Aerojet FAR Robotic Payload – RC Rover

Justice Cordova, James Ellison, Wesley Fletcher, and Joshua Kissoon

Dept. of Electrical and Computer Engineering, University of Central Florida, Orlando, Florida, 32816-2450

***Abstract* — At the dawn of a Second Space Age, there is high demand for robust, small remote-controlled and/or semi-autonomous vehicles capable of exploring the surfaces of planets and other astronomical objects. This paper presents the design of a small, ruggedized, semi-autonomous rover that is capable of traversing difficult desert terrain and relaying live video and telemetry data to a ground station. Designed to meet the size and weight constraints of a rocket payload, this rover is capable of remote control, and can still operate in a high latency environment, where delays in transmission of control commands do not impair rover operations as it can navigate short distances without direct guidance from the ground station.**

***Index Terms* — autonomy, LoRa, mobile robotics, navigation, ROS**

I. INTRODUCTION

Aerospace company Aerojet Rocketdyne has organized a multidisciplinary project where UCF students compete in the Friends of Amateur Rocketry (FAR) 2022 competition. As one of the payload teams, we have designed and developed a rover payload for the rocket, under the supervision of Dr. Felix Soto Toro. To meet FAR requirements, our rover must be remote-controlled, continuously transmit video and operate at a range of up to 600 meters. Inspired by modern rovers used in space exploration like the Perseverance, the rover is a ruggedized design made to handle rough, sandy terrain and reliably communicate with the ground station at long range, even in a high latency environment where real-time control cannot be achieved.

II. SYSTEM OVERVIEW

To meet the challenges of difficult communication and potentially high latency, we made the decision to develop our RC rover into a semi-autonomous mobile robot. In this way, the rover could be trusted to competently navigate short distances, not requiring constant real-time intervention from the operator. Doing this also opened the door to more advanced robotics concepts and software challenges.

We envisioned the basic operation of the rover to be as such: after landing and deploying from the canister (whose design was outside our scope), the operator would have access to a ground station GUI and live video feed. Armed with telemetry data and the video feed transmitted by the rover, the operator could command the rover by either providing manual movement commands in the form “one unit forward, turn one unit left, two units reverse” or by providing a local-frame geometric coordinate and allowing the rover to make its own way there.

To implement this vision, we needed to provide a mobile robot capable of seeing it’s environment and understanding its own place within that environment. So, we needed to implement sensors to provide that information: depth clouds from a stereo camera to “see” obstacles, wheel encoders to track our translation, IMU to track our orientation, and so on. We then had to supply that data back to the operator and allow the operator to send commands: handled by a 5.8GHz video transmitter and a pair of LoRa transceivers. Then, we created a custom interface for the operator to command the rover with, which we call the Ground Station software in this paper. The final piece of the puzzle is the software that enables the rover to act out the operator commands. Our software stack makes extensive use of the Robot Operating System, a broad ecosystem of robotics software and utilities designed for this purpose. The contextual behaviors of the rover are determined by a finite state machine, wherein each state makes extensive use of the packages and functionalities of ROS to execute behaviors.

The requirements of the FAR competition state that our rover be capable of RC control and live video streaming. In terms of actual motion, the competition only requires ten feet in any direction. To satisfy these requirements, we developed an intentionally slow mobile base, using high-torque and low-RPM motors. The high torque makes us more capable of dealing with the potential inclines and gradients of uncontrolled dessert terrain, while the low RPM makes it easy to keep track of ourselves as we move, aiding our semi-autonomous navigation.

III. SYSTEM COMPONENTS

In this section, we’ll go into details about the selection of our components and their intended purposes. A simplified block diagram of the system as designed is shown in figure 1.

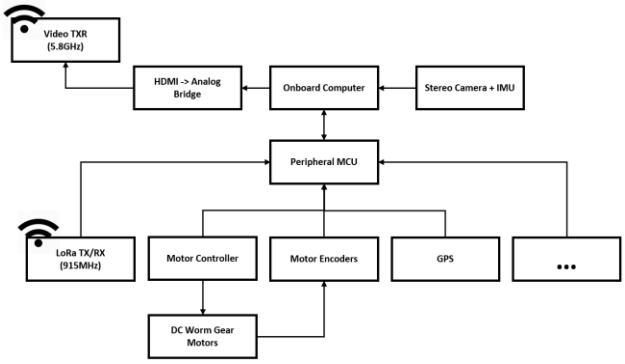


Figure 1: a component level block diagram of our rover

A. Rover Components

Single-board Computer (SBC): the SBC will be the "brain" of the rover, providing the needed computational resources for intensive applications like image processing and autonomy software. For this component, we selected the Jetson Nano Dev. Kit for its high computational performance and broad port selection. The Jetson Nano can run a full OS (a custom fork of Ubuntu called L4T), making it compatible with the Robot Operating System (ROS) and easing development.

Further, the Jetson Nano provides two CSI-2 camera interface connectors, which allows us interface with two cameras or a single stereo camera without needing an extra component. Another benefit of the Jetson Nano: it was the cheapest of the options we considered, since we already had one available to us, making it effectively free.

The Jetson Nano does have some disadvantages: it has a large footprint, is quite heavy, and contains a significant amount of unneeded functionality that adds weight and complexity to our design. Ultimately, the benefits of the platform outweigh these negatives.

Peripheral Microcontroller: the peripheral MCU is intended to provide the SBC with a single "point-of-access" to all the peripherals onboard the rover. As such it needs to have sufficient IO to support these components, as well as a simple communications interface. For this task, we selected the Raspberry Pi Pico. The Pico provides a high clock speed (~125MHz), plenty of IO, and a built-in USB controller for programming and serial communication.

While it is possible to interface with all peripheral components using the built in GPIO headers on the Jetson Nano, we decided to include the peripheral MCU to abstract the specifics of interfacing with the peripherals away from the SBC. This mitigates our dependence on the Jetson Nano specifically to be our SBC. As we're currently contending with a global chip shortage, we wanted to maximize our flexibility in case of damage to our SBC. The peripheral MCU ensures that even if we were to swap the Jetson Nano with a completely different device, so long as it has a USB port, we won't lose any of our rover's basic functionality.

Stereo Camera (and IMU): The requirements of the FAR contest state that we must have a live video stream, necessitating some sort of camera to be included in our design. To simplify our design, we selected a stereo camera, the Waveshare IMX219-83 Stereo Camera, to both provide live video feed(s) and provide environmental information to build our world model. The Waveshare Stereo Camera is designed to be used with the Jetson Nano Dev. Kit and its two CSI-2 camera connectors, simplifying integration.

By processing the stereo image disparity, we can generate a "depth cloud" to be used for generating our map. In this way, we can build our autonomy system with just the single sensor, rather than a combination of LiDAR, optical, and ultrasonic sensors as is typical for mobile robots.

GPS Module: The GPS module will provide world-fixed, discrete, absolute position data for localizing our rover. This not only allows us to track the rover during and after deployment, but can also serve to augment our localization solution, potentially correcting the inherent build-up of positional error that is typical with continuous positioning sensors, like an IMU.

For the GPS, we selected a NEO-6M module. It communicates over a simple UART connection, provides a reasonable position estimate (within about 20 feet, in our experience), and is cheap and readily available.

Wheel Encoders: wheel encoders will provide information on the velocity of our wheels by way of "ticks." By tracking the number of positive or negative ticks of the wheel and knowing the wheels physical properties, we can derive the wheel velocities, and thus the velocity of our robot.

For wheel encoders, we selected CUI Devices ENC-AMT102V. These are incremental (relative rotation only) capacitive encoders with configurable ticks-per-revolution.

Motor Controller: the motor controller will be responsible for providing commands to the wheel motors. Motor commands will be determined by the rover software processing user commands and generating PWM duty cycles. These duty cycles will be passed to the peripheral MCU, which will handle direct communication with the motor controller.

For this purpose, we selected the Cytron MDD10A, a brushed DC motor controller rated for 10A of continuous current. The MDD10A uses four pins

Motors: for motors, we were looking for a large amount of torque (for dealing with potential rugged terrain) at a relatively low RPM. Low speeds are easier to track via odometry and make the robot easier to control. To fit these needs, we selected worm gear motors, rated at 12VDC and

250RPM and providing 10Kg*cm of torque. There are a few benefits to worm gear motors: first, they have a high output torque to input current ratio; and second, they "brake by default," i.e., when the motor is not energized, it will resist movement to the point of mechanical failure. Finally, the motors selected have dual output shafts, one on either side of the gear box. This allows us to mount the wheel encoders directly to the motors themselves, without interfering with the wheels.

Battery: The battery is an off-the-shelf, 7.4 V 4600mAh Lithium Polymer (LiPo) battery. It is in a sturdy case that will protect it from damage during rover operations. A battery management system is connected to the terminals, preventing an over discharge of the battery during deployment.

Video Transmitter (TX): The video transmitter is a small 5.8 GHz transmitter which has an analog input for video. Because the stereo camera only had digital output and the video transmitter had an analog input, we had to make an analog bridge using an HDMI to analog adapter. The video transmitter and analog bridge are compact, and field testing confirms a transmission range of ~600m.

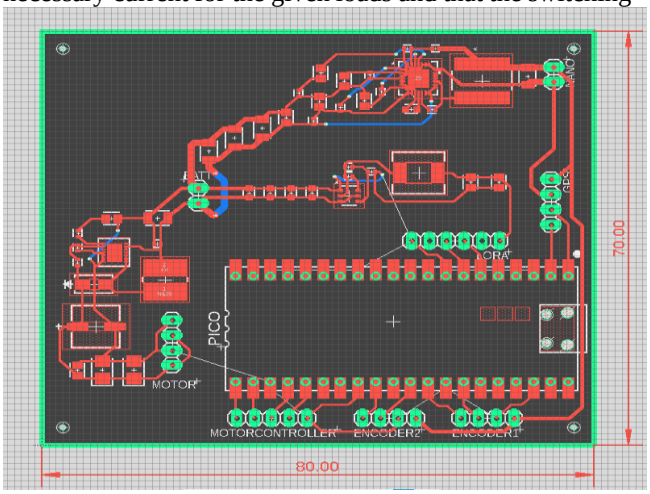
Radio Transceiver (TX/RX): The rover uses Reyax RYLR896 transceiver modules for remote control and data transmission. These modules are very compact, satisfying size constraints for the rover. They transmit on the 33-centimeter band, a band that doesn't require a license and does not interfere with rocket telemetry on the 70-centimeter band. These modules use the LoRa protocol, which is designed for long range, low power applications, perfect for rover communications. The modules interface with the rover hardware over UART and are configured and controlled with a small set of AT commands. The ground station and rover software handle the AT commands, creating a more user-friendly interface for control of the rover. Field testing has confirmed that the transceivers meet range requirements.

Connectors, Fasteners, and Misc.: for connectors to be used on the rover, we decided to go with locking connectors wherever possible. For data and low-current connections, this meant JST-XH connectors. These connectors use crimped wire connections and a semi-locking "shroud" to protect the connections and hold the male-female pair together strongly. For power and high-current, we used Amass XT30 connectors. On top of being rated at a (much more than sufficient) 30A of continuous current, these locking connectors use solder cups, allowing us to easily use stranded wires and thus increasing our resilience to strain and vibration.

IV. ELECTRICAL DETAILS

For our PCB design, we had two major concerns to address: the power electronics, and interfacing with our MCU.

The rover design requires significant power electronics to manage the various subsystems. The subsystems operate at three different voltage levels: 3.3 V, 5 V, and 12 V. We have two buck converters and one boost converter on our PCB. Due to supply chain issues and the chip shortage, we could not rely on standard TPS series Texas Instruments chips in our design and had to resort to more elaborate DC to DC converter designs. Our design was significantly constrained by the limited availability of chips and passive components. The biggest challenge was the 5 V DC to DC converter for the Nano: we had to use a 28-pin chip with 20 passive components instead of a 9-pin TPS series chip with 7 passive components. We had to make other substitutions for the 3V and 12 V DC to DC converters, but we made sure that trace widths were wide enough to carry the necessary current for the given loads and that the switching



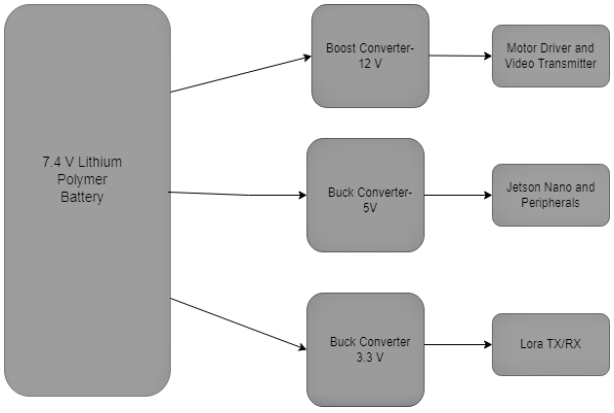


Figure 2: a) PCB Schematic, b) power diagram

frequencies would not interfere with the rover transmissions.

For interfacing with the MCU, we provided a set of through-hole pads connected to the Pico GPIO pins for each of our peripherals, as well as a footprint for our MCU to sit in. During assembly, all peripheral connections were populated with JST-XH connectors. The Pico itself is connected to a set of female pin headers, allowing us to potentially hot swap in case of damage to the microcontroller.

The final PCB design, shown in figure 2a, consists of the three DC to DC converters and the Raspberry Pi Pico and its associated pins.

V. MECHANICAL DETAILS

Frankly, mechanical design was not our focus for this project. As ECE majors, the motto was "as little mechanical design as possible." So, when designing our rover's enclosure and drivetrain, we focused on adapting commercial-off-the-shelf components in tandem with custom 3D-printed components to fit our needs, rather than a complete, ground-up design

The robot is differential drive, making use of two front "drive wheels" directly attached to electric motors and a back pivot point. Usually, this pivot is implemented with a caster wheel, but since we're designing for use in sand, we're instead using custom-designed "skis" designed to ride over sand. These skis were modelled in Fusion 360 and printed out of PLA filament. The modelled skis are shown below, in figure 3a. Placed on the back of the robot in place of the caster wheel, they allow the robot to rotate about its center to turn in place.

To attach the wheels to the D-shaped motor shafts, we needed custom shaft couplers with space for set screws, printed in PLA, shown below in figure. 3b.

The enclosure for the robot was adapted from commercial-off-the-shelf "project boxes," made of ABS plastic and rated at IP67. Modified via drill and rotary tool, these boxes were cut to fit our needs. For motor mounts, a hole-drilling guide was modelled, and 3D printed, then used to drill motor mounts in the side of the project box. Doing it this way removed the need for purchasing or printing motor mounts, using the sides of the box instead. Cable pass-throughs were in the lids and sides of the boxes to maximize interior space.

For the members of this group, this was a first experience in the building of a mobile robot. There were quite a few hard lessons learned in the process of getting to a working prototype.

An important consideration in the design of our robot was its footprint: it needed to fit a rather strict size requirement,

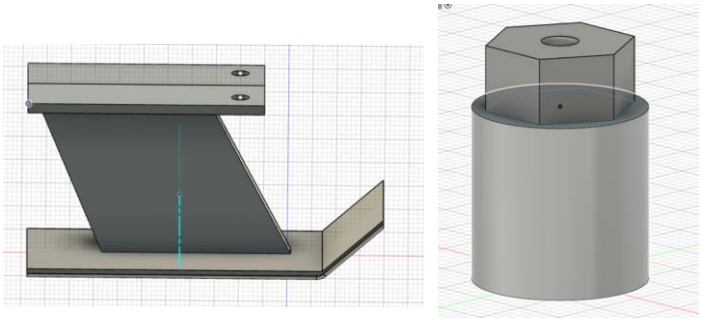


Figure 3: a) "ski" model, b) motor shaft adapter

and we were trying to fit a lot of functionality (and therefore components) into that small space. Unfortunately, we under-estimated the amount of space taken up by standard electronics connectors, like HDMI connectors and USB plugs. By failing to properly account for the footprints of the connectors and cables, we ended up with a final product that is quite difficult to assemble and disassemble cleanly. The fix to this was to route cables outside of the robot wherever necessary, running antennae, USB cables, and power lines on the exterior of the robot. While this potentially compromises the integrity of the box, and almost certainly makes the enclosure not fully IP67-compliant, it was necessary to ensure that everything would fit in the space provided.

Another issue: some quirk of our mechanical assembly causes a rightward drift when the robot is meant to be driving straight. While it's easily corrected by the operator in real-time control, it's proven to be quite the thorn in our side for our semi-autonomous behaviors. It's really hammered home the lesson that software isn't enough to implement complex systems. Seemingly small issues in other areas, like mechanical assembly, can drastically affect the overall effectiveness of our solution. In this case, by making our robot much less accurate at tracking its own position.

VI. SOFTWARE DETAILS

The software developed for this project can be split into three sub-systems: the rover autonomy software, the embedded peripheral MCU code, and the ground station.

B. Embedded Peripheral MCU Software

The peripheral MCU is intended to provide the SBC a single point of access to the array of peripheral devices on-board the rover. The embedded software on the peripheral MCU interfaces with each of the components attached to the MCU (e.g., wheel encoders, LoRa transceiver, motor controller, etc.) and exposes an interface where the rover can send commands and receive data.

The peripheral MCU software is written in C, with convenience functions and macros provided by the Raspberry Pi Pico SDK. While the use of Pico-specific functions makes our code less portable, it provides the advantage of faster development and a more robust (read: less likely to mysteriously break) implementation.

The interface between the rover software and the peripheral MCU is implemented as a simple serial connection (115200-8-N-1), sending plain-text messages in pre-defined message formats. On either side of the serial connection, message prefixes determine how a specific message is handled. For example, the "$MTR" prefix denotes a motor PWM command to be carried out on the peripheral MCU by sending PWM commands to the motor controller, while the "$CMD" prefix denotes a message from the Ground Station, received by the LoRa transceiver and passed through to the SBC for interpretation. On the MCU side, each message received is passed through a switch-statement that determines how the message is handled based on the prefix.

The details of how the peripheral MCU interfaces with each component follow:

The LoRa transceiver: the LoRa transceiver communicates via a UART (115200-8-N-1) protocol. Since messages can be sent/received at any time during the lifecycle of the peripheral MCU program, reading from and writing to the LoRa transceiver is handled on the Pico's second core. This approach allows us to constantly poll for newly available data without sacrificing the responsiveness of other components. When a message is read in, it's added to an inter-core queue for asynchronous processing by our main core. Likewise, when our main core generates a message to be written to the LoRa receiver, it's added to an inter-core queue to be handled asynchronously by the secondary core.

The motor controller: the Cytron MDD10A motor controller onboard the rover is configured to use four pins for controlling the motors: two digital (binary) GPIO pins for controlling motor directions, and two PWM pins for controlling motor speeds. Using functions provided by the Pico SDK, incoming motor commands (those with the "$MTR" prefix) are unpacked into variables and passed to a convenience function that sets the pin outputs.

The wheel encoders: the wheel encoders generate two waveforms on channels A and B. By configuring a timer at a high frequency (~250Hz), we can poll both channels and generate wheel ticks. These wheel ticks get passed to the SBC during the main loop, at a more reasonable 20Hz, after being prepended with the "$ENC" prefix.

Ultimately, the wheel ticks that are output by the encoders get turned into RPM readings on the SBC, then wheel velocities. After being fused into a "unicycle model" of our robot's current position and velocity, these wheel velocities are an integral part of our odometry solution, and thus our mobile navigation solution.

The GPS module: the GPS module communicates via UART (9600-8-N-1). Data is formatted as standard NMEA sentences. Software interrupts are configured to fire every time data is available to be read from the GPS (roughly 1Hz). After being read in, data is prepended with the "$GPS" prefix and passed to through the serial connection to the SBC be processed by the rover software.

C. Rover Autonomy Software

The rover autonomy software is a ROS-based software stack built around a simple finite state machine and a publish-subscribe message passing model. This software stack provides for the "advanced" functionality of the rover: semi-autonomous waypoint navigation, real-time telemetry data, and image processing, among other functionalities. Ultimately, it's this system that is responsible for carrying out operator commands to reach targets.

The finite state machine: The autonomy software is built around a simple finite state machine (FSM), pictured below in Figure 4., implemented using the smach Python library. The FSM coordinates the actions of the rover software throughout the phases of operation. In the BOOT state, the FSM ensures that we have access to all needed input streams (e.g., from the cameras, from the IMU, from the GPS, etc.). STANDBY waits for command inputs from the user, most often in the form of a waypoint. After receiving a waypoint, the FSM transitions to WAYPOINT, where semi-autonomous navigation to that waypoint begins, with the help of the move_base ROS package. Successful navigation to the waypoint or user-initiated preempt return us to STANDBY to wait for new commands. The WARN state fires when we come across issues during execution of other states. Finally, END is intended to spin down all associated processes and even power-off the rover, if needed.

Along with the FSM, several ROS-enabled "nodes" are running simultaneously, processing data and inputs from the various sensors (e.g., stereo_image_proc and gscam for camera streams) as well as converting data between useful forms. These nodes are designed take advantage of the publish-subscribe architecture of ROS, processing inputs and generating outputs wholly independent of one another. If one node fails, the rest stay alive.

Some of the important custom nodes are listed here, in no particular order:

- **pico_bridge** (Python): the "rover-end" of the serial connection to the Pi Pico. Converts ROS messages to appropriate (read: prefixed) strings for the peripheral

MCU and reads strings from peripheral MCU into ROS messages.

- **imx219_stereo_cam_node** (CPP): interfaces with the Waveshare stereo camera, publishing camera streams and IMU data as ROS messages to be consumed by the rest of the system.
- **telemetry_pub** (python): packages information about the current state of the rover into a custom-defined Telemetry message that then gets pushed to the peripheral MCU for transmission to ground station.
- **encoder_odom** (python): given encoder ticks by way of the peripheral MCU, generates an Odometry message containing an estimate of our current position and speed.

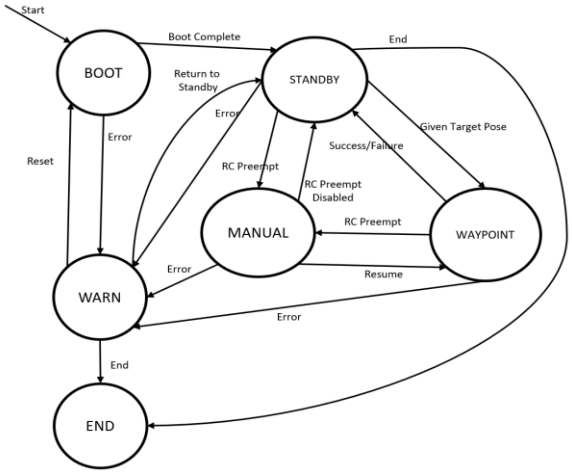


Figure 4: a diagram of the FSM implemented in our rover software stack

- **motor_control** (python): a node that converts the contents of a ROS Twist message into appropriate commands for our motor controller (using the Motors custom message definition).

Waypoint navigation: A key component of our approach to this design is waypoint navigation, i.e., providing the rover with a waypoint and trusting it to navigate there with minimal intervention, contained within the WAYPOINT state of the FSM. A benefit of this approach is that it is inherently resilient to latency issues. Using low-power radio at significant distances with no guaranteed line-of-sight can introduce significant communication delays. If we empower the rover to handle simple waypoint targets, we reduce the number of commands we need to send and responses we need to process on the ground station, and thus ameliorate issues of responsiveness and latency.

Our approach to waypoint navigation makes extensive use of the ROS Navigation stack, a collection of ROS packages, messages, and services designed to provide autonomy to robots [1]. Central to this navigation approach is the OccupancyGrid (costmap) world model, which contains a simple geometric representation of the immediate area round the robot. With this "map" of our environment, we can then use a standard path planning algorithm (implemented within the Navigation stack) to determine the best means for navigating through the map. With that, the move_base package within the Navigation stack will generate commands for our robot base to execute with our motors. Much of the software efforts for this portion of the project are aimed at providing the correct input streams and configurations needed for smooth functioning of the Navigation stack. Figure 5 depicts the details of our WAYPOINT state's navigation approach , including inputs and outputs.

The Navigation stack's OccupancyGrid package is costmap_2d. costmap_2d requires inputs that specify, among other things, the geometry of the robot (tf [2]) and its sensors and sensor information about our surroundings. The main sensor stream for our costmap will be the depth information provided by our stereo camera's disparity image processed by stereo_image_proc [3]. Combined with the relative geometry of our sensors, we can generate a map of our surroundings.

We will be using the default path planner to generate paths through our map. An important input to the path planner is robot odometry ("where are we relative to starting position?"). We will generate this odometry through sensor fusion, using the robot_localization package's ekf_localization_node, which implements an Extended Kalman Filter (EKF) to generate more accurate localization information. The EKF will be used to fuse odometry information from: the motor encoders odometry generated by encoder_odom, the IMU data provided by imx219_stereo_cam_node, and (potentially) GPS information; generating a local estimate of our position. For this application, the rover is designed to travel relatively short distances. So, we made the decision to use

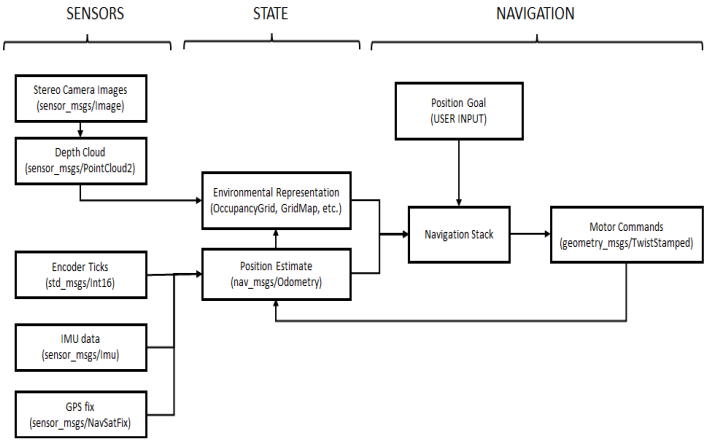


Figure 5. Diagram of the navigation components of WAYPOINT state

the local map only, and thus the local planner and local position estimates only.

The final piece of the puzzle is move_base, which generates motor commands from a given path plan and position estimate. We need only provide an interface that converts the output of move_base (called /cmd_vel) into appropriate commands for our motors. This is handled by our motor_control node.

With these components in place, we can provide a "goal" to our robot via Command message and trust the WAYPOINT state of our FSM to negotiate with move_base and navigate the robot, without our further intervention.

Message Definitions: A fundamental benefit of ROS is the large library of standardized ROS messages, each with a pre-defined format. These messages can be used to pass data back and forth between nodes. However, for the custom functionalities of our rover, we implemented a few bespoke message types, summarized in Figure 6. By defining custom messages using the ROS message functionality, we get the benefit of using the ROS API for message passing, rather than having to implement a custom IPC solution for our data. The Command message acts as the primary user input to our FSM and its associated nodes. As of the writing of this document, it allows the user to provide a waypoint target, RC message, and a few command flags. If additional information is needed, it's trivial to update the message definition (.msg file) and re-compile the message. The Telemetry message, shown in figure 6b, captures relevant real-time status information about the rover, such as what state the FSM is in, for transmission to the ground station. The Motors message, shown in figure 6c, is used to communicate PWM data in a format that matches the motor controller configuration. Finally, the RC message, shown in figure 6d, contains simple integers allowing the user to specify movement in four directions. Upon processing, the rover will then move X "units" in that direction, with units being defined as either time intervals at a certain PWM speed, or a distance/rotation interval.

```
geometry_msgs/Point target
rover_msg/RC rc
######## flags ########
std_msgs/Bool start          # init FSM (if all is well)
std_msgs/Bool cancel         # cancel current actions
std_msgs/Bool shutdown       # stop robot execution
std_msgs/Bool rc_preempt     # preempt action with motor cmd
std_msgs/Bool pose_preempt   # preempts with waypoint target
```

```
# state machine state
std_msgs/String state
# relative (odom) pose
geometry_msgs/PoseStamped pose
# gps fix
sensor_msgs/NavSatFix fix
```

```
std_msgs/Byte   dir1
std_msgs/Int8   pwm1
std_msgs/Byte   dir2
std_msgs/Int8   pwm2
```

```
int8 forward
int8 reverse
int8 left
int8 right
```

Figure 6. Custom ROS message definitions: a) Command message, b) Telemetry message, c) Motors message, and d) RC message

The simulation: as a means of speeding the development of the above functionality while we worked on the physical prototype, we developed a software simulation of our robot using the popular Gazebo simulator. Gazebo allows us to write software that leverages the same message definitions and interfaces as our physical robot, thus creating a high-fidelity test bed for our software stack. Most of our software was developed in this simulation environment before being "ported" into the messier conditions of the real-world.

As convenient as this simulation can be, it truly is a double-edged sword. There's no such thing as a perfect simulation of real-world conditions, and the use of the simulator abstracted away some significant real-world issues that needed to be dealt with. Chief among these issues was sensor inputs. In the simulation environment, getting access to sensor inputs is as easy as configuring them in the robot's configuration file. From there, we have essentially perfect sensor data, with no noise and no latency concerns, and without any of the (significant) pains of getting this data from real-world sensors. Frankly, the use of the simulator caused us to vastly underestimate just how much work it would be to get, for example, the IMU of our actual robot to output usable data. While this likely isn't a groundbreaking revelation to the reader, it was a lesson we had to learn: the real world is quite a bit messier and more difficult to handle than a simulation can prepare you for.

C. Ground Station Software

Ground Station: is the command center used to receive and relay controls to the rover. It will relay instructions to the rover and retrieve live video and GPS data. The host machine running the Ground Station software will be a laptop that will use Ubuntu 20.04 LTS. We will control the rover and receive telemetry using our custom GUI (Graphical User Interface).

GUI (Graphical User Interface): is a system of interactive visual components for computer software. The GUI was developed using PyQt5. Qt is a widget toolkit for creating cross-platform graphical user interfaces. PyQt5 is a cross-platform GUI toolkit that provides Python bindings for Qt v5. PyQt5 has a tool called QtDesigner to design the front-end by drag and drop method so that development can become faster and easier because it allows us more time for developing reliable communication. PyQt5 helped streamline the start of the ground station by making it easier for us to design a GUI. Once the initial front end was done it was, we needed a map to display the rover's current

position. The interactive map was generated using an open-source map library called Pyqtlet.

The ground station GUI, shown in figure 7, has three tabs. The first tab is the controls/map section, the second tab is the communication, and the third tab is settings/debugging. The first tab deals with the initial setup and rover telemetry. For all intents and purposes, this is the ground station's primary tab. To begin operating the rover, the LoRa port must first be selected. This will configure the LoRa and establish a connection with the rover. Afterwards, one of the following control modes can be selected: Blind Drive, Manual 1, and Manual 2. Blind Drive takes two latitude and longitude coordinates and transmits it to the rover. In terms of accuracy for longitudinal and longitudinal positioning it is down to 5 decimal places for accuracy. Manual 1 will relay XYZ coordinates, as well as rover state parameters, to the rover. Manual 2 allows travel by specifying the number of units and cardinal direction in which the rover should move; these directions are relative to the rover's orientation. Any incoming telemetry will be shown below the controls. Further down, we have two buttons that will pan the map to the position of the ground station and rover. The last button will toggle the map style between a minimalistic, dark map and a realistic, world map. The map shows the current location and path of the rover. The second tab allows us to see the state of communication between the rover and ground station. It will also tell us if the message needs to be retransmitted or the connection is lost. The third tab allows us to change settings related to the LoRa serial port, LoRa configuration, and map. By default, the most optimal LoRa parameters are set. If needed, we can use this tab for debugging hardware or communication issues.

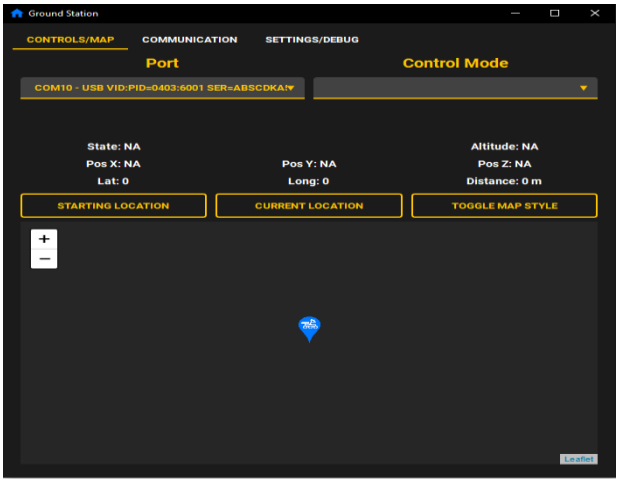


Figure 7: Ground Station GUI

ABOUT US

Justice Cordova is a senior at the University of Central Florida. He plans to graduate in May of 2022 with a Bachelor of Science Degree in Electrical Engineering. His interests include RF electronics and control systems, and he plans to continue his studies with a MSEE.

James Ellison is a senior at the University of Central Florida. He plans to graduate in May of 2022 with a Bachelor of Science Degree in Computer Engineering.

Wesley Fletcher intends to receive a Bachelor's in Computer Engineering in May of 2022. His interests include mobile robotics and large-scale autonomous systems. During his time at UCF, he served as the President of the Robotics Club. After graduation, he will begin work as a Robotics Software Engineer for a consulting firm in MA while pursuing further education.

Joshua Kissoon is a senior at the University of Central Florida. He plans to graduate in May of 2022 with a Bachelor of Science Degree in Computer Engineering. After graduation, he plans to obtain his Master's in Computer Engineering, specializing in VLSI design.

ACKNOWLEDGEMENT

Our group would like to thank our faculty advisor, Dr. Felix Soto-Toro, for his guidance throughout this project. Further, we'd like to thank Drs. Richie and Wei for their support during Senior Design.

We'd also like to thank Aerojet Rocketdyne Coleman Aerospace for sponsoring us, as well as UCF's MAE department for providing the opportunity to work on this multidisciplinary project.

Finally, we'd like to thank the Robotics Club of Central Florida for donating lab space, tools, components, and shoulders to cry on.

REFERENCES

[1] E. Marder-Eppstein, "navigation," *ros.org*, 2020. [Online]. Available: http://wiki.ros.org/navigation. [Accessed: 12-Apr-2022].

[2] T. Foote, E. Marder-Eppstein, and W. Meeussen, "tf2," *ros.org*, 2019. [Online]. Available: http://wiki.ros.org/tf2. [Accessed: 12-Apr-2022].