NASA Challenge Team
Final Report

Ben Bernhard, Riley Egan, Brendan Kopf, George O'Sullivan, & Cara Ravasio

Table of Contents:

1 Introduction (about 3-5 pages)

As humans begin to spend more time in harsh, extraterrestrial environments, new technologies for applications often reserved for Earthbound missions must emerge. One such mission is the search and retrieval of some known object by a team of robots. The first segment of this mission, the search, is what motivated the NASA Centennial Challenges Program (CCP) Space Robotics Phase 2 (SRC2). The original challenge is for virtual simulations of robotic systems, but we adapted it for physical rovers. Moreover, we focus specifically on the electrical engineering aspects of the mission and only require a trivial rover mechanical design. Essentially, our modified version of this project tasks us with designing and constructing a system of two autonomous rovers capable of navigating through a partially unknown lunar environment, finding a predetermined object, and estimating the position of the object relative to the rovers' starting location. Each aspect of this design reference mission (DRM) is largely directed by a couple of main design constraints.

Firstly, communication latencies between Earth and the rovers and the limited bandwidth that exist in such a channel dictate that the robotic system be able to function completely autonomously for long periods of time. While long distance communication between the Earth and Moon is outside the scope and budget of this project, we still require that the rovers be autonomous.

Additionally, the location of the object will likely be in a dimly lit or completely dark area (e.g. in a crater or on the dark side of the moon). So, the rovers must be able to function in these lighting conditions.

Furthermore, we consider a time sensitive mission, in which the object must be located as quickly as possible.

Lastly, NASA pays special attention to the radiation and thermal environments in which the device must operate. Both of these environmental hurdles will affect the processing and sensing capabilities of the solution. While thermal concerns are well within our scope, we do not have the resources to properly address radiation concerns, so we simplify this constraint by only requiring simple shielding.

We propose a two rover system to accomplish the mission outlined in the problem description. These two rovers work together to find and report the location of the object of interest in the following way.

First, each rover searches a different area of the moon for the target object while remaining within communication range of one another. Each rover navigates the environment, keeping track of its location relative to its starting location and avoiding obstacles detectable by a range finder.

Once the rover detects the object using object recognition technology, the rover reports the picture to Earth or a nearby lunar gateway for confirmation. Once the object in the image is confirmed to be the object of interest, the rover estimates the object's position using the rangefinder data and motor encoder data and transmits this location to the other rover.

When the other rover receives this signal, it travels to the communicated location, finds the object of interest, and makes its own estimate of the object’s location. Using these two independent estimates of the object’s location, a final estimate will be made and recorded. This concludes the mission.

An overview of a given rover in the system is illustrated in Figure 1.

Usually, lunar missions would require the rovers to have a long term power supply (generally nuclear). However, we consider the question of a reliable long term power source outside the scope of our project since our focus is on object identification, autonomy, and communication.

We will address radiation and temperature concerns by operating our rovers only on the dark side of the moon. Since the rovers are already expected to operate in dark environments, this is a reasonable simplification. The dark side of the moon will shield us from the moon’s higher temperatures and most of the strongest radiation from the sun. With only low temperatures and ambient radiation to worry about, we can reasonably use simple radiation shielding and escape the need for electronics working at high temperatures.

We will demonstrate basic swarm communication by creating two robots to communicate with each other. In theory if two robots can work together, a larger swarm could communicate in the same general way. We will be using Zigbee for our communication protocol.

To successfully identify and locate the prespecified object of interest, the rovers will patrol their paths and use a camera and flash light to identify obstacles and potential objects. The distance between the rover and obstacle or object will be determined by implementing a laser rangefinder (see figure 1). If a potential candidate for the object of interest is found, a picture will be taken for identification purposes. A convolutional neural net will be taught to identify the specific object of interest; we plan to use OpenCV for this step and train it using simulated pictures. If the target is determined to be the object, then it's location will be recorded and communicated to the other rover.

While the rover is travelling through the partially known terrain, there may be some objects in the rover's path that the rover needs to avoid. We plan on using the laser rangefinder as the obstacle detection for the rover. The laser rangefinder will be directed at obstacles in the rover's path so the rover knows the location of the obstacles and can take the proper measures to avoid unknown obstacles.

We plan on having the rovers cover a partially known terrain which means the rover can have a predetermined path programmed in. The rovers will be deployed from a known starting location. The partially known environment will have the basic topology of the moon so we can avoid mountains and craters for our rover paths. We will have the rover traverse this terrain while avoiding any obstacles that may come up and then moving back to the programmed path.

Due to the global pandemic, we were unable to construct our final design of the rovers. However, given our experience with the subsystems we speculate that our final design would have met our expectations. We can thank the very sophisticated and expensive LIDAR system we planned to implement for our optimism. We speculate that the most likely part of our design to underperform would be the motor control subsystem if only because we are inexperienced in designing mechanical structures and accounting for all possible sources of stress on our driving system.

2 Detailed System Requirements (several pages)

Given time and monetary restrictions, our project involves the minimum number of rovers to be considered a swarm (two rovers). However, our system should be easily scaled up, so our demo of two rovers provides a basic proof of concept for a larger system. It is possible to think of each of our rovers as a node in the swarm which communicates with it's adjacent nodes so the swarm is completely integrated.

PIC32:

We used a PIC32 on our custom board that is responsible for low-level motor control, stepping down the power source voltage, and low-level communication between the two robots. The low-level motor control consists of applying PWM motor drive signals to individual motors according to high-level commands sent from the Raspberry Pi via serial communication. The power will be received from the LiPo battery and stepped down to various voltages according to Figure 1 by using DC-DC converters.

Raspberry Pi:

The Raspberry Pi reads and interprets the data from the IMU, Decawave and Camera subsystems. The Pi communicates via I2C with our custom PCB. The PCB relays the range finder information, information from the servo motors and communication from the zigbee module. The Pi communicates with the IMU through UART and the Decawave module through SPI. The camera communicates by CSI. The camera module we are using comes with the Pi and it works natively with the Pi. The Pi is also in charge of doing the math for finding the location of the rovers. The Pi receives information from the Decawave and calculates the location of the rovers based on this information.

DecaWave:

We are using the Decawave 1001 Module which gives the position of each rover relative to 4 known and fixed anchors. According to decawave, the x-y position is usually accurate to within 20cm of the actual location. So, we should be able to meet our requirement of determining an object’s position to within 2 meters of its actual location. The decawave module uses time of flight sensing to determine the position. This meets the constraint being feasible in a lunar environment, not relying on GPS.

Zigbee:

For our local communication between our rovers we are using a Zigbee module. Zigbee is a local communication protocol that can use mesh networking, P2P networking, and several other protocols. We are using the ZIgbee modules in a mesh network. Although we are only demonstrating two rovers, the mesh networking provides a scalable solution. The Microchip SAMR30M18A Zigbee module has a max range of 400 ft and operates at 2.4 GHz. The Zigbee module communicates over SPI with the PIC32.

Pi Camera & Computer Vision:

The camera is used to identify whether an object is an obstacle or an object of interest. The camera is constantly scanning the area in front of the rover and using OpenCV to locate objects. The camera is attached to the Raspberry Pi which runs OpenCV for real time computer vision.

LIDAR:

The range finder is used to find the distance from the rover to the object of interest and for obstacle avoidance. Once the camera identifies an object, the camera will analyze the object to determine if it is an object of interest. If it is not an object of interest, the rover's path will adjust accordingly. If the object is the object of interest, then the range finder will be pointed at the object and the distance measured. The exact location of the rover will be known with our location system. With the location of the rover and the distance from the rover to the object, the location of the object of interest will be easy to find. The range finder is able to communicate with the Pi via I2C.

IMU:

We are using an IMU (BNO055) that provides estimation for the angle of the rover. The IMU includes the 3 standard IMU sensors: gyroscope, accelerometer, and magnetometer. However, a magnetometer would not work on the moon, since there is no magnetic field on the moon. Therefore, we will first attempt to obtain accurate angle estimation through the gyroscope and accelerometer, only. Our IMU choice provides the output of all three sensors separately. This way, we are able to choose between using all 3 sensors or only 2 sensors.

Servo Motors:

Our bots use servo motors for the drive train. The servo motors are 360 degree continuous rotation servo motors controlled with a simple PWM signal that will be sent from the PIC32. The velocity of the servo motors can be varied and controlled with the PWM signal.

Path-Planning ROS node:

We are using path-planning ROS nodes to set waypoints for our rovers to move to. The path is pre-programmed into the rovers as the search location is known. The various subsystems also act as ROS nodes that can update the Path-Planning ROS node if an object of interest is found. Once the object of interest is found, the object's rough location will be set as a waypoint and the rover will move to the object's location according to the path-planning ROS node. The rover's previous location on the programmed path will be set a another waypoint so that after the rover is done finding the location of the object it can move back to the path

Multicell Lipo Battery:

We determined that a 7.4V (5000mAh) Lipo battery provides the required amount of power for all of our different systems. The battery is rechargeable so that the rovers can be retrieved, powered up, and sent back out on another mission. Considering the power draw of our entire system, the LiPo battery should be able to power everything for 3 hours. The LiPo battery is also lightweight which is essential for our budget and motor power.

3 Detailed project description

3.1 System theory of operation

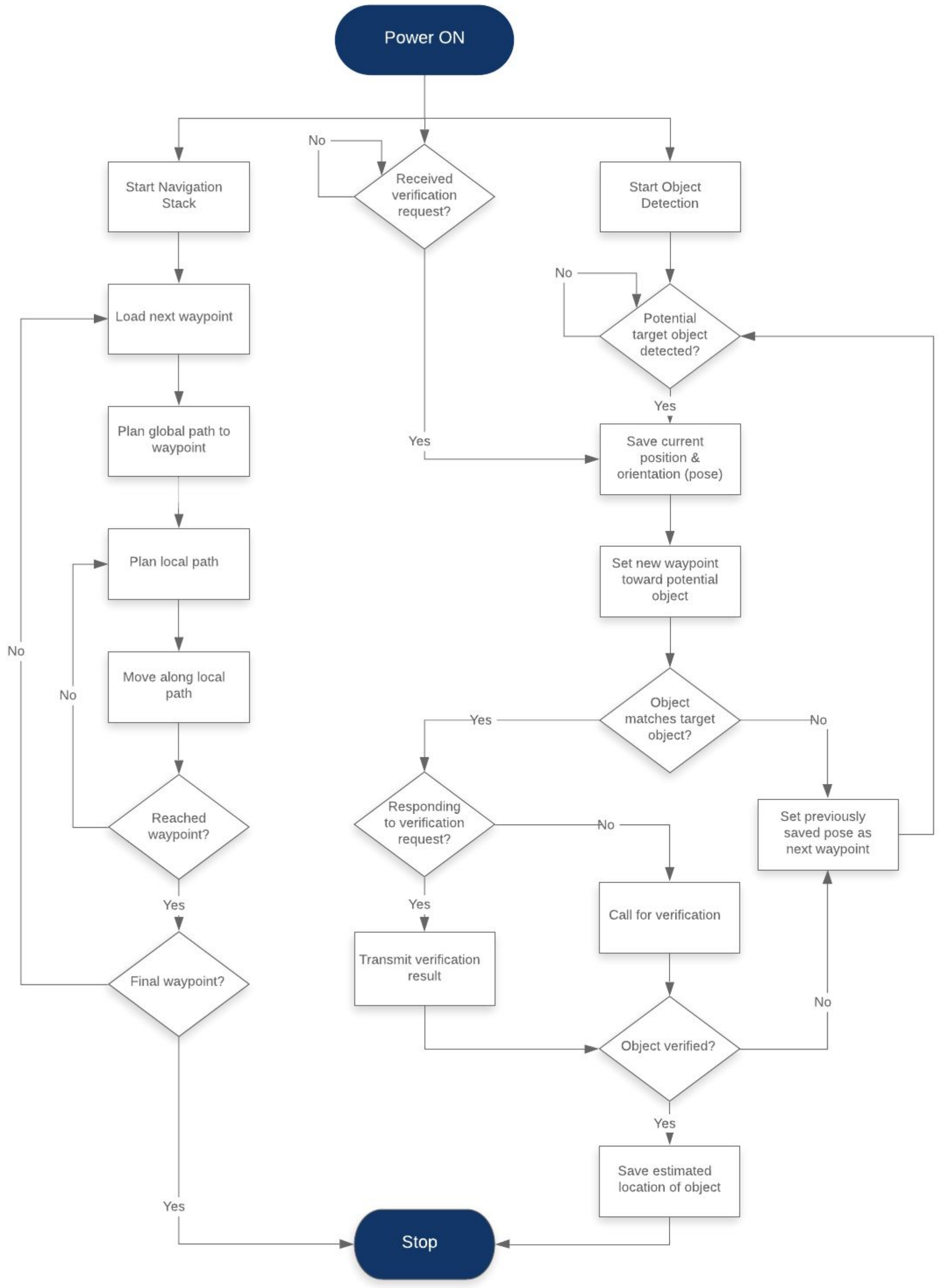

Power ON
Start Navigation Stack
Received verification request?
No
Start Object Detection
Load next waypoint
Potential target object detected?
No
Yes
Plan global path to waypoint
Yes
Save current position & orientation (pose)
Plan local path
Set new waypoint toward potential object
No
Move along local path
No
Object matches target object?
Yes
No
Reached waypoint?
Responding to verification request?
No
Set previously saved pose as next waypoint
Yes
Yes
Call for verification
Final waypoint?
Transmit verification result
No
Object verified?
Yes
Save estimated location of object
Yes
Stop

Figure 1. Overall Rover Swarm System State Machine Diagram

3.2 System Block diagram

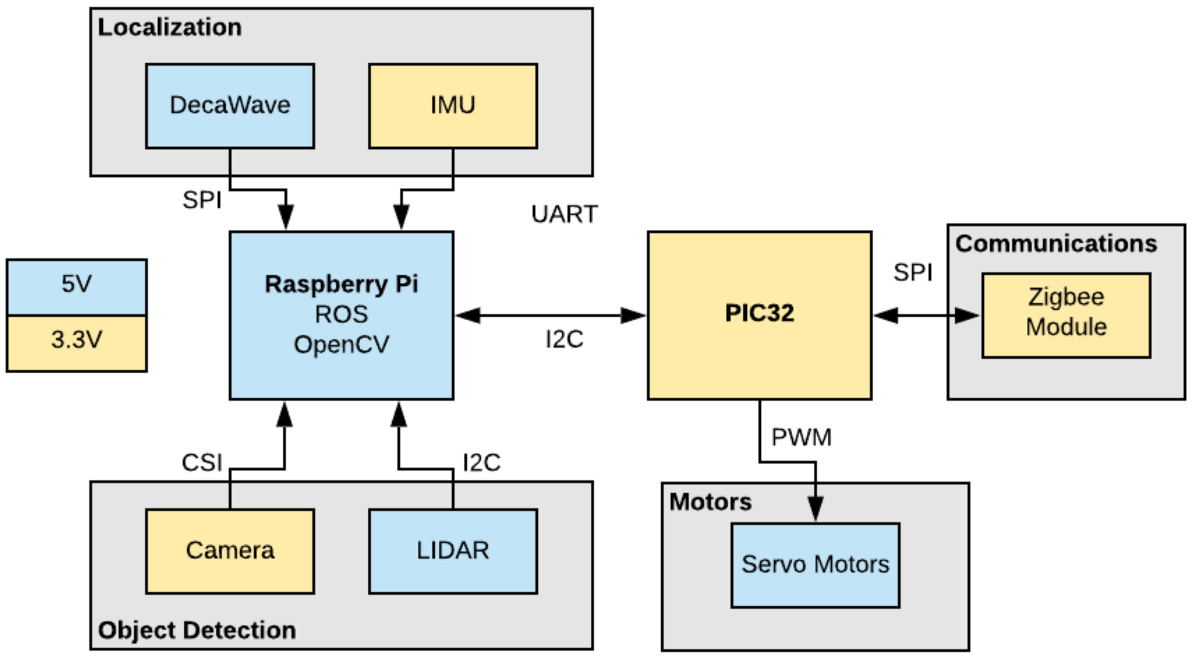


Figure 2. Proposed electrical system overview of a single rover in a two-rover system

3.3 Subsystem 1: Localization

DecaWave:

Decawave 1001 Module gives the position of each rover relative to 4 known and fixed anchors. According to decawave documentation, the x-y position is usually accurate to within 10cm of the actual location. So, we should be able to meet our requirement of determining an object's position to within 2 meters of its actual location. The decawave module uses time of flight sensing to determine the position. This meets the constraint being feasible in a lunar environment by not relying on GPS. The DecaWave will receive power of 5V from the Raspberry Pi and a clock signal from the Pi for SPI.

The DecaWave (dwm1001) module on each rover is used for self-localization (accuracy within 10cm) by wirelessly connecting to other DecaWaves in the area

(~30m radius for each module outdoors) using Ultra Wide Band (UWB) communication. The DecaWave system requires four anchors and as many tags as the user would like. The anchors must be placed in known locations in order to get a baseline for calculating distance. The tags are then able to self-locate (within 10 cm) by measuring its distance to each of the anchors. The self-localization is updated every 100ms (rate of 10Hz). Our project demonstrates this system working with four anchors and two tags, but the number of tags can be scaled up easily.

We have not done testing, but online user groups and the DecaWave product manufacturers recommend increasing the number of anchors as the size of your area increases, the number of objects in your area increases, and as the number of tags increases significantly in order to maintain desired performance and accuracy.

The DecaWave communicates with our Raspberry Pi via I2C as one ROS node (an executable capable of communicating with other nodes via ROS). This information is used in path-planning/following and when an object of interest is identified. During path-planning/following, the DecaWave's self-localization is used to verify that the rover's trajectory correctly corresponds to its pre-planned path. Also, when a rover detects an object that is not an object of interest and must avoid it, the DecaWave location information is used to navigate around the object and to return to the pre-planned path after successful avoidance. Finally, when the rover identifies an object of interest, the self-localization is used to identify not only the location of the rover but also the location of the object (in conjunction with computer vision). The location of the object of interest is transmitted from the DecaWave to the Pi which sends this information to the Pic32 to be packaged and sent wirelessly to the other rovers in the swarm via the zigbee module.

We chose to use the DecaWave module for its accuracy and speed of localization, and for its relatively easy implementation in the updated DWM1001 model. The old DWM1000 model, used by many other senior design groups, was difficult to integrate and communicate with; it was often the road-blocker for these senior design projects and barred the group from successfully completing their project. Most of the issues the past groups faced are addressed by the DWM1001's easy user interface and built in data computation.

We explored the idea of using the DecaWave modules to transmit rover locations to other rovers, thus removing the need for a Zigbee wireless communication module, but the DWM1001 module does not allow customization of messages sent via the DecaWave. It is possible that the messages could be customized using the DWM1000 model, but we determined the integration was not worth our time when Zigbee modules are cheap and user friendly.

IMU:

The IMU (BNO055) will provide estimation for the angle of the rover. The IMU will include the 3 standard IMU sensors: gyroscope, accelerometer, and magnetometer. However, a magnetometer would not work on the moon, since there is no magnetic field on the moon. Therefore, we will first attempt to obtain accurate angle estimation through the gyroscope and accelerometer, only. The IMU will receive power from the Raspberry Pi and a clock signal from the Pi for I2C.

The IMU (BNO055), which stands for inertial measurement unit, also supplies data to the navigation stack by providing acceleration and gyroscope data. The BNO055 supports a variety of configurations that combine the accelerometer, gyroscope, and magnetometer to generate sensor fusion data on relative and absolute orientation.

The information from the IMU is sent to the navigation stack to be used by other ROS nodes accessing the navigation stack. In particular, the accelerometer and gyroscopic data can be used to determine the speed and orientation of the robot and contributes to the path-planning system. Without this information, we would not be able to determine whether our robot is correctly oriented to follow the path or determine the distance it has traveled along the path.

We chose the BNO055 as our IMU because of its ability to perform sensor fusion of the accelerometer and gyroscope data to produce relative orientation estimations without any additional hardware or software processing. The BNO055 package contains an ARM CPU that fuses the data and provides standard UART and I2C interfaces for retrieving orientation estimations.

Subsystem Test:

Due to the global pandemic we were unable to fully integrate the IMU and DecaWave into a complete localizalization subsystem for testing. However, if we were to test the functionality of the subsystem we would execute the following steps:

1) Place four DecaWave anchors in known locations
2) Place the rover in a known location and orientation
3) Verify the orientation and location information of the rover gathered by the Pi by comparing it to the rover's known orientation and location

If we pass this verification test, then the localization subsystem is operating as expected.

3.4 Subsystem 2: Object Detection/Identification

Range Finder:

In addition to the Pi Cam, the laser rangefinding system, the LIDAR, is used to detect objects in the path of the rover. The LIDAR works by sending out a laser pulse and detecting the scattered light to learn about objects in the environment. We used the Hokuyo URG-04LX-UG01 Scanning Laser Rangefinder which costs $1,080. Dr. Hai Lin was kind enough to let us borrow these expensive pieces of equipment from his lab for our project. Our backup LIDAR was the Garmin LIDAR Lite-v3, which costs $130 and two are already owned by the Senior Design lab. Our LIDAR receives 5V and a clock signal for I2C from the Pi.

The Hokuyu LIDAR has an impressive detectable range of 20mm to 5600mm, scans 10 times a second (100ms/scan), and a 240° scan area with 0.36° angular resolution. This high quality scan gives us information about both path blocking objects and potential objects of interest. When combined with the visual information from the camera, objects can then be classified as either objects of interest or not. If they are not, the LIDAR information can be used to determine how the rover should deviate from its pre-planned path in order to avoid the object before returning to its pre-planned path.

Our LIDAR has been used in enough robotics projects that a ROS node has been implemented for it to act as a driver. The data from our LIDAR ROS node is fed into the rest of the navigation stack to be made available to other ROS nodes accessing this stack.

The Garmin LIDAR is much lower quality than the Hokuyu LIDAR, so we do not believe we could reasonably improve our current setup given the senior design budget. However, if money were not an obstacle, there are even higher quality

LIDARs from the same manufacturer as the Hokuyu LIDAR. This would give our system even greater range and accuracy.

Camera:

As previously mentioned, the DecaWave module does not act alone in the localization of an object of interest. In addition to the DecaWave, we use a Pi Camera attached to the rover to obtain a measurement of the distance to the object given its size in the camera's field of view. The Pi Cam receives 3.3V and a clock signal for CSI from the Pi.

The Pi Camera is also responsible for object identification. We calibrate the ROS computer vision program to identify specific objects (objects of interest) given their shape, size, and color so that when that object of interest enters the camera's field of view, the program can track the object and identify the distance from the camera to the object. This distance information is transmitted to the Pi and is combined with the DecaWave information to give the location of the object of interest. This location is then passed from the Pi to the Pic32 to be transmitted to the nearest rover via Zigbee.

We chose to use a Pi Camera because it is designed to be used with a Raspberry Pi and lends itself to the ROS computer vision program we implemented. To improve results, more time could be spent calibrating the ROS program to the objects of interest, and a higher quality camera could be used.

Subsystem Testing:

Due to the global pandemic, we were unable to integrate the LIDAR and the Pi Cam for a complete object detection/identification subsystem. However if we were to test it, we would follow these steps:

1) Physically place a large object (not an object of interest) somewhere in the rovers' pre-planned path
2) Set the rover at its starting position, turn it on, allow it to follow its path
3) Verify that the rover detects the object and subsequently avoids it by observing whether it alters its path to compensate for the object.
4) Stop the rover
5) Replace the large object in the path with an object of interest
6) Set the rover at its starting position, turn it on, allow it to follow its path

7) Verify that the rover detects and identifies the object of interest by observing whether it stops to evaluate the position of the object before returning to its path

If we meet these criteria, then our rover is detecting and identifying objects as expected.

3.5 Subsystem 3: Raspberry Pi/Path Planning

Raspberry Pi:

In addition to the Pic32 processor, we use a Raspberry Pi (Model 3B) to run ROS (robot operating software), an open source library to ease the programming of robots. ROS handles the operation and computation of 5 different systems and sensors; (1)path planning for the rovers' baseline movement and object avoidance, (2)computer vision for object detection/identification/avoidance, (3)LIDAR for object detection/avoidance, (4)IMU (BNO055) for acceleration and gyroscopic information, and (5)DecaWave (dwm1001) module for location acquisition. We use one SPI, one UART, two I2C, and one CSI interface on the Pi to control/communicate with the DecaWave, IMU, PIC32, LIDAR, and Pi Camera respectively. The Raspberry Pi will receive power from the PIC32 Custom Board. The on board voltage converter will step down our input voltage from 5V to 3.3V for the logic. The Raspberry Pi has an on-board clock which will be used to control the Decawave and IMU.

We chose to use a Raspberry Pi specifically to operate ROS. ROS is not technically an operating system in itself, and requires a base operating system in order to operate, so the Linux OS on a Raspberry Pi offered a convenient solution. So, the Pi handles all systems pertaining to ROS, including the computation of localization and object detection data, and the Pic32 handles all other systems requiring computation.

Path Planning ROS Node:

The path planning ROS node is one of the most important aspects of our project, even though it does not use any hardware directly. This node uses the open source package 'nav_core' to plan a given robot's trajectory. It creates a global path based on a number of waypoints on a pre-loaded, low detail map of the search area's topography. These waypoints are selected by the user, such that the search path for each rover will cover the entire search area. Then the algorithm uses the LiDar node, IMU node, and DecaWave node data available to continuously adjust the path of each rover to avoid obstacles not defined in the stored map. The path is adjusted to avoid obstacles while still hitting each waypoint. Low-level controls are computed based on the high-level plan by using the open source package 'move_base.' The appropriate output is then communicated to the servo motor node through ROS 'topics.'

We chose to use ROS specifically for this portion of our project because it is so well equipped to handle problems like these. Without the ROS node, we would waste time designing our path-planning control algorithm from the ground up.

Subsystem Test:

Due to the global pandemic, we were unable to construct a rover capable of moving, and we were therefore unable to physically verify the path planning/following component of our design. We were able to make a virtual simulation of our path planning/following which verified our path-planning algorithm. However, if we wanted to test that our robot was following its designated path we would follow these steps:

1) Plan a simple path for the rover
2) Layout tape on the ground that approximates the planned path for the rover given its starting location (make sure the path is unobstructed)
3) Set the rover down, power it up, and verify that it begins to follow the designated path

If we successfully completed these steps, then the Raspberry Pi is successfully executing the path-planning subsystem

3.6 Subsystem 4: PIC32 Custom Board

Custom Board:

The custom board itself will be responsible for low-level motor control, stepping down the power source voltage, and local communication between the two robots via zigbee. The low-level motor control will consist of applying PWM motor drive signals to individual motors according to high-level commands sent from the Raspberry Pi via I2C communication. The custom board will get its clock signal from the internal PIC32 clock.

The power circuitry onboard will consist of two TI TPS63061 DC-DC Buck-Boost Converters. These supplies have programmable outputs, so one supply provides 3.3V to sensors and the PIC32 MCU, while the other provides 5V to the servo motors. A non-programmable 3.3V version of this device (TPS63060) exists. Using this non-programmable version would cut down the amount of peripheral passive components required. We did not use the non-programmable version in order to reduce the number of unique parts on the design. This allowed us to recycle the layout from one regulator for the other. Additionally, the TPS63061 provides an input voltage range of 2.5V-12V. This is essential, since our 2S LiPo battery has a nominal voltage of 7.4V, which drops as the battery is depleted. The TPS63061 is also capable of providing up to 2A, which is enough for our sensors/MCU at 3.3V and for the servo motors at their peak current draw (~1A each) at 5V.

When designing with the TPS63061, peripheral passive components must be chosen carefully, according to the part data sheet. The datasheet recommends several specific power inductors, of which we chose the Coilcraft XFL4020-102. This specific inductor seems to be TI's default choice in their documentation. Additionally, the resistors used in the output-programming voltage divider (R11/R12 and R14/R16) had to be selected based on the desired output voltage. TI provides the pertinent resistor value equation in the data sheet.

When laying out the device on a PCB, the data sheet example layout must be closely followed. It is essential to manage ground noise by providing separate control and power ground pours. The output power planes must be quite large to handle maximum output currents.

In addition to providing power to onboard components via these buck converters, our PCB is also responsible for providing power to the Raspberry Pi via a UBEC (universal battery-eliminating circuit) with input and output wires. This battery-eliminating circuit allows us to power the Raspberry Pi with clean 5V power. The Pi consistently requires 2.5A, and this UBEC provides a maximum of 3A, making it an ideal choice for this application. 16-gauge wire pads have been provided on our PCB to connect the input of the UBEC to the battery voltage. The output of the UBEC can be connected to the external power pins on the Pi.

Local communications are handled by a Microchip ATSAMR30M18 Sub-GHz 802.15.4 RF Module. This device provides an all-in-one solution for local wireless communications. It includes an onboard 16MHz crystal oscillator, balun, RF filtering, and proper shielding, allowing us to be efficient with available board area. The ability to transmit in the sub-GHz range (915MHz specifically) is essential, since transmitting in the more common 2.4GHz range would likely cause interference with the DecaWave modules used for rover self-localization.

The SAM30 data sheet sets specific instructions for the RF output routing. To make the layout process easier, we decided to use an SMA antenna rather than a chip antenna. The SMA connector provides the 50-ohm matched impedance required for efficient transfer of power between the module and antenna. Had we chosen a chip antenna, we would have had to pay much closer attention to the effect of our PCB stackup on the impedance of a much more complicated RF trace element. We eliminated this concern. Additionally, nothing could be routed in the area under the device on either layer of the board for fear of causing interference.

Our PCB also hosts an IMU (inertial measurement unit), which provides the Raspberry Pi, which is running ROS, orientation information to aid in path planning. This device was chosen based on its ubiquity, availability, and relative ease of configuration. Additionally, this device handles data synthesis onboard and outputs usable data, removing this burden from our software.

Though the BNO055 will be passing information directly to the Raspberry Pi via a UART interface, it must be hosted on our custom board. The connection from our board to the Pi will be made with jumpers between headers. In retrospect, we should have interfaced the BNO055 with the PIC32 and simply passed the sensor data across the I2C interface that connects the PIC32 to the Pi. This is an obvious improvement in space management.

PIC32:

This system is built around a PIC32 (PIC32MX795f512H) processor, which manages communications between rovers and coordinates movement (motor control). We chose this microprocessor because of our familiarity with the programming interface and its diverse and numerous modes of operation and communication. In particular, we required at least two output comparator pins to control the PWM signals sent to our servo motors, and at least one set of I2C and SPI pins for communication to the Raspberry Pi and Zigbee Module respectively. We code the PIC32 using C programming.

Originally, we had chosen an Atmel microprocessor (from the SAMC21 series) for its advertised ease of use with motor control, but we struggled to learn how to use a new programming interface in a reasonable time, and we had trouble getting the hardware setup/code uploaded using the SAMC21 bootloader.

The PIC32 microcontroller is arguably overqualified for the small number of tasks we are using it for given it's large package size and high pin count. With more research, a different PIC32 package could be chosen to accomplish the same tasks with a smaller package.

Multicell LiPo Battery:

The entire system is powered by a 7.4V (5000mAh) multi-cell LiPo rechargeable battery. Given the maximum current draw of our system, the battery should be able to power our system for around 3 hours. We determined 3 hours is more than enough time for our rovers to find the object of interest in the current search area of our project.

In addition to the target voltage and its longevity, we chose this battery because it is rechargeable and specifically made for RC cars which means it is designed to handle sudden acceleration. Additionally, it is also relatively lightweight.

If money were not an object, a higher quality power source could be chosen. In an ideal world, our rovers would operate off of MMRTGs which harness heat from the radioactive decay of plutonium into electricity.

Subsystem Testing:

Due to the global pandemic, we were unable to construct and test the Custom Board. However, our test plan would include the following steps to verify functionality:

1) Check custom board for shorts between ground and 5V, 3.3V using a multimeter
2) Verify LiPo Battery voltage is the expected 7.4V using a multimeter
3) Attach LiPo Battery to DC-DC converters, then verify the output voltage is the expected 5V using a multimeter.
4) Verify all 3.3V and 5V points on the Custom board using a multimeter
5) Test functionality of custom board by attaching servo motors and operating both at the same time (run simple script)

6) Test functionality of custom board by sending message via zigbee to a receiver (run simple script)

If all of these tests were completed successfully, then the custom board is operating properly.

3.7 Subsystem 5: Communication

Zigbee:

Since we decided to save time and effort by using the DWM1001 model DecaWave, we also use a Zigbee module to communicate information between the rovers wirelessly.

Zigbee is a local communication protocol that uses mesh networking. Although we are only demonstrating two rovers, the mesh networking provides a scalable solution. The Microchip SAMR30M18A Zigbee module has a max range of 400 ft and operates at 2.4 GHz. The Zigbee module communicates over SPI with the PIC32. The Local Communication will receive 3.3 V from a DC to DC converter on the PIC32 custom board which will step the original 7.4V from the LiPo battery to 3.3V. The Local Communication will receive a clock signal from the PIC32 internal clock and will communicate over SPI with the PIC32.

The Zigbee module has a maximum communication range of 400 ft. We determined this range is acceptable since we would eventually be working with a rover swarm and can therefore communicate through the swarm by simply transmitting information to the nearest adjacent rover and allowing it to propagate through the system in this manner. However, to conserve power, we would like to operate at a maximum range of around 200 ft.

The Zigbee module constantly checks for reception of a signal from another bot, but only transmits information when the rover has located an object of interest. Once the rover has determined the location of the object of interest using a combination of the DecaWave and Pi Camera, that location is transmitted to the other rover. The other rover must then save its current location, deviate from its pre-planned path, drive over to the transmitted location, and take its own measurement of the location of the object of interest. Once this process is completed, it returns to its previous location on the pre planned path and continues to follow this route to completion.

We chose the Zigbee module for its ample documentation, ease of use, and because there are two different operating frequencies for zigbee communication. Since both the DecaWave and the Zigbee are communicating wirelessly, it's important to ensure they are communicating on two different frequencies and not interfering with each other. The DWM1001 operates at 6.5GHz, but also has an onboard bluetooth module that operates at 2.4GHz, so we planned to avoid both of these operating frequencies. Later we discovered this bluetooth module could be deactivated, but given our original knowledge we chose Zigbee modules that operate at 916MHz instead of 2.5GHz.

In the future, since the DecaWave's bluetooth module can be deactivated, either Zigbee module could be implemented.

Subsystem Test:

Testing individual zigbee modules is easy as it requires a simple script that sends and receives information from other zigbee modules which can be indicated by the onboard LEDs. However, we did not have an opportunity to test the operation of our zigbee modules in the fully integrated rover system. If we were going to verify the Zigbee operation, we would execute the following steps:

1) Place an object of interest in the pre planned path of one of the rovers
2) Set both rovers down at their respective starting positions, turn them on and allow them to follow their pre-planned paths
3) Verify that the two rovers' zigbee modules are operating correctly by observing whether the rover that finds the object of interest calls over the other rover for verification and then whether that second rover returns to its previous location

If we pass this test, then the zigbee module is operating as expected in our system.

3.8 Subsystem 6: Motors

Motors:

We use two servo motors (DS04-NFC) to drive the movement of our robot. The DS04-NFC is a 360° continuous rotation servo motor with a maximum torque of 5.5kg-cm at ~5V. We chose wheels with a 3.3cm radius, so we determined that the rover could weigh no more than 4 lbs, though we would like to have a safety factor of at least 0.5 lbs, so our target weight would be 3.5 lbs. To limit the weight, we decided to 3D print the chassis of the rover. The servo motors will receive power directly from the 7.4V LiPo battery.

The Servo motors are controlled by PWM signals that come from the Pic32's output comparators. The PWM signal controls both the speed and direction of the servos. The PWM signal to be sent to the servos must be determined by the needs of the path-planning ROS node then sent via the Pi to the Pic32 to control the output to the motors. For example, if an object needs to be avoided, the path-planning ROS node must determine how far to the right or left it would like the rover to turn, and then how far forward it must drive before turning back to get on to the path. Similarly, if the rover is traveling through a curve in the pre-planned path, a message will need to be sent to the motors to control the speed at which the rover turns, the turn radius, and the direction of the turn.

We chose servo motors to drive the rovers because of the accuracy in controlling the servos' speed. By using servos, we removed the need for encoders and/or motor controllers associated with DC motors which reduces cost and weight of our rovers. We chose the DS04-NFC servos in particular because there are about six already owned by the senior design lab, they operate at 5V, and have a reasonably high torque.

Servo motors are a good low cost option, however, if money were not an obstacle, higher torque DC motors with encoders and a decent motor controller (probably a sabretooth) would be a superior option for a fully functional rover. Obviously on the moon, more power would be needed for the rovers to move around than is provided by the small servos we have chosen.

Subsystem Test:

Verifying the proper operation of individual servo motors is very easy and only requires a simple script to be run on the PIC32 to send a proper PWM signal to each motor. However, we were unable to complete a more meaningful evaluation of our drive train since we could not construct our final design. If we were to test our system though, we would follow these steps:

1) Write and run a script on the PIC32 to execute a straight path, 90° turns in both directions, and 45° turn in both directions
2) Turn on the system, observe that the behavior executed by the rover follows this script and is accurate
3) Plan a straight path for the rover

4) Turn on the rover, verify that it is moving in a straight path
5) Plan a path with 90° turns in it for the rover
6) Turn on the rover, verify that it completes these turns in the correct direction
7) Plan a path with gradual curves in it for the rover
8) Turn on the rover, verify that it is able to complete these curves

If all of these tests are passed, then we have verified that the servos are capable of accurate manipulation of the rover and that the PIC32 is receiving and properly responding to path planning instructions sent to it by the Pi.

3.9 Interfaces

Our project uses three main serial interfaces UART, SPI, and I2C. We would use the following very basic setups for each:

UART:

- 9600 Baud rate
- 8 bit data
- No Parity bit
- 1 Stop bit
- No flow control

SPI:

- 8bit data transfers
- Sample in the middle
- Output on the falling edge
- Clock idle state low

I2C:

- Baud rate: BRG=31

4 System Integration Testing

4.1 Describe how the integrated set of subsystems was tested.

Due to the global pandemic, we were unable to integrate and test our subsystems. However, we would execute each subsystem test as laid out in Section 3 of this document. After these were each completed we would begin our Full System Test by executing the following steps:

1) Plan paths for each rover in a given 15sqft area
2) Lay out tape to identify the square area and the preplanned-paths for an observer
3) Place objects (not of interest) randomly in each path
4) Place a few objects of interest randomly in the square area
5) Set the two rovers down at their respective starting positions
6) Turn on the rovers, allow them to begin to follow their pre planned paths
7) Observe and document each rover's behavior when following and/or returning to its path and when confronting an object of interest or obstacle
8) Verify that each object of interest is identified by the rovers and that both rovers make assessments of the object's location
9) Repeat as necessary/desired with different object placements within the 15sqft area
10) Quantify the rovers' accuracy at locating objects of interest by comparing their measurements of the identified objects' locations to the known locations of the objects of interest and tracking how many objects of interest are missed by the system (if any)

By following this plan, we will be able to quantify the performance of our rover swarm system and verify that it is working as expected and meeting our expectations.

4.2 Show how the testing demonstrates that the overall system meets the design requirements

Our design requirements stated that we should design and construct a system of two autonomous rovers capable of navigating through a partially unknown lunar environment, finding a predetermined object, and estimating the position of the object relative to the rovers' starting location.

The autonomy of our rovers is demonstrated by the path-following and object avoidance behaviors of our rovers. Presently, the user only has to turn on the rovers when in their starting positions, and the rovers will then fully function and operate as expected.

The navigation of a partially unknown lunar environment is demonstrated by the object-avoidance behavior of our rovers. If the rovers are able to navigate around detected obstacles, then we have successfully navigated the partially unknown environment.
The discovery of the predetermined object of interest is demonstrated by the object detection/identification behavior of our rovers. If the rover detects and identifies the predetermined object of interest, then calls over the other rover, we will have proven that the rovers are capable of finding the predetermined objects.
The estimation of the object of interest's position relative to the rover's starting location is demonstrated by the object identification and localization behavior of our rovers. Both rovers will document the location of the objects of interest, and this information will be saved to a .txt file in each rover's memory. These locations can be accessed by the user once the rovers have completed their mission, and in our test scenario, we can verify their accuracy, which we would expect to be within around +/- 10cm given the DecaWave and PiCam's combined information. By gathering a document of these locations, we will prove the rovers can estimate the positions of predetermined objects.

5 Users Manual/Installation manual

5.1 How to install your product

So, you just bought a Baymax and Baymin? To install your new state-of-the-art robots, you must first be on the moon. If you are currently on some other celestial body, quickly fly to the moon before resuming installation. Make sure to bring all components of your Baymax and Baymin quick starter kit.

First, you will need to determine where to set up your system on the moon. If you have any general idea where your lost object could be, try to pick a relatively close location. Once you have picked a spot, go ahead and start unpacking your kit. Inside, you will find two robots (Baymax and Baymin) and a number of DecaWave anchors.

Before unboxing the robots, you must deploy the anchors. The anchor points should be placed in known locations. Record the position of each anchor point and input the coordinates into the friendly user interface. Next, you may unbox the rovers and proceed to product setup.

5.2 How to setup your product

Training for an object
First, you need to select the object you would like Baymax and Baymin to find. Your object may be one of the predefined objects stored in each of the robot's memory. To check, navigate to the "Predefined Objects" tab of the robot user interface. If your object is one of the objects listed, select it. If not, you will need to train for a custom object. Although this capability is not currently available (as of May 2020), a future software update will allow you to train a neural network to recognize your custom object. To train for a custom object, you will need many pictures of the object in various lighting conditions and backgrounds. It is recommended that you upload at least 10,000 images of your desired object. However, the more images you upload, the better the performance of your robots will be.

Setting up path
Next, you will need to participate in determining the search path of Baymax and Baymin. The user interface will prompt you to select a number of *waypoints*. The waypoints are locations that either Baymax and Baymin will navigate toward, while searching for the object along the way. Select your way points by picking an even distribution of locations within the area you would like the robots to search.

Initiate the Search
Now you are ready to begin searching for your lost item. Simply power each robot on and set them down on the surface. Make sure to place Baymax and Baymin facing opposite directions, so that they search opposite ends of the search area. Once this is complete, the robots will take it from there.

5.3 How the user can tell if the product is working / troubleshooting

If your robots begin to navigate to the selected waypoints, then your product is working. If you find that your robots are correctly navigating to the selected waypoints, but they are still not locating the object of interest, then your search area may be too small. Begin troubleshooting by increasing the search area. This can be done by selecting more waypoints through the user interface.

If you notice that your robots come close to the object of interest but do not recognize that it is the object of interest, then the robot's camera may be malfunctioning. First, check to see if there is any dust or debris obstructing the camera's field of view. If so, remove it using a lint-free cloth.

If you notice that your robots are not correctly navigating to the selected waypoints, then your product is not functioning properly. Begin troubleshooting by visually inspecting the robots' wheels. If you notice a flat tire, you will need to get that replaced.

6 To-Market Design Changes

There are a number of changes we would make to our design before taking it to "market" (the moon). Due to budget constraints, our current rovers do not use a 3-dimensional LiDAR system, which would allow us to identify obstacles such as craters that would not fall into our current 2-dimensional field of vision. Including this higher 3D LIDAR in our design would improve both object avoidance and path planning.

Additionally, we would like to provide better shielding for our electronics, given the EMI levels on the moon. With multiple sensitive RF components onboard, our design is especially susceptible to interference. The ATSAMR30M18 is shielded adequately for use on earth under normal conditions, but this stock shielding may not be adequate for the moon.

We would also like to increase the operating temperature range of our components. For instance, the TPS63061 can function in a maximum free air temperature of 85℃ down to -40℃, yet the temperature on the moon can fluctuate between about 127℃ and -173℃. Clearly, this one flaw would render our entire system useless.

As for the power source, we certainly would not use a LiPo battery on a final design meant for long-term operation on the moon. Ideally, our design would be nuclear- or solar-powered.

Currently, our rovers save the location of the object in a .txt file in on-chip memory. We would prefer to have this information transmitted wirelessly to a control station so that the information can be obtained and used quickly.

In our mechanical design, we would use a more robust chassis that is able to surmount small-to-medium sized obstacles. This could be achieved through the use of a rocker-bogie design, which uses six wheels, with two of the wheels mounted on swing arms.

7 Conclusions

While our time working on our beloved rovers, Baymax and Baymin, was cut short due to COVID-19, we feel that we have sufficient groundwork that our rovers could easily be constructed and up and running with minimal extra effort. We hope that this document could act as a guide to future Notre Dame senior EE design groups who may be looking to do a similar project. As our project stands, we have chosen every component that will be in the rovers and written all the code for the various subsystems.

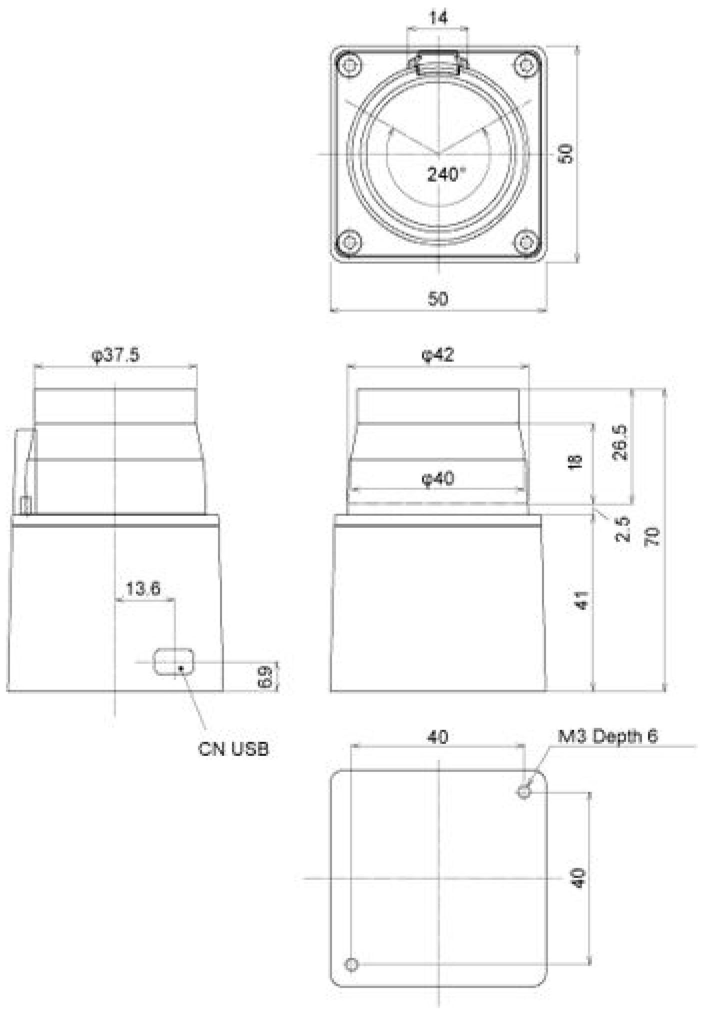
At this stage of the project, we find ourselves reflecting on what possible improvements we could have made to make our design that much better. One improvement we identified was that we could have used a microprocessor that was less beefy than the PIC32 we chose. The PIC32 is capable of so many different functions and our application did not require many of these functions. We also would have had the IMU connect to the PIC32 instead of the PI. The PI has many different sensors attached to it as it is and by connecting the IMU to the PIC32 we could lighten the PI’s load while keeping the same functionality which would ultimately lead to a more reliable design.

Even though our rovers would have been made and tested on Earth, we feel our rovers would be Moon Ready™. Our solution to the specified NASA challenge is a lightweight, cost-effective, and efficient design. In addition, our solution is easily scalable, a feature we thought was highly important for any NASA mission. With minimal components and maximal effectiveness, our sleek design is one that could be in use for many years to come.

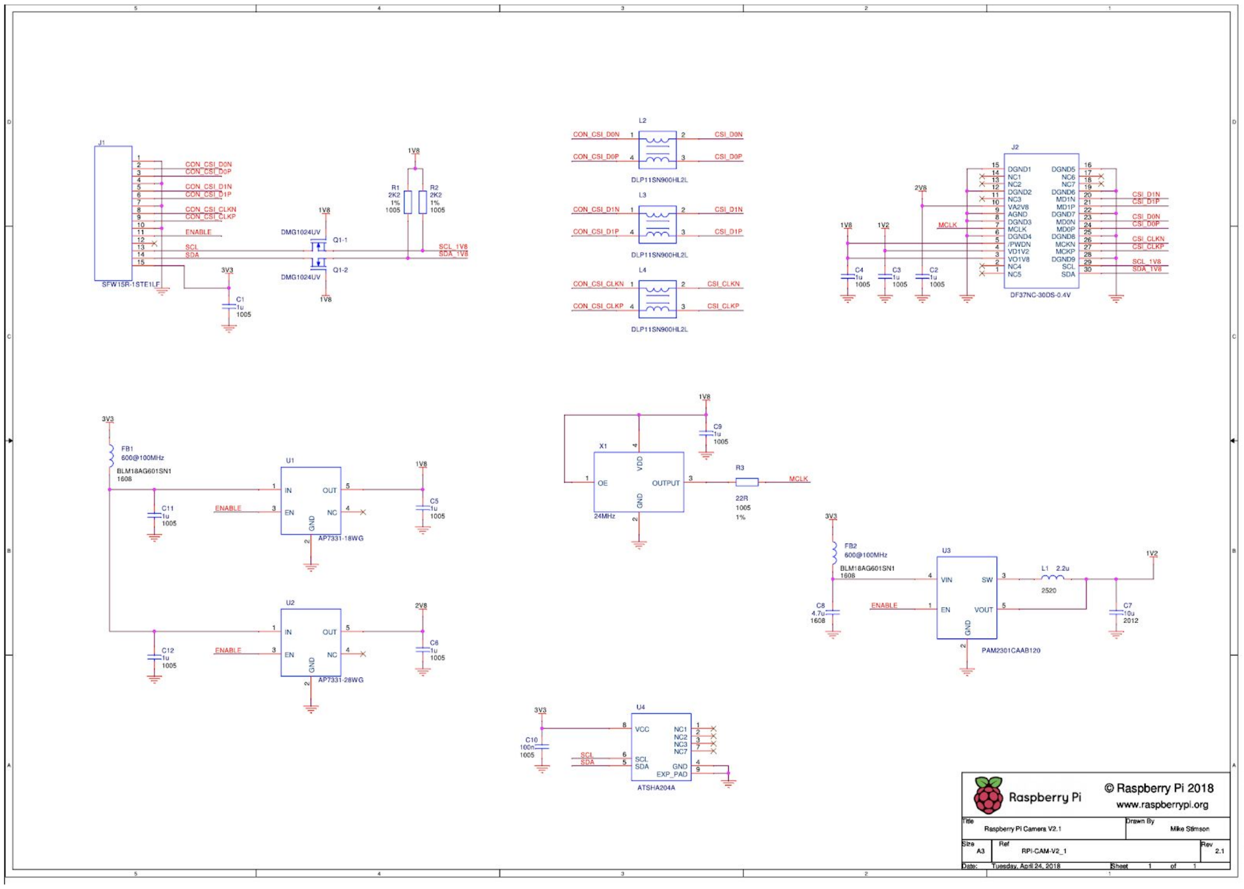
8 Appendices

8.1 Hardware Schematics

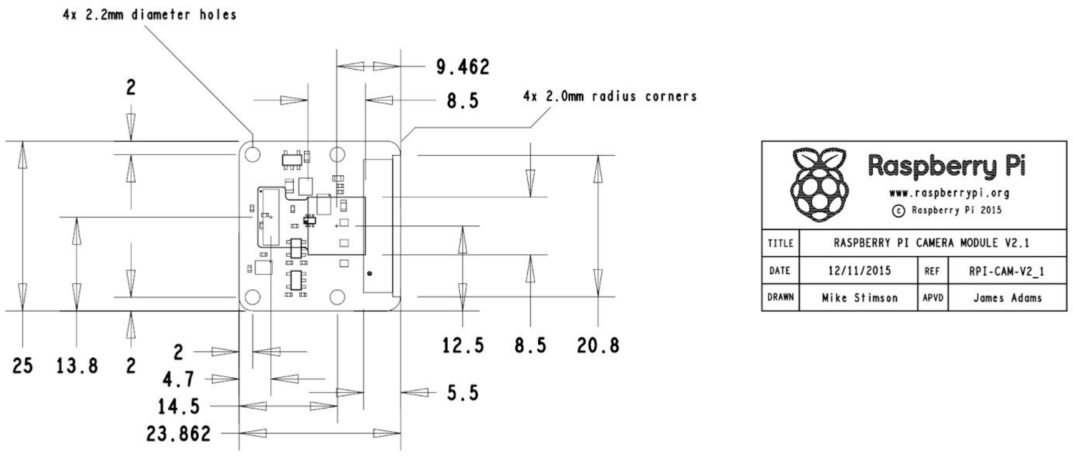
LiDAR Schematic: Hokuyo's URG-04LX-UG01

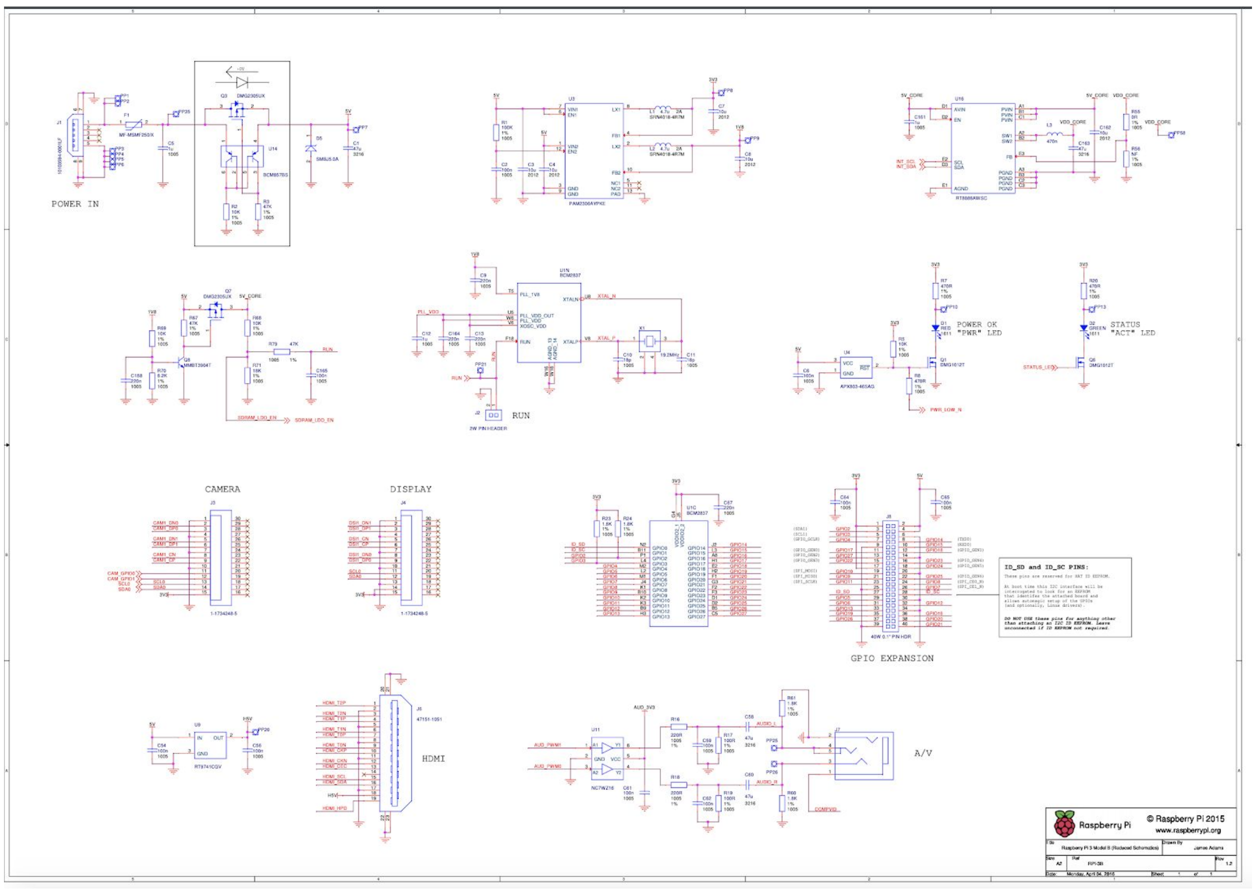


Camera Schematic: Pi Camera V2.1

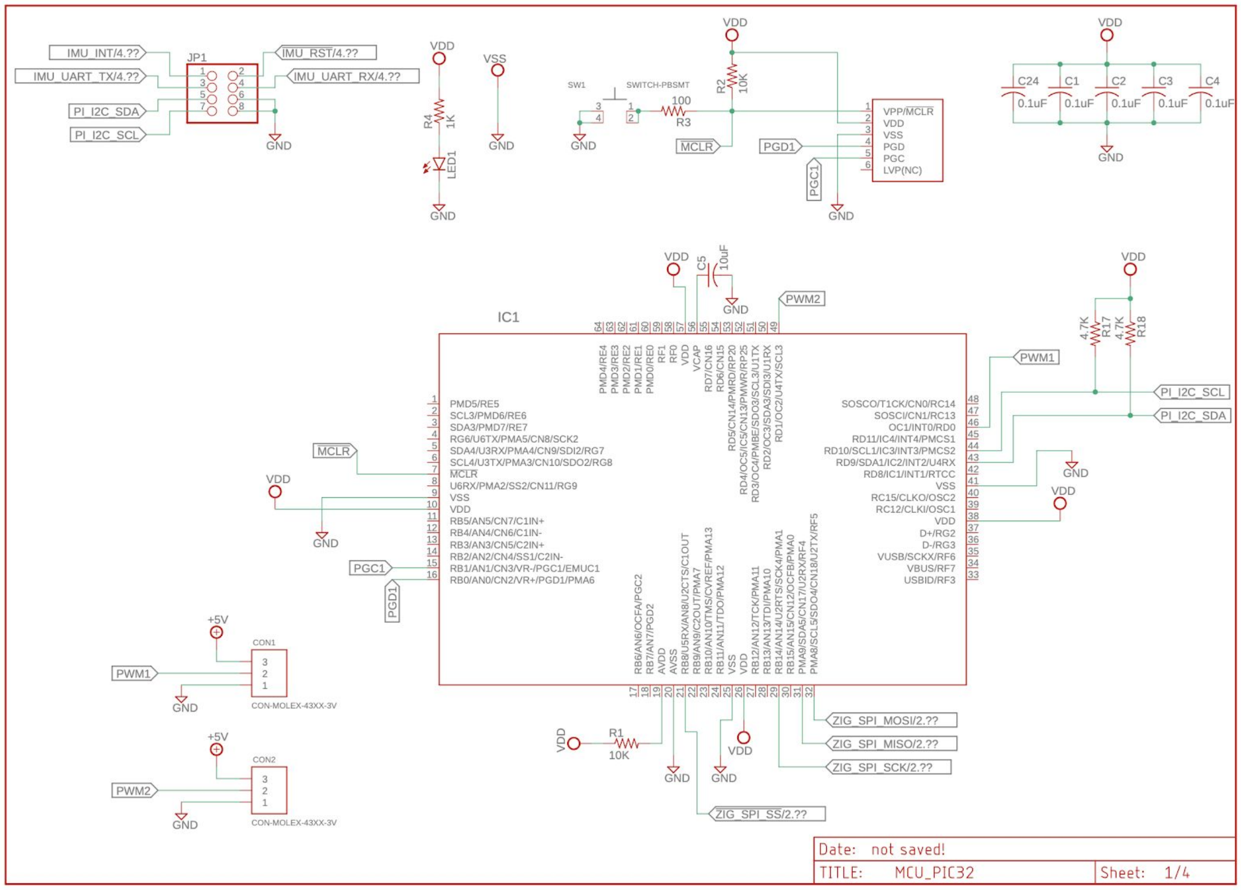


Raspberry Pi Schematic: Model B

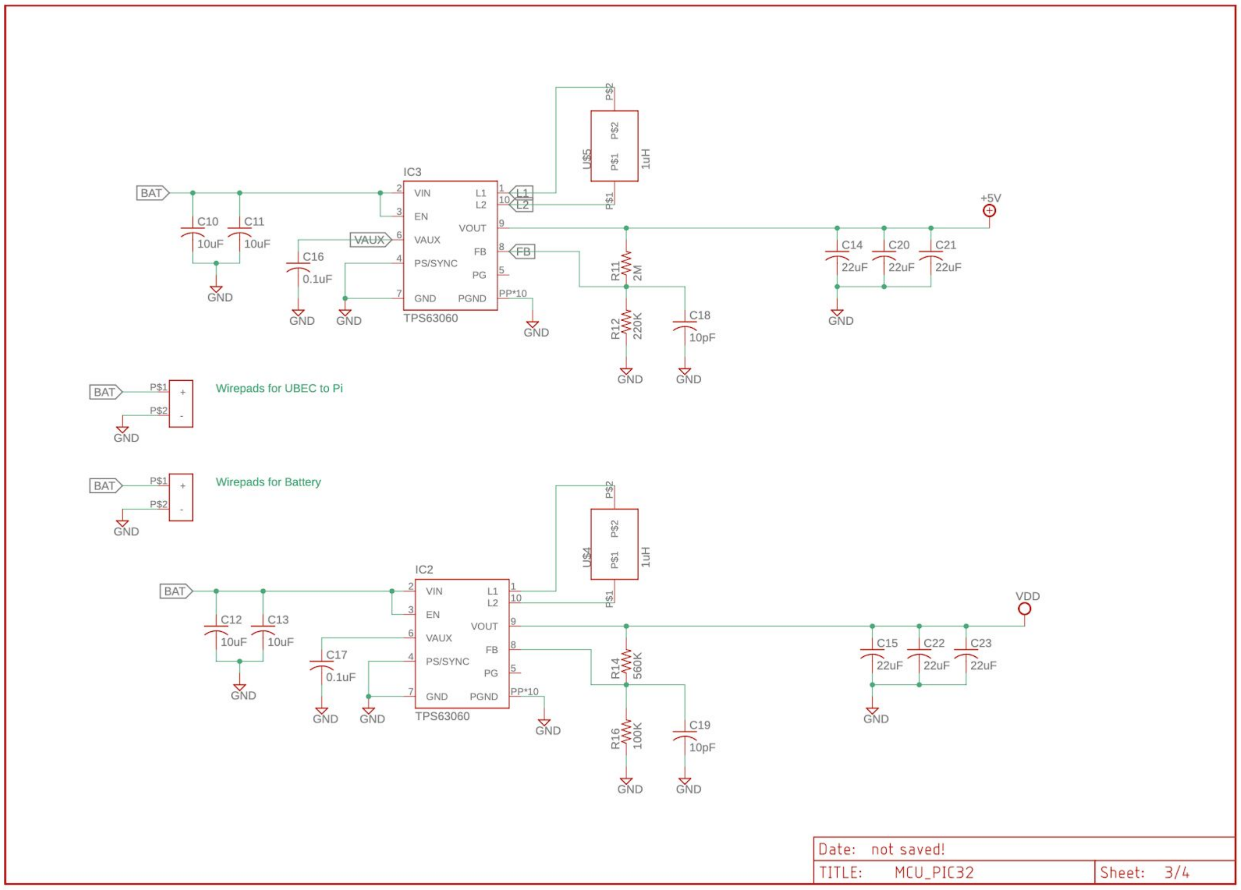




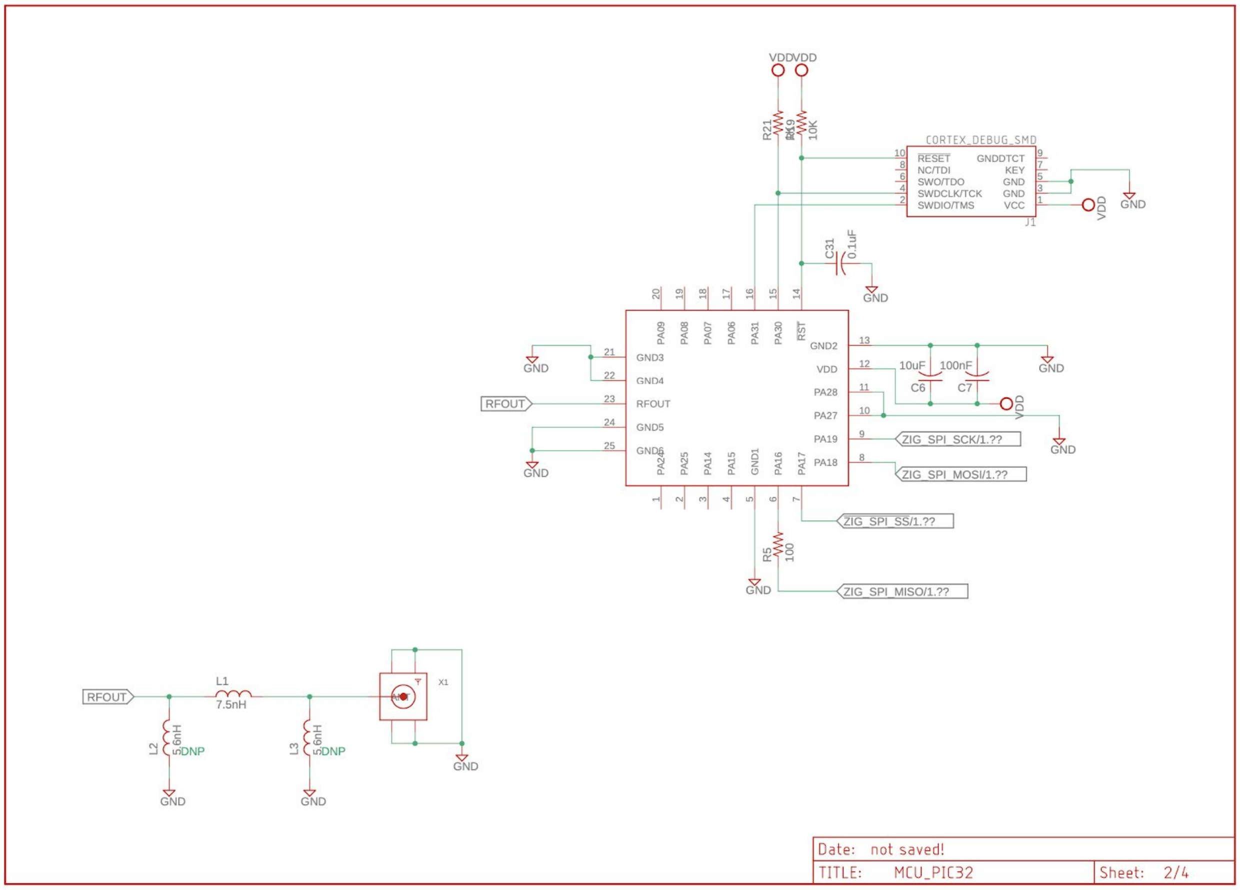
Custom PCB - MCU



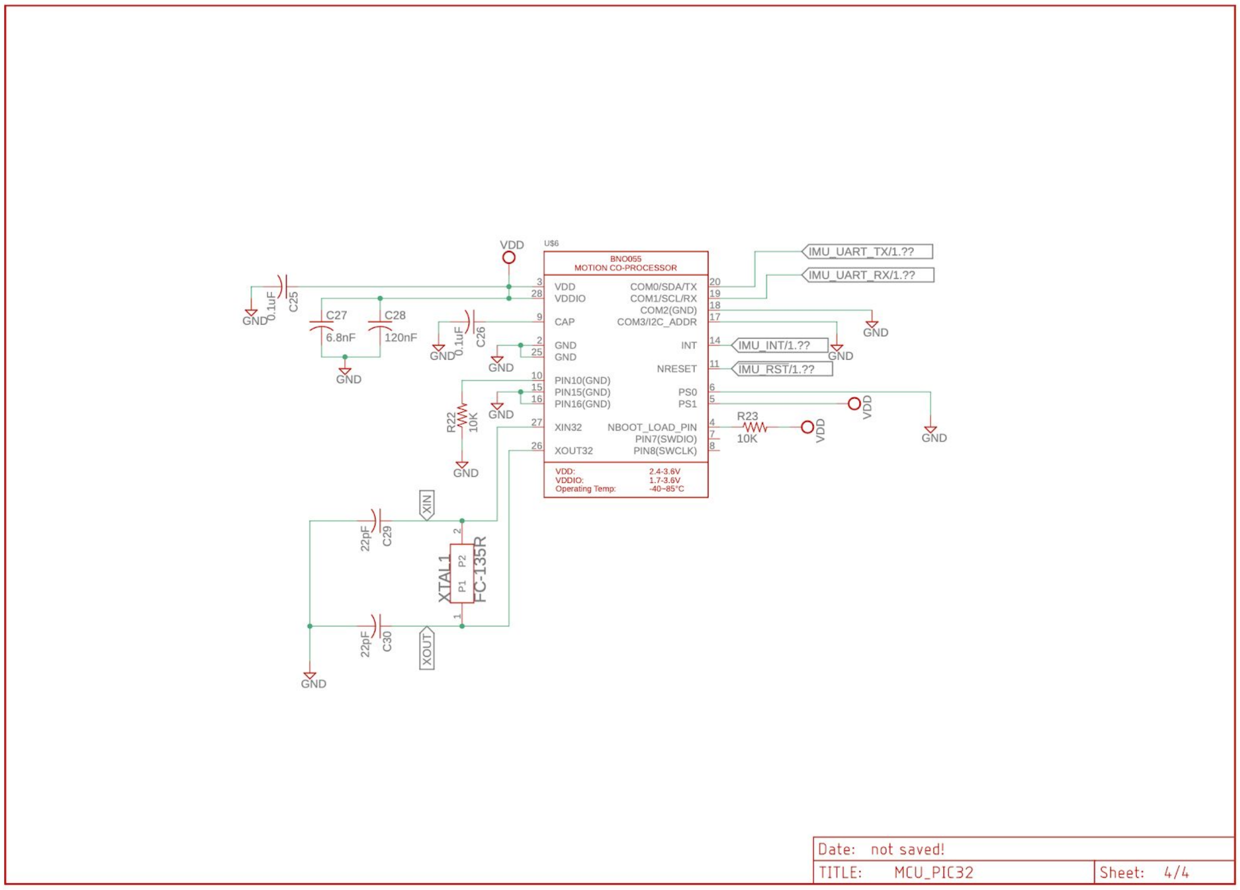
Custom PCB - Power



Custom PCB - ATSAMR30M18



Custom PCB - BNO055



8.2 Code

Object recognition python code

```
# USAGE
# python ball_tracking.py --video ball_tracking_example.mp4
# python ball_tracking.py

# import the necessary packages
from collections import deque
from imutils.video import VideoStream
from picamera.array import PiRGBArray
from picamera import PiCamera
from threading import Thread
import numpy as np
import argparse
import cv2
import imutils
import time
```

```
class PiVideoStream:
        def __init__(self, resolution=(320, 240), framerate=32):
                # initialize the camera and stream
                self.camera = PiCamera()
                self.camera.resolution = resolution
                self.camera.framerate = framerate
                self.rawCapture = PiRGBArray(self.camera, size=resolution)
                self.stream = self.camera.capture_continuous(self.rawCapture,
                        format="bgr", use_video_port=True)
                # initialize the frame and the variable used to indicate
                # if the thread should be stopped
                self.frame = None
                self.stopped = False

        def start(self):
                # start the thread to read frames from the video stream
                Thread(target=self.update, args=()).start()
                return self
        def update(self):
                # keep looping infinitely until the thread is stopped
                for f in self.stream:
                        # grab the frame from the stream and clear the stream in
                        # preparation for the next frame
                        self.frame = f.array
                        self.rawCapture.truncate(0)
                        # if the thread indicator variable is set, stop the thread
                        # and resource camera resources
                        if self.stopped:
                                self.stream.close()
                                self.rawCapture.close()
                                self.camera.close()
                                return
        def read(self):
                # return the frame most recently read
                return self.frame
        def stop(self):
                # indicate that the thread should be stopped
                self.stopped = True

# construct the argument parse and parse the arguments
ap = argparse.ArgumentParser()
ap.add_argument("-v", "--video",
help="path to the (optional) video file")
ap.add_argument("-b", "--buffer", type=int, default=64,
        help="max buffer size")
args = vars(ap.parse_args())

# define the lower and upper boundaries of the "green"
# ball in the HSV color space, then initialize the
# list of tracked points
greenLower = (29, 86, 6)
greenUpper = (64, 255, 255)
pts = deque(maxlen=args["buffer"])
```

```
vs = PiVideoStream().start()

# allow the camera or video file to warm up
time.sleep(2.0)

############### Calibration ####################
# For calibration of distance calculation
KNOWN_DISTANCE = 12.0
KNOWN_WIDTH = 1.5

image = cv2.imread("ball_calibration.jpg")

def distance_to_camera(knownWidth, focalLength, perWidth):
        # compute and return the distance from the maker to the camera
        return (knownWidth * focalLength) / perWidth

# resize the frame, blur it, and convert it to the HSV
# color space

image = imutils.resize(image, width=600)
blurred = cv2.GaussianBlur(image, (11, 11), 0)
hsv = cv2.cvtColor(blurred, cv2.COLOR_BGR2HSV)

# construct a mask for the color "green", then perform
# a series of dilations and erosions to remove any small
# blobs left in the mask
mask = cv2.inRange(hsv, greenLower, greenUpper)
mask = cv2.erode(mask, None, iterations=2)
mask = cv2.dilate(mask, None, iterations=2)

# find contours in the mask and initialize the current
# (x, y) center of the ball
cnts = cv2.findContours(mask.copy(), cv2.RETR_EXTERNAL,
        cv2.CHAIN_APPROX_SIMPLE)
cnts = imutils.grab_contours(cnts)
center = None

# find the largest contour in the mask, then use
# it to compute the minimum enclosing circle and
# centroid
c = max(cnts, key=cv2.contourArea)
((x, y), radius) = cv2.minEnclosingCircle(c)
M = cv2.moments(c)
center = (int(M["m10"] / M["m00"]), int(M["m01"] / M["m00"]))

# draw the circle and centroid on the frame,
# then update the list of tracked points
cv2.circle(image, (int(x), int(y)), int(radius),
        (0, 255, 255), 2)
cv2.circle(image, center, 5, (0, 0, 255), -1)

# show the frame to our screen
cv2.imshow("Image", image)
```

```
focalLength = (radius * KNOWN_DISTANCE) / KNOWN_WIDTH
##################################################

# keep looping
while True:

	# grab the current frame
	frame = vs.read()

	# handle the frame from VideoCapture or VideoStream
	frame = frame[1] if args.get("video", False) else frame

	# if we are viewing a video and we did not grab a frame,
	# then we have reached the end of the video
	if frame is None:
		break

	# resize the frame, blur it, and convert it to the HSV
	# color space
	frame = imutils.resize(frame, width=600)
	blurred = cv2.GaussianBlur(frame, (11, 11), 0)
	hsv = cv2.cvtColor(blurred, cv2.COLOR_BGR2HSV)

	# construct a mask for the color "green", then perform
	# a series of dilations and erosions to remove any small
	# blobs left in the mask
	mask = cv2.inRange(hsv, greenLower, greenUpper)
	mask = cv2.erode(mask, None, iterations=2)
	mask = cv2.dilate(mask, None, iterations=2)

	# find contours in the mask and initialize the current
	# (x, y) center of the ball
	cnts = cv2.findContours(mask.copy(), cv2.RETR_EXTERNAL,
		cv2.CHAIN_APPROX_SIMPLE)
	cnts = imutils.grab_contours(cnts)
	center = None

	# only proceed if at least one contour was found
	if len(cnts) > 0:
		# find the largest contour in the mask, then use
		# it to compute the minimum enclosing circle and
		# centroid
		c = max(cnts, key=cv2.contourArea)
		((x, y), radius) = cv2.minEnclosingCircle(c)
		M = cv2.moments(c)
		center = (int(M["m10"] / M["m00"]), int(M["m01"] / M["m00"]))

		# only proceed if the radius meets a minimum size
		if radius > 10:
			# draw the circle and centroid on the frame,
			# then update the list of tracked points
			cv2.circle(frame, (int(x), int(y)), int(radius),
				(0, 255, 255), 2)
```

```
                        cv2.circle(frame, center, 5, (0, 0, 255), -1)

        # update the points queue
        pts.appendleft(center)

        # loop over the set of tracked points
        for i in range(1, len(pts)):
                # if either of the tracked points are None, ignore
                # them
                if pts[i - 1] is None or pts[i] is None:
                        continue

                # otherwise, compute the thickness of the line and
                # draw the connecting lines
                thickness = int(np.sqrt(args["buffer"] / float(i + 1)) * 2.5)
                cv2.line(frame, pts[i - 1], pts[i], (0, 0, 255), thickness)

        # Update distance
        inches = distance_to_camera(KNOWN_WIDTH, focalLength, radius)
        cv2.putText(frame, "%.2fft" % (inches / 12),
                (frame.shape[1] - 200, frame.shape[0] - 20),
cv2.FONT_HERSHEY_SIMPLEX,
                2.0, (0, 255, 0), 3)

        # show the frame to our screen
        cv2.imshow("Frame", frame)
        key = cv2.waitKey(1) & 0xFF

        # if the 'q' key is pressed, stop the loop
        if key == ord("q"):
                break

# stop the camera video stream
vs.stop()

# close all windows
cv2.destroyAllWindows()
```

Low Level Servo Motor C code for PIC32

```
#include <stdio.h>

#include <stdlib.h>

#include <xc.h>

#include "configbits-16ex8.h"

#include "SDlib16.h"
```

```
#include <sys/attribs.h>



#define init_duty_cycle 3870   //(cw=994-3560; stall= <990 and 3562-3845 and >6405; ccw= 3848-6405)
#define period 49999;         //20ms
/*
 *
 */
int main(int argc, char** argv) {

    //Servo 1 Setup
    T2CONbits.TCKPS=2;         //1:4 pre-scaler
    PR2=period;                //Set the Period for Timer2
    TMR2=0;                    //Choose 32 bit count register for Timer2
    OC1CONbits.OCM=0b110;      //OC1 set for PWM without fault pin
    OC1RS=init_duty_cycle;     //Set secondary register duty cycle (update this register)
    OC1R=init_duty_cycle;      //Set primary register with initial duty cycle

    //Servo 2 Setup
    T3CONbits.TCKPS=2;         //1:4 pre-scaler
    PR4=period;                //Set the period for Timer4
    TMR4=0;                    //
    OC2CONbits.OCM=0b110;      //OC2 set for PWM without fault pin
    OC2RS=init_duty_cycle;     //Set secondary register duty cycle (update this)
    OC2R=init_duty_cycle;      //Set primary register with initial duty cycle

    //Enable Timers and Output Compare Registers
```

```
   T2CONbits.ON=1;         //Turn on Timer 2
   OC1CONbits.ON=1;         //Turn on OC1
   T4CONbits.ON=1;         //Turn on Timer 4
   OC2CONbits.ON=1;         //Turn on OC2

   while(1);              //infinite while loop
return (EXIT_SUCCESS);
}
```

Main.c file for Zigbee code

(For full code go to http://seniordesign.ee.nd.edu/2020/Design%20Teams/nasa/Zigbee/)

```
/************************ HEADERS ****************************************/
#include "task.h"
#include "asf.h"
#include <asf.h>
#include "sio2host.h"

#if defined(ENABLE_NETWORK_FREEZER)
#include "pdsDataServer.h"
#include "wlPdsTaskManager.h"
#endif

#if ((BOARD == SAMR30_XPLAINED_PRO) || (BOARD == SAMR21_XPLAINED_PRO))
#include "edbg-eui.h"
#endif
```

```
/************************** DEFINITIONS **********************************/
#if (BOARD == SAMR30_MODULE_XPLAINED_PRO)
#define NVM_UID_ADDRESS   ((volatile uint16_t *)(0x0080400AU))
#endif

// location found indicates that the other rover found an object
#define LOCATION_FOUND (1)

/************************** PROTOTYPES ***********************************/
void ReadMacAddress(void);
void init_spi(SercomSpi spi);
unsigned char write_spi(SercomSpi spi, unsigned char val);

int main ( void )
{
    bool freezer_enable = false;
    irq_initialize_vectors();

   system_init();
   delay_init();

    cpu_irq_enable();

#if defined (ENABLE_CONSOLE)
    sio2host_init();
#endif
```

```
    // Read the MAC address from either flash or EDBG

    ReadMacAddress();


    // Initialize system Timer

    SYS_TimerInit();


    // Demo Start Message

    DemoOutput_Greeting();


#if defined(ENABLE_NETWORK_FREEZER)

    SYS_TimerInit();

    nvm_init(INT_FLASH);

    PDS_Init();

    demo_output_freezer_options();

    // User Selection to commission a network or use Freezer

    freezer_enable = freezer_feature();

#endif


    // Commission the network

    Initialize_Demo(freezer_enable);


            init_spi(SERCOM1->SPI); // our pic is connected to SPI 1


    while(1)

    {

        Run_Demo();
```

```
  }
}
void ReadMacAddress(void)
{
  uint8_t i = 0, j = 0;
  for (i = 0; i < 8; i += 2, j++)
  {
    myLongAddress[i] = (NVM_UID_ADDRESS[j] & 0xFF);
    myLongAddress[i + 1] = (NVM_UID_ADDRESS[j] >> 8);
  }
}


void init_spi(SercomSpi spi){
        //SercomSpi spi = SERCOM4; // sercom 4 controls transceiver?
        // initialize spi
        spi.CTRLA.bit.ENABLE = 1;
        spi.CTRLA.bit.MODE = 0x3; // master mode (0x02 for slave)
        spi.CTRLA.bit.CPHA = 1; // clock phase
        spi.CTRLA.bit.CPOL = 1; // clock polarity
        spi.CTRLA.bit.DORD = 0; // MSB first
        spi.CTRLB.bit.MSSEN = 1; // slave select
        spi.BAUD.bit.BAUD = 0x0F; // baud rate
        spi.CTRLB.bit.RXEN = 1; // enable sending
}


unsigned char write_spi(SercomSpi spi, unsigned char val){
        spi.CTRLB.bit.MSSEN = 0;
        spi.DATA.bit.DATA = val;

        while(!spi.INTFLAG.bit.DRE);

        nop();

        return spi.DATA.bit.DATA;

}
```

8.3 Data Sheet Links

LiDAR Hokuyo's URG-04LX-UG01
https://www.robotshop.com/media/files/pdf/hokuyo-urg-04lx-ug01-specifications.pdf

Raspberry Pi 3 Model B
https://www.alliedelec.com/m/d/4252b1ecd92888dbb9d8a39b536e7bf2.pdf

Pi Cam V2.1
https://www.raspberrypi.org/documentation/hardware/camera/

TI TPS63061

https://www.ti.com/lit/ds/symlink/tps63061.pdf?ts=1588272468062

BNO055 IMU

https://ae-bst.resource.bosch.com/media/_tech/media/datasheets/BST-BNO055-DS000.pdf

ATSAMR30M18

http://ww1.microchip.com/downloads/en/DeviceDoc/70005384A.pdf

PIC32MX795F512H

http://ww1.microchip.com/downloads/en/DeviceDoc/PIC32MX5XX6XX7XX_Family)Datasheet_DS60001156K.pdf

DecaWave DWM1001

https://www.decawave.com/wp-content/uploads/2019/01/DWM1001-DEV_Datasheet-1.2.pdf