child node exists/is valid

        // === 10a ===
        // Check if child is in the CLOSED list
        if( closed_list[children[k]] == true ) { // child is in CLOSED list
          continue; // go to next child
        }

        // === 10b ===
        // Get child position and A* matrix values
        index_to_grid(children[k]);                 // get position of child
node
        int xc = temp_pos[0]; int yc = temp_pos[1]; // " "
        g[yc][xc] = g[y][x] + 1;           // g value of child node (g child =
g parent + 1)
        f[yc][xc] = g[yc][xc] + h[yc][xc]; // f = g + h

        // === 10c ===
        // Check if child is in the OPEN list
        if( open_list[children[k]] == true ) { // child is in OPEN list
          if( g[yc][xc] > g[y][x] ) {
            continue; // go to next child
          }
        }

        // === 10d ===
        // Put child in open list
        open_list[children[k]] = true; open_list_size += 1; // add to open list

      }
    }
```

```
  }

  if( x != endx || y != endy ) { // failed to reach end, throw error
    Serial.print("Could not reach goal!");
  }

}
```

Code F2. Remote control functionality of the TRIDENT vehicle. (Top) Code uploaded to the vehicle. (Bottom) Code uploaded to the Arduino at the base station/remote controller.

```
C/C++
// TITLE: Remote Controlled Functionality
// AUTHOR: Andrew Smith, Noah Etienne, Dylan Taylor, and Venkata Sai Phanindra
Mandadapu
// UPDATED: May 16th, 2024
//
// VERSION HISTORY:
// - (5/11) Initial version, forward march program and turning protocol
// - (5/13) Added fire detection function for IR sensors
// - (5/13) Improved forward motion functionality
// - (5/16) Replaced Servo motor for ladder control
//
// NOTES:
// -



//
================================================================================
================================================================
// INITIALIZE PARAMETERS (Define all constants and variables used in code)
//
================================================================================
================================================================



// System variables
unsigned long time;                // current system time in ms
const int start_delay_time = 3000; // delay time before motor movement (in ms)
const int step_delay_time = 1000;  // time to wait after moving one tile
forward (in ms)

// Define wheel motors
#include <Servo.h>
Servo leftMotor;
Servo rightMotor;

// Define LED pins
const int dsp_LEDs[] = {47, 45, 43, 41, 39, 37, 35}; // connection LEDs for all
segments of display
```

```
const int onLED = 33;                                // "ON" indicator -> dot
on 7-segment display
const int rmtLED = onLED;                            // remote control
functionality indicator (same on ON LED except it blinks)
const int fireLED = 46;                              // LED to sync with the
fire IR emission


// Define sensors
int snsL, snsR;              // raw QTI sensor reading values
int stsL, stsR;              // readings of QTI sensors -> 0 = grey, 1 = line
int prev_stsL, prev_stsR;    // previous readings " "
const int left_QTI_pin =  3; // left QTI sensor connection pin
const int right_QTI_pin = 2; // right " "

// Line detection settings
int num_lines = 0;                  // number of lines passed over last
const int threshold[2] = {105, 95}; // light threshold to use for line
detection -> one for each sensor {L, R}
int line_dtc_time;                  // amount of time to collect all line
detection signals (in ms) -> scales with vehicle speed
const int line_dtc_time_max = 1000; // " " MAXIMUM value -> based on maximum
speed of both wheels
unsigned long LED_start_time;       // starting time for LED display
const int LED_on_time = 600;        // amount of time to turn LED indicator
lights on (in ms)
bool oob = false;                   // whether vehicle has reached
out-of-bounds or not
unsigned long oob_start_time;       // starting time for out-of-bounds line
detection
int oob_dtc_time;                   // amount of time for continuous line
detection to indicate out-of-bounds line (in ms)
const int oob_dtc_time_max = 350;   // " " MAXIMUM value -> based on maximum
speed of both wheels


// Motion control parameters
const int motor_pin_L = 30;          // attachment pin of motor
const int motor_pin_R = 31;          // " "
float r_spd, l_spd;                  // speed of each wheel [-1, 1] -> % of max
speed
const float r_spd_initial =  0.185;  // set initial wheel speeds ("straight"
value)
const float l_spd_initial =  0.170;  // " "
```

```
float oob_spd_threshold = 0.07;      // threshold to prevent moving out of
bounds
float max_spd = 0.2;                 // maximum wheel speed, used to throttle
control

// Set up remote control
#include <SPI.h>
#include "RF24.h"
#define CE_PIN 7
#define CSN_PIN 8
RF24 radio(CE_PIN, CSN_PIN);
uint8_t address[][6] = { "1Node", "2Node" };
bool radioNumber = 1;  // 0 uses address[0] to transmit, 1 uses address[1] to
transmit
bool role = false;     // true = TX role, false = RX role
int payload[2];
int i;
float p0;
float p1;

// Remote control parameters
const int rmt_btn_pin = 48;       // pin for remote control button
bool rmt_on = false;              // whether or not remote control
functionality is on
const float rmt_blnk_freq = 1.0;  // frequency to blink "on" LED when remote is
on (in Hz)
const int rmt_btn_threshold = 10; // any signal below this value will count as
the button being pressed in


// Define ladder servo
#include <Servo.h>
Servo ladder;
const int ladder_pin = 49;    // pin for ladder servo motor
const int ladder_pos_0 = 180; // default position (upright)
const int ladder_pos_1 = 100; // engaged position (horizontal)
bool ladder_down = false;     // whether to engage ladder mechanism

// Set up IR detection sensor
#include <IRremote.hpp>
const int IR_RECEIVE_PIN[3] = {42,24,40};    // IR sensor connection pins
(fire/building detection) -> 0 = left, 1 = front, 2 = right
bool prox_sns[3] = {false,false,false};      // reading from each IR sensor
bool prev_prox_sns[3] = {false,false,false}; // previous readings " "
```

```
// Fire detection variables
const int fire_dtc_time_max = 1000;
// amount of time to wait before determining whether (1) building is on fire or
(2) building is near
long fire_dtc_time[3] = {-fire_dtc_time_max, -fire_dtc_time_max,
-fire_dtc_time_max}; // time of first signal detection of each IR sensor -> set
as high value initially
bool fire_timer_exp[3] = {true, true, true};
// whether fire detection timer has expired (for each sensor)
int building_dtc[3] = {0,0,0};
// values read from fire detection function -> 0 = no signal, 1 = building
(blank face), 2 = fire detected


// Forward Step Calibration
const float dv = 0.001; // step size for changing wheel speed during
re-alignment
bool tile_calibrate = true;   // setting to calibrate distance measurement
bool tile_recalibrate = true; // " "
bool done_move = false;     // continue cycle until process is complete
unsigned long t0; // time of first line detection
unsigned long t1; // time to move vehicle to center of next tile (after first
line detection)
unsigned long tn; // time to move 1 tile
unsigned long tc; // time to move vehicle to next line (used for recalibration)
const float d0 = 14.5; // distance from QTI to first line at start (cm)
const float d1 = 24.0; // distance to move vehicle from first line to center of
next tile (cm)
const float dn = 30.5; // distance from line-to-line (tile dimension, cm)
const float dc = 6.3;  // distance from sensor at tile center to line (used for
recalibration)

// Yaw Turning Variables
const float w_spc = 14.4; // distance between wheel centers (cm)
const float d_wc = 5.0;   // distance from wheel axle to center of vehicle body
(cm)
const float half_pi = 1.5707963267948; // value of PI/2 (used for timing
calculations)


// A* Path Finding Parameters
// Initialize default grid
const int ROW = 10; // size of grid (8x8 grid with a 1 tile border)
```

```
const int COL = 10; // " "
int grid[ROW][COL] =
  { { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
              { 0, 1, 1, 1, 1, 1, 1, 1, 1, 0 },
              { 0, 1, 1, 1, 1, 1, 1, 1, 1, 0 },
              { 0, 1, 1, 1, 1, 0, 1, 1, 1, 0 },
              { 0, 1, 1, 1, 1, 1, 0, 1, 1, 0 },
              { 0, 1, 1, 1, 1, 1, 1, 1, 1, 0 },
              { 0, 1, 1, 1, 1, 1, 1, 1, 1, 0 },
              { 0, 1, 1, 1, 1, 1, 1, 1, 1, 0 },
    { 0, 1, 1, 1, 1, 1, 1, 1, 1, 0 },
              { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 }
};
/* Grid above is numbered ***upside down***:
            South
      [0,0] [1,0] [2,0] ...
West  [0,1] [1,1] [2,1] ...  East
      [0,2] [1,2] [2,2] ...
       ...   ...   ...  ...
            North
0 = building/out-of-bounds, 1 = free tile
*/
// Initial position
int pos[2] = {2,8}; // enter starting position (x,y) using (0,9) for South-West
corner
int end[2] = {6,3}; // end (x,y) coordinate " " -> TEMP used for verifying path
finding works
String dir = "S";   // enter vehicle starting direction -> N = North (+y), E =
East (+x), S = South (-y), W = West (-x)
// Define A* computation matrices
int h[ROW][COL]; // heuristic matrix -> uses manhattan distance from grid point
to end target
int g[ROW][COL]; // path matrix      -> cost of the path from start point to
grid point
int f[ROW][COL]; // cost matrix      -> sum of path and heuristic matrix values


// Define Search Path
int path_index = 0; // current step of path to take
String path = "";   // instructions of path to follow (e.g., "SFSFRFSFRF")
//int s1[2] = {6,2};
//int PATH[15][2] = {pos,s1,s2,s3,s4,s5,s6,s7,s8,s9,s10,s11,s12,s13,s14};
```

```
//
================================================================================
================================================================
// SETUP FUNCTION (Performs once at beginning of code)
//
================================================================================
================================================================



void setup() {
  Serial.begin(9600);

  // Declare LED pins as outputs
  pinMode(onLED, OUTPUT);
  pinMode(rmtLED, OUTPUT);
  pinMode(fireLED, OUTPUT);
  for(i=0; i<8; i++){
    pinMode(dsp_LEDs[i], OUTPUT); // set all display pins as outputs
  }

  // Set motor attachments
  leftMotor.attach(motor_pin_L);  // left rear wheel
  rightMotor.attach(motor_pin_R); // right " "

  // Initiate speed control
  r_spd = r_spd_initial; // default motor speed
  l_spd = l_spd_initial; // " "

  radio_setup();

  // Initiate ladder
  ladder.attach(ladder_pin);  // attach servo motor for ladder
  ladder.write(ladder_pos_0); // default position

  // Initiate QTI sensor readings
  prev_stsL = 0;
  prev_stsR = 0;

  // Initiate IR sensors
  IrReceiver.begin(IR_RECEIVE_PIN[0], ENABLE_LED_FEEDBACK);
  IrReceiver.begin(IR_RECEIVE_PIN[1], ENABLE_LED_FEEDBACK);
  IrReceiver.begin(IR_RECEIVE_PIN[2], ENABLE_LED_FEEDBACK);
```

```
  // Delay at beginning
  display_char("-"); // loading display
  delay(start_delay_time); // wait 5s before beginning program
}



//
================================================================================
================================================================
// MAIN LOOP (Repeats each timestep)
//
================================================================================
================================================================



void loop() {

  // SYSTEM VARIABLES
  if( !rmt_on ) { // turn on "ON" LED (constant = on, blinking = remote on)
    digitalWrite(onLED, HIGH);
  }
  time = millis() - start_delay_time; // update current time


  // HARDWARE INPUT
  set_sensors();   // set sensor values
  toggle_remote(); // toggle remote control functionality


  // RADIO TRANSMISSION
  radio_receive();
  if(rmt_on) {
    l_spd = -1*( 2*max_spd*(payload[0])/1023.0 - max_spd );
    r_spd = 2*max_spd*payload[1]/1023.0 - max_spd;
  }
  else {
    r_spd = 0; // stop wheels when no signal is detected
    l_spd = 0; // " "
  }

  // MOVE VEHICLE
```

```
  if(rmt_on){
    left_wheel(l_spd);
    right_wheel(r_spd);
  } else{
    left_wheel(l_spd);
    right_wheel(r_spd);
  }


  // BUILDING DETECTION
  // Determine detection type for each sensor
  for(int i=0; i<3; i++){
    building_dtc[i] = fire_detection(i);
  }
  // Turn on LED if any IR signal is detected
  set_sensors(); // update sensor values
  if(prox_sns[0] || prox_sns[1] || prox_sns[2]) { // any sensor detects signal
    digitalWrite(fireLED, HIGH);
  } else {
    digitalWrite(fireLED, LOW);
  }

  // LADDER CONTROL
  if(building_dtc[1] == 2) {
    ladder.write(ladder_pos_1); // set to load position
    display_char("F");
    delay(750); // wait for signal to end
  } else {
    display_char(" ");
    ladder.write(ladder_pos_0); // return to default position
  }


  // UPDATE MAP
  //cartography(); // continuously add buildings to grid as they are found
  //print_grid(grid, ROW, COL, "Grid");
  //Serial.println("");


  // DISPLAY DEBUG INFORMATION
  debug("null"); // print debug information -> [sensor, status, line, remote,
fire, calibration, position, null]
}
```

```
//
================================================================================
============================================================
// DEFINE GLOBAL FUNCTIONS - Part 1 (Basic vehicle inputs and outputs)
//
================================================================================
============================================================


long RCTime(int sensorIn){
  // Read QTI sensor at pin and output sensor signal strength
  long duration = 0;
  pinMode(sensorIn, OUTPUT);     // Make pin OUTPUT
  digitalWrite(sensorIn, HIGH);  // Pin HIGH (discharge capacitor)
  delay(1);                      // Wait 1ms
  pinMode(sensorIn, INPUT);      // Make pin INPUT
  digitalWrite(sensorIn, LOW);   // Turn off internal pullups
  while(digitalRead(sensorIn)){  // Wait for pin to go LOW
    duration++;
  }
  return duration;
}



int left_wheel(float spd) {
  // Set motor rotation rate based on input motor speed
  int spd_us = int( spd * 500.0 + 1500.0 ); // map microsecond PWS input for
motor
  leftMotor.writeMicroseconds(spd_us);      // move motor set amount
}
int right_wheel(float spd) {
  // See above
  int spd_us = int( -1*spd * 500.0 + 1500.0 ); // map microsecond PWS input for
motor -> use positive values for forward movement
  rightMotor.writeMicroseconds(spd_us);        // move motor set amount
}



void set_sensors() {
  // Read QTI sensors and determine whether line is detected
```

```
  snsL = RCTime(left_QTI_pin);  // left QTI sensor value
  snsR = RCTime(right_QTI_pin); // right " "
  if( snsL < threshold[0] ) { stsL = 1; } else { stsL = 0; } // left line
detection status -> 1 = line, 0 = grey
  if( snsR < threshold[1] ) { stsR = 1; } else { stsR = 0; } // " "
}



void get_timing() {
  // Calculate the timing variables based on the speed of the right sensor.
  float w = 14.8;                    // width between wheels (in cm)
  float d = 1.5;                     // distance between right sensor and
right wheel
  float r = d / w;                   // ratio of distances, used for linear
scaling
  float wq = l_spd*r + r_spd*(1-r);   // effective speed at the sensor ->
ranges from [-1, 1] just like wheel speeds
  // Return scaled value
  if( wq <= oob_spd_threshold ) { // prevents divide by zero error
    wq = oob_spd_threshold;
  }
  float s = wq / max_spd; // ratio of sensor ground speed to maximum speed
  line_dtc_time = line_dtc_time_max / s; // linearly scale detection time ->
slower speeds = more detection time
  oob_dtc_time = oob_dtc_time_max / s;   // " "
}



bool curr_but = 1; // current button value -> initialize as OFF
bool prev_but = 1; // previous button value -> initialize as OFF
void toggle_remote() {
  // Turn remote control functionality on and off

  curr_but = digitalRead(rmt_btn_pin); // current button value

  if( curr_but == 0 && prev_but == 1 ) { // button is pressed
    if ( rmt_on == true ) { rmt_on = false; } // if on -> turn off
    else if ( rmt_on == false ) { rmt_on = true; } // if off -> turn on
  }

  // Blink "on" LED when remote is on
```

```
  float x = 2*(2*floorf(rmt_blnk_freq*time/1000) -
floorf(2*rmt_blnk_freq*time/1000) ) + 1;
  if( rmt_on ) {
    if( x > 0 ) {
      digitalWrite(rmtLED, HIGH);
    }
    else {
      digitalWrite(rmtLED, LOW);
   }
  }

  // Update previous value
  prev_but = curr_but; // update previous button value


  // Prevent boot-up signal errors
  if( time < 100 ) { // ignore signal in first 100ms
    rmt_on = false;
  }

}



unsigned long currentTime = 0;  // time at current line detection
int temp_num = 0;               // number of lines detected (temporary
variable)
bool reset_counter = false;     // used to reset temporary counter only once
int prev_sts = 0;               // previous line status (temporary variable)
int line_detection(int sts) {
  // Return the number of lines detected from the input sensor status
  //    0 -> plain mat, no lines
  //    1 -> single line
  //    2 -> double line
  //   -1 -> out-of-bounds line

  // Set detection time
  if (sts == 1) { // line IS being detected
    if( prev_sts == 0 && time > currentTime + line_dtc_time ) {
      oob = false;
      temp_num = 0;               // reset counter if this is the first
detection in a while
    }
```

```
    currentTime = time;           // save time of any line detection encounter
  }
  else { // line is NOT being detected
    oob_start_time = time;        // if line detection stops we are no longer
at out-of-bounds line, so don't let timer run out
  }

  // Output I: out-of-bounds
  if( time > oob_start_time + oob_dtc_time ) { // line has been continously
detected for set amount of time
    LED_start_time = currentTime + line_dtc_time; // set starting time for LED
display as the end of this timer
    return -1; // out-of-bounds indicator value
  }

  // Output II: line count
  if( (time > currentTime + line_dtc_time) && !oob) { // time has passed for
line detection
      LED_start_time = currentTime + line_dtc_time; // set starting time for
LED display as the end of this timer
      return temp_num;                            // return number of lines
detected
  }
  else { // still within line detection timer
    if ( (sts - prev_sts) == 1 && !oob) { // sensor went from grey to line
      temp_num += 1;
    }
  }

  prev_sts = sts; // update previous value

}



void line_display(int num) {
  // Use indicator light to show the line detection state

  if ( time > (LED_start_time + LED_on_time) ) { // indicator time has elapsed
    display_char(" "); // turn off all LEDs, return to grey indicator
    num_lines = 0;     // reset number of lines variable
  }
  else {
    if ( num == 0 ) { // no line
```

```
      display_char(" "); // turn off all LEDs, return to grey indicator
    }
    else if ( num == 1 ) { // single line
      display_char("1"); // display "1"
    }
    else if ( num == 2 ) { // double line
      display_char("2"); // display "2"
    }
    else if ( num == -1) { // out-of-bounds
      oob = true;
      display_char("---"); // display 3 dashes
    }
  }

}



void display_char(String s){
  // Recieve input character and display that character on the 7-segment
display
  if( s == "0") { // not currently used
    digitalWrite(dsp_LEDs[0], HIGH);
    digitalWrite(dsp_LEDs[1], HIGH);
    digitalWrite(dsp_LEDs[2], HIGH);
    digitalWrite(dsp_LEDs[3], LOW);
    digitalWrite(dsp_LEDs[4], HIGH);
    digitalWrite(dsp_LEDs[5], HIGH);
    digitalWrite(dsp_LEDs[6], HIGH);
  }
  else if( s == "1") { // single line
    digitalWrite(dsp_LEDs[0], LOW);
    digitalWrite(dsp_LEDs[1], LOW);
    digitalWrite(dsp_LEDs[2], HIGH);
    digitalWrite(dsp_LEDs[3], LOW);
    digitalWrite(dsp_LEDs[4], LOW);
    digitalWrite(dsp_LEDs[5], LOW);
    digitalWrite(dsp_LEDs[6], HIGH);
  }
  else if( s == "2") { // double line
    digitalWrite(dsp_LEDs[0], LOW);
    digitalWrite(dsp_LEDs[1], HIGH);
    digitalWrite(dsp_LEDs[2], HIGH);
    digitalWrite(dsp_LEDs[3], HIGH);
```

```
  digitalWrite(dsp_LEDs[4], HIGH);
  digitalWrite(dsp_LEDs[5], HIGH);
  digitalWrite(dsp_LEDs[6], LOW);
}
else if( s == "3") { // misc. numerical value
  digitalWrite(dsp_LEDs[0], LOW);
  digitalWrite(dsp_LEDs[1], HIGH);
  digitalWrite(dsp_LEDs[2], HIGH);
  digitalWrite(dsp_LEDs[3], HIGH);
  digitalWrite(dsp_LEDs[4], LOW);
  digitalWrite(dsp_LEDs[5], HIGH);
  digitalWrite(dsp_LEDs[6], HIGH);
}
else if( s == "4") { // misc. numerical value
  digitalWrite(dsp_LEDs[0], HIGH);
  digitalWrite(dsp_LEDs[1], LOW);
  digitalWrite(dsp_LEDs[2], HIGH);
  digitalWrite(dsp_LEDs[3], HIGH);
  digitalWrite(dsp_LEDs[4], LOW);
  digitalWrite(dsp_LEDs[5], LOW);
  digitalWrite(dsp_LEDs[6], HIGH);
}
else if( s == "5") { // misc. numerical value
  digitalWrite(dsp_LEDs[0], HIGH);
  digitalWrite(dsp_LEDs[1], HIGH);
  digitalWrite(dsp_LEDs[2], LOW);
  digitalWrite(dsp_LEDs[3], HIGH);
  digitalWrite(dsp_LEDs[4], LOW);
  digitalWrite(dsp_LEDs[5], HIGH);
  digitalWrite(dsp_LEDs[6], HIGH);
}
else if( s == "6") { // misc. numerical value
  digitalWrite(dsp_LEDs[0], HIGH);
  digitalWrite(dsp_LEDs[1], HIGH);
  digitalWrite(dsp_LEDs[2], LOW);
  digitalWrite(dsp_LEDs[3], HIGH);
  digitalWrite(dsp_LEDs[4], HIGH);
  digitalWrite(dsp_LEDs[5], HIGH);
  digitalWrite(dsp_LEDs[6], HIGH);
}
else if( s == "7") { // misc. numerical value
  digitalWrite(dsp_LEDs[0], LOW);
  digitalWrite(dsp_LEDs[1], HIGH);
  digitalWrite(dsp_LEDs[2], HIGH);
```

```
  digitalWrite(dsp_LEDs[3], LOW);
  digitalWrite(dsp_LEDs[4], LOW);
  digitalWrite(dsp_LEDs[5], LOW);
  digitalWrite(dsp_LEDs[6], HIGH);
}
else if( s == "8") { // misc. numerical value
  digitalWrite(dsp_LEDs[0], HIGH);
  digitalWrite(dsp_LEDs[1], HIGH);
  digitalWrite(dsp_LEDs[2], HIGH);
  digitalWrite(dsp_LEDs[3], HIGH);
  digitalWrite(dsp_LEDs[4], HIGH);
  digitalWrite(dsp_LEDs[5], HIGH);
  digitalWrite(dsp_LEDs[6], HIGH);
}
else if( s == "9") { // misc. numerical value
  digitalWrite(dsp_LEDs[0], HIGH);
  digitalWrite(dsp_LEDs[1], HIGH);
  digitalWrite(dsp_LEDs[2], HIGH);
  digitalWrite(dsp_LEDs[3], HIGH);
  digitalWrite(dsp_LEDs[4], LOW);
  digitalWrite(dsp_LEDs[5], LOW);
  digitalWrite(dsp_LEDs[6], HIGH);
}
else if( s == "-") { // no line (grey/mat)
  digitalWrite(dsp_LEDs[0], LOW);
  digitalWrite(dsp_LEDs[1], LOW);
  digitalWrite(dsp_LEDs[2], LOW);
  digitalWrite(dsp_LEDs[3], HIGH);
  digitalWrite(dsp_LEDs[4], LOW);
  digitalWrite(dsp_LEDs[5], LOW);
  digitalWrite(dsp_LEDs[6], LOW);
}
else if( s == "XX") { // left and right both 0
  digitalWrite(dsp_LEDs[0], LOW);
  digitalWrite(dsp_LEDs[1], LOW);
  digitalWrite(dsp_LEDs[2], LOW);
  digitalWrite(dsp_LEDs[3], LOW);
  digitalWrite(dsp_LEDs[4], HIGH);
  digitalWrite(dsp_LEDs[5], LOW);
  digitalWrite(dsp_LEDs[6], HIGH);
}
else if( s == "LR") { // left and right both 1
  digitalWrite(dsp_LEDs[0], HIGH);
  digitalWrite(dsp_LEDs[1], LOW);
```

```
  digitalWrite(dsp_LEDs[2], HIGH);
  digitalWrite(dsp_LEDs[3], LOW);
  digitalWrite(dsp_LEDs[4], LOW);
  digitalWrite(dsp_LEDs[5], LOW);
  digitalWrite(dsp_LEDs[6], LOW);
}
else if( s == "LX") { // left is 1, right is 0
  digitalWrite(dsp_LEDs[0], HIGH);
  digitalWrite(dsp_LEDs[1], LOW);
  digitalWrite(dsp_LEDs[2], LOW);
  digitalWrite(dsp_LEDs[3], LOW);
  digitalWrite(dsp_LEDs[4], LOW);
  digitalWrite(dsp_LEDs[5], LOW);
  digitalWrite(dsp_LEDs[6], HIGH);
}
else if( s == "XR") { // right is 1, left is 0
  digitalWrite(dsp_LEDs[0], LOW);
  digitalWrite(dsp_LEDs[1], LOW);
  digitalWrite(dsp_LEDs[2], HIGH);
  digitalWrite(dsp_LEDs[3], LOW);
  digitalWrite(dsp_LEDs[4], HIGH);
  digitalWrite(dsp_LEDs[5], LOW);
  digitalWrite(dsp_LEDs[6], LOW);
}
else if( s == "F") { // F for fire
  digitalWrite(dsp_LEDs[0], HIGH);
  digitalWrite(dsp_LEDs[1], HIGH);
  digitalWrite(dsp_LEDs[2], LOW);
  digitalWrite(dsp_LEDs[3], HIGH);
  digitalWrite(dsp_LEDs[4], HIGH);
  digitalWrite(dsp_LEDs[5], LOW);
  digitalWrite(dsp_LEDs[6], LOW);
}
else if( s == "---") { // three dashed lines, out-of-bounds
  digitalWrite(dsp_LEDs[0], LOW);
  digitalWrite(dsp_LEDs[1], HIGH);
  digitalWrite(dsp_LEDs[2], LOW);
  digitalWrite(dsp_LEDs[3], HIGH);
  digitalWrite(dsp_LEDs[4], LOW);
  digitalWrite(dsp_LEDs[5], HIGH);
  digitalWrite(dsp_LEDs[6], LOW);
}
else if( s == " ") { // turn off all LEDs
  digitalWrite(dsp_LEDs[0], LOW);
```

```
    digitalWrite(dsp_LEDs[1], LOW);
    digitalWrite(dsp_LEDs[2], LOW);
    digitalWrite(dsp_LEDs[3], LOW);
    digitalWrite(dsp_LEDs[4], LOW);
    digitalWrite(dsp_LEDs[5], LOW);
    digitalWrite(dsp_LEDs[6], LOW);
  }
}



void debug(String type) {
  // Print statements to check functionality of the code
  // type: "sensor" = print raw sensor readings
  //       "status" = print status readings
  //       "line" = print line detection variables
  //       "remote" = show on display direction of wheel motion
  //       "fire" = print whether digital read of IR sensor is high/low
  //       "position" = x and y coordinates of vehicle
  //       "calibrate" = forward movement calibration timing variables

  if( type == "sensor" ) {
    Serial.println("QTI_L: " + String(snsL) + " , QTI_R: " + String(snsR) ); //
print raw sensor readings
    if(stsL == 1 && stsR == 1) { display_char("LR"); }
    if(stsL == 1 && stsR == 0) { display_char("LX"); }
    if(stsL == 0 && stsR == 1) { display_char("XR"); }
    if(stsL == 0 && stsR == 0) { display_char("XX"); }
  }

  else if( type == "status" ) {
    Serial.println("STS_L: " + String(stsL) + " , STS_R: " + String(stsR) ); //
print sensor status readings
    if(stsL == 1 && stsR == 1) { display_char("LR"); }
    if(stsL == 1 && stsR == 0) { display_char("LX"); }
    if(stsL == 0 && stsR == 1) { display_char("XR"); }
    if(stsL == 0 && stsR == 0) { display_char("XX"); }
  }

  else if( type == "line" ) {
    Serial.println("Num Lines = " + String(num_lines) + ", LineDtcTime = " +
String(line_dtc_time) + ", OOBDtcTime = " + String(oob_dtc_time));
  }
```

```
  else if( type == "remote" ) { // turn on LEDs on display based on value of
joystick transmission sent
    Serial.println("Payload values: " + String(payload[0]) + ", " +
String(payload[1]) + ", " + String(payload[2]));
    float pL_temp = 2.0*payload[0]/1023.0 - 1.0; // value of left joystick ->
range from -1 to 1
    float pR_temp = 2.0*payload[1]/1023.0 - 1.0; // " " right joystick
    if(pL_temp > 0 && pR_temp > 0) { display_char("LR"); }
    if(pL_temp > 0 && pR_temp < 0) { display_char("LX"); }
    if(pL_temp < 0 && pR_temp > 0) { display_char("XR"); }
    if(pL_temp < 0 && pR_temp < 0) { display_char("XX"); }
  }

  else if( type == "fire" ) {
    //Serial.println("Fire Detection (Front) -> Building: " +
String(building_dtc[1]) + ", IR Signal: " + String(prox_sns[1]) + ", IR Signal
(prev.): " + String(prev_prox_sns[1]) + ", Signal Expire: " +
String(fire_timer_exp[1]) + ", Detection Time: " + String(fire_dtc_time[1]) );
    if(building_dtc[0] == 1 || building_dtc[1] == 1 || building_dtc[2] == 1) {
// any sensor detects building (blank face)
      display_char("1");
    }
    else if(building_dtc[0] == 2 || building_dtc[1] == 2 || building_dtc[2] ==
2) { // any sensor detects fire
      display_char("F"); // Fire
      delay(500);
    }
    else {
      display_char("0");
    }
  }

  else if( type == "position" ) {
    Serial.println("(x,y) : (" + String(pos[0]) + ", " + String(pos[1]) + ")"
);
    display_char(String(pos[1])); // display y-position
  }

  else if( type == "calibrate") {
    Serial.println("[t0,t1,tn] = [" + String(t0) + ", " + String(t1) + ", " +
String(tn) + "]");
  }
  else if( type == "wheels") {
    Serial.println("l_spd: " + String(l_spd) + ", r_spd: " + String(r_spd));
```

```
  }

  else {
    ; // do nothing
  }

}



//
================================================================================
============================================================
// DEFINE GLOBAL FUNCTIONS - Part 2 (Remote control functionality)
//
================================================================================
============================================================



void radio_setup(){

    // initialize the transceiver on the SPI bus
  if (!radio.begin()) {
    Serial.println(F("radio hardware is not responding!!"));
    while (1) {}  // hold in infinite loop
  }

  // print example's introductory prompt
  Serial.println(F("Connection created"));

  // Set the PA Level low to try preventing power supply related problems
  // because these examples are likely run with nodes in close proximity to
  // each other.
  radio.setPALevel(RF24_PA_LOW);  // RF24_PA_MAX is default.

  // save on transmission time by setting the radio to only transmit the
  // number of bytes we need to transmit a float
  radio.setPayloadSize(sizeof(payload));  // float datatype occupies 4 bytes
  //radio.setPayloadSize(sizeof(rightPayload));

  // set the TX address of the RX node into the TX pipe
  radio.openWritingPipe(address[radioNumber]);  // always uses pipe 0
```

```
  // set the RX address of the TX node into a RX pipe
  radio.openReadingPipe(1, address[!radioNumber]);  // using pipe 1

  // additional setup specific to the node's role
  if (role) {
    radio.stopListening();  // put radio in TX mode
  } else {
    radio.startListening();  // put radio in RX mode
  }
}



void radio_receive(){

  // Automatically set vehicle to be in "listening" mode
  uint8_t pipe;
  if (radio.available(&pipe)) {              // is there a payload? get the
pipe number that recieved it
    uint8_t bytes = radio.getPayloadSize();  // get the size of the payload
    radio.read(&payload, bytes);             // fetch payload from FIFO
  radio.startListening();
  }

/*
  // Parse ladder information
  if(payload[2] < rmt_btn_threshold) { // allow threshold near 0
    ladder_down = true;
  }
  else {
    ladder_down = false;
  }
*/

}



//
================================================================================
================================================================
// DEFINE GLOBAL FUNCTIONS - Part 3 (Motion Control Elements)
```

```
//
================================================================================
================================================================


void move_forward_calibrate(unsigned long curr_time){
  // Calibrates the motion of the vehicle for moving n tiles. Input calibration
mode and time of function call.
  // Notes: Only gets performed once at start of code to (1) position vehicle
to center of tile and (2) calculate required delay times

  // Course correction
  while(stsL != stsR){ // sensors read different values -> correct course

    // Get sensor readings
    set_sensors();   // set sensor values

    // Determine turn direction
    if(stsL == 1 && stsR == 0) { // first detect line only on LEFT side -> turn
to LEFT
      if(prev_stsL == 0 && prev_stsR == 0) { // prevent trailing edge bugs
        l_spd = 0;
        r_spd = r_spd_initial;
      }
    }
    else if (stsR == 1 && stsL == 0) { // first detect line only on RIGHT side
      if(prev_stsL == 0 && prev_stsR == 0) { // prevent trailing edge bugs ->
turn to RIGHT
        l_spd = l_spd_initial;
        r_spd = 0;
      }
    }

    // Update previous readings
    prev_stsL = stsL;
    prev_stsR = stsR;

    // Move vehicle
    left_wheel(l_spd);
    right_wheel(r_spd);
  }

  // Revert back to default speeds
```

```
  l_spd = l_spd_initial;
  r_spd = r_spd_initial;

  // Initial Calibration
  left_wheel(l_spd);        // move forward until line is met
  right_wheel(r_spd);       // " "
  if(stsL == 1 || stsR == 1) { // either sensor detects line
    //Serial.println("Repositioning vehicle.");
    // pause
    left_wheel(0);
    right_wheel(0);
    delay(step_delay_time);
    // calculate parameters
    t0 = curr_time; // save time of detection (set t=0 at start, ignore time
used for correcting course)
    t1 = t0 * (d1/d0); // calculate delay for moving vehicle from line to tile
center
    tn = t0 * (dn/d0); // " " line-to-line (one tile)
    // stop calibration
    tile_calibrate = false; // prevents re-calibration
    // move to center of next tile
    left_wheel(l_spd);  // move forward
    right_wheel(r_spd); // " "
    delay(int(t1));
    // Update position
    if(dir == "N") {
      pos[1] += 1;
    }
    else if(dir == "E"){
      pos[0] += 1;
    }
    else if(dir == "S"){
      pos[1] -= 1;
    }
    else if(dir == "W"){
      pos[0] -= 1;
    }
    // turn off wheels (await further commands)
    left_wheel(0);  // stop wheels
    right_wheel(0); // " "
    delay(step_delay_time);
  }
}
```

```
void move_forward(int n, unsigned long curr_time) {
  // Move vehicle forward n squares. Works in collaboration with calibration
function (see above).
  // Input: number of tiles (n) and current time of program (curr_time)

  // System variables
  unsigned long move_time = curr_time; // time actually spent moving vehicle
(accounts for pause during course correction)

  // Move vehicle n tiles
  while( move_time < curr_time + tn*n ) {

    // 0 GET START CONDITION
    unsigned long t1_a = millis(); // time at start

    // 1 UPDATE SENSORS
    set_sensors(); // set sensor values

    // 2 RECALIBRATE
    if(tile_recalibrate) {
      if(snsL==1 || snsR==1) {
        tc = (millis() - start_delay_time) - curr_time; // save time of
occurence since last center position (curr_time always called when vehicle is
centered)
        tn = tc * (dn/dc); // recalibrate center-to-center delay time
      }
      tile_recalibrate = false; // prevent recalibration
    }

    // 3 COURSE CORRECTION
    unsigned long t1 = millis(); // time before course correction
    while(stsL != stsR){ // sensors read different values -> correct course

      // Get sensor readings
      set_sensors();   // set sensor values

      // Determine turn direction
      if(stsL == 1 && stsR == 0) { // first detect line only on LEFT side ->
turn to LEFT
        if(prev_stsL == 0 && prev_stsR == 0) { // prevent trailing edge bugs
          l_spd = 0;
          r_spd = r_spd_initial;
```

```
        }
      }
      else if (stsR == 1 && stsL == 0) { // first detect line only on RIGHT
side
        if(prev_stsL == 0 && prev_stsR == 0) { // prevent trailing edge bugs ->
turn to RIGHT
          l_spd = l_spd_initial;
          r_spd = 0;
        }
      }

      // Update previous readings
      prev_stsL = stsL;
      prev_stsR = stsR;

      // Move vehicle
      left_wheel(l_spd);
      right_wheel(r_spd);

    }
    unsigned long dt = millis() - t1; // time elapsed during correction

    // 4 REVERT TO DEFAULT SPEEDS
    l_spd = l_spd_initial;
    r_spd = r_spd_initial;

    // 5 MOVE FORWARD
    left_wheel(l_spd);
    right_wheel(r_spd);

    // 6 UPDATE TIME
    unsigned long t1_b = millis(); // time at end
    move_time += (t1_b - t1_a) - dt; // total time elapsed during loop - time
elapsed during course correction

  }


  // Update position
  if(dir == "N") {
    pos[1] += 1;
  }
  else if(dir == "E"){
    pos[0] += 1;
```

```
  }
  else if(dir == "S"){
    pos[1] -= 1;
  }
  else if(dir == "W"){
    pos[0] -= 1;
  }


  // Turn off wheels to prevent further movement
  left_wheel(0);
  right_wheel(0);
  delay(step_delay_time); // wait 1s after moving


  // Reset calibration variable for next cycle
  tile_recalibrate = true;

}



void turn_angle(int ang){
  // Turn the vehicle in place a specified angle.
  // Input: 90 = clockwise quarter-turn (right), 270 = counter-clockwise
quarter-turn (left), 180 = U-turn
  // NOTE: ***requires vehicle speed values (l_spd, r_spd) and forward motion
timing parameters (tn) to be calibrated to work***

  switch(ang) {
    case 90: // right turn
      // set wheel speeds
      left_wheel(l_spd);
      right_wheel(-r_spd);
      // calculate turning time
      float turn_time = tn * (half_pi*(w_spc/2))/dn; // use distance of travel
(quarter circle) and calibrated data to scale delay
      delay(turn_time);
      // pause
      left_wheel(0);
      right_wheel(0);
      delay(step_delay_time);
      // re-center vehicle
      left_wheel(l_spd);
```

```
      right_wheel(r_spd);
      float readj_time = tn * d_wc/dn;              // see above
      delay(readj_time);
      // pause
      left_wheel(0);
      right_wheel(0);
      delay(step_delay_time);
      // update direction
      if(dir=="N") { dir="W"; }
      else if(dir=="E") { dir="N"; }
      else if(dir=="S") { dir="E"; }
      else if(dir=="W") { dir="S"; }
      break;

    case 270: // left turn -> see above code
      // set wheel speeds
      left_wheel(-l_spd);
      right_wheel(+r_spd);
      // calculate turning time
      turn_time = tn * (half_pi*(w_spc/2))/dn; // use distance of travel
(quarter circle) and calibrated data to scale delay
      delay(turn_time);
      // pause
      left_wheel(0);
      right_wheel(0);
      delay(step_delay_time);
      // re-center vehicle
      left_wheel(l_spd);
      right_wheel(r_spd);
      readj_time = tn * d_wc/dn;              // see above
      delay(readj_time);
      // pause
      left_wheel(0);
      right_wheel(0);
      delay(step_delay_time);
      // update direction
      if(dir=="N") { dir="E"; }
      else if(dir=="E") { dir="S"; }
      else if(dir=="S") { dir="W"; }
      else if(dir=="W") { dir="N"; }
      break;

    case 180: // U-turn
      // set wheel speeds
```

```
      left_wheel(l_spd);   // turn around clockwise
      right_wheel(-r_spd);
      // calculate turning time
      turn_time = tn * (2*half_pi*(w_spc/2))/dn; // use distance of travel
(quarter circle) and calibrated data to scale delay
      delay(turn_time);
      // re-center vehicle
      left_wheel(-l_spd);
      right_wheel(-r_spd);
      readj_time = tn * 2*d_wc/dn;              // see above
      delay(readj_time);
      // update direction
      if(dir=="N") { dir="S"; }
      else if(dir=="E") { dir="W"; }
      else if(dir=="S") { dir="N"; }
      else if(dir=="W") { dir="E"; }
      break;
  }

  // reset wheel speeds
  left_wheel(0);
  right_wheel(0);

}



//
================================================================================
================================================================
// DEFINE GLOBAL FUNCTIONS - Part 4 (Building detection)
//
================================================================================
================================================================



int fire_detection(int i) {
  // Reads IR sensor signals and determines whether a building or a fire is
detected.
  // Input: sensor to read values from -> 0 = left, 1 = front, 2 = right
  // Output: detection type -> 0 = no detection, 1 = building (blank face), 2 =
fire (window face)
```

```
  // Read sensor value
  prox_sns[i] = (digitalRead(IR_RECEIVE_PIN[i]) == LOW); // true = signal
detected, false = no detection (detects around 4cm away)

  // Determine whether timer has expired
  fire_timer_exp[i] = (time > fire_dtc_time[i] + fire_dtc_time_max); // true =
timer has expired, false = timer is still going

  if(prox_sns[i] && !prev_prox_sns[i] && fire_timer_exp[i]) { // signal is
detected for first time (timer has not expired)
    fire_dtc_time[i] = time;        // set timer value
  }

  if(!fire_timer_exp[i]) { // timer has not yet expired
    prev_prox_sns[i] = prox_sns[i]; // update previous value
    if(!prox_sns[i]) {
      return 2; // signal is lost during timer, fire has been detected
    }
    return 1; // assume blank face of building until fire is detected
  } else{
    prev_prox_sns[i] = prox_sns[i]; // update previous value
    return 0; // timer has expired
  }

  // Default output
  prev_prox_sns[i] = prox_sns[i]; // update previous value
  return 0;                       // assume no signal detected otherwise

}



void cartography() {
  // Updates grid matrix based on readings from IR sensors.

  if(dir == "N") { // North heading
    if(fire_detection(0)==1){ // left sensor
      grid[pos[1]][pos[0]-1] = 0; // add building
    }
    if(fire_detection(1)==1){ // front sensor
      grid[pos[1]+1][pos[0]] = 0; // " "
    }
    if(fire_detection(2)==1){ // right sensor
      grid[pos[1]][pos[0]+1] = 0; // " "
```

```
    }
  }
  else if(dir == "E") { // East heading
    if(fire_detection(0)==1){ // left sensor
      grid[pos[1]+1][pos[0]] = 0; // add building
    }
    if(fire_detection(1)==1){ // front sensor
      grid[pos[1]][pos[0]+1] = 0; // " "
    }
    if(fire_detection(2)==1){ // right sensor
      grid[pos[1]-1][pos[0]] = 0; // " "
    }
  }
  else if(dir == "S") { // South heading
    if(fire_detection(0)==1){ // left sensor
      grid[pos[1]][pos[0]+1] = 0; // add building
    }
    if(fire_detection(1)==1){ // front sensor
      grid[pos[1]-1][pos[0]] = 0; // " "
    }
    if(fire_detection(2)==1){ // right sensor
      grid[pos[1]][pos[0]-1] = 0; // " "
    }
  }
  else if(dir == "W") { // West heading
    if(fire_detection(0)==1){ // left sensor
      grid[pos[1]-1][pos[0]] = 0; // add building
    }
    if(fire_detection(1)==1){ // front sensor
      grid[pos[1]][pos[0]-1] = 0; // " "
    }
    if(fire_detection(2)==1){ // right sensor
      grid[pos[1]+1][pos[0]] = 0; // " "
    }
  }
}



//
================================================================================
================================================================
// DEFINE GLOBAL FUNCTIONS - Part 5 (Path finding)
```

```
//
================================================================================
================================================================


int grid_to_index(int x, int y) {
  // Converts grid coordinate into a node name (used for A* algorithm).
  return x+y*COL;
}

int temp_pos[2]; // temporary coordinate (x,y) for function output
void index_to_grid(int i) {
  // Converts index position to grid coordinates (x,y)
  // Note: "temp_pos" variable is used instead of a function output.
  temp_pos[0] = i%COL;         // update temp variable (x value)
  temp_pos[1] = (i-i%COL)/COL; // " " (y value)
}



void print_grid(int arr[ROW][COL], int r, int c, String type){
  // Print an array of values as text characters in the Serial for debugging
purposes.
  // Input: 2D array "arr", number of rows, number of columns, type [Grid,
Cost, Path]

  // print header
  Serial.println();
  Serial.println(type + ": ");

  // loop over each element of 2D array
  //for(int i=r-1; i>=0; i--){ // print from top to bottom -> reverse order!!!
  for(int i=0; i<r; i++){ // print from bottom to top
    for(int j=0; j<c; j++){
      int a = arr[i][j]; // value to print
      if ( type == "Grid" ) {
        if( a == 0 ) { Serial.print("X "); } // convert integers to grid
information
        if( a == 1 ) { Serial.print("0 "); } // " "
      }
      if ( type == "Cost" ) {
        if(a < 10) { Serial.print("00"); }        // print leading zeroes
(format as 3 digit number)
```

```
        if(a>=10 && a<100) { Serial.print("0"); } // " "
        Serial.print(a + String(" "));            // display number
      }
      if ( type == "Path" ) {
        ; // print path information ?
      }
    }
    Serial.println();
  }

}



String get_directions(int p1, int p2, String d) {
  // Instructions for moving vehicle, used during path reconstruction of A*
algorithm.
  // Input: first position index (p1), second position index (p2), and
direction at first position (d)
  // Output: first character = new direction heading [N = north, S = south, E =
east, W = west], second character = turn direction [R = right, L = left, U =
turn around, S = straight/no-turn]
  // Notes: forward movement is implied after each instruction

  String s = ""; // string to output

  // get coordinate values
  index_to_grid(p1);                          // p1 = (x1, y1)
  int x1 = temp_pos[0]; int y1 = temp_pos[1]; // " "
  index_to_grid(p2);                          // p2 = (x2, y2)
  int x2 = temp_pos[0]; int y2 = temp_pos[1]; // " "

  // Check all starting directions
  // North
  if( d == "N" ) { // Towards user (increasing y)
    if( x1 == x2 ) {
      if( y1 < y2 ) { s = "NS";  }
      if( y1 > y2 ) { s = "SU";  }
    }
    else if( x1 < x2 ) { s = "EL"; }
    else if( x1 > x2) {  s = "WR";  }
  }
  // East
  else if( d == "E" ) { // Right (increasing x)
```

```
    if( y1 == y2 ) {
      if( x1 < x2 ) { s = "ES"; }
      if( x1 > x2 ) { s = "WU"; }
    }
    else if( y1 < y2 ) { s = "NR"; }
    else if( y1 > y2 ) { s = "SL"; }
  }
  // South
  else if( d == "S" ) { // Away from user (decreasing y) -> starting direction
    if( x1 == x2 ) {
      if( y1 < y2 ) { s = "NU";  }
      if( y1 > y2 ) { s = "SS";  }
    }
    else if( x1 < x2 ) { s = "ER"; }
    else if( x1 > x2) {  s = "WL";  }
  }
  // West
  else if( d == "W" ) { // Left (decreasing y)
    if( y1 == y2 ) {
      if( x1 < x2 ) { s = "EU"; }
      if( x1 > x2 ) { s = "WS"; }
    }
    else if( y1 < y2 ) { s = "NL"; }
    else if( y1 > y2 ) { s = "SR"; }
  }

  return s; // default value to return, throws an error in system

}



String aStar(int startx, int starty, int endx, int endy, String dir){
  // Performs A* search algorithm using the current grid.
  // Input: grid of values (0 = wall, 1 = free space), start position (x, y),
end position, and starting direction
  // Output: list of instructions on how to move vehicle -> [F = forward 1
tile, R = turn 90deg CW, L = turn 90 deg CCW]
  // See reference [1]
https://medium.com/@nicholas.w.swift/easy-a-star-pathfinding-7e6689c7f7b2

  // === 1 ===
  // Initialize A* matrices
  for(int i=0; i<ROW; i++){
```

```
    for(int j=0; j<COL; j++){
      h[i][j] = abs(j - endx) + abs(i - endy); // compute manhattan heuristic
      if( i == starty && j == startx ) { // starting node
        g[i][j] = 0;       // set path distance as 0 for starting node
        f[i][j] = h[i][j]; // starting cost function is just heuristic value
      } else { // all other nodes
        g[i][j] = 999;     // assign path matrix as "infinite" to start
        f[i][j] = 999;     // assign cost function also as infinite
      }
    }
  }

  // === 2 ===
  // Initialize variables used
  // positioning
  int x, y;                 // position of current node checked
  // parent references
  int parents[ROW*COL];     // parents[i] = k -> node i has a parent at node k
  for(int i=0; i<ROW*COL; i++) {
    parents[i] = -1; // set as -1 to start
  }
  // open and closed lists
  int open_list_size = 0;   // size of lists, used for adding nodes to search
query
  int closed_list_size = 0; // " "
  // initialize open and closed lists (whether node indices are in list or not)
  bool open_list[ROW*COL];   // value at index = 1 if node index is in list,
elsewise it is 0
  bool closed_list[ROW*COL]; // " "
  // set all values as false to start (lists are both empty)
  for(int i=0; i<ROW*COL; i++) {
    open_list[i] = false;
    closed_list[i] = false;
  }

  // === 3 ===
  // Add starting node to open list
  open_list[grid_to_index(startx, starty)] = true; // set starting position
node to be in open list
  open_list_size += 1;


  // === 4 ===
  // Perform search algoritm
```

```
  while( open_list_size > 0 ) { // loop through nodes until all nodes have been
visited and expanded
    // === 5 ===
    // Get the current node (node with minimum f value from priority queue)
    int temp_min_f = 999; // minimum f value
    int temp_min_node;    // node index of above minimum
    int temp_h_at_min;    // h value of above minimum (used for tiebreakers)
    for(int i=0; i<ROW*COL; i++) {
      if( open_list[i] ) { // only look at nodes in open list
        index_to_grid(i);                          // get position of node in
list
        int xa = temp_pos[0]; int ya = temp_pos[1]; // " " (store as temporary
variables)
        int f_val = f[ya][xa]; // score of current node to check (prevents
repeated array calls)
        int h_val = h[ya][xa]; // " "
        if( f_val < temp_min_f ) { // choose value of lowest f score
          temp_min_f = f_val;    // update minimum
          temp_h_at_min = h_val; // " ""
          temp_min_node = i;     // update node
        }
        else if ( f_val == temp_min_f && h_val < temp_h_at_min ) { // break
ties if present
          temp_min_f = f_val;    // update minimum
          temp_h_at_min = h_val; // " ""
          temp_min_node = i;     // update node
        }
      }
    }
    // save current cell position
    index_to_grid(temp_min_node);     // get position of current node
    x = temp_pos[0]; y = temp_pos[1]; // " "

    // === 6 ===
    // Move current node from OPEN list to CLOSED list
    open_list[temp_min_node] = false;  open_list_size -= 1;   // remove from
open list
    closed_list[temp_min_node] = true; closed_list_size += 1; // add to closed
list

    //Serial.println("Current node: " + String(temp_min_node)); // print
current progress (debugging)

    // === 7 ===
```

```
    // End condition (target reached)
    if( x == endx && y == endy ) { // current node = end node
      // === 7a ===
      // Set up reconstruction algorithm
      int start_id = grid_to_index(startx, starty); // retrieve start/end node
      int end_id = grid_to_index(endx, endy);       // " "
      int path[ROW*COL];                            // return path (list of
indices)
      for(int i=0; i<ROW*COL; i++) { path[i]=-1; }  // initialize to be -1
(default value)
      int i = end_id; // current node in path -> start at end to work backwards
      int j = 1;      // counter variable

      // === 7b ===
      // Reconstruct path based on parent node information
      path[0] = end_id; // first index should be end value
      while( i != start_id ) {
        path[j] = parents[i]; // path follows heirarchy of parent nodes
        i = parents[i];       // new i = parent node of former i
        j += 1;               // increase counter
      }
      // print path (for debugging)
      for(int k=j-1; k>=0; k--){
        Serial.print(String(path[k]) + ", ");
      }

      // === 7c ===
      // Go backwards through path to get instructions
      String guide = ""; // directions for vehicle to follow -> F = forward, R
= 90deg CW turn, L = 90 degree CCW turn
      String d = dir;       // current heading of vehicle in path -> start in
indicated start direction
      for(int k=j-1; k>=0; k--) { // go backwards through path starting at
first node
        String dir_output = get_directions(path[k], path[k-1], d); // output of
directions function -> use current position, next position, and current
direction to get instructions of travel
        d = dir_output[0];      // first character is new direction heading ->
update heading
        guide += dir_output[1]; // second character is turn data (i.e., R
(right), L (left), or S (straight))
        guide += "F";           // forward direction is implied with each step
      }
      //Serial.println(guide); // print guide information (for debugging)
```

```
      return guide; // return list of instructions
    }

    // === 8a ===
    // Generate children of current node
    int children[4]; // list of node indices that neighbor current node
    // North
    int ci = grid_to_index(x, y+1);                      // child index (temp
variable)
    if( grid[y+1][x] == 1 && closed_list[ci] == false ) { // check if child
node is not a barrier AND it hasnt be explored before
      children[0] = ci;
    } else {
      children[0] = -1;                                  // use default value
otherwise (skips child)
    }
    // South
    ci = grid_to_index(x, y-1);                          // see above
    if( grid[y-1][x] == 1 && closed_list[ci] == false ) { // " "
      children[1] = ci;
    } else {
      children[1] = -1;                                  // " "
    }
    // East
    ci = grid_to_index(x+1, y);                          // see above
    if( grid[y][x+1] == 1 && closed_list[ci] == false ) { // " "
      children[2] = ci;
    } else {
      children[2] = -1;                                  // " "
    }
    // West
    ci = grid_to_index(x-1, y);                          // see above
    if( grid[y][x-1] == 1 && closed_list[ci] == false ) { // " "
      children[3] = ci;
    } else {
      children[3] = -1;                                  // " "
    }

    // === 8b ===
    // set children as successors of parents
    for(int i=0; i<4; i++) {
      int n = children[i]; // child node
      if( n != -1) {
```

```
        parents[n] = temp_min_node; // set parent node
      }
    }

    /*
    // print current progress (debugging)
    String c_print = ""; // print children for
    c_print += "Parent: " + String(temp_min_node);
    c_print += ", Children: [";
    c_print += String(children[0]) + ", " + String(children[1]) + ", " +
String(children[2]) + ", " + String(children[3]);
    c_print += "]";
    Serial.println(c_print);
    */

    // === 9 ===
    // Check each child cell
    for(int k=0; k<4; k++) {
      if( children[k] > -1) { // make sure child node exists/is valid

        // === 10a ===
        // Check if child is in the CLOSED list
        if( closed_list[children[k]] == true ) { // child is in CLOSED list
          continue; // go to next child
        }

        // === 10b ===
        // Get child position and A* matrix values
        index_to_grid(children[k]);                 // get position of child
node
        int xc = temp_pos[0]; int yc = temp_pos[1]; // " "
        g[yc][xc] = g[y][x] + 1;           // g value of child node (g child =
g parent + 1)
        f[yc][xc] = g[yc][xc] + h[yc][xc]; // f = g + h

        // === 10c ===
        // Check if child is in the OPEN list
        if( open_list[children[k]] == true ) { // child is in OPEN list
          if( g[yc][xc] > g[y][x] ) {
            continue; // go to next child
          }
        }

        // === 10d ===
```

```
        // Put child in open list
        open_list[children[k]] = true; open_list_size += 1; // add to open list

      }
    }

  }

  if( x != endx || y != endy ) { // failed to reach end, throw error
    Serial.print("Could not reach goal!");
  }

}
```

Code F2 (continued).

```
C/C++
// TITLE: Remote Controller Transmission
// AUTHOR: Andrew Smith, Noah Etienne, Dylan Taylor, and Venkata Sai Phanindra
Mandadapu
// UPDATED: May 16th, 2024
// NOTES:
// - (4/25) Original functionality
// - (4/26) Simplified code, allowed automatic transmission
// - (5/16) Final design, minor bug fixes



//
================================================================================
=========================



#include <SPI.h>
#include "printf.h"
#include "RF24.h"

#define CE_PIN 7
#define CSN_PIN 8

RF24 radio(CE_PIN, CSN_PIN);

uint8_t address[][6] = { "1Node", "2Node" };
bool radioNumber = 0;  // 0 uses address[0] to transmit, 1 uses address[1] to
transmit
bool role = false;  // true = TX role, false = RX role
int payload[3];
int i;
int buttonPin = 6;
int buttonState = 0;



void setup() {
  // put your setup code here, to run once:
  Serial.begin(9600);
  pinMode(buttonPin, INPUT);
```

```
  while (!Serial) {
    // some boards need to wait to ensure access to serial over USB
  }

  // initialize the transceiver on the SPI bus
  if (!radio.begin()) {
    Serial.println(F("radio hardware is not responding!!"));
    while (1) {}  // hold in infinite loop
  }

  // print example's introductory prompt
  Serial.println(F("Connection created!"));


  // role variable is hardcoded to RX behavior, inform the user of this
  Serial.println(F("*** PRESS 'T' to begin transmitting to the other node"));

  // Set the PA Level low to try preventing power supply related problems
  // because these examples are likely run with nodes in close proximity to
  // each other.
  radio.setPALevel(RF24_PA_LOW);  // RF24_PA_MAX is default.

  // save on transmission time by setting the radio to only transmit the
  // number of bytes we need to transmit a float
  radio.setPayloadSize(sizeof(payload));  // float datatype occupies 4 bytes
  //radio.setPayloadSize(sizeof(rightPayload));

  // set the TX address of the RX node into the TX pipe
  radio.openWritingPipe(address[radioNumber]);  // always uses pipe 0

  // set the RX address of the TX node into a RX pipe
  radio.openReadingPipe(1, address[!radioNumber]);  // using pipe 1

  // additional setup specific to the node's role
  if (role) {
    radio.stopListening();  // put radio in TX mode
  } else {
    radio.startListening();  // put radio in RX mode
  }

}
```

```
void loop() {

  // Auto-assign to transmit information
  role = true;
  radio.stopListening();

  if (role) {
    // Read controller information
    payload[0] = analogRead(A0); // left joystick (up = 1023, down = 0)
    payload[1] = analogRead(A1); // right " "
    payload[2] = analogRead(A2); // right joystick button (0 = pressed in)

    unsigned long start_timer = micros();    // start the timer
    bool report = radio.write(&payload, 6);  // transmit & save the report
    unsigned long end_timer = micros();      // end the timer

    if (report) {
      Serial.print(F("Transmission successful! "));  // payload was delivered
      Serial.print(F("Time to transmit = "));
      Serial.print(end_timer - start_timer);  // print the timer result
      Serial.print(F(" us. Sent: "));
      Serial.print(payload[0]);
      Serial.print(", ");
      Serial.print(payload[1]);
      Serial.print(", ");
      Serial.println(payload[2]);

      //Serial.println(payload);  // print payload sent
      //payload += 0.01;          // increment float payload
    } else {
      Serial.println(F("Transmission failed or timed out"));  // payload was
not delivered
    }

    // to make this example readable in the serial monitor
    delay(100);  // slow transmissions down by 1 second

  }

}
```