Indoor Autonomous Navigation System

Kelvin Silva, Daniel Hunter, Kevin Beher, Kyle Ebding, Juan Huerta

April 3, 2018

Contents

Abbreviations

API	Application programming interface
GCE	Google Compute Engine
IANS	Indoor Autonomous Navigation System
IMU	Inertial Measurement Unit
LiDAR	Light Detection and Ranging
MDF	Medium Density Fiberboard
MQTT	Message Queuing Telemetry Transport
PWM	Pulse Width Modulation
ROS	Robotic Operating System

1 | System Introduction

1.1 Abstract

We present the Indoor Autonomous Navigation System (IANS) as a prototype and proof of concept to be implemented onto a wheelchair. The use of wheelchairs is increasing and will continue to increase in the coming years: a study in England and Wales showed that between 1986 and 1995, there was a 100% increase in the number of wheelchair users[2]. With this increase comes significant personal and social challenges. Those who have used a wheelchair as a method of transportation report frequent tipping or ejection. A Nova Scotian study reported that out of 577 wheelchair users, 57% of them reported having completely tipped over[1]. Currently this risk can be mitigated by medical assistants who provide help in pushing the user's wheelchair. While this can be an effective solution, users report that they feel a loss of personal sovereignty and dehumanization due to their dependence on such medical assistance[2].

In recent years the technology for autonomous indoor navigation has become feasible, and the time has come to deliver a cost efficient autonomous wheelchair as a safety enhancement, and to empower individuals who have lost the ability to walk. Due to the already high cost of electric wheelchair mechanical components, our system is built with low cost electronics to be incorporated into existing electric wheelchairs, providing an affordable solution. The system is applicable for a variety of environments, instead of predesigned for a specific floor plan as some products are.

1.2 System Description

The Indoor Autonomous Navigation System consists of a web server that communicates with a mobile robot (see Figure 2.2 for a system diagram). Using the mobile web interface, a user can direct the robot to a list of destinations in a pre mapped environment. When the robot receives a new destination from the user, it uses the map and onboard sensors to avoid obstacles and navigate to the destination.

While we did not have the funds to prototype with a wheelchair, the system presented is designed with extensibility in mind, and can be installed onto a wheelchair with minimal modification.

Onboard the robot, we use a Raspberry Pi 3 with the Robot Operating System (ROS) framework for navigation, localization, and mapping the floor plan.

A Light Detection and Ranging (LiDAR) unit is the main sensor used to determine the robot's position and avoid obstacles. The LiDAR creates a two dimensional grid map of the indoor environment for navigation and localization purposes. Other sensors include wheel encoders and an inertial measurement unit (IMU) to track the movement of the robot.

Indoor Autonomous Navigation has been extensively studied[3], and the system presented here uses previously established techniques and best practices. To maximize the final effectiveness of the system and following the style of industry, we leverage prebuilt software throughout the project, making heavy use of the ROS framework and several libraries to interface with the onboard sensors.

1.3 Sample Use Case

This section presents an example of a user action and the resulting data flow throughout the robotic system. The user clicks a button on the web interface, which sends a message to the robot. Once the robot receives the message, it launches a set of ROS nodes[1] to control the motors, read odometry data, and scan the environment with the LiDAR. The ROS navigation and localization nodes receive LiDAR, odometry and IMU data to compute a path plan, avoiding any unmapped obstacles (such as pedestrians). Once the robot

[1]A ROS node is a programming construct used to contain specific system functionality.

has arrived to the destination, the navigation algorithm stops the robot and signals to the web interface that the robot has arrived.

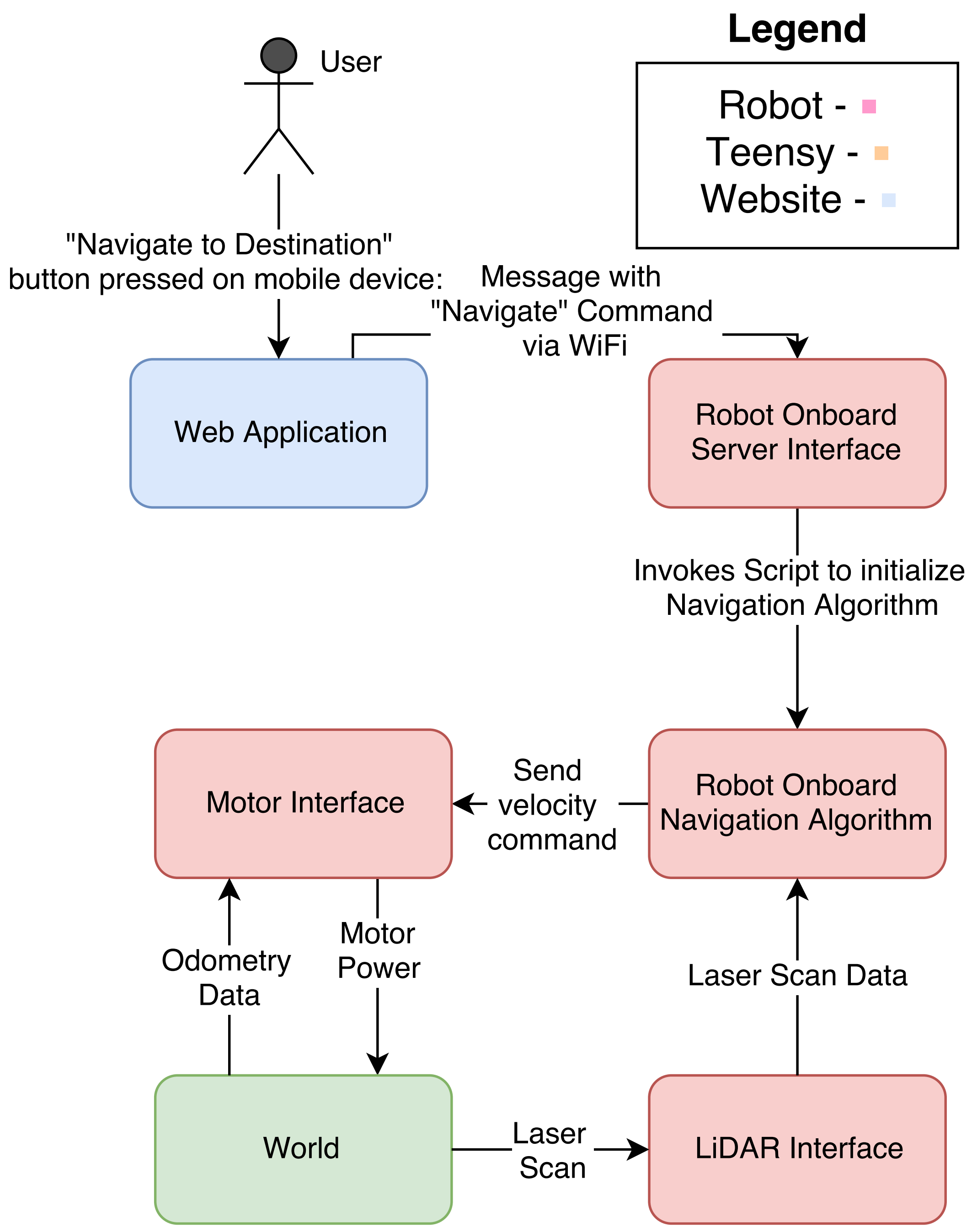

Legend
Robot - 🟥
Teensy - 🟧
Website - 🟦
User
"Navigate to Destination" button pressed on mobile device:
Message with "Navigate" Command via WiFi
Web Application
Robot Onboard Server Interface
Invokes Script to initialize Navigation Algorithm
Motor Interface
Send velocity command
Robot Onboard Navigation Algorithm
Odometry Data
Motor Power
Laser Scan Data
World
Laser Scan
LiDAR Interface

Figure 1.1: Example Use Case

2 | System Architecture

This section describes the robot system architecture, visually presenting the connections between components, both hardware and software.

2.1 Introduction

Figure 2.1 shows how all the components of the indoor autonomous navigation system interface with another. There are two onboard computers: a Raspberry Pi that calculates a route to a waypoint selected by a user via a website interface and uses data read from a LiDAR(Light Detection and Ranging) sensor for obstacle avoidance; and, a Teensy 3.6 that sends the appropriate power to the motors to follow the route provided by the Raspberry Pi and relays back IMU sensor and encoders readings to the Raspberry Pi for navigation correction.

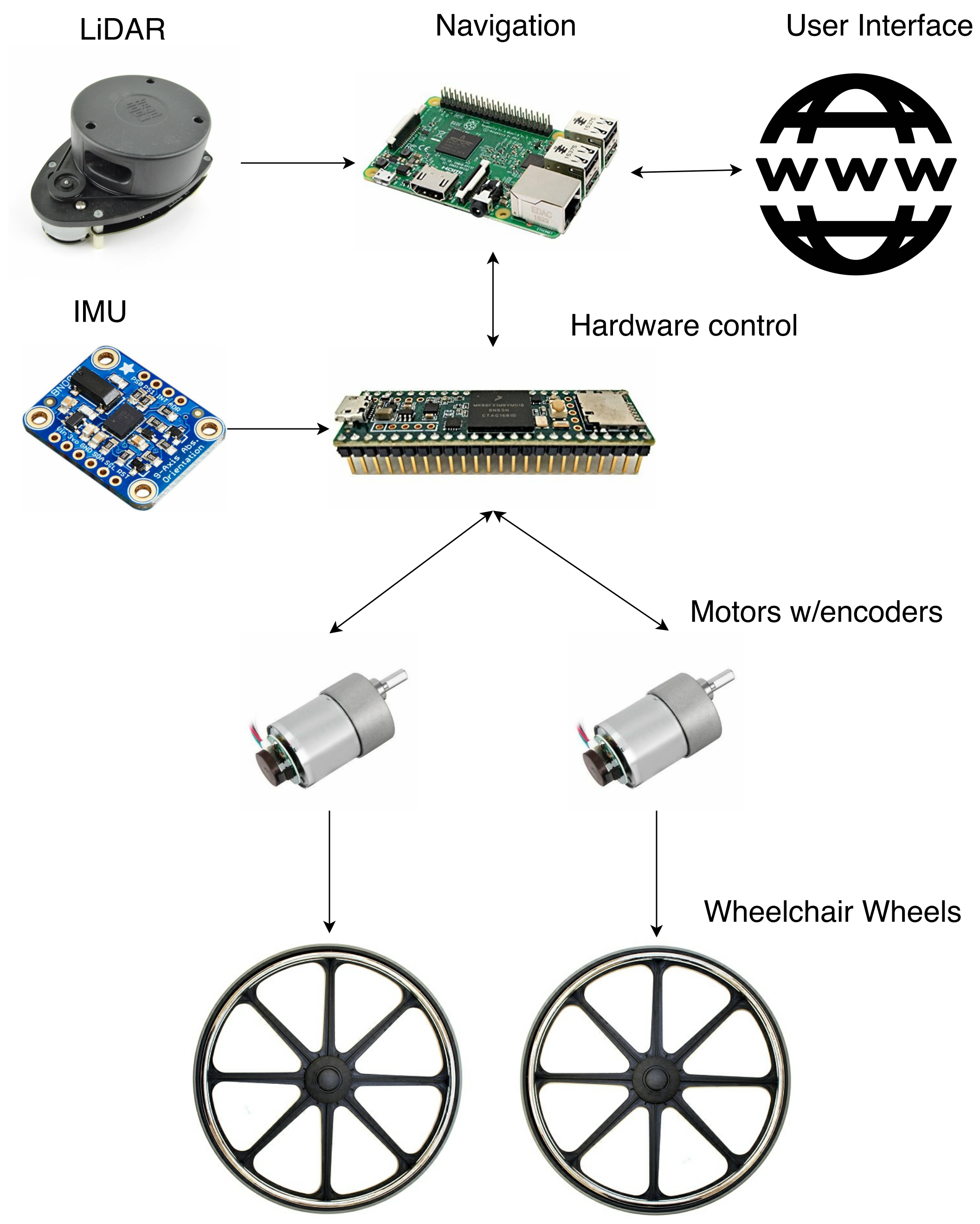


Figure 2.1: Component Diagram

The system architecture consists of three main components: the user, the web interface / server, and the robot, as shown in figure 2.2.

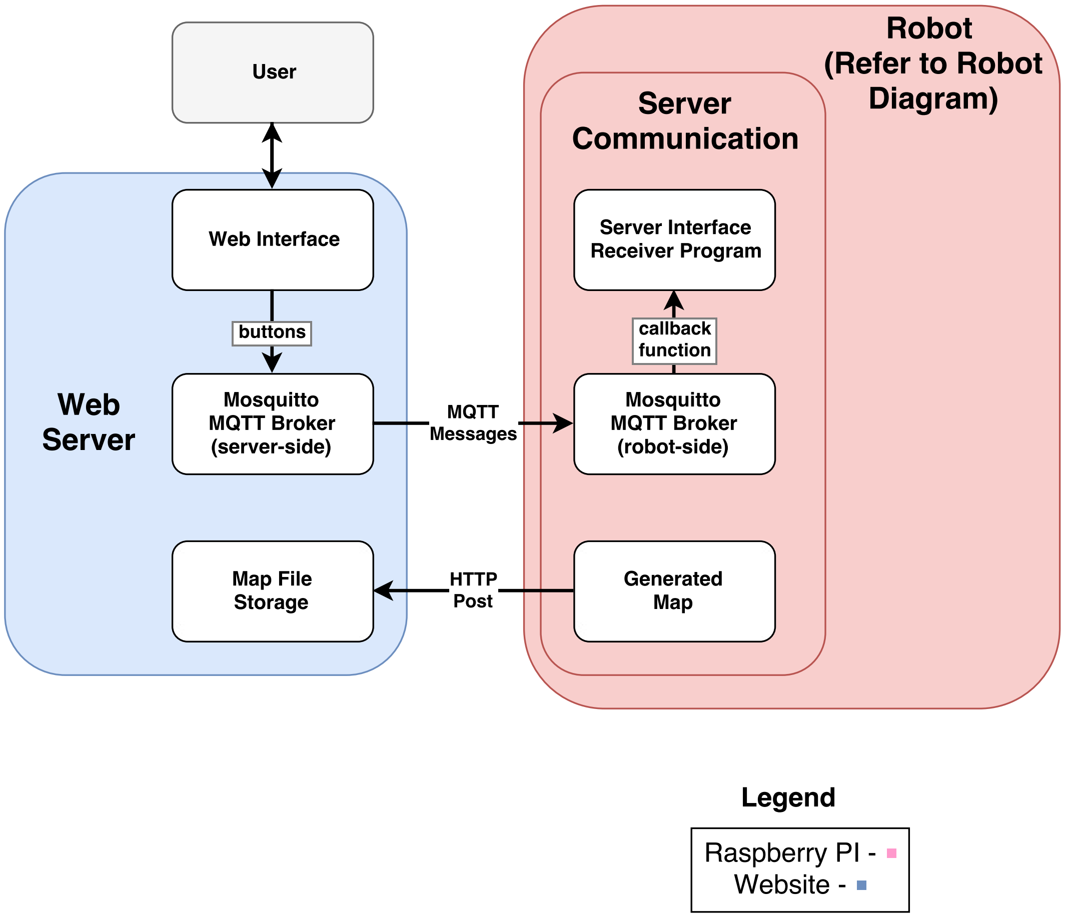
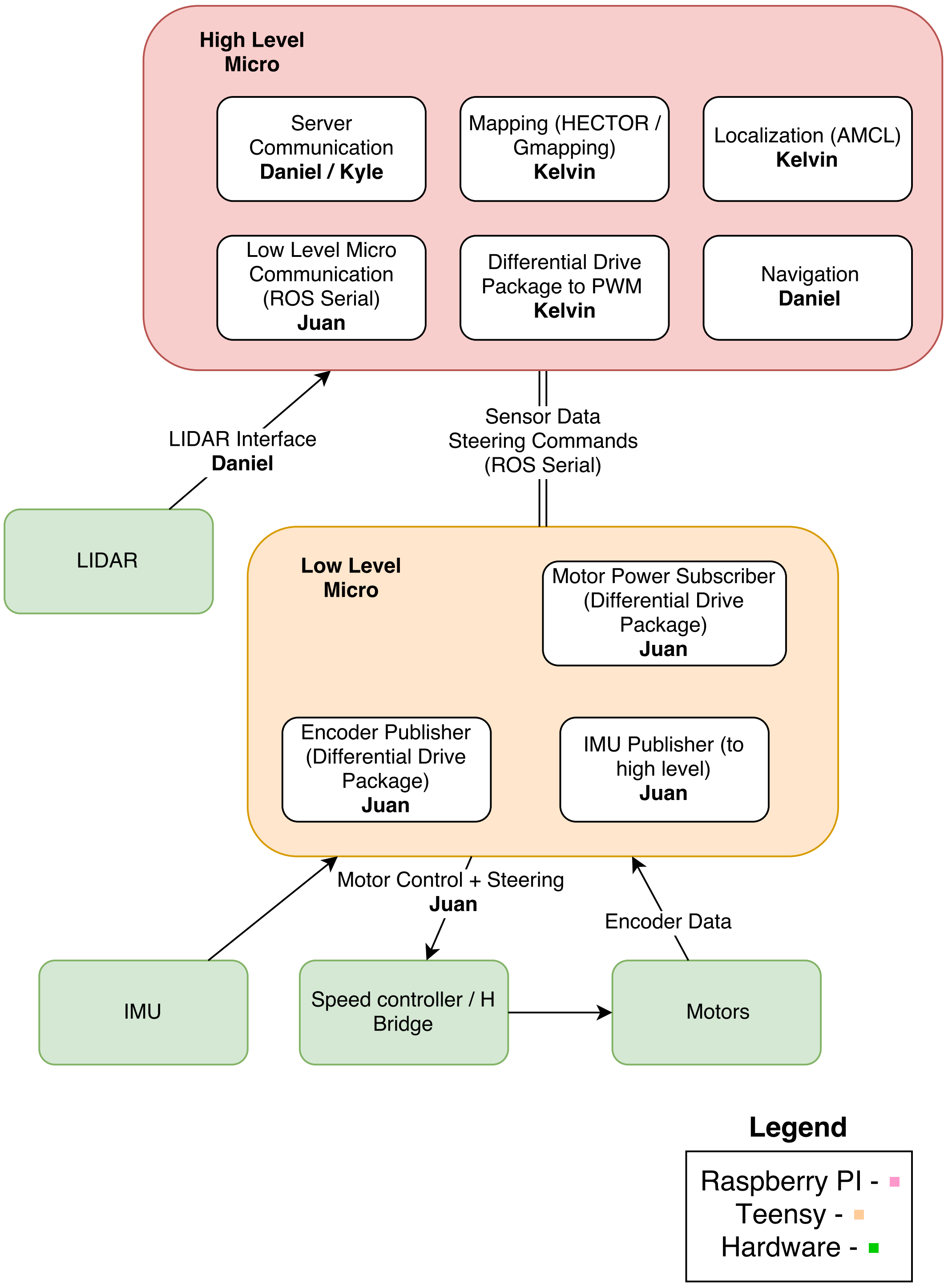


Figure 2.2: Top Level System Diagram

Figure 2.3 shows a detailed robot system diagram including sensors, motors, and both microcontrollers.



High Level Micro
Server Communication Daniel / Kyle
Mapping (HECTOR / Gmapping) Kelvin
Localization (AMCL) Kelvin
Low Level Micro Communication (ROS Serial) Juan
Differential Drive Package to PWM Kelvin
Navigation Daniel
LIDAR Interface Daniel
Sensor Data Steering Commands (ROS Serial)
LIDAR
Low Level Micro
Motor Power Subscriber (Differential Drive Package) Juan
Encoder Publisher (Differential Drive Package) Juan
IMU Publisher (to high level) Juan
Motor Control + Steering Juan
Encoder Data
IMU
Speed controller / H Bridge
Motors
Legend
Raspberry PI - 🟪
Teensy - 🟧
Hardware - 🟩

Figure 2.3: Robot System Diagram

Figure 2.4 shows the flow of how the user can interact with the interface and how the interface reacts to send the input to the robot.

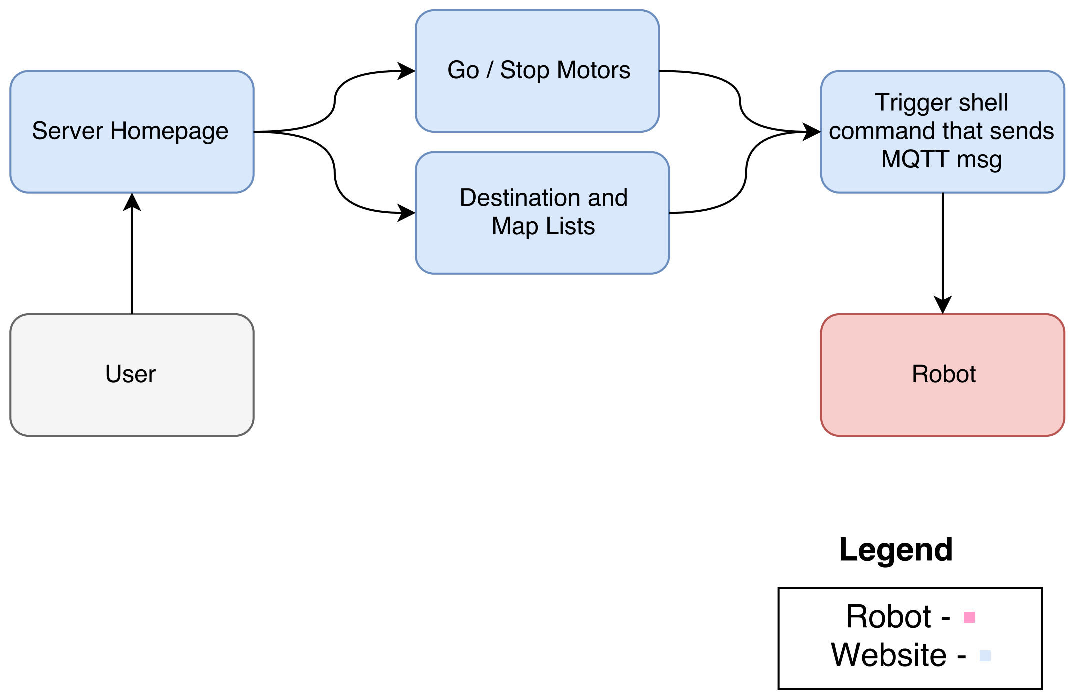

Server Homepage
Go / Stop Motors
Destination and Map Lists
Trigger shell command that sends MQTT msg
User
Robot
Legend
Robot - ■
Website - ■

Figure 2.4: Cloud Interface Flow Chart

2.2 Application Programming Interface

Figure 2.3 demonstrate several distinct hardware components that must be integrated: the web server, the high level micro, the low level micro, and the sensors (LiDAR, IMU, encoders, etc). For each integration, an Application Programming Interface (API) was created. The details of the API can be found in the appendix of this document.

2.3 User Interface

The user selects a destination on a map from a control panel on the website. The cloud server running the website sends the destination to the robot, which then navigates towards the destination. The user can toggle a motor disable switch via the website to halt the robot, and toggle it back off to tell the robot to continue.

2.4 System Software

This section discusses the purpose of the cloud software and the local software, and how they interact.

2.4.1 Software - Cloud

The website contains a floorplan of the environment, a dropdown list of the possible destinations the robot can be sent to, and a "Go" button that initiates movement to the destination. Additionally, there is a "Stop" button that disables the robot in case of an emergency. The website sends the information entered by the user to the robot automatically. The robot receives this information and uses it to determine its destination, whether or not it is allowed to move, and what map to use.

2.4.2 Software - Local

The tasks carried out repeatedly by the robot can be split into two logical groups:

1. High level tasks, such as navigation and localization,
2. Low level tasks, such as polling sensors and sending data to motors.

As particle filtering and similar localization techniques are computationally intensive, we moved low level tasks to a separate micro controller with dedicated Pulse Width Modulation (PWM) modules and more General-Purpose Input/Output (GPIO) than a Raspberry Pi. This creates two separate physical microcontrollers, and two separate codebases: the low level running on a Teensy 3.6 microcontroller, and the high level running on a Raspberry Pi 3. The high level interacts with the web server and runs the algorithms for navigation, localization, and mapping. The low level collects sensor readings from motor encoders and IMU and controls motor actuation based on commands sent from the high level.

The high level codebase consists of ROS nodes that handles different functionalities of the autonomous system. Each node is a separate system process which communicate to each other. This architecture is versatile in the case of a system crash; for example, if the LiDAR node encounters a segmentation fault the node handling the "Stop" command from the web interface can continue running and stop the robot if needed.

2.5 System Hardware

We used Solidworks to design the frame of the robot and constructed it from medium-density fibreboard (MDF), chosen for its strength-to-weight ratio.

As the scope of this project does not include high speed navigation or large mass transport, the frame was kept as simple as possible. We used a laser cutter to cut the robot's frame and as mounting brackets. The robot uses two wheel drive system with a caster for easy turning, a type of steering is known as differential drive.

We use the Raspberry Pi 3 as the main onboard computing device: it is simple to use and meets our project budget constraints. The LiDAR is the primary sensor in use on the robot. It is used to scan the surroundings and avoid obstacles. In order to simplify odometry tracking and navigation commands, we use electric motors with encoders. Additionally, there is an IMU on the robot which is used for localization and navigation purposes.

3 | System Software - Cloud

3.1 Cloud Platform

For this project we used Google Cloud Services, specifically the Google Compute Engine. We chose this service because it is scalable, easy to use, and price efficient.

3.2 Cloud Environment

The Google Compute Engine runs a Debian virtual machine (Ubuntu 16.04). This machine has a Python Flask server running on it that the user connects to through any web browser via HTTP.

3.3 Web Application

The main function of the cloud environment is to run a simple web server. The server acts as a control panel for the user to control the robot. When control actions are sent as input from the user, the server sends corresponding messages to the robot that tell the robot how to behave and what to do.

3.3.1 Web Application Frontend

The button *Toggle Movement* is used to control whether or not the robot is allowed to move, regardless of whether or not the robot has reached its destination. To issue a command, the user simply clicks or taps on the corresponding button.

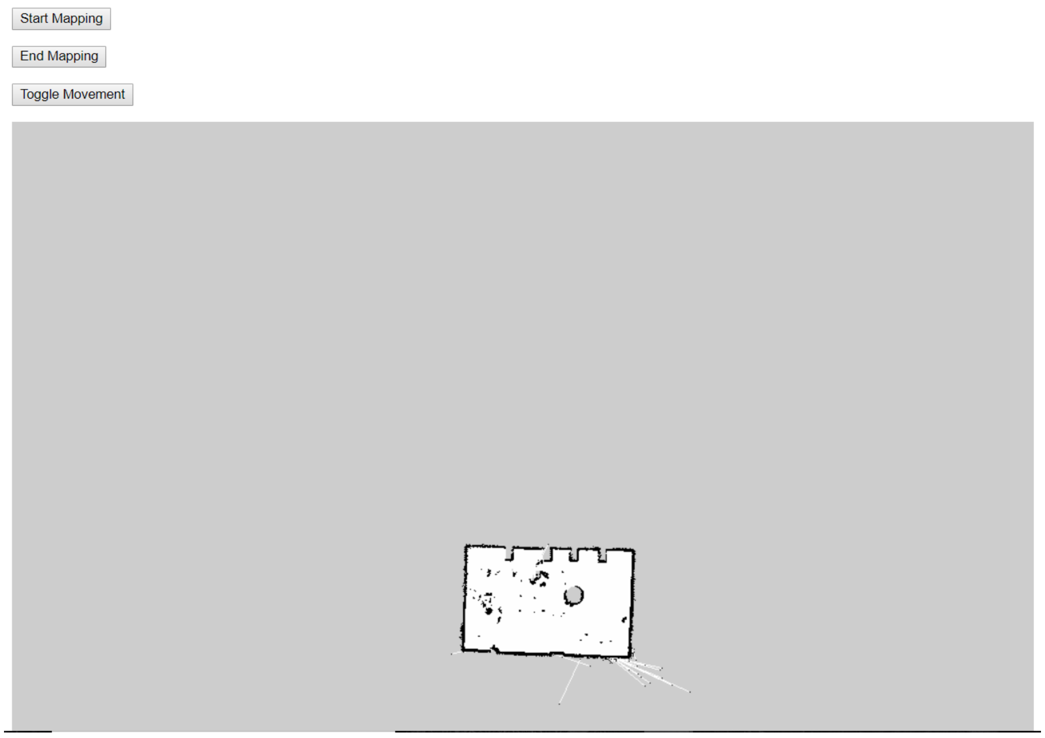


Figure 3.1: A screenshot of the web interface.

3.3.2 Web Application Backend

The server is run by a Python Flask application that serves the HTML homepage and the command endpoints (which do not serve a file to the client). Flask also receives map files from the robot after it generates maps with its LiDAR.

The home page has a button called "Toggle Movement," a map, and a dropdown list of destinations the robot can be sent to. Whenever the Toggle Movement button is pressed or a destination is selected, the server sends an Message Queue Telemetry Transport (MQTT) message to the robot. MQTT is designed for small messages via an unreliable network with low bandwidth, and is often used for sending sensor data or thermometer readings, as well as Internet of Things messages for simple devices, so it is well-suited for this project. The robot has a background program that receives MQTT messages and reacts to them accordingly by calling shell scripts that control the robot's behavior, such as by initializing the ROS nodes used to generate a map with the LIDAR.

The server has an endpoint `/v1/robot_receive_map` that accepts a .png file sent by the robot via an HTTP Post. After receiving the map the endpoint saves it to a file `map.png`, which is displayed on the homepage of the web interface. The map can be updated or replaced by simply going through the steps to generate a map again.

The server does not need to find the robot's network address. Instead, the robot will connect to the server on startup to subscribe to the topics used to control the robot. To send a message to the robot, the server merely has to publish the MQTT message to the topic corresponding to the message, and the message is sent to the robot automatically via Mosquitto, the MQTT broker used in this project.

4 | System Software - Local (On-board)

4.1 Hardware Software Integration

The Raspberry Pi communicates with the sensors via a Teensy. The Raspberry Pi handles calculating a path using the ROS navigation stack, while the Teensy handles applying the motor power needed to follow that path. We use the *differential_ drive* ROS package that converts messages provided by the navigation stack containing the current target robot velocity into left and right motor target velocities in units of m/s, and then publishes target velocities to ROS topics *lwheel_ vtarget* and *rwheel_ vtarget*. The *differential_ drive* package provides a PID controller node called *pid_ velocity*. *pid_ velocity* converts the target wheel velocities into PWM values with nine bits of precision and publishes them to the topics *lwheel* and *rwheel*. The PWM signal drives the motor angular velocity forward and reverse, proportional to the signal magnitude.

The Teensy uses subscription template functions from the Teensy ROS library that read the data published to the *lwheel* and *rwheel* topics. When the subscription functions detect a message published to the topics, they trigger a callback function that applies the motor power and direction. Using the encoders on the motors, the Teensy reads and publishes the number of encoder pulses since the start of tracking to the *rmotor* and *lmotor* topics. The *pid_ velocity* node subscribes to the *rmotor* and *lmotor* topic and uses the encoder pulse values to correct the power output back to the motors.

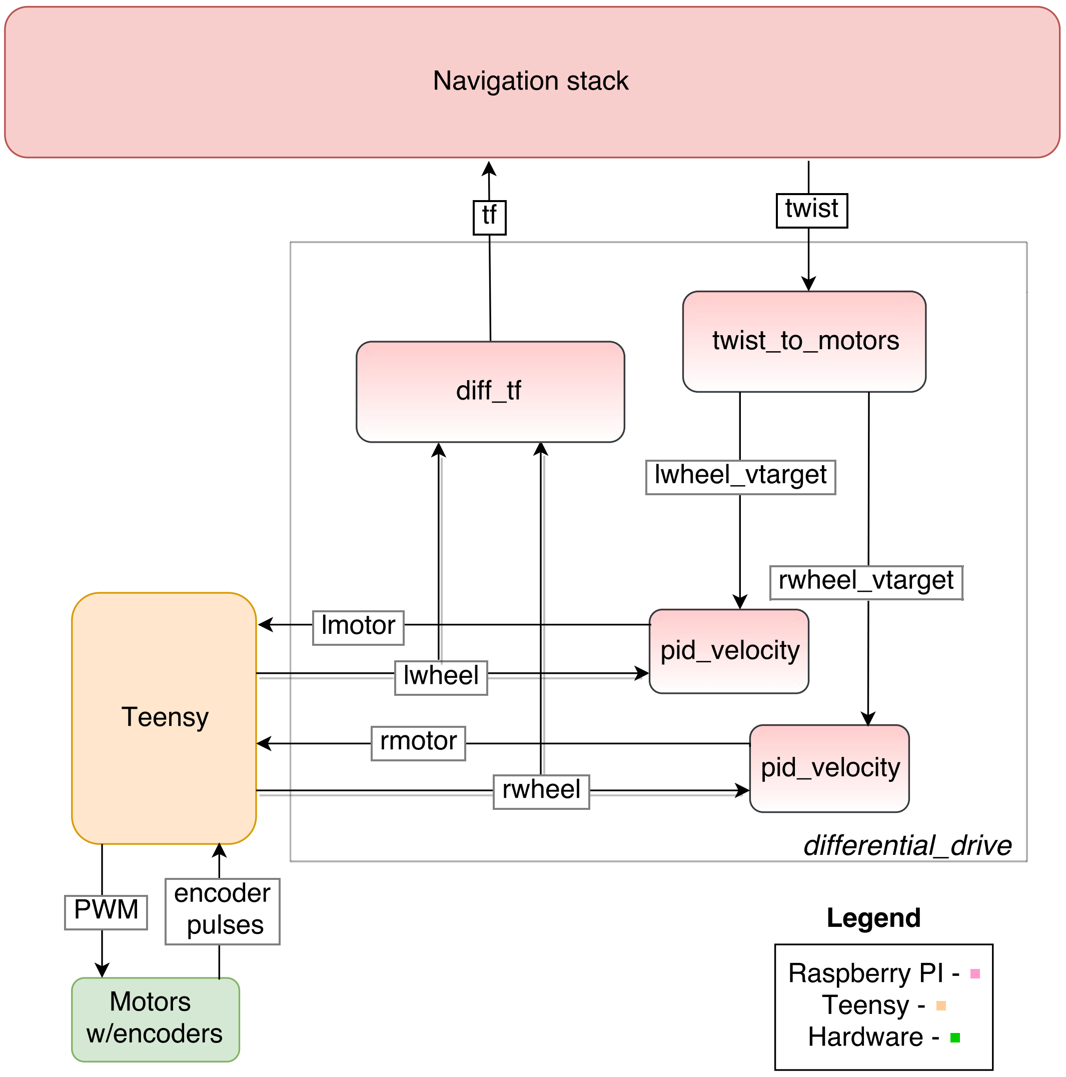


Figure 4.1: The path from the ROS navigation stack down to the Teensy 3.6 through the nodes of the ROS package *differential_ controller*.

4.2 Robot Operating System (ROS)

In our initial research of robotic navigation and localization, ROS came up repeatedly. As we investigated it, the team decided that the existing packages available were surprisingly well targeted for our needs, and the benefits far outweighed the learning curve.

ROS was the appropriate choice because we wanted to avoid writing an unwieldy program which would be hard to maintain. Such an unwieldy program would have an unmanageable code base, as well as an extremely tight coupling between different areas of the system's functionality. The decoupled nature of the ROS framework is an excellent example of software inter-process communication and modularization (introduced in the next section).

Since there are many interfaces and components in our system (perhaps in different programming languages) that raise integration complexity, a framework like ROS was a good solution. In addition, in the event that one of the system's components malfunctions, a decoupled approach allows safety critical components to continue running.

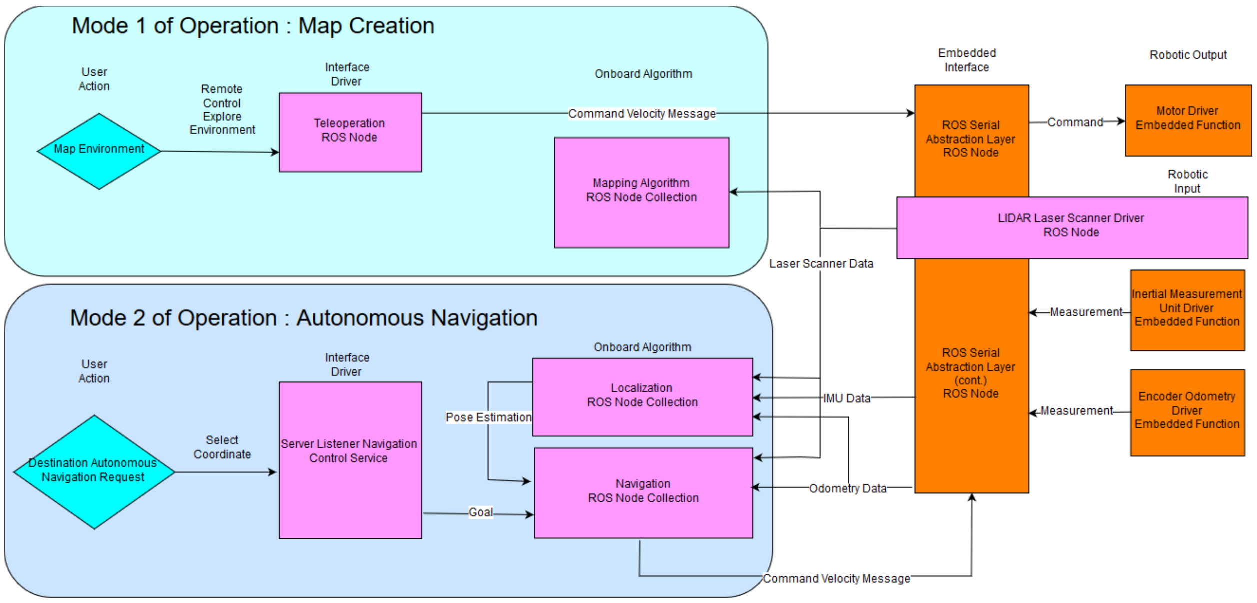
4.3 Inter-Process Communication and Modularization

ROS is middleware software that sits on top of Linux and is used as a collection of software frameworks for robotic control and communication. Packages are a unit of organization of ROS code that contains source code for ROS nodes. Nodes are executables that run the software for the system and have the ability to communicate with other nodes; each node is a system process.

To communicate between nodes, ROS uses a publisher/subscriber model. A node can publish (post data) or subscribe (receive data) from *Topics* that contain ROS *Messages* which are ROS Specific data types built from C++ primitives that are used when publishing or subscribing to a topic.

An example of of such a message would be a "std_msgs/String" ROS message which consists of C++ String type.

4.3.1 Software Architecture Schematic



Mode 1 of Operation : Map Creation
User Action
Map Environment
Remote Control Explore Environment
Interface Driver
Teleoperation ROS Node
Onboard Algorithm
Command Velocity Message
Mapping Algorithm ROS Node Collection
Embedded Interface
ROS Serial Abstraction Layer ROS Node
Command
Robotic Output
Motor Driver Embedded Function
Robotic Input
LIDAR Laser Scanner Driver ROS Node
Laser Scanner Data
Mode 2 of Operation : Autonomous Navigation
User Action
Destination Autonomous Navigation Request
Select Coordinate
Interface Driver
Server Listener Navigation Control Service
Onboard Algorithm
Localization ROS Node Collection
Pose Estimation
Navigation ROS Node Collection
Goal
IMU Data
Odometry Data
Command Velocity Message
ROS Serial Abstraction Layer (cont.) ROS Node
Measurement
Inertial Measurement Unit Driver Embedded Function
Measurement
Encoder Odometry Driver Embedded Function

Figure 4.2: Software Flow Chart

4.4 Raspberry Pi Environment

The Raspberry Pi has ROS Kinetic installed on LUbuntu, a lightweight Linux distribution based on Ubuntu.

4.4.1 Launch ROS nodes on boot

We set up auto-login and ROS node launch procedures on bootup of the Raspberry Pi to execute autonomous car functionality as soon as it is powered on. To do this we modified LightDM settings which is a display manager running in Ubuntu that starts the user sessions and login screen. The lightdm.conf file was edited to set auto-login.

We created a custom script to startup ROS nodes for autonomous driving on boot. This sets up the environment and connects to the web interface, allowing the user to being operation with minimal hassle.

4.5 Teensy Environment

We used the Arduino IDE to program the Teensy 3.6 by using an add-on for Teensy devices called teensyduino. C++ is used to implement our low level libraries used on the Teensy. To enable ROS functionality, a ROS library for the Arduino environment is installed. It uses the rosserial protocol to interface with the Raspberry Pi. More information on the rosserial protocol is provided in the Low Level section below.

4.6 Simulation and Testing

In robotics, it is industry standard to perform a physical simulation of the system being proposed due to uncertainty in the system's success before investing in the costs of development. In particular, for the Indoor Autonomous Navigation System (IANS) we also wanted to install and use the mapping, localization and navigation software to see the feasibility and technique of its usage.

With our chosen framework of ROS, we used the Gazebo physical simulator to create a 3D model of the IANS robotic car. We used a ROS wrapper for Gazebo to send commands to the physical simulation which is generated using the ROS navigation and localization algorithms. This allowed full simulation of IANS. In the simulation, a 25x25 meter room was built with hallways and the robotic car was placed inside this room. A LiDAR scanner with 6 meter range and 360 degree scan was placed on the car (simulating the RPlidar). Data from encoders were supplied automatically by Gazebo's publishing to the odometry topic.

Lastly, we used RViz, a data visualization program which grabs data from both the navigation and simulation to create a visualization of the data (such as map diagram, pose in the map, and an interface to send waypoints). This was necessary as we planned to use RViz with the final product to visualize its position on the map from sensor data.

The main benefit of the simulation is to provide the team with a running prototype of the mapping, navigation, and localization high level algorithms before progressing further in the project. It also trained the team on the intricacies and fine technical details of ROS Mapping, Localization, and Navigation software packages, so that when hardware was obtained (reality), the team was ready test and implement.

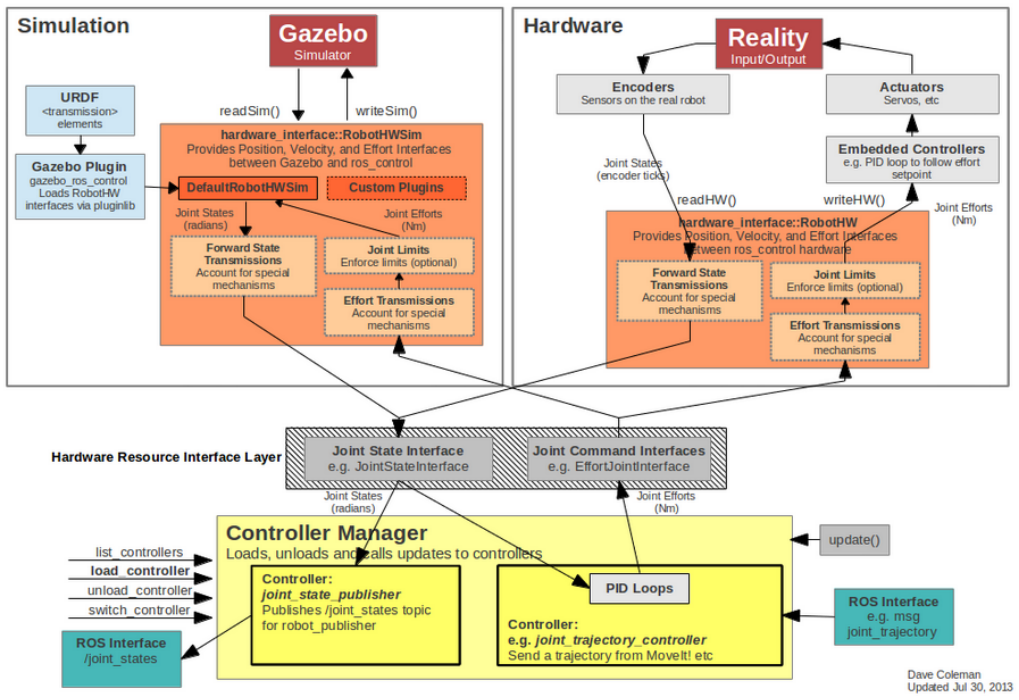

Simulation
Gazebo
Simulator
URDF
<transmission> elements
Gazebo Plugin
gazebo_ros_control
Loads RobotHW interfaces via pluginlib
readSim()
writeSim()
hardware_interface::RobotHWSim
Provides Position, Velocity, and Effort Interfaces between Gazebo and ros_control
DefaultRobotHWSim
Custom Plugins
Joint States (radians)
Joint Efforts (Nm)
Forward State Transmissions
Account for special mechanisms
Joint Limits
Enforce limits (optional)
Effort Transmissions
Account for special mechanisms
Hardware
Reality
Input/Output
Encoders
Sensors on the real robot
Actuators
Servos, etc
Embedded Controllers
e.g. PID loop to follow effort setpoint
Joint States (encoder ticks)
readHW()
writeHW()
Joint Efforts (Nm)
hardware_interface::RobotHW
Provides Position, Velocity, and Effort Interfaces between ros_control hardware
Forward State Transmissions
Account for special mechanisms
Joint Limits
Enforce limits (optional)
Effort Transmissions
Account for special mechanisms
Hardware Resource Interface Layer
Joint State Interface
e.g. JointStateInterface
Joint Command Interfaces
e.g. EffortJointInterface
Joint States (radians)
Joint Efforts (Nm)
Controller Manager
Loads, unloads and calls updates to controllers
update()
list_controllers
load_controller
unload_controller
switch_controller
Controller:
joint_state_publisher
Publishes /joint_states topic for robot_publisher
PID Loops
Controller:
e.g. joint_trajectory_controller
Send a trajectory from MoveIt! etc
ROS Interface
/joint_states
ROS Interface
e.g. msg
joint_trajectory
Dave Coleman
Updated Jul 30, 2013

Figure 4.3: Gazebo ROS and Reality Interface

Gazebo is designed to be interchangable with reality. Specifically, the interface between Gazebo and the Navigation software (GeometryTwist Messages, as well as receiving data from LIDAR and encoders) is analogous to the interface between reality and the Navigation software, which would be a large focus of the team's efforts. Lastly the data obtained from hardware needs to be properly calibrated for the navigation software to achieve a functioning system.

4.6.1 Simulation Procedure

The simulation was run on a Ubuntu Virtual Machine (VMware).

There are three main steps in our simulation procedure.

1. Create simulation environment

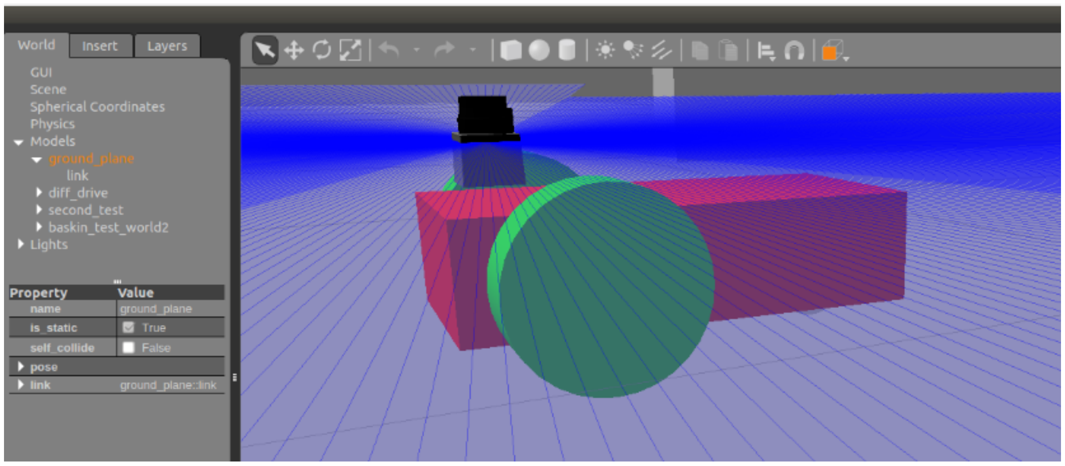
2. Create 2D Occupancy Grid representing environment

3. Autonomously navigate in simulation environment based on map obtained in step 2

The first step was to create the floorplan and robotic car using a Unified Robot Description Format (URDF) model. The floorplan was a 25x25m floor with various hallways (see figure 4.5). It was built using the model editor in Gazebo. The model we created is shown below.

The colors chosen are for high contrast between the wheel and chassis, so that the robot's pose was easy to visualize on screen. The LiDAR Scanner has a scan radius of 6 meters using the hokuyo shape model.

The simulation of the LIDAR was a libgazebo_ros_laser element built into Gazebo. Gazebo comes with various primitives for robotic simulation, such as Lasers, Cameras, IMU modules, as well as a differential car drive primitive (libgazebo_ros_diff_drive) for the robotic car.



Property	Value
name	ground_plane
is_static	☑ True
self_collide	☐ False
pose	
link	ground_plane::link

Figure 4.4: Model of the robotic car differential drive

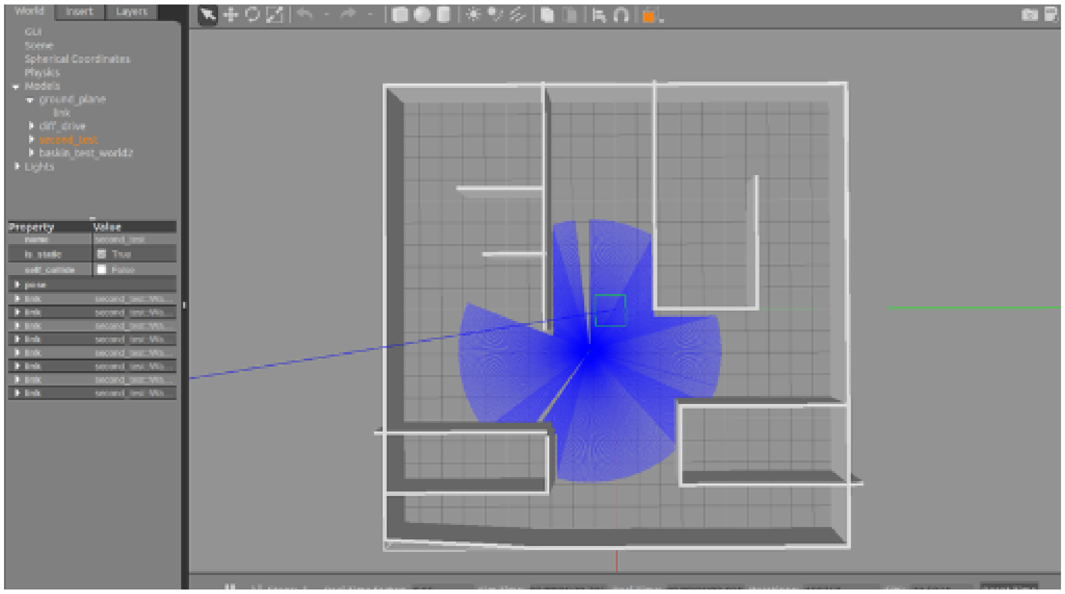


Figure 4.5: Test environment for robotic car. Car is placed in center. Overhead view of simulation.

The second step was to run the SLAM package to create a 2D Occupancy grid map of the environment. In our simulation we used the *slam_gmapping* package as well as a teleoperation package to drive the car around. To create the map (shown below), the simulation was started, as well as *slam_gmapping* package was initiated for the mapping of the environment. Next RViz and the map server was brought up, which helped visualize the process and saved the output of *slam_gmapping*. Lastly a teleoperation node was invoked which allowed us to control the robotic car in simulation using keyboard input. To build the map, the car needed to be driven throughout the environment.

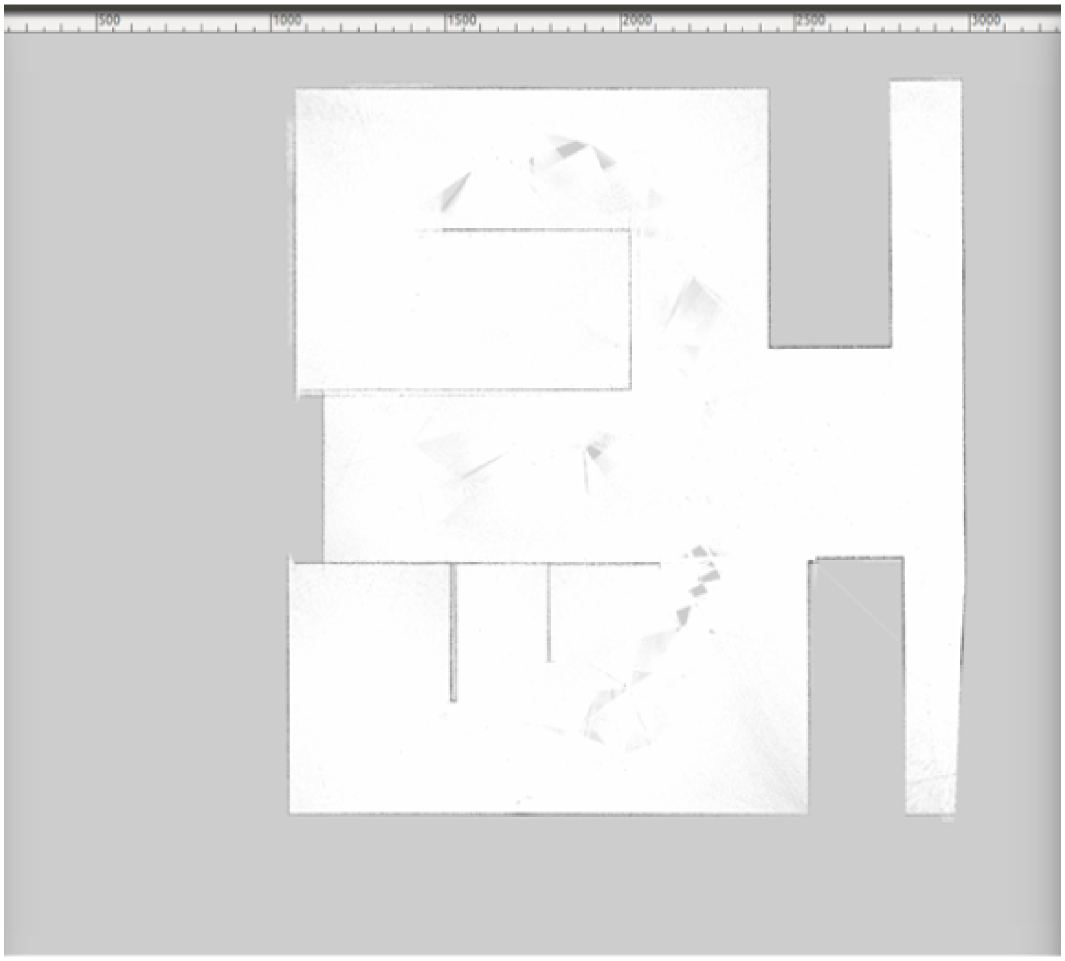


Figure 4.6: 2D Occupancy Grid created based on the environment

The 2D Occupancy grid is stored as a .pgm (Portable Graymap) image and a .yaml file for metadata. In the previous figure, the file was opened in an image editor. Non white pixels in the image represent obstacles. Note that in any map building scenario, there could be small errors. An example of this is in the figure where there are grey spots within the hallway (signifying obstacles). To fix these errors, one could paint the gray portions of the image a white color (RGB: 255, 255, 255).

The third step is to initialize the navigation packages, invoking the following programs in order:

1. Gazebo Simulator

2. Navigation Launch Package
   (a) Map Server
   (b) AMCL Localization Package
   (c) Move_Base Navigation Package
3. RViz Visualization Tool

Reference: http://wiki.ros.org/

Now that the map has been created and saved, the ROS localization packages localize the robot in the environment using LiDAR scan data, and the user can use the RViz graphical user interface to select waypoints for the robot to navigate.

A good experiment was to drop a rolling sphere inside the physical simulation to act as a moving object. This tested the navigation algorithm's local path planning to detect and avoid obstacles.

4.7 High Level Software

4.7.1 Mapping Software

The mainstay of the mapping functionality consists of the *slam_gmapping* package. The simulation outputs laser scan data to *slam_gmapping*, as well as odometry data. *slam_gmapping* subscribes to a *laser_scan topic*, as well as a *tf* Transform topic. *tf* is a ROS builtin library which performs transforms from one coordinate frame to another and provides this service to any ROS Node.

For example, assume that the LiDAR scan data is received from LiDAR unit that is mounted 5 meters in front of the robots center base. A *tf* transform relating this data can be "broadcasted" or published the *tf* topic. Any ROS node that requires this relation, (for example a SLAM node), can request it by using *tf* library to request it from the *tf* topic. In short, *tf* is used in relating map, odometry, and robotic position frames of references.

For *tf* transform, *slam_gmapping* requires the following transforms to be available:

1. *the frame attached to incoming scans* -> *base_link*

2. *base_ link* -> *odom*

The first item is so that *slam_ gmapping* can relate the position of the LIDAR attached to the robot's frame to the Laser scans that it receives. The second item is so that *slam_ gmapping* can relate *base_ link* to odometry information.

Running the data through *slam_ gmapping* results in output to */map* topic, which is a 2D Occupancy grid.

The following ROS Nodes were running during *slam_ gmapping* execution with a *Gazebo* simulation.

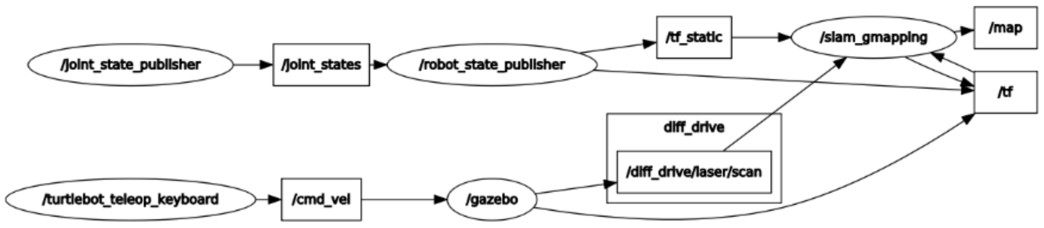


Figure 4.7: ROS Nodes Interaction During GMapping SLAM Execution

From the left side, notice the *teleop_ keyboard* node is being reused from the turtlebot package to publish driving commands to the robot in *Gazebo*. The teleop node takes in keyboard input from the user and publishes *geometryTwist* messages on the *cmd_ vel* topic. As a result, the *Gazebo* simulation will move the robotic car through its environment. It will also publish LiDAR scanner data to the *diff_ drive/laser/scan* topic.

Next, notice the nodes *joint_ state_ publisher* and *robot_ state_ publisher*. These two nodes are involved in the kinematic model representation of the robot from the URDF model. They are not absolutely necessary for the application of our robotic car, but they are there because of their required usage in SLAM algorithms. *joint_ state_ publisher* is used to publish data regarding the position of a robots moving joints. This data is then inputted to *robot_ state_ publisher*, which creates all the necessary transform data for the robot model with the given joint positions. For the robotic car, the joint movements and kinematic representation are very simple, particularly, in the URDF model of the car, the position of the LIDAR unit is placed above the wheels in the robotic car of the Gazebo simulation. The roles of the two state publisher nodes are similar for any application, and are not specific to this discussion, thus it is important to keep in mind the concepts learned about these ROS Nodes as they will be present in other areas of ROS nodes discussions.

Lastly, notice the *slam_ gmapping* node. This node takes in LiDAR scan data along with odometry values (published as transform from *tf* topic) and correlates the changes in odometry value to the current and previous scans taken from the environment. In the context of *ROS*, *slam_ gmapping* receives the LiDAR scan data from *Gazebo*, as well as odometric data represented as a transform from *base_ link -> odom* published by *Gazebo* in the *tf* topic. With these inputs, *slam_ gmapping* publishes the data for the 2D Occupancy grid to the map topic. Data from this topic can be saved and used for future localization and mapping.

4.7.2 Navigation Software

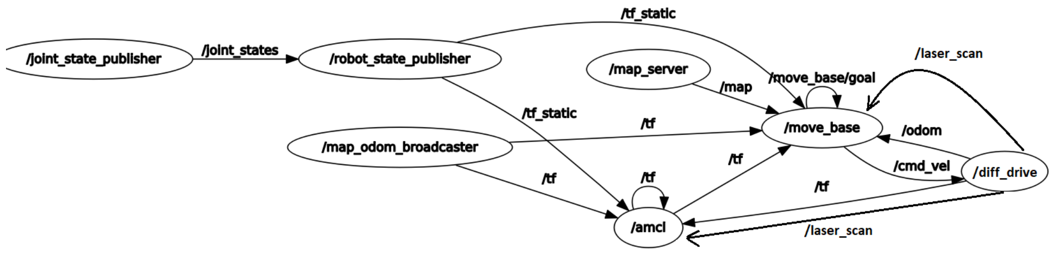


Figure 4.8: ROS Nodes Interaction During Navigation

The Navigation Launch Package consists of three main components.

The *map_ server* node takes in a 2-D occupancy grid map publishes its data to the */map* topic. The message in the */map* topic is of *nav_ msgs/OccupancyGrid ROS* message type, which translates into an *int8* array, where each element is a number from -1 to 100, where each number represents the level of occupancy in the map, and its index, the position of the map, where -1 is unknown.

The *Adaptive Monte Carlo Localization (AMCL)* package contains the algorithm for determining pose within the 2d occupancy grid. It is a particle filtering algorithm, where it calculates many pose estimates based on map information which creates a particle cloud. Afterwards the unlikely poses are filtered out and the particles which best match the current readings from

LIDAR and wheel encoder data are kept. The accumulation of the probable particles determines the robot's position within the map. *AMCL* takes into account odometry data drift, and tries to use laser pattern matching to fix this error over time. Whenever the robotic car moves and obtains sensor readings, AMCL recalculates the particles and creates an even better pose estimate.

The *move_ base* component is the mainstay of the navigation algorithm. *move_ base* is a package which integrates various aspects of navigation, each of which adheres to a standard API defined in the *ROS Navcore* standard. This standard interface allows the user to easily implement their own algorithms for each aspect of robotic navigation (Local and Global planning, Recovery, Costmaps). Move base consists of the following aspects or components:

1. Costmap Software: *costmap_ 2d*
2. Navigation Standard Interface: *nav_ core*
3. Local Planner : *base_ local_ planner*
4. Global Planner: *navfn*
5. Costmap Recovery: *clear_ costmap_ recovery*
6. Rotation Recovery Software: *rotate_ recovery*

Reference: http://wiki.ros.org/

The *move_ base* node subscribes to the *move_ base_ simple/Goal* topic which is a *geometry_ msgs/PoseStamped* message consisting of pose information. This is the input to the algorithm, and sets the destination.

move_ base also publishes to *cmd_ vel* topic which is a *geometry_ msgs/Twist* ROS message. This is the output of the algorithm and gives turn-by-turn directions on how to get to the destination. This ROS message consists of an angular heading in which the robot should be oriented, and a linear component for where the robot should traverse. This is key for navigation, as the main problem to solve is to convert each message in *cmd_ vel* to low level hardware commands for the robot's navigation.

The costmap consists of static objects (like walls, specified in the 2-D occupancy grid), as well obstacles detected by sensors. A 2-D occupancy grid is inputted to the Costmap software to "inflate" each obstacle in the occupancy grid (black dot in pgm file). Inflation means to make each obstacle appear bigger than it is. This allows for a configurable version of the 2-D occupancy grid where a wall in the occupancy grid would become thicker such that path planning algorithm would be configured to stay away from a certain distance from obstacles. The navigation algorithms navigate based on the costmap.

Next, the Local and Global planner (*base_local_planner* and *navfn*) are used to create a "Global" path, and a "Local" path. The Global Planner *navfn*, uses Djikstra's shortest path algorithm to determine a path. The Global path is the path from the robot's current position to the destination. The Local path is the path where the robotic car should follow to stay on course to follow the Global path. The Local path planner's main job is to generate velocity commands that the robotic car should execute to stay on course to the Global path. Another job of the Local path planner is for obstacle avoidance. For example if an object is plotted on the costmap, the Local path planner would adjust the car's course heading, avoid the obstacle, and readjust velocity to follow the Global path.

The Costmap Recovery node (*clear_costmap_recovery*) is used to clear obstacles in the costmap outside a user defined radius. Obstacles should not remain in the costmap after a certain amount of time since they can be assumed to be dynamic. For example, if a person is sensed by the robot, it will be plotted on the costmap. After avoiding the obstacle, the obstacle should be removed from the costmap since the person would (likely) not be present or their position would have changed in the environment.

Lastly, the Rotation Recovery node (*rotate_recovery*) is used for when the robot detects too many obstacles in its costmap for navigation (if it is surrounded by many people for example). It is a procedure for the robotic car to maximize its sensor readings to obtain a solution for navigation. The robotic car will stop, and rotate in place if possible to first clear and update the obstacles in its costmap, to find a suitable route for navigation. If this recovery fails (too many objects detected in costmap, meaning robot cannot navigate), it will signal failure.

4.8 Low Level Software

4.8.1 rosserial

rosserial is a protocol that sends ROS over a serial port. We use *rosserial* as the communication protocol between the Raspberry Pi and the Teensy 3.6. With *rosserial*, the Raspberry Pi sees the Teensy and it's libraries as another node within ROS infrastructure. *rosserial* is a metapackage which means it is a group of multiple packages in a single logical location.

One package it contains is called *rosserial_python*. This package contains Python implementation of the host side *rosserial* connection. It handles the setup, publishing and subscribing for the Teensy with the node *serial_node.py*. This node spins up subscribers and publishers. To run rosserial type this command with a specified serial port, for examples with /dev/ttyACM0:

```
rosrun rosserial_python serial_node.py /dev/ttyACM0
```

Another package it contains that we use is *rosserial_arduino*. This contains the client side implementation of rosserial for the Teensy. It is designed for microcontrollers that can be programmed with the Arduino IDE. *rosserial_arduino* requires a ROS library called *ros_lib*, to be installed within the Arduino libraries to allow ROS publishing and subscribing functionality.

4.8.2 Motor Interface

We wrote a motor library to integrate the motors within Teensy low level code base. The motors library is initialized by creating an object of the class with the Teensy pin values that are connected to H-bridge as parameters. The motor library functions are used to rotate left and right motors forwards, in reverse, and to stop the motors. These functions are called when the Teensy receives an *lmotor* or *rmotor* message from the Raspberry Pi. These messages contain the motor power values from -255 to 255: negative values call reverse motor functions, positive values call forward motor functions and zero call brake motor functions. The function definitions are documented in the API section of the Appendix.

5 | System Hardware

5.1 Introduction

The first iteration of our robot was an Ackerman steering platform with a Raspberry Pi and a motor controller. We controlled the drive speed and steering using keyboard inputs and custom built ROS nodes. Although this served as a good way to familiarize team members with ROS and Python, the Ackerman design would be more problematic than a differential drive robot in later stages of development. Since many ROS packages for navigation, mapping, and LiDAR interfacing were designed for differential drive robots, we made the choice start with a new differential drive robot design.

The computing units chosen for this design were a Raspberry Pi 3 as our high level computing device, and the Teensy 3.6 microcontroller as our low level computing device.

5.2 Differential Drive Robot Design

Shown here is the SolidWorks design of the robot. The base plate of the robot is 16" by 12". The motor mounts were designed so that they could simultaneously fasten the motors with screws while supporting the top platform. The robot is constructed from medium density fiberboard (MDF) which we fabricated using a laser cutter. The rear end of the robot is supported with a ball bearing caster.

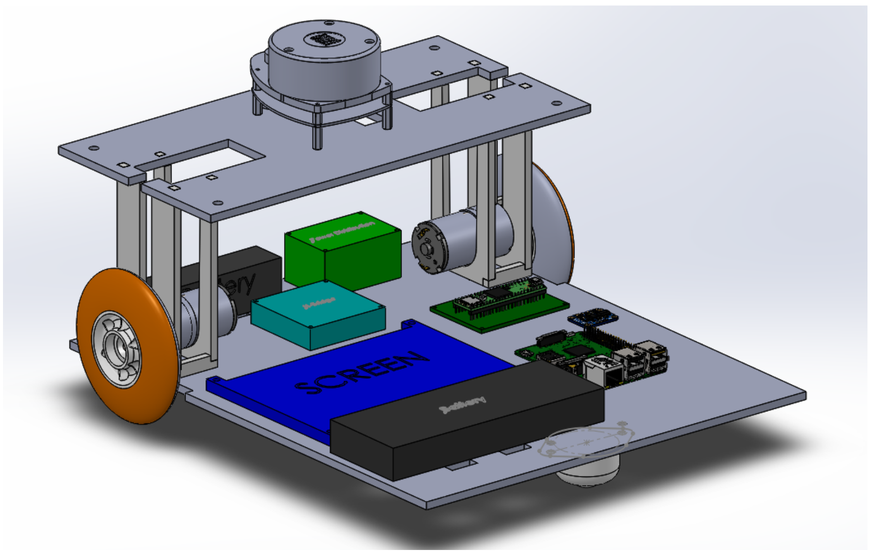
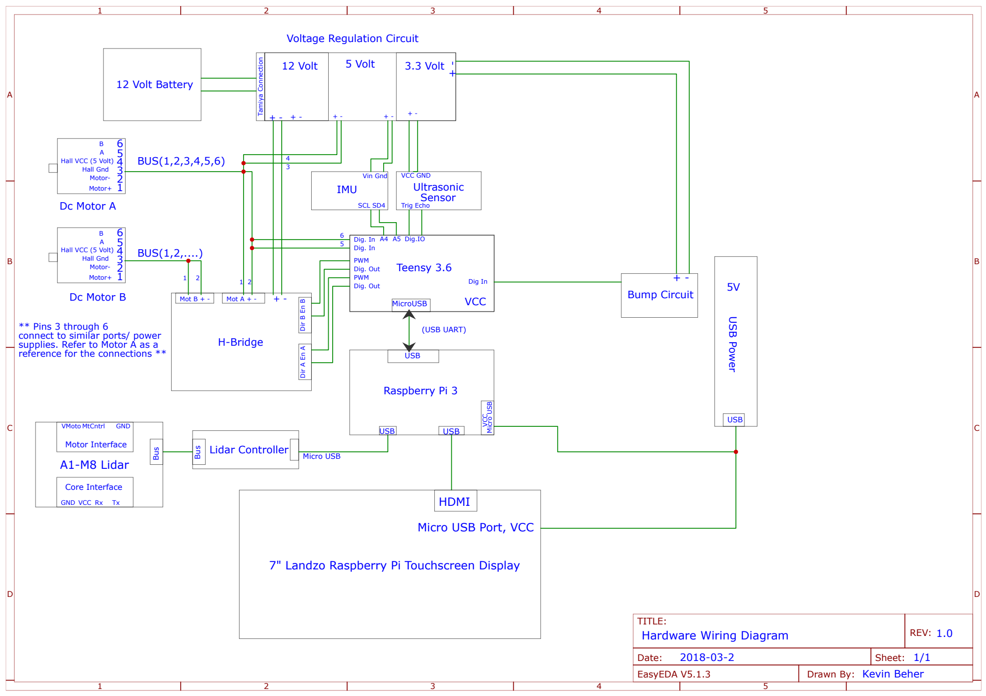


Figure 5.1: Solid Works Design of Robot

5.3 System Wiring Diagram

Shown here is our system wiring diagram. This diagram includes the specific types of connectors used for certain modules. If no type of connection is stated, this means the connection is a jumper wire.

4/2/2018 CMPE 123



blob:https://easyeda.com/8a744994-adff-44ac-945b-7726a19ae909 5/5

Figure 5.2: System Wiring Diagram

5.4 Power Analysis

We calculated the power consumption of our system to be approximately 25 watts. We use a 20,000 mAh USB power bank with a maximum 18W output to power both of the microcontrollers, the touch screen, and the LiDAR. The motors are powered using a 12 volt Ni-Mh 2000mAh battery. The approximate amount of time our robot can run is 1 hour and 25 minutes. Shown below is how this amount of time was calculated. The total current load on the Ni-Mh battery is 1.4 amps. This amperage was calculated using the information on our electronics datasheets.

$$\frac{2000mAh}{\sum Currents} = 1.42$$

Table 5.1: Power Consumption Analysis

Device	Volts	Amps	Watts	Watts Consumed
Uxcell 166 RPM 12 V (1)	12	0.7	8.4	25.09009
Uxcell 166 RPM 12 V (2)	12	0.7	8.4	
RP A1-M8 Lidar	5	0.1	0.5	
RP A1-M8 Lidar	5	0.3	1.5	
7" Display Screen	5	1	5	
BNO055 IMU	3.3	0.0123	0.04059	
HC-SR04 Active Sounding Sensor	3.3	0.015	0.0495	
Raspberry Pi 3	5	0.24	1.2	
Teensy 3.6	12	0.08	0.96	

The power distribution circuit supplies power from a 12 volt battery and steps it down to 5 volts and 3.3 volts. This board is used to power the motors with its 12 volt line, and the encoders with its 5 volt line. The voltage regulators chosen for this application are CUI V7805 switching regulators. These regulators were chosen for their 90 percent efficiency, 2 amp output, and ease of installation. They were chosen over linear regulators because linear regulators only have roughly 50 percent efficiency, a lower amp output, and generally require a heat sink. In the end we chose these regulators because we believed it would save us time in the long run by providing a higher amp output and the ease of installation.

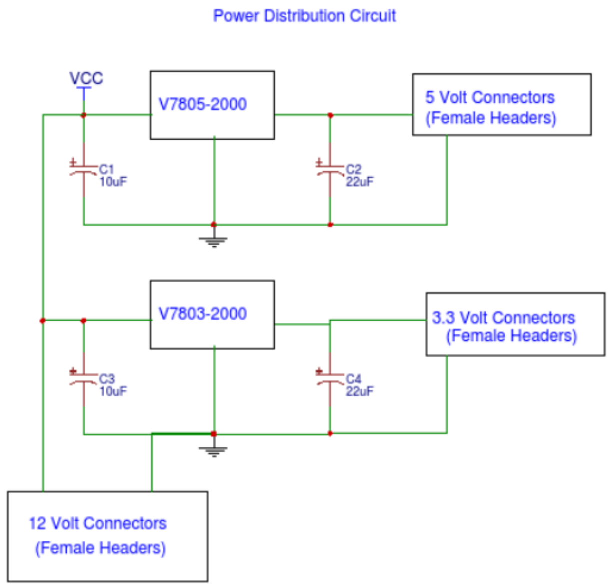


Figure 5.3: Power Distribution Schematic

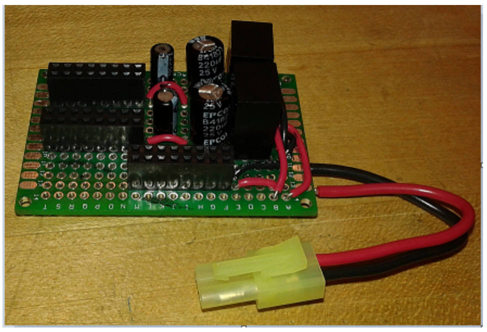


Figure 5.4: Power Distribution Board

5.5 Onboard Hardware

The Raspberry Pi 3 was chosen as our high level micro controller for its USB port and resource availability. Additionally, all of the team members were familiar with this microcontroller from previous course work. The main reason for choosing the Teensy 3.6 as our low level microcontroller was the availability of resources for interfacing with sensors like the IMU, the encoders, and sonic sensors. For our purposes like controlling the motors, polling encoders, and interfacing with the IMU, this platform works perfectly as its processing speed is fast, and most of these tasks are not computational heavy. Additionally, the high number of pins on the Teensy 3.6 give us room for expandability in the future for ultrasonic sensors, bump sensors, and more.

The motors we chose for this project were encoded Uxcell motors. We chose to use the DROK L298 Dual H Bridge Motor Driver to control our motors because it is capable of handling larger amounts of current than most other H-bridges. The encoder provides a resolution of 64 counts per revolution of the motor shaft. For every count the encoders output two square waves from terminals a, and b which are out of phase. The voltages of these waves range from 0 to VCC. Shown below are the resulting waveforms for a clockwise rotation. Here, signal A is blue and signal B is yellow.

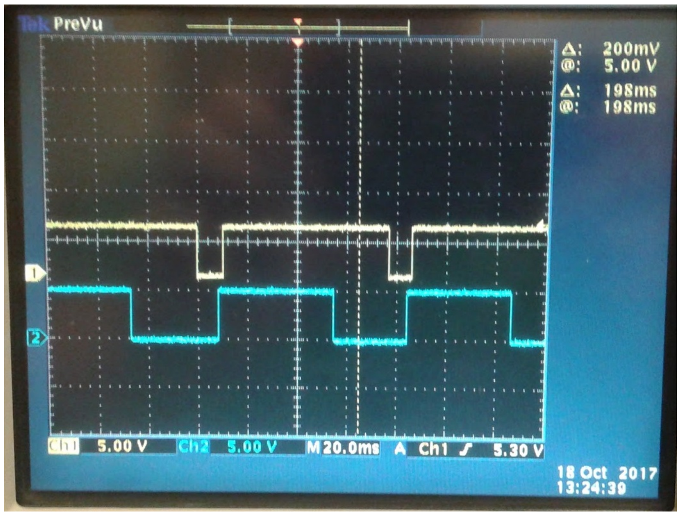


Figure 5.5: Encoder Wave Form. Signal A is blue, Signal B is yellow

In addition to the encoders we also use a LiDAR and an IMU for localization. The Adafruit BNO055 IMU was chosen for of its capability of internally processing magnetometer, gyroscope, and accelerometer data and returning a stable three axis orientation. Since our SLAM algorithms benefit from precise measurements of magnetic field, angular velocity, and linear acceleration we chose to integrate this IMU into our design. The Slamtek

360 degree rangefinder LiDAR is our main navigation and obstacle avoidance sensor. It is used to relate laser scan data to a predetermined map using a particle filter.

6 | Conclusion & Results

6.1 Acknowledgements

We would like to extend our sincere gratitude towards Professor Anujan Varma, Sam Mansfield, and Sargis Yonan for their support and mentorship throughout this project. Additionally, we would like to thank Baskin Engineering Lab Support for their contributions to our development.

Appendices

Bibliography

[1] Kirby RL 1 , Ackroyd-Stolarz SA , Brown MG , Kirkland SA , MacLeod DA. (1994) Wheelchair-related accidents caused by tips and falls among noninstitutionalized users of manually propelled wheelchairs in Nova Scotia.

[2] Emma M. Smith, Brodie M. Sakakibara, William C. Miller. (2016) A review of factors influencing participation in social and community activities for wheelchair users. Disability and Rehabilitation: Assistive Technology 11:5, pages 361-374.

[3] Thrun, Sebastian, *Learning metric-topological maps for indoor mobile robot navigation*, Artificial Intelligence 99.1 (1998): 21-71.

A | Low Level API

These functions are used for communication between the Raspberry Pi, the Teensy 3.6, and the robot sensors. The Raspberry Pi computes the route path and sends appropriate velocity messages to the *differential_ drive* ROS package. The *differential_ drive* package contains a PID controller that sends motor control messages to the Teensy. These messages contain Float32 values.

The Teensy reads from the left and right motor encoders and sends the output to the Raspberry Pi to use for route correction and localization. The values sent provide the number of pulses from the start of navigation. The definition of *lwheel* and *rwheel* messages are as follows:

```
Int16 encoderVal
```

The Teensy publishes IMU readings to the Raspberry Pi to use in the localization/navigation algorithm. The message definition of an IMU message is as follows:

```
geometry_msgs/Quaternion orientation
# Row major about x, y, z axes
float64[9] orientation_covariance

geometry_msgs/Vector3 angular_velocity
# Row major about x, y, z axes
float64[9] angular_velocity_covariance

geometry_msgs/Vector3 linear_acceleration
# Row major x, y z
float64[9] linear_acceleration_covariance
```

Pi ↔ Teensy connection

```
| Author: Juan Huerta
| Param: Float32 message that was published to the lwheel topic
| Return: Void
| Remark: The Teensy calls this callback function when an message
|    containing a PWM value, is published by the Pi to the topic lmotor.
|    When the message is a positive PWM value, Motor library function
|    left_motor_forward is called. When the message is a negative
|    PWM value, Motor library function left_motor_reverse is called.
void lmotor_callback(const std_msgs::Float32& msg);

| Author: Juan Huerta
| Param: Float32 message that was published to the rwheel topic
| Return: Void
| Remark: The teensy calls this callback function when an message
|    containing a PWM value, is published by the Pi to the topic
|    rmotor. When the message is a positive PWM value, Motor library
|    function right_motor_forward is called. When the message is
|    a negative PWM value, Motor library function right_motor_reverse
|    is called.
void rmotor_callback(const std_msgs::Float32& msg);

| Author: Juan Huerta, Kevin Beher
| Param: None
| Return: Void
| Remark: This function reads encoder pulses using Arduino Encoder
|    library and publishes the total number of pulses to lwheel
|    and rwheel ROS topics at a rate of 10HZ
void ROS_encoder_publisher();

| Author: Juan Huerta, Kevin Beher
| Param: An empty message published to
| Remark: Reset encoder values to zero
void encoder_reset_callback(const std_msgs::Empty& reset_msg);

| Author: Juan Huerta
| Param: None
| Return: Void
| Remark: This function calls the Read_IMU_msg() function from the IMU
|    API and publishes the data to imu ROS topic
void ROS_IMU_publisher();

| Author: Juan Huerta
| Param: Digital Pins values on the Teensy used as PWM and
|    motor direction and direction enables
| Remark: This constructor sets the pin numbers given as private
|    variables of the class
Motors(int right_pwm_Pin,int rmotor_direction_pin1,
int rmotor_direction_pin2, int left_pwm_pin,
int lmotor_direction_pin1, int lmotor_direction_pin2);

| Author: Juan Huerta
| Param: motor_speed, 0-255
| Return: Void
| Remark: This function sets PWM duty cycle of the right
|    motor and sets directions pins to rotate motors forward
void Motors::right_motor_forward(std_msgs::Float32 motor_speed);

| Author: Juan Huerta
| Param: motor_speed, 0-255
| Return: Void
| Remark: This function sets PWM duty cycle of the right
|    motor and sets directions pins to rotate motors in reverse
void Motors::right_motor_reverse(std_msgs::Float32 motor_speed);

| Author: Juan Huerta
| Param: motor_speed, 0-255
| Return: Void
| Remark: This function sets PWM duty cycle of the left
|    motor and sets directions pins to rotate motors forward
void Motors::left_motor_forward(std_msgs::Float32 motor_speed);

| Author: Juan Huerta
| Param: motor_speed, 0-255
| Return: Void
| Remark: This function sets PWM duty cycle of the left
|    motor and sets directions pins to rotate motors in reverse
void Motors::left_motor_reverse(std_msgs::Float32 motor_speed);

| Author: Juan Huerta
| Param: None
| Return: Void
| Remark: This function sets motor direction pins to zero
|    and PWM duty cycle to zero to brake left motor
void Motors::left_motor_brake();

| Author: Juan Huerta
| Param: None
| Return: Void
| Remark: This function sets motor direction pins to zero
|    and PWM duty cycle to zero to brake right motor
void Motors::right_motor_brake();

| Author: Juan Huerta
| Param: None
| Return: Void
| Remark: Initialize IMU input pins.
initialize_IMU();

| Author: Juan Huerta
| Param: None
| Return: Array of vectors
| Remark: Reads data from IMU using Adafruit library, which will
|         then be published  to the Raspberry Pi
Vector<3>[] read_IMU_msg();
```

A.1 ROS Library Template Functions

These are ROS template functions that are used by the Teensy. They are used to define what topics the Teensy will publish and subscribe to and what message type they will be handle. In the Teensy main function, we call a ROS function, **spinOnce**, that handles the communication callbacks. The Subscriber template functions get called when data is published to a topic the function is subscribed to; they then call callback functions that are documented in the API above. The Publisher template functions is called when the ROS publish function, $[topicname].publish(\&msg)$, is called within the Teensy code base.

```
| Author: Juan Huerta
| Param: The topic to subscribe to
| Param: Callback, the callback function that gets executed
|       when a message is received from the Pi
| Return: The lmotor message that was received from the Pi
| Remark: This is a built in ROS function that subscribes to
|         the lmotor topic and the lmotorCallback function is
|         then executedon the Teensy when that topic is published
|    by the Pi.
ros::Subscriber<std_msgs::Float32t> lmotor_sub("lmotor",
&lmotorCallback);

| Author: Juan Huerta
| Param: The topic to subscribe to
| Param: Callback, the callback function that gets executed when
|        a message is received from the Pi
| Return: The rmotor message that was received from the Pi
| Remark: This is a built in ROS function that subscribes to
|        the rmotor topic and the rmotorCallback function is
|        then executed on the Teensy when that topic is published
|        by the Raspberry Pi.
ros::Subscriber<std_msgs::Float32t> rmotor_sub("rmotor",
&rmotorCallback);

| Author: Juan Huerta
| Param: The topic to subscribe to
| Param: Callback, the callback function that gets executed when
|        a message is received from the Pi
| Return: The empty reset_encoders message that was received
|        from the Pi
| Remark: This is a ROS template function that subscribes to
|        the reset_encoders topic. The encoder_reset_callback
|        function is then executed on the Teensy when that topic is
|        published by the Raspberry Pi.
ros::Subscriber<std_msgs::Empty> reset_encoder_sub("reset_encoders", &encoder_reset_

| Author: Juan Huerta
| Param: The topic to publish to
| Param: A built in ROS message data type
| Return: Unknown
| Remark: This is a built in ROS function that publishes to the
|         lwheel topic from the Teensy
ros::Publisher lwheel("lwheel", &lwheel_msg);

| Author: Juan Huerta
| Param: The topic to publish to
| Param: A built in ROS message data type
| Return: Unknown
| Remark: This is a built in ROS function that publishes to a
|         rwheel topic from the Teensy
ros::Publisher rwheel("rwheel", &rwheel_msg);

| Author: Juan Huerta
| Param: The topic to publish to
| Param: A built in ROS message data type
| Return: Unknown
| Remark: This is a built in ROS function that publishes to the
|         imu topic from the Teensy
ros::Publisher IMU_data("imu", &IMU_msg);
```

B | Server API

This API describes the communication between the robot and the cloud server.

Server-Side Communication API

This section describes the functions used by the web server to interact with the robot. Note: the server-side API is written in Python.

```
| Author: Kyle Ebding
| Return: void
| Remark: This function is called when the user presses the
|     "Toggle motorDisable" button on the web interface. It
|     sends an MQTT message to the topic "robot/motorDisable"
|     which the robot is subscribed to.
toggle_motor_disable();

| Author: Kyle Ebding
| Param: an int representing the ID number of a destination
| Return: void
| Remark: This function is called when the user selects a
|    destination on the web interface. This function sends an
|    MQTT message to the robot telling it to use the function
|    input as its new destination.
send_destination(dst);

| Author: Kyle Ebding
| Return: void
| Remark: This function is called when the user presses the
|    "Start Mapping" button on the web interface. It sends an MQTT
|    message "robot start mapping" to the topic "robot/map\_msgs"
|    which the robot is subscribed to.
```

```
start_mapping();

| Author: Kyle Ebding
| Return: void
| Remark: This function is called when the user presses the
|    "Stop Mapping" button on the web interface. It sends an MQTT
|    message "robot stop mapping" to the topic "robot/map\_msgs"
|    which the robot is subscribed to.
stop_mapping();

| Author: Kyle Ebding
| Param: an int representing the ID number of the map
| Return: void
| Remark: This function sends an MQTT message to the robot on
|    the topic "robot/map_msgs." It is called when the user
|    selects the map on the web interface corresponding to the
|    robot's location (e.g. Baskin Engineering ground floor). The
|    robot takes this information and uses it to download the map
|    via a separate connection over HTTP since MQTT is not well
|    suited for file transfer.
send_map(mapID);
```

Robot-Side API

This section describes the functions used by the robot to communicate with the server. Note: the robot-side API is written in C++.

```
| Author: Kyle Ebding
| Return: integer value representing success (0) or failure (-1)
| Remark: This function tells the robot to upload the PGM file it has
|    generated to the cloud server by connecting to an endpoint
|    that receives a map file and gives it an ID number.
int upload_map();

| Author: Kyle Ebding
| Return: integer value representing success (0) or failure (-1)
| Remark: This function is called when the map topic receives a
|    publication, which is caused by the server calling sendMap().
|    This function connects to an endpoint that uploads the PGM
|    file specified by the mapID to the robot.
int download_map(char* mapID);
```

```
| Author: Kyle Ebding
| Return: void
| Remark: This function is called when the string "robot start mapping"
|    is received on the mapping topic. It runs a shell script that
|    starts the necessary ROS nodes for the mapping operation.
void start_mapping();

| Author: Kyle Ebding
| Return: void
| Remark: This function is called when the string "robot stop mapping"
|    is received on the "robot/mapping" topic. It runs a shell script
|    that stops the ROS nodes used in the mapping operation then
|    converts the map to .png format and uploads it to the cloud
|    server via an HTTP Post request to an endpoint designed
|    for receiving maps this way.
void stop_mapping();
```

C | Bill of Materials

Table C.1: Bill of Materials

Part Name	**Price**	**Quantity**	**Shipping**	**Total Spent so**
CUI 3.3 Voltage Regulator	$10.00	1	$3.75	$433.71
CUI 5.0 Voltage Regulator	$10.00	1	$0.00	
A1-M8 Lidar	$200.00	1	$0.00	
7" Raspberry Pi Touch Screen	$42.88	1	$0.00	
Wheel - Shaft Mounting Hub (6mm)	$4.95	2	$3.95	
Battery 2000 mAh	$23.92	2	$0.00	
DROK L298 Dual H Bridge Motor Driver	$16.00	1	$0.00	
Tenergy Battery Charger	$15.00	1	$0.00	
Arduino	$34.89	1	$0.00	
BNO05 Absolute Orientaion IMU	$34.95	1	$4.55	
Uxcell 12 Volt Encoded Motor 166 RPM	$22.09	2	$0.00	