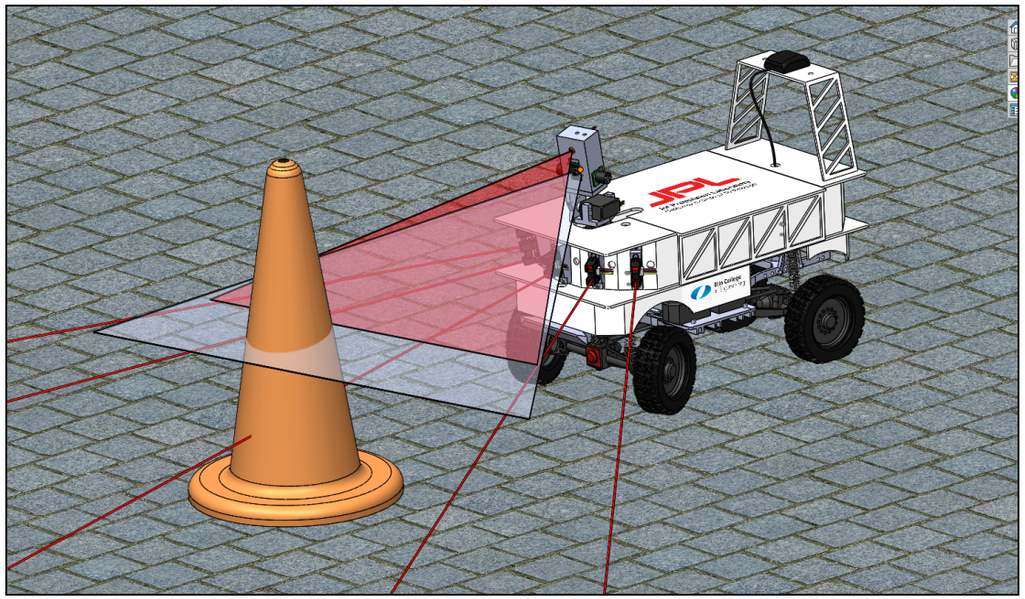
ENGR3390: Fundamentals of Robotics Final Project Rev A

Each year, Fun-Robo has a set of formative **Sense-Think-Act** labs that introduce you to the basic hardware of modern robotics as well as to help you build your software skills in using that hardware. Following this heavily scaffolded set fundamental skill building lab part of the course, we always have a summative, **build an actual robot**, team-based design-built project experience. This year it will be the a (relatively un-scaffolded) design-build-demo of a fully autonomous planetary rover doing outdoor missions around the Olin Oval.



Your course Ninjas have created a **great planetary rover contest** to both stretch your current skill set and to help you integrate the many new skills you have picked up from the labs into an actual working robot that you will carefully design and build to perform a set of complex multifaceted representative missions.

Project Overview

A long time ago in a galaxy far, far away, NASA discovered a planet with geodes made from elements we've never seen before. Back then, they sent small probes to explore in hopes of bringing these elements back to earth for scientific study, but the probes all mysteriously went missing before they were able to return with samples. Rumors have it that aliens reside on this planet and are responsible for these disappearances, but this rumor has never been confirmed. Seeking to learn more about this anomaly, NASA has asked you and your team of world-class engineers to design and construct a small rover that will be able to complete the unfinished work of your predecessors. However, completing this work will be no small feat - rovers will need to be fully autonomous due to the communication barrier posed by the insurmountable distance to this planet while being robust and adaptable enough to navigate around the challenging terrain.

Project goals, Your team should:

1. Design & fabricate a body for your rover (you're provided with the chassis, but you need a body to mount your sensors and electronics to)
2. Create sensor mounts to attach sensors to your rover
3. Design and construct a mechanism to drop a payload
4. Wire up sensors to the raspi & create a wiring diagram
5. Write code that will allow your rover to autonomously complete the following tasks at least at the MVP level:
    a. Waypoint navigation/obstacle avoidance
    b. Docking
    c. Payload delivery
6. Choose an additional 3-4 stretch goals to attempt based on your team's interests

ENGR3390: Fundamentals of Robotics Final Project Rev A

Demo day format:

1. There will be three rounds, each focusing on a different behavior (waypoint navigation/obstacle avoidance, docking, and scientific sampling)
2. Between each round, teams will be able to (optionally) re-upload code to their rover
3. Each team will have 5 minutes during each round to showcase their rover's abilities related to that behavior
4. Completing each task will provide your team with 10 "mission points" - Note that your grade is NOT dependent on the number of points your team ends up with, but rather the quality of the report and the level of effort put into this project
5. At the conclusion of the event, the following awards will be awarded by a panel of guest judges:
    a. Best engineered rover
    b. Prettiest rover
    c. Most daring rover design

6. We will keep track of progress using a table similar to this:

	Team 1	Team 2	Team 3
Waypoint Navigation			
MVP: Navigate through the "wasteland" (no obstacles)	✓	✓	✓
Navigate through the "rocky road" (stationary obstacles)	✓	✓	✓
Navigate through the "Alien territory" (moving obstacles)	✓	✓	
Docking			
MVP: Reach the supply station	✓	✓	✓
Park within the medium box	✓	✓	✓
Park within the small box			✓
Return to starting dock	✓		
Scientific Payload Delivery			
MVP: Deliver the payload to any of the science drop boxes	✓	✓	✓
Deliver the payload to a specific science drop box	✓	✓	
Retrieve a payload			✓
Return to starting dock after delivery/retrieval		✓	✓

ENGR3390: Fundamentals of Robotics Final Project Rev A

Hardware, Project supplies list:

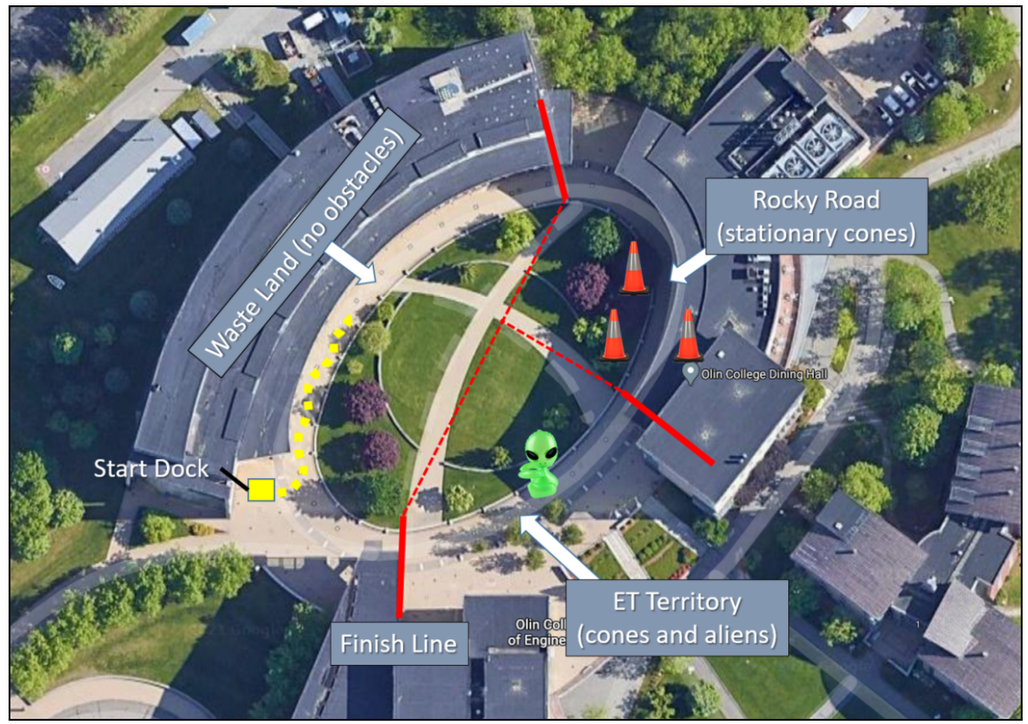
- A rover that has the low-level systems built, where the rover's steering and velocity can be controlled with PWM signals from the raspi.
- 2 sheets of Sintra board to make rover hull.
- 2 Maxbotix Ultrasonic sensors (MB1340-000 sensors, see appendix for detection range).
- 5 Sharp IR sensors (GP2Y0A41SK0F Model, 4-30cm detection range)
- 1 Raspberry Pi Camera Module v2 (8MP camera)
- GPS
- Servo Pan-Tilt subassembly
- 1 Additional Servo
- Two batteries

Note: Your team can use as many of your student Raspberry-Pis as you like. The above supplies will be available to your team, but you are not required to use all of the supplies listed. Spec sheets for each of these sensors, sample code, rover details, and other specific hardware-related info will be available on the Canvas website..

Mission 1: Waypoint Navigation & Obstacle Avoidance

The first thing your rover will need to be able to do in order to complete its deep space sampling mission is to navigate between waypoints, and to avoid obstacles along the way to prevent getting stuck. Your rover should at least be able to navigate between set waypoints, and ideally be able to adjust it's planned path to route around any obstacles it encounters. Beware of the mobile ET!

For this challenge, we've sectioned off the oval into three parts - a "wasteland" with no obstacles, a "rocky road" with large stationary obstacles, and "ET territory" where there will be large stationary obstacles and a mobile ET to avoid.



The bright-green ET lifeforms on this planet stand out from their environment and may look harmless, but be careful! They are mobile, and love to pick on unsuspecting rovers to steal parts for their own rovers - avoid them at all costs! (Note: the ET's rover will be controlled by one of your friendly neighborhood CAs, and will generally attempt to follow/block your rover) (Note 2: we won't actually steal your hardware, but if you get caught by the ET you won't get mission points for crossing the finish line)

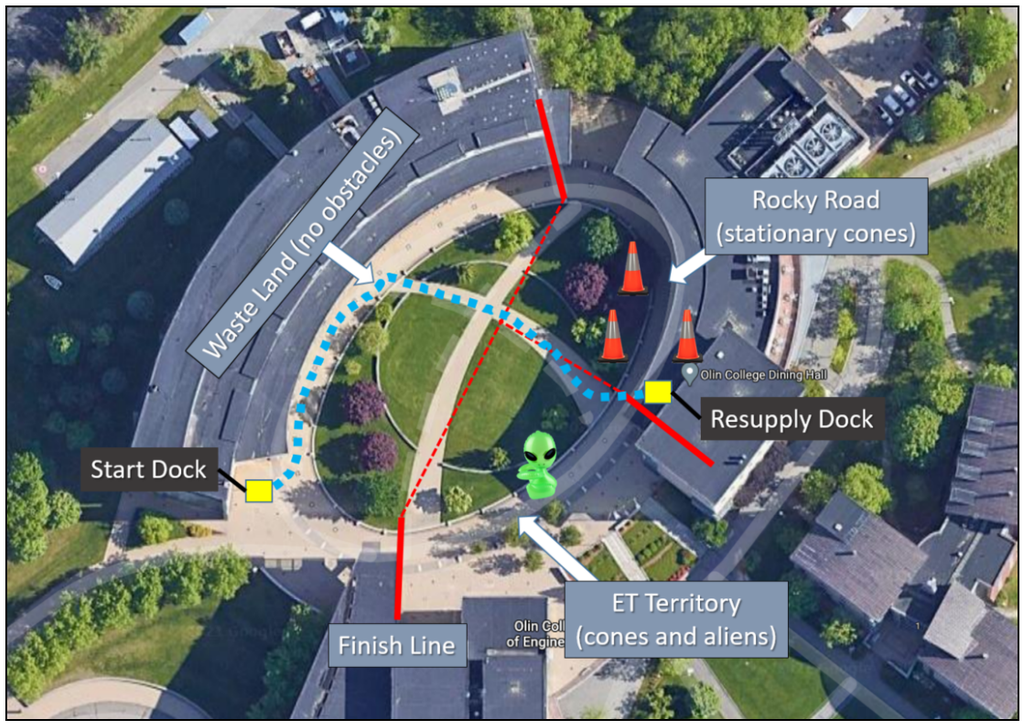
Mission Objectives Summary:

- MVP (required): make it through the first third of the oval with no obstacle's "wasteland"
- Stretch goal: make it through the second third of the oval with stationary obstacles (orange cones)

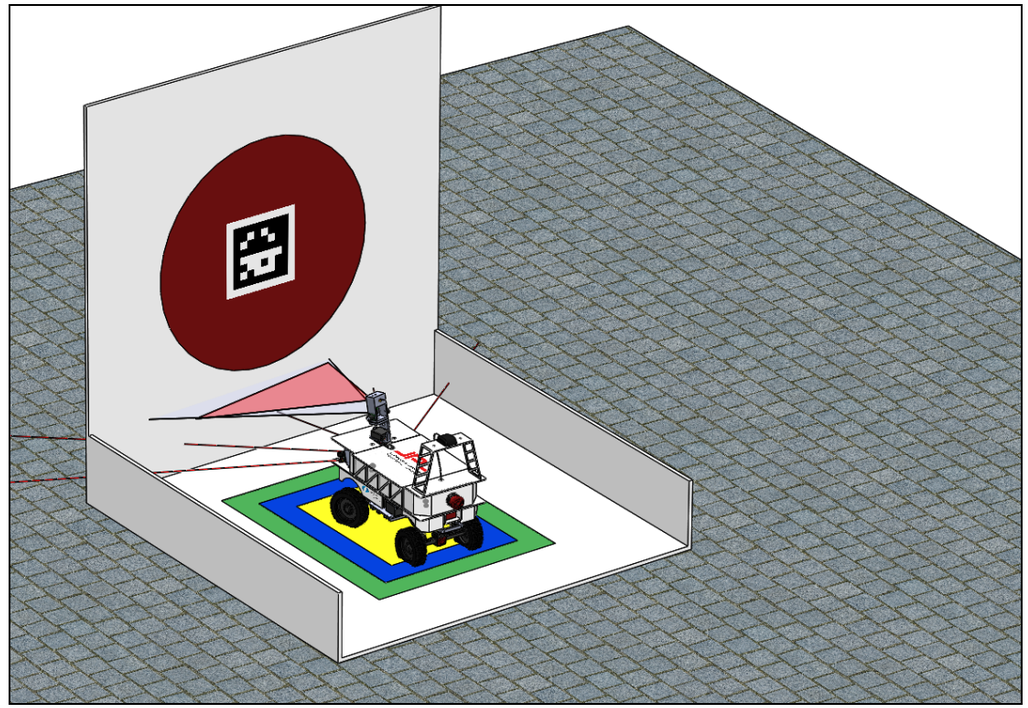
- Stretch goal: make it through the final third of the oval with stationary and moving obstacles (like the previous section, but with a green CA-driven ET roaming around)

Mission 2: Docking

Being able to precisely park your rover to resupply stations is key for a successful deep space sampling mission! Your rover should be able to drive across the oval from the starting dock to the resupply station (and if you'd like a challenge, have your rover also return to the starting dock!)



To help with alignment, we've provided you a board with a large red dot in the center, and included side walls on the docking platform.



To evaluate your rover's docking accuracy, the station's base has an "archery" target with a small, medium, and large box. The small box has 1in of clearance on all sides of the rover's wheelbase, the medium box has 3in of clearance, and the large box has 6in of clearance.

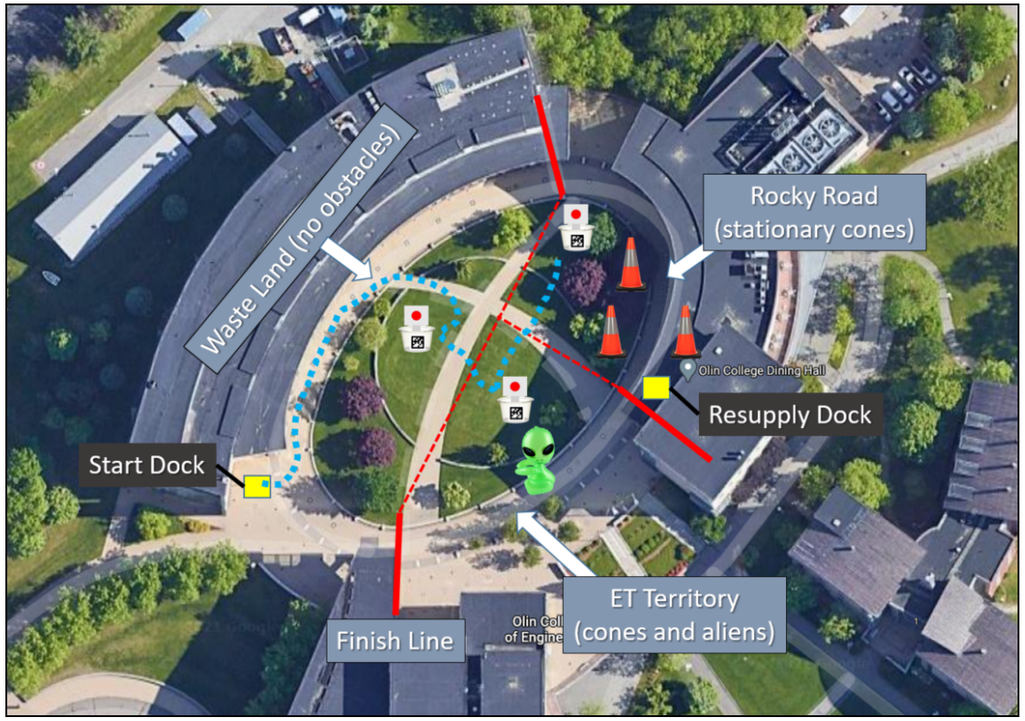
Mission Objectives Summary:

- MVP (required): successfully reach the resupply station and have all 4 wheels on the station's base
- Stretch goal: Precisely park the rover such that all 4 wheels are within the medium box

- Stretch goal: Precisely park the rover such that all 4 wheels are within the small box
- Stretch goal: After reaching the resupply station, be able to return to the starting dock

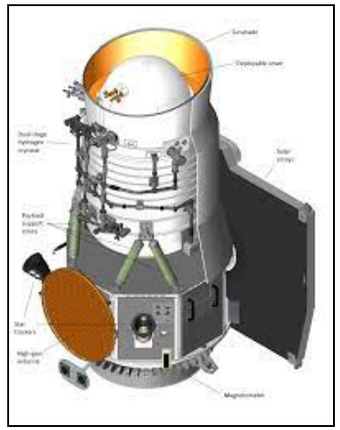
Mission 3: Scientific Payload Delivery

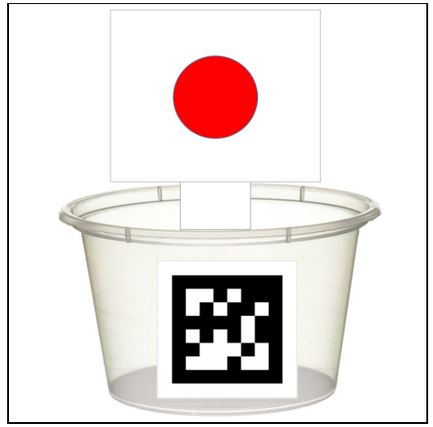
Being able to successfully deliver a scientific payload is crucial to any good exploration rover! While the rover is at the resupply dock, you can pre-load a geode container onto it, but your rover should be able to autonomously deliver the container to one of the science drop boxes.



This mission will be occurring on one of the larger grassy areas of the oval, where the terrain may pose a bit more of a challenge. The drop boxes should be in visible range of the starting dock, but don't count on a hard-coded path since the exact locations of the dock and the boxes won't be exact.

The "scientific payload" will be a small 3D printed object that's (hopefully) designed to be easy to grip.





The drop boxes will be a short (~2in tall) tupperware box with a red dot and apriltag to help with alignment. For the MVP, we'd like your rover to be able to drop the scientific payload into any one of these boxes. Each team will have a separate box with a unique AprilTag identifier. Dropping the payload into any one of the boxes is good enough for the MVP, but delivering the payload to your team's box will provide you with more mission points! You can also retrieve the extra payload on the closed drop box for additional points.After a successful delivery/recovery mission, your team will also get bonuses for returning to the starting dock!

Mission Objectives Summary:

- MVP (required): Deliver the payload to any of the science drop boxes
- Stretch goal: Deliver the payload to your team's specific science drop box (note: you only get mission points for this if it occurs intentionally, not randomly)
- Stretch goal: Retrieve a payload
- Stretch goal: Return to starting dock after delivery/retrieval

ENGR3390: Fundamentals of Robotics Final Project Rev A

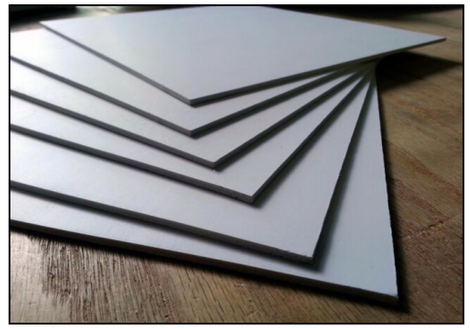
Having reviewed the contest, the following three sections will provide you with a bit of background and some technical help in the mechanical, electronic and software design of your planetary rover.

Mechanical Design

Your team's planetary rover will be built on top of a slightly used **Axial SCX10 II Deadbolt 4WD RC Rock Crawler Off-Road**

https://www.amazon.com/Axial-Deadbolt-Off-Road-Electric-Crawler/dp/B07CQ8YNHM/ref=dp_prsubs_2?pd_rd_i=B07CQ8YNHM&psc=1

This base vehicle has phenomenal low-speed traction and maneuverability as well as being based on a 1979 International Scout, perhaps the best COTS off-road vehicle ever made (yes I drove one for many years). The fancy plastic Scout shell has been removed and your team's design task is to make a reasonably weatherproof robot shell out of 3mm Sintra board to house your electronics and sensors.



Sintra is an easily worked, more robust version of the foam core panels used for posters in science fairs and Olin's EXPO. It can be cut with an Exacto knife, solvent bonded, cnc-routed, hot-melted and thermally formed into a wide variety of 3d shapes. It is commonly used for making cosplay armour. Please see embedded videos for working with Sintra, before starting on your robot hull design.

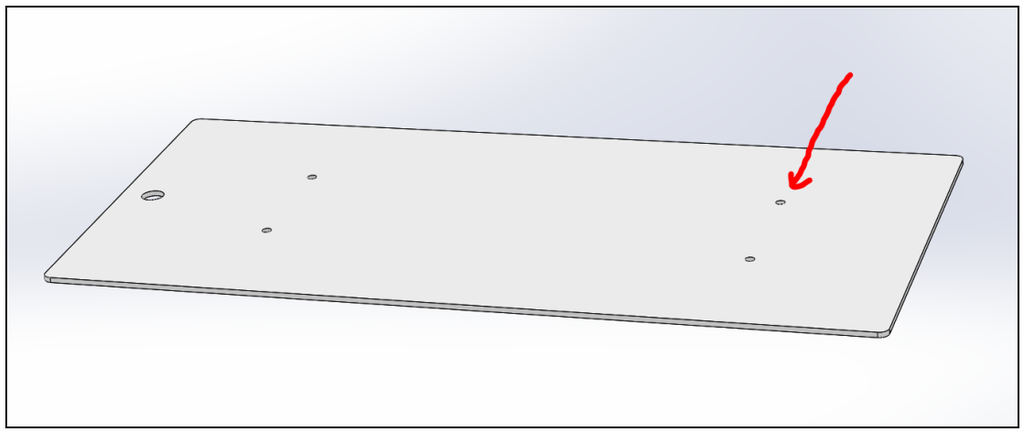
Bottom Hull Design

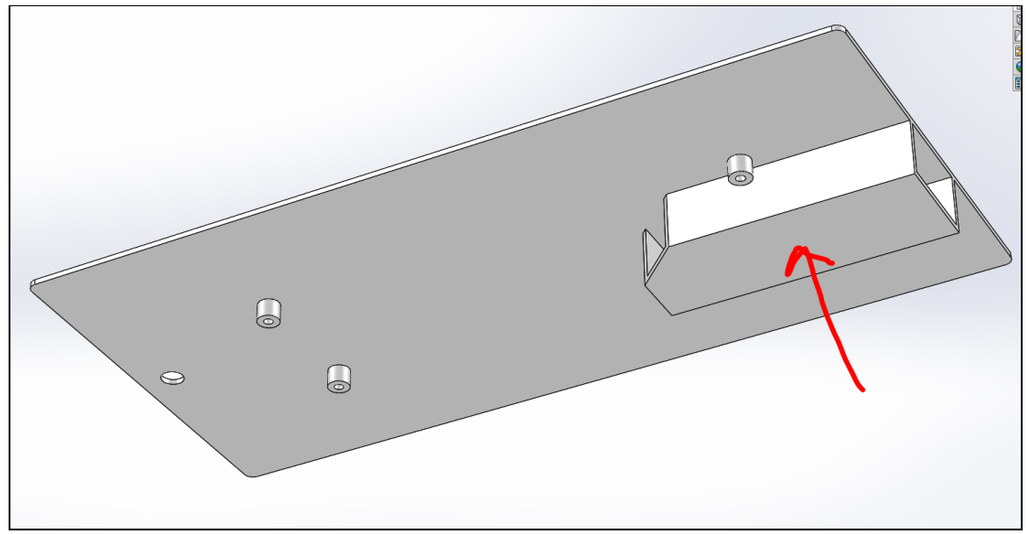
Your team can design any size and shape bottom hull for your rover. Your goal is to build a nice looking Sintra box that can hold and protect all of your electronics while providing appropriate attachment points for your sensor suites. You will want to make sure you have hand clearance to assemble components into the box and that you leave ample space for wiring so that you can connect up all of your electronic parts. Please also consider leaving space to add and remove batteries and to plug your Raspberry Pi into shore power when you are coding for long periods of time.

The Axial SCX10 chassis has 4 hardpoints that sit above the shock absorber towers to hold the shell in place. We will insert 4 threaded inserts in them and use that to attach the baseplate of your Bottom Hull to it.

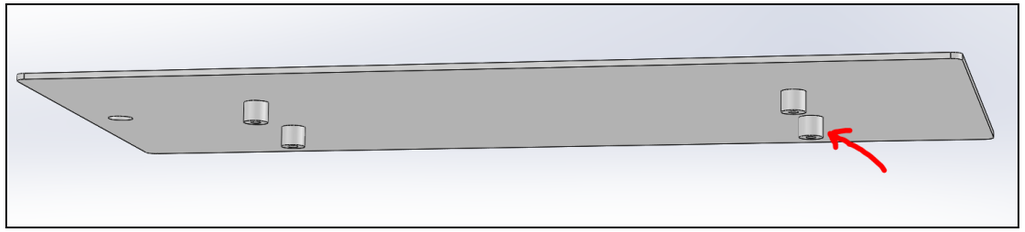
ENGR3390: Fundamentals of Robotics Final Project Rev A

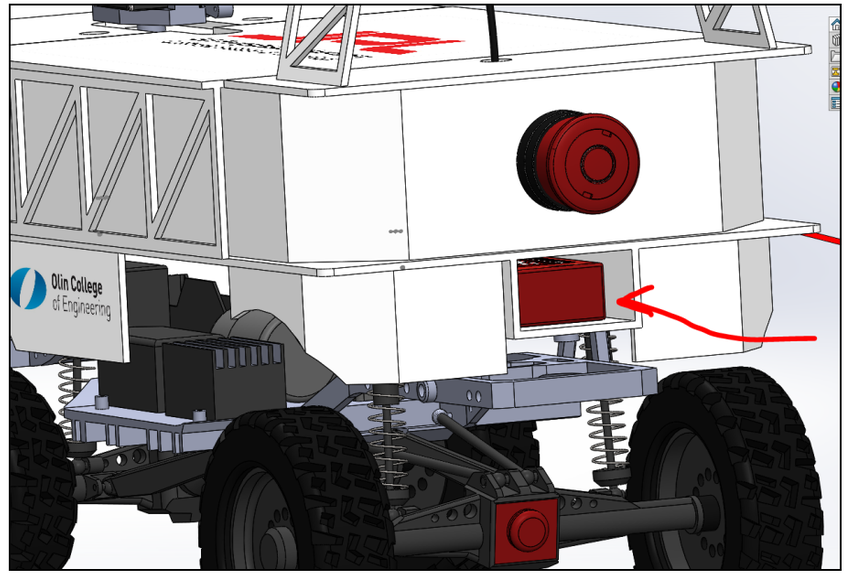
To start your design, lay out a 3mm thick (Sintra sheet is 3mm) bottom plate with the 4 attachment holes in place (there is a complete bottom hull uploaded to the final project file folder in Canvas that you can take reference dimensions from):





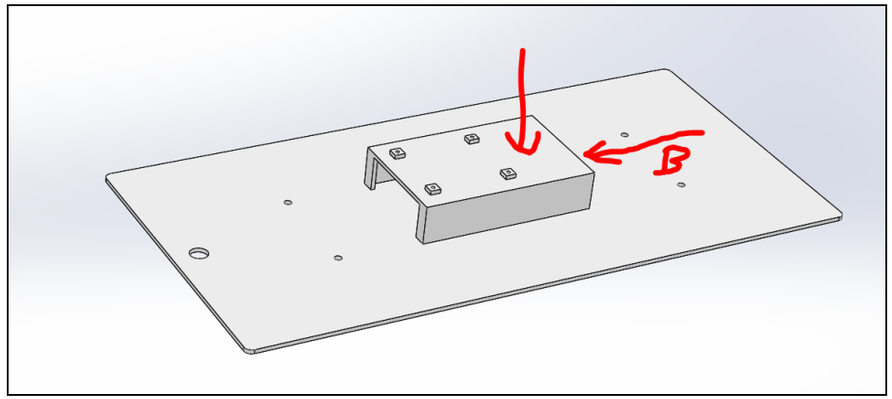
Add about 9mm (three sheets of 3mm Sintra) spacers on bottom to get your baseplate up over the Rovers differential:

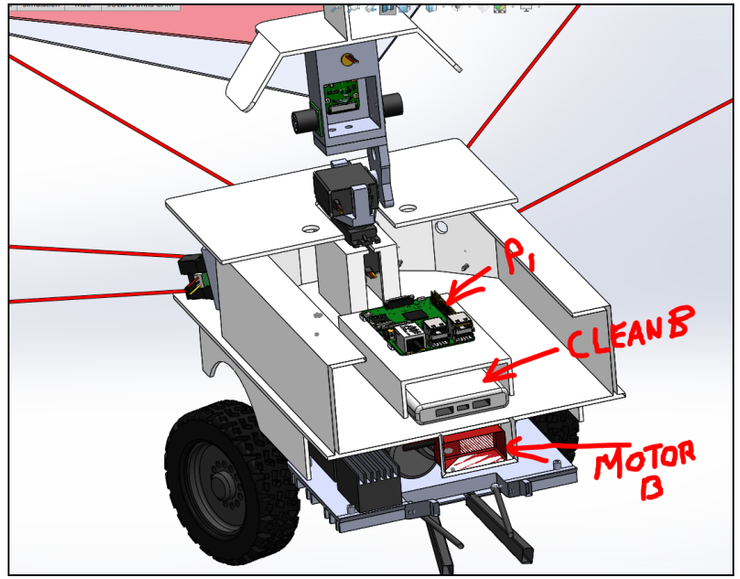




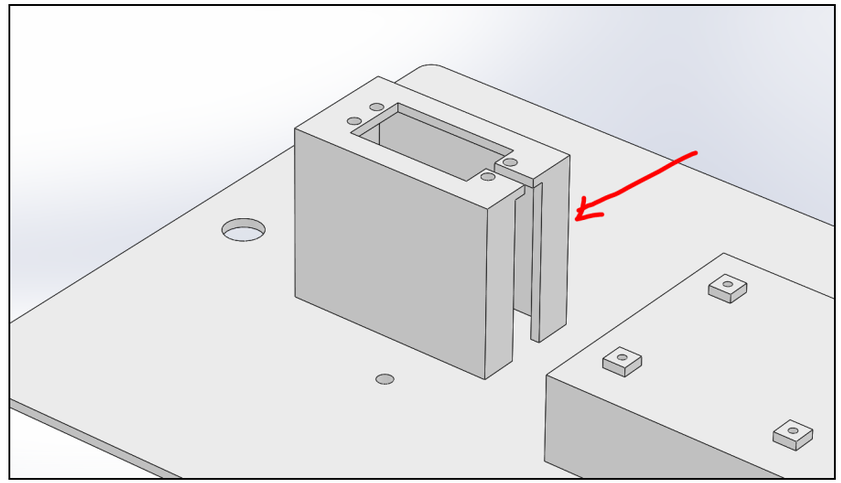
Then add a rover drive battery (Venom 5000, part file in Canvas) holder down below the stern to keep the weight low and centered. Make sure you leave a spot for power wires to exit the battery holder toward its front end. All construction needs to be done with cut and welded 3mm Sintra panels. Using tabs and slots for alignment is good.

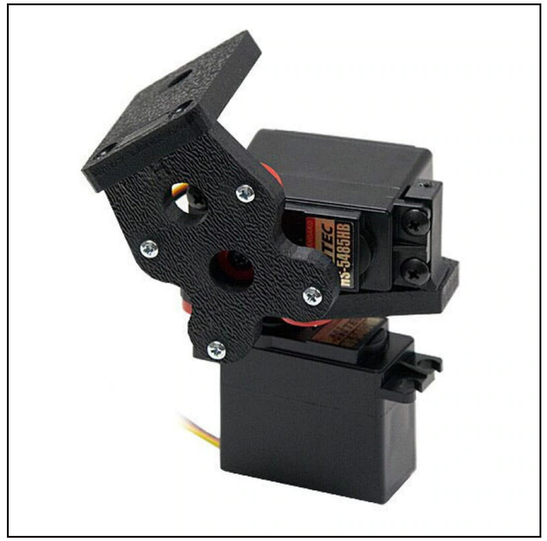
Next add a raised platform in the center of your Rover to hold your Team's Raspberry-Pi and to provide a captive battery holder for its battery. Every Rover will have one battery for clean computer and sensor power and one dirty battery for drive motor power. This raised platform will hold the clean battery,



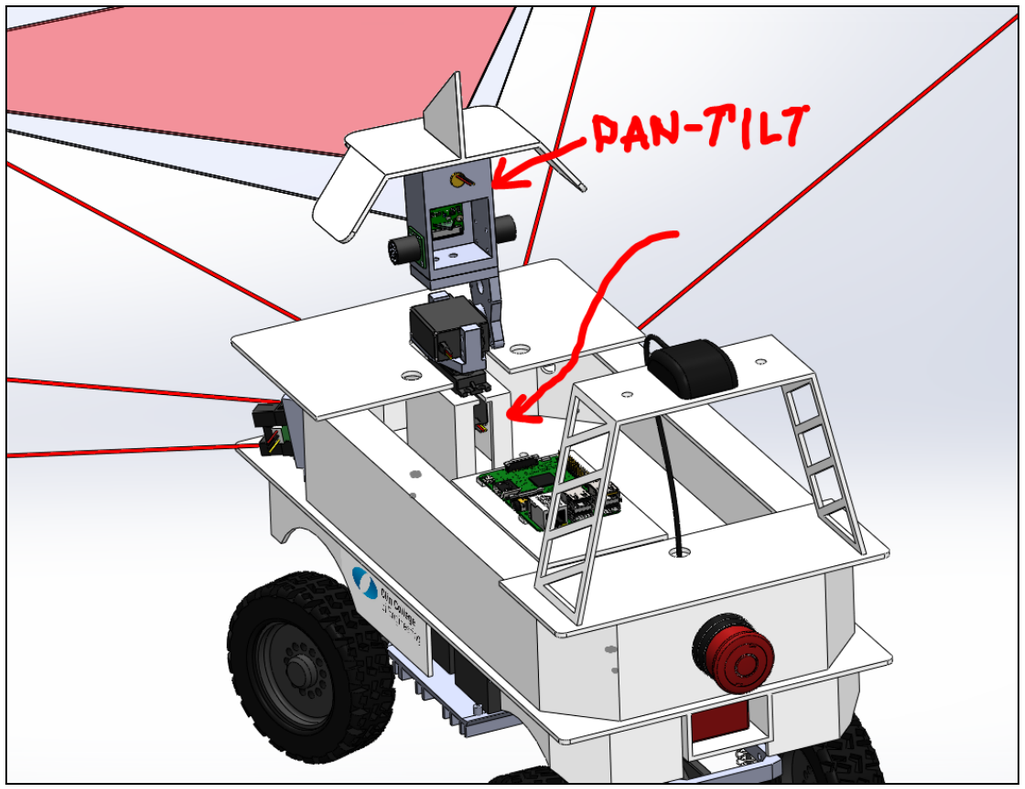


Right in front of Pi platform, create a raised platform for the pan motor of the supplied Servo-City pan-tilt unit.



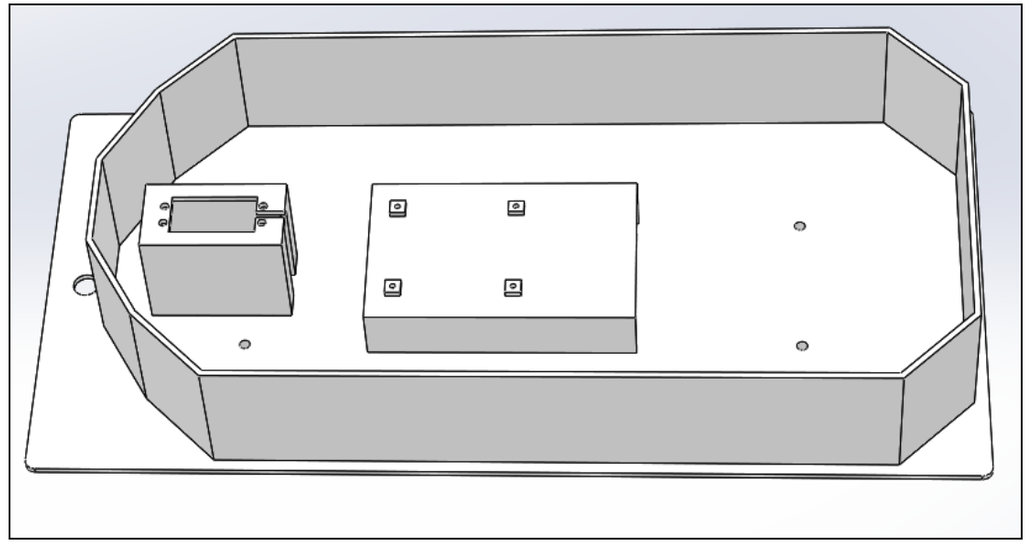


Make sure you leave space for wires and servo body to drop through platform

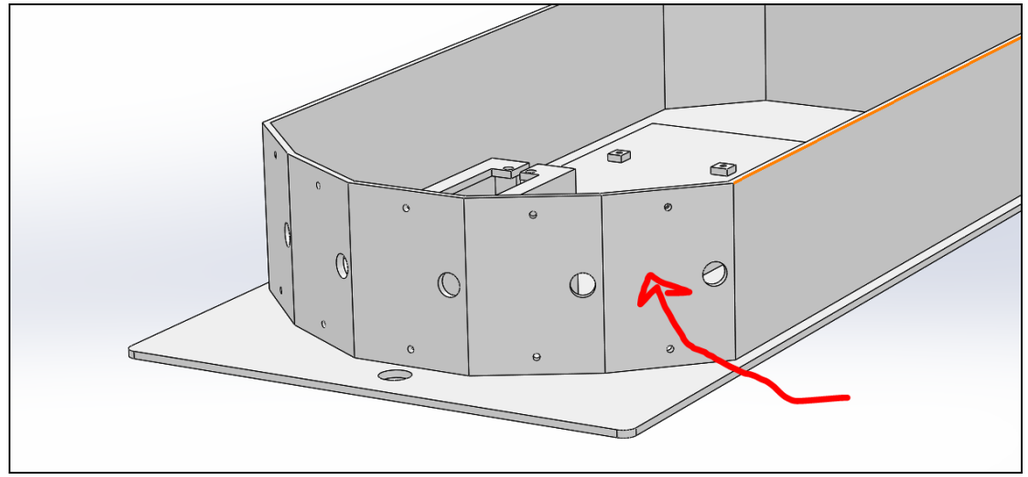


The pan-tilt unit is supplied with all of its parts made, your team just needs to assemble them. You will need to design a sensor head and an anti-sun blinding hat to go on its tilt stage (as shown above, once your team has figured out what sensors to put where.

Having designed the main subassembly mounting features of your lower hull part, its time to come up with a design that will both let you mount most of your stationary sensors as well as provide a dirt, dust and moisture barrier for the delicate electronics that sit inside your rover. This box doesn't need to be downpour grade watertight, but it would be advisable to get it as sealed up as you can, while making it look as cool as your team is able. Please create a set of bottom hull walls to solvent bond to bottom plate as shown:

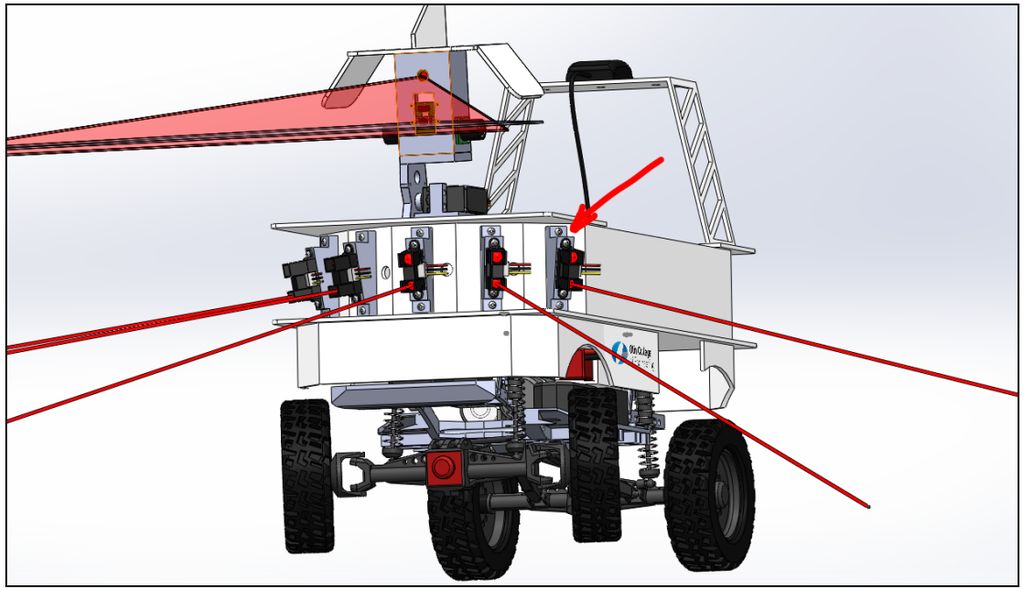


Once hull shell design is done, please punch out mounting holes for yoru fixed sensor array as shown below:

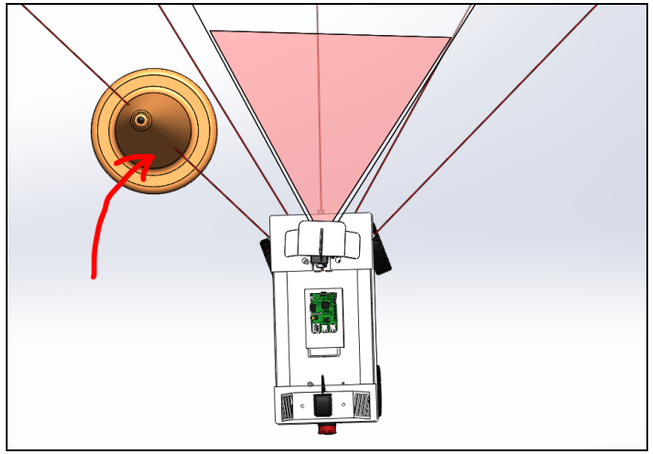


Don't forget the wire passthrough holes.

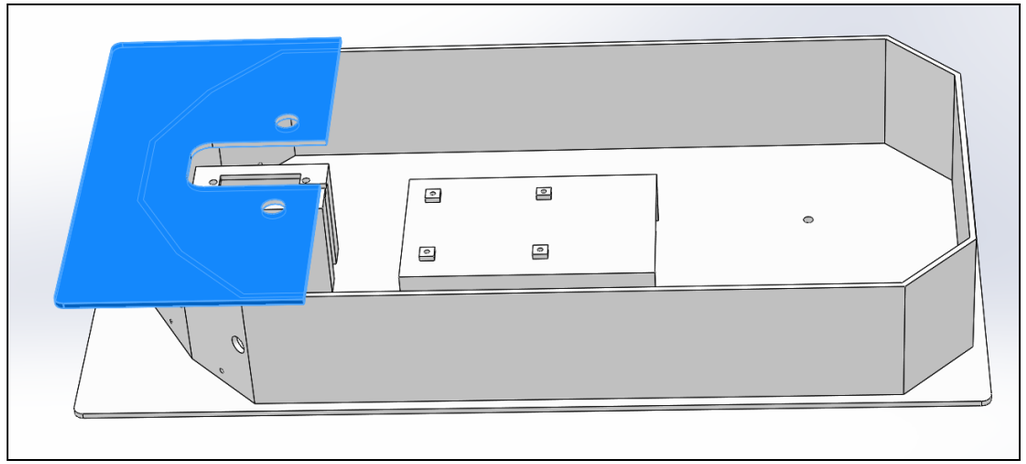
You will use self tapping plastic screws and some 3d printed mounting blocks to mount your team's sensors to the Sintra bottom shell:



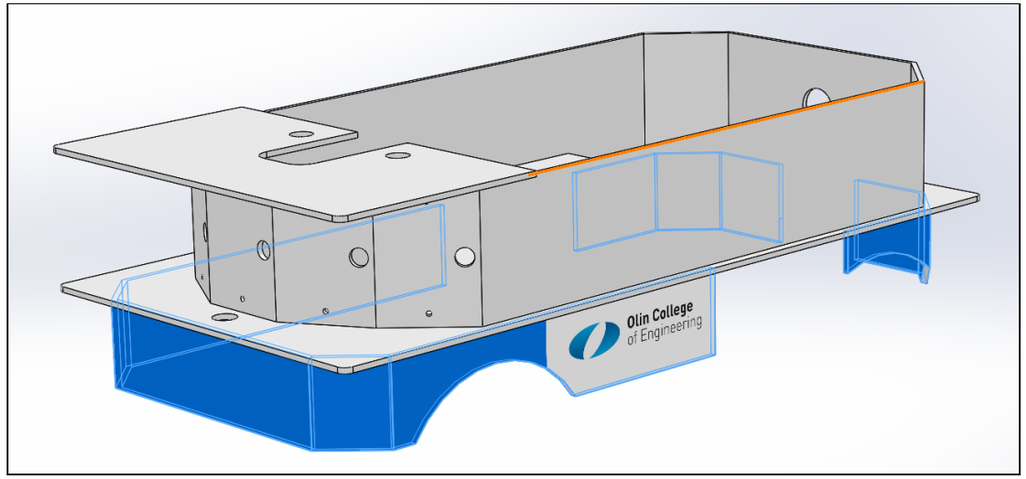
Think carefully about sensor placement, you want to be able to find and range to docks and cones easily:



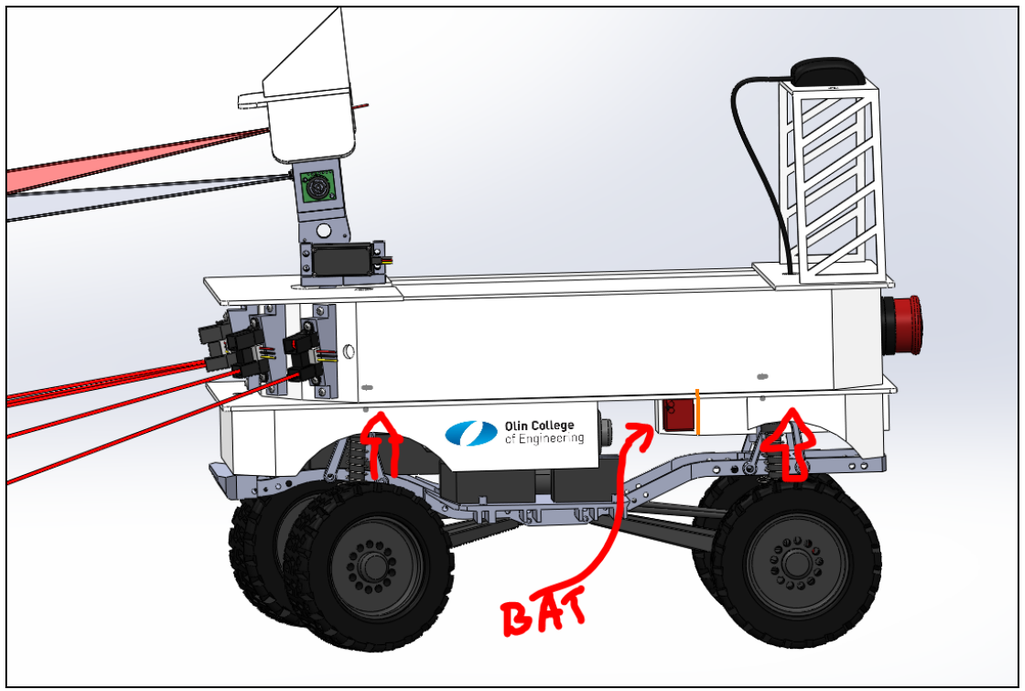
Add on a foredeck to protect sensors from solar glare and rain and to close off front part of shell:



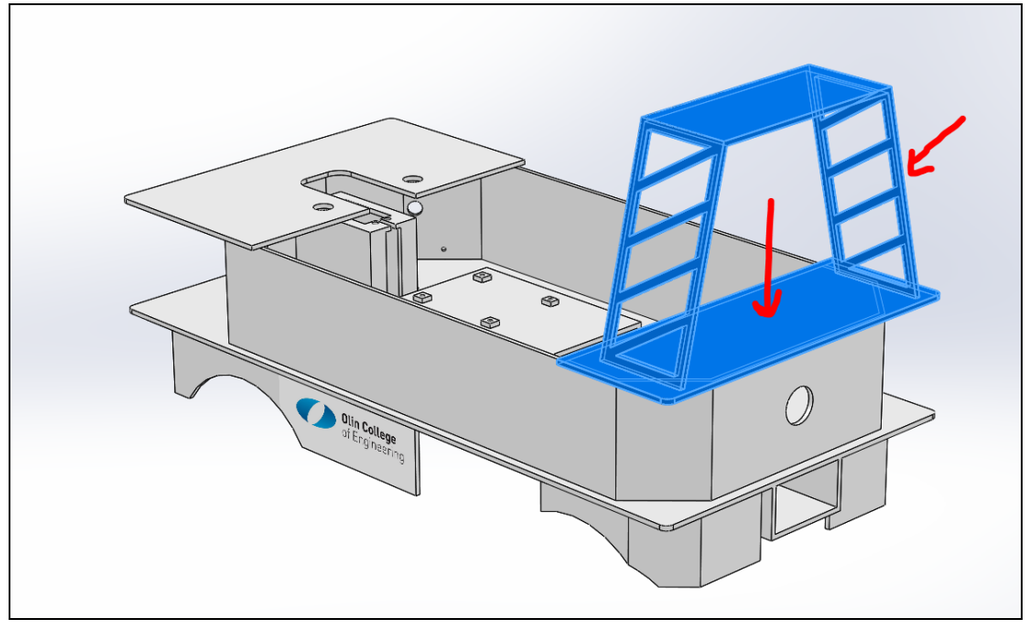
Add some below decks rocker panels to keep puddle splashing down and to give you a good place to put your team's name and Olin Logo:



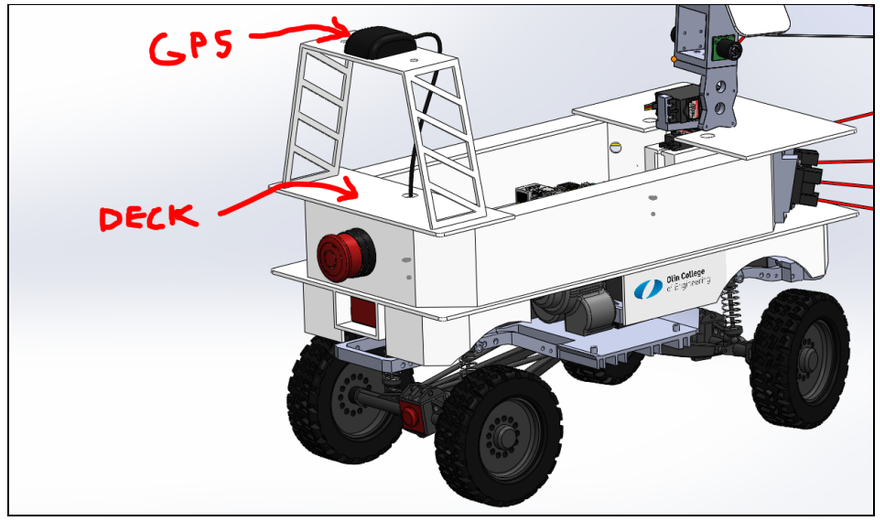
Make sure you have enough of a wheel cut out to clear your rover's monster tires and soft suspension (you will need it for the third mission!). And leave finger access to plug rovers motor battery into rover as shown:



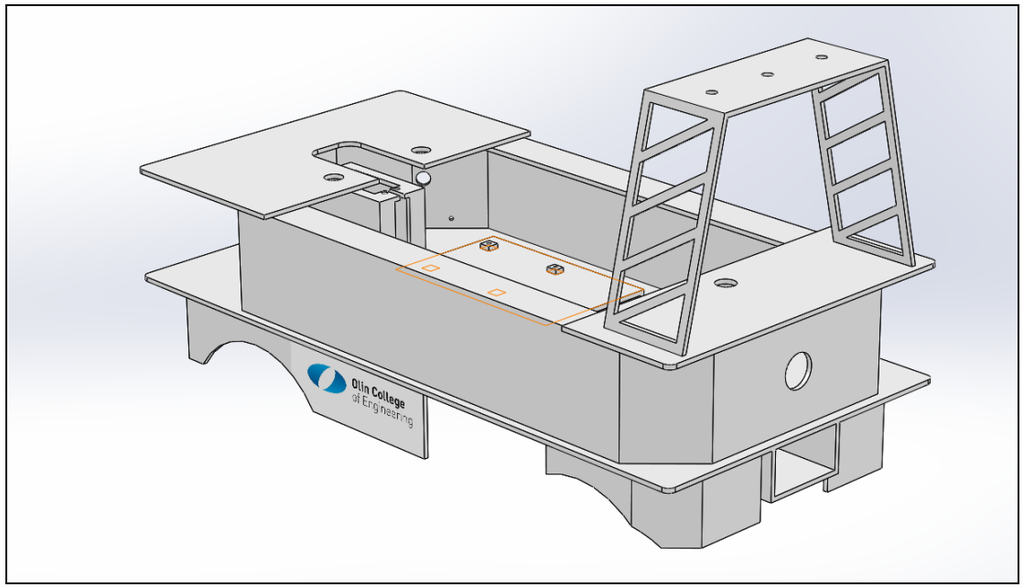
Your bottom hull is almost done at this point. You still need to add a rear deck to close off the stern of the hull and then put a lightweight GPS tower on it to get your rover's GPS as far off the ground as you can. Please plan to put some form of payload deployment system under the GPS so that you can precisely drop scientific payloads when required. Overall your system should look somewhat like this:



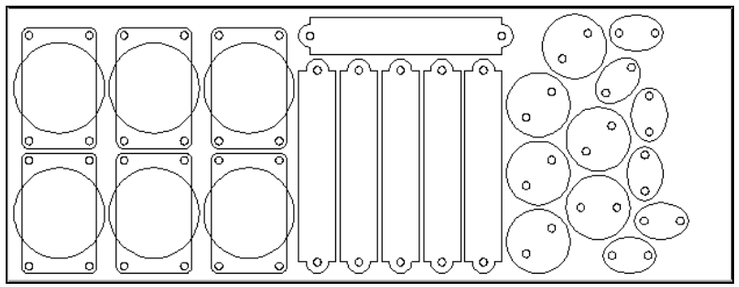
And when assembled look like this:



Finally, punch in a mounting hole for your big red emergency stop button (all robots should have one!), any cable tie points, wire pass throughs, etc.Ideally all of your wires will stay with the bottom shell and not need to be removed after assembly. Then go through and add structural reinforcements wherever anything looks like it might be too flexible or wobbly, like the hull side walls.:

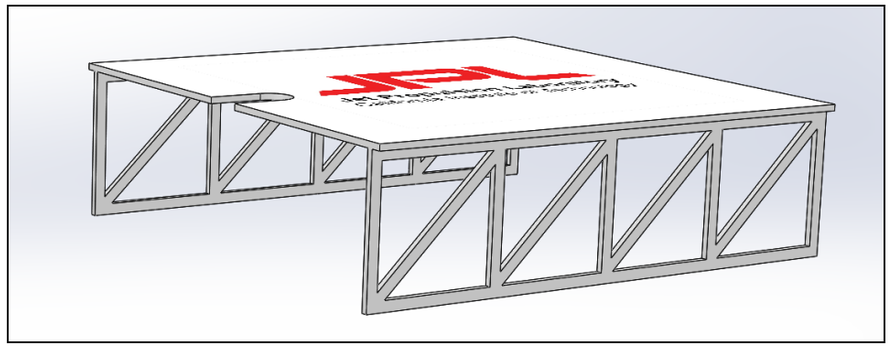


When done, lay out all parts hull parts flat in a single SolidWorks assembly drawing, packed tight, minimizing material waste and then print out full size and tape down to your Sintra sheet and start cutting them out with your Exacto knife:

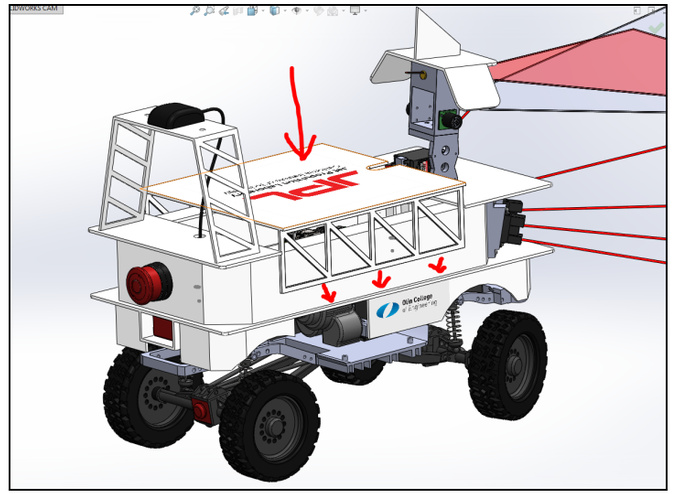


Top Cover Design

The top cover of your hull is going to be going on and be taken off all the time, so design it to be easy to remove. It should also be as light as possible, while closing off the box to provide environmental protection for the fragile electronics inside. And you should take this opportunity to make it look as cool as possible to help win your tema prize for best looking rover. A simple U-shaped shell will do:

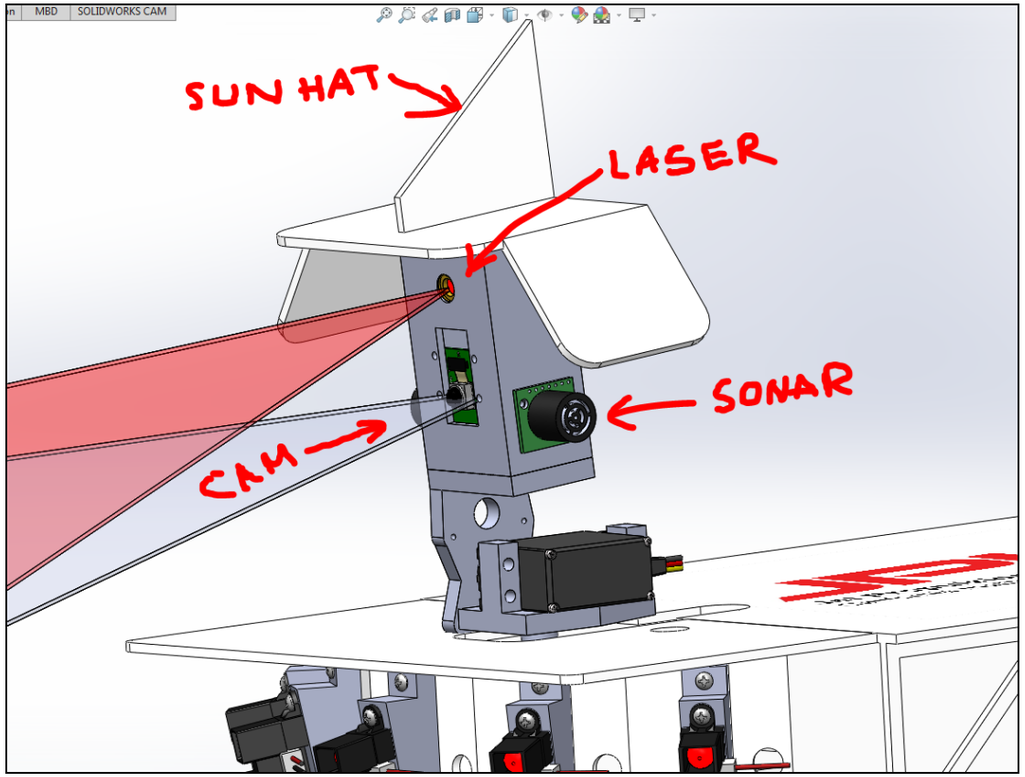


Make sure to leave wire clearance holes for the sensor head and a thumb hole to help get a grip on it to pull it up and off. Make from remaining Sintra sheet:

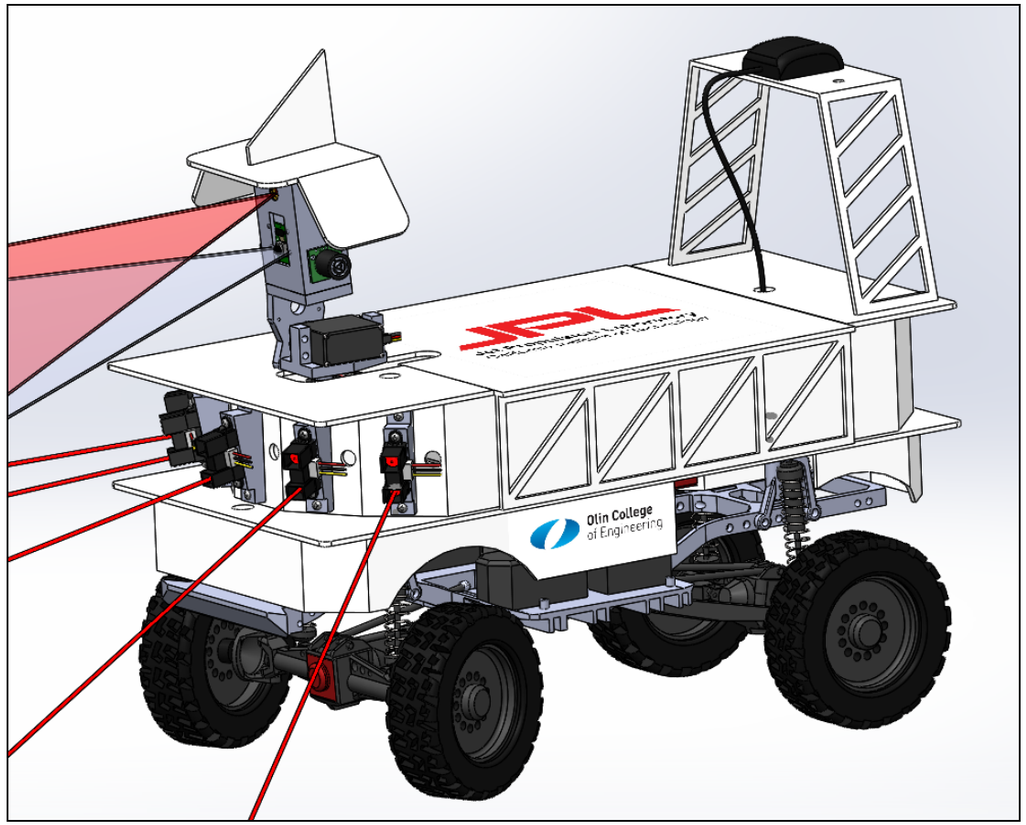


Sensor Head Design

When you only have a few expensive sensors, it makes a lot of design sense to place then in a pan-tilt head (this is why your eyes,ears,nose and tongue are all in your head!). Think carefully about how to maximize your sensor coverage. Remember you need to detect cones, docks and aliens and then design your sensor head to contain a structured light laser, the Raspberry V2 Pi-Cam and possibly your two sonars. Take some time and draw out the options and have a good team discussion about the best mix. When ready, create a single 3d-printed pan-tilt sensor head as shown:



Your completed Sintra rover body shells should look something like this:

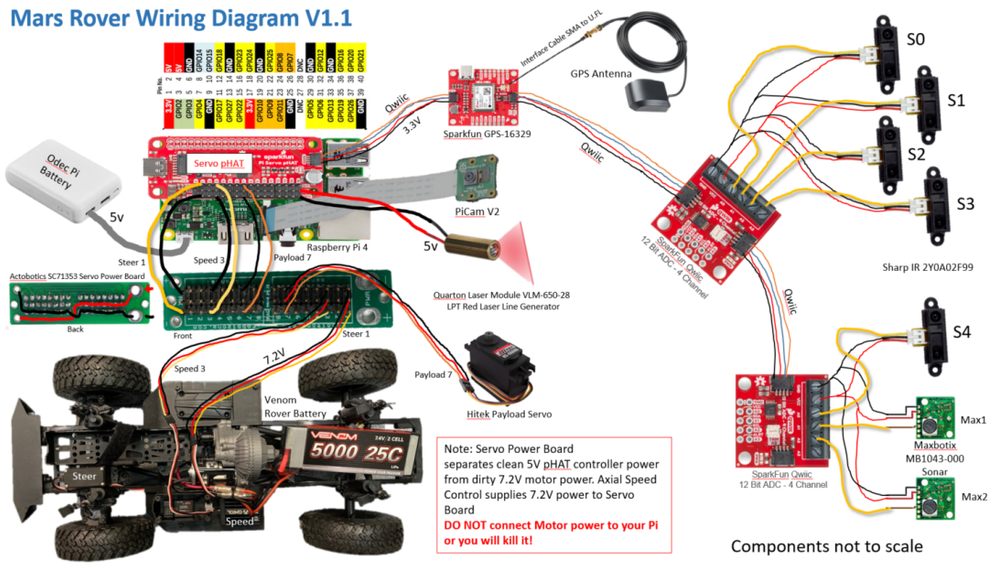


This task is purposely only lightly structured to give your team's MEs and builders something substantial to work on after a semester of software and wiring. Please feel free to design your team's shells any way you like. It just has to be functional, light and good looking (no ugly robots in Fun-Robo!). The payload deployment system is fully unstructured to give your teammates a chance to design an important sub-system from scratch. You have one servomotor, a lot of Sintra and an Exacto knife, go for it!

ENGR3390: Fundamentals of Robotics Final Project Rev A

Electronic Design

Your planetary rover has a lot of electronic components, fortunately the electrical hook up is pretty straightforward. An overall electrical wiring diagram can be found downloaded from Canvas website and is given below:



It is recommended that you print out a full size version, when wiring your rover up. Please note: **YOUR RASPBERRY PI IS VERY DELICATE, NEVER HOOK UP ANYTHING HOT !!!**

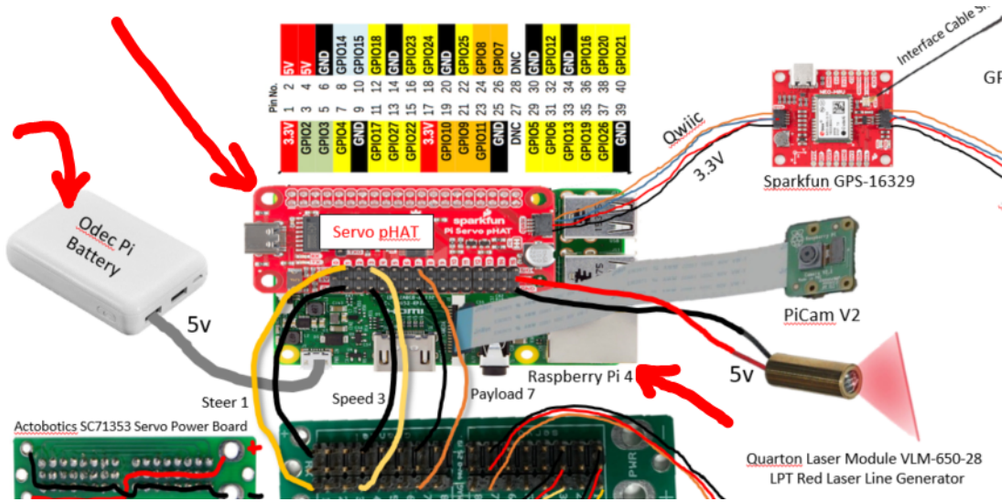
Unlike the Arduino class of embedded controllers, yoru Arduino uses fairly fragile CMOS logic and as such voltages above 5V of static discharge from you wiggling your but in a chair can easily burn out input pins or the whole board. It is good practice to never put your board down on anything conductive, never touch it except by the edges and never ever connect or disconnect anything to it when it is powered.

A really sad way to screw the final demo is to hotplug something in the morning of demo day and burn out your whole board. Never plug in hot, you have been warned.

Lets walk through the electronic hook up part by part, so that you can design around each accordingly.

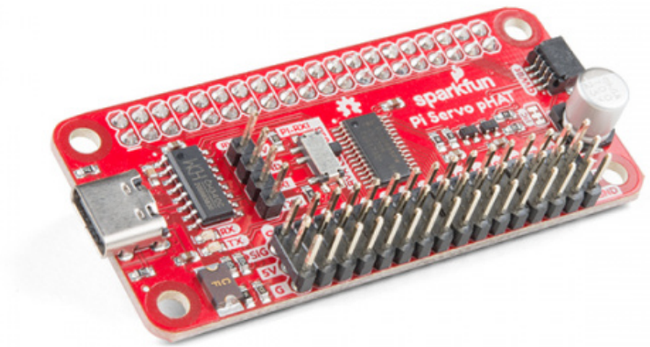
Raspberry Pi Controller

Yor rover will be controlled by a student supplied **Raspberry Pi 4** powered via its micro USB jack from a 5VDC **Odec Lithium battery pack**. You will get two Odec battery packs, it's recommended you keep one on charge while developing with the other so that you don't need to recharge during active testing. It is estimated that each battery pack will be good for at least an hour long mission. Your Pi's IO capabilities will be enhanced by adding a SparkFun **Servo pHAT shield** that will both drive your PWM servos and provide a Qwiic (i2c bus) connector to support yoru rovers i2c sensor network:



Details on the Sparkfun Servo pHAT can be found here: https://learn.sparkfun.com/tutorials/pi-servo-phat-v2-hookup-guide?_ga=2.129754255.2089808247.1616942563-915018044.1593288880

The SparkFun Pi Servo pHAT provides your Raspberry Pi with 16 PWM channels that can be controlled over I2C. These channels are broken out in a header combination that is perfect for connecting servo motors. Additionally, the PWM channels can control other PWM devices as well.

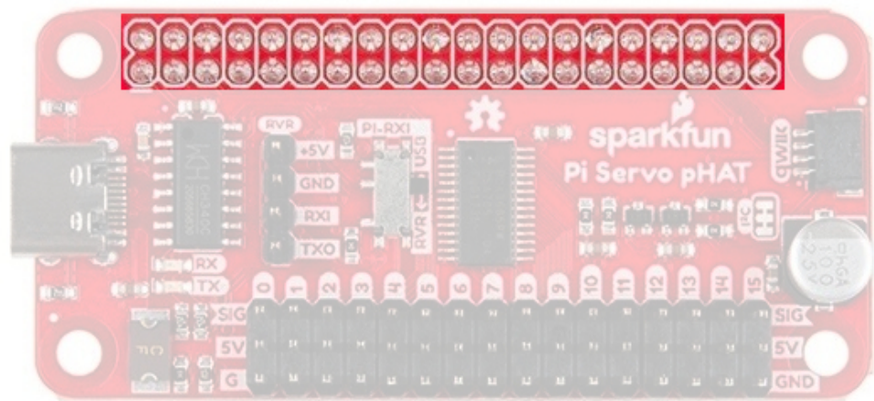


There are several functional components of the hat, which are designed to be as foolproof as possible. Although we have attempted to take precautions to guard against the most common user errors, users should still be wary. Most users may already be aware of these common pitfalls. However, if you have forgotten or maybe you have never used a Raspberry Pi (or similar single board computer), they are highlighted throughout this section just in case. There is a lot of detailed content in this section; in general, users primarily need to:

- Be cautious of any loose connections and avoid shorting or bridging 5V and 3.3V on the Raspberry Pi.
- Be aware of any potential current draw limitations.

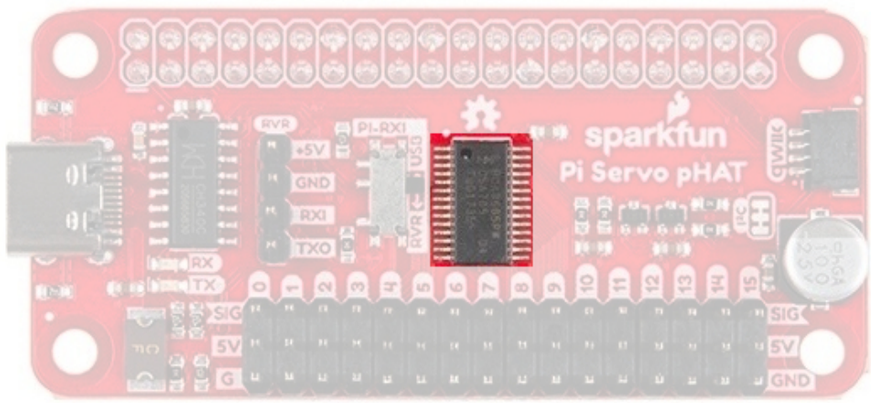
> **Danger:** With connections provided to both the **5V** and **3.3V** pins of the Raspberry Pi, please be cognizant of any loose wires. **A short between the 5V and 3.3V lines will permanently put your Raspberry Pi out of commission**.

On the Pi Servo pHAT, 3.3V power is drawn from the Raspberry Pi 3.3V pin, on the 40-pin GPIO header, to power your Qwiic devices and for logic level translation. You cannot power the Raspberry Pi through the 3.3V pin; in fact, you should never connect another power source to this pin. It is intended to be used only as a power output for the Qwiic connect system.



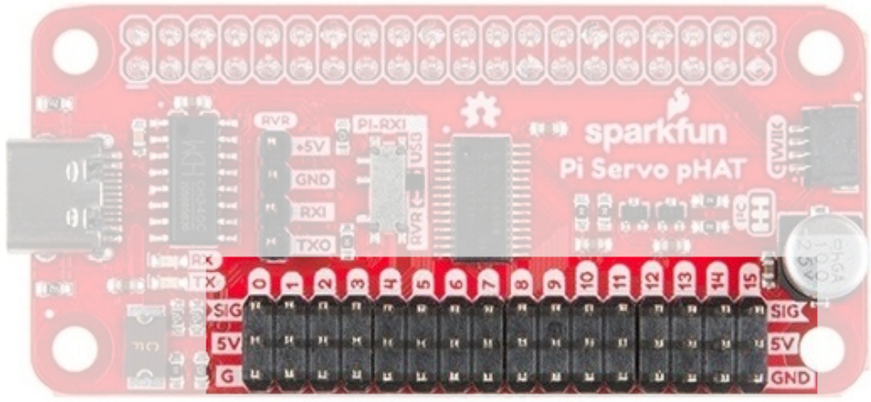
The 3.3V line is primarily used to power any connected Qwiic devices. In general most users will not reach the current limitations of the Raspberry Pi. However, if you intend to sink a bunch of current or have a ton of Qwiic devices, make sure to double check your current draw and the limitations of the Raspberry Pi you are using. As an example, the Raspberry Pi Zero W uses a PAM2306 switching 3.3V regulator (found in the schematic for power regulation), which is rated up to a 1A output current.

The PCA9685 provides I2C control over the 16-channels of 12-bit pulse width modulation (PWM) on the Pi Servo pHAT. The PCA9685 is designed primarily for LED control, but can be used for other PWM devices like servos.

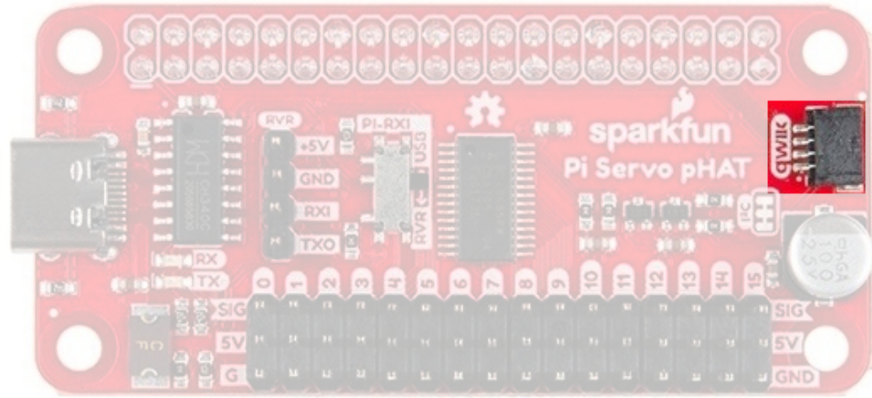


Although the PWM frequency of all the channels are shared, the duty cycle of the 16 channels of PWM output can be controlled individually. This allows the PCA9685 to control servos or LEDs on each of the outputs. For LEDs, this allows the channels to control or drive the brightness; or the position on servos.

Servo/PWM Headers:The Pi Sevro pHAT can be utilized with various PWM devices, the most typically application will be servos and LEDs. These headers are spaced out to make it easier to attach servo motors to them. Additionally, they are broken out in the standard 3-pin configuration for most hobby-type servo motor connectors.



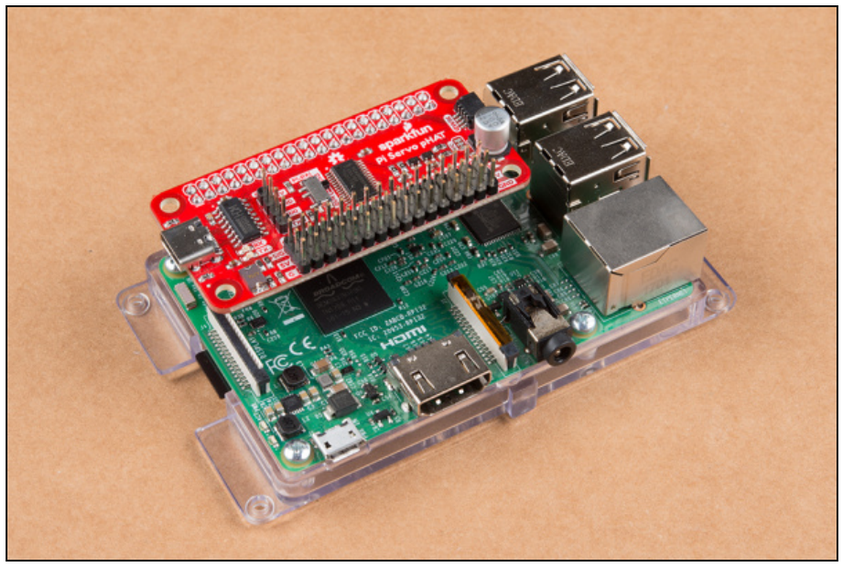
A Qwiic connector is provided for additional compatibility with SparkFun's Qwiic line of products. The Qwiic system is intended to be a quick, hassle-free cabling/connector system to conveniently add your favorite I2C devices. The Qwiic connector is tied to the I2C pins of the Raspberry Pi and draws power directly from the 3.3V pins.



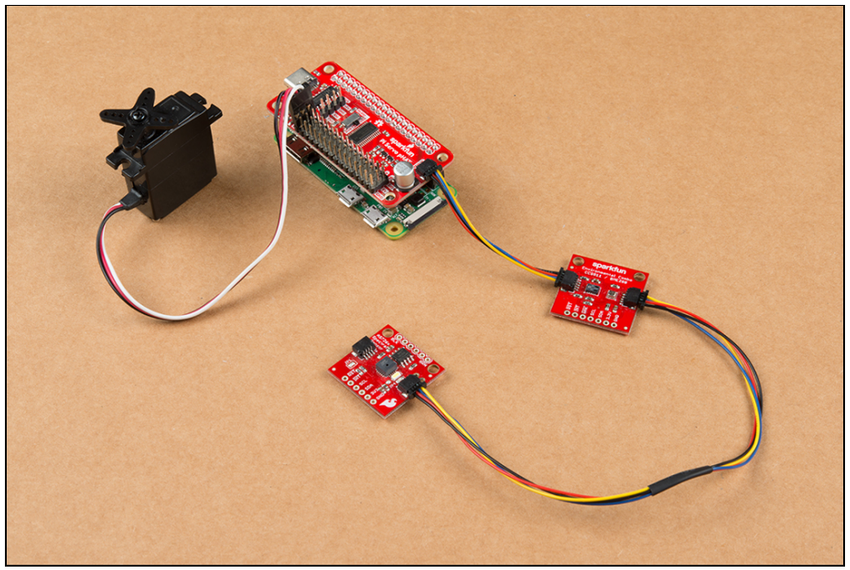
Assembling the Pi Servo pHAT (v2) with a Raspberry Pi is pretty straight forward. First make sure that nothing is powered. Then, just stack the board onto the Raspberry Pi so that the PCBs line up on top of each other; the Pi Servo pHAT is NOT meant to protrude out to the side.

- Stacking the Pi Servo pHAT with the PCB protruding to the side of the Raspberry Pi instead of on top (or above) will incorrectly connect pins and will most likely damage something.
- Although usually minimal, even when connecting a Raspberry hat there is a chance to short, brown-out, and/or damage something. As a best practice, you should never plug in a hat while anything is powered (including the hat, servos, LEDs, or additional qwiic boards).
- Make sure you aren't using a Raspberry Pi 4 or the Debian Buster image. Both have yet to be tested for use with this product.

Never plug in a servo while your Raspberry Pi is running. The sudden current spike needed to power your servo will reset your Raspberry Pi. Always plug in all of your servos first, and then boot up.

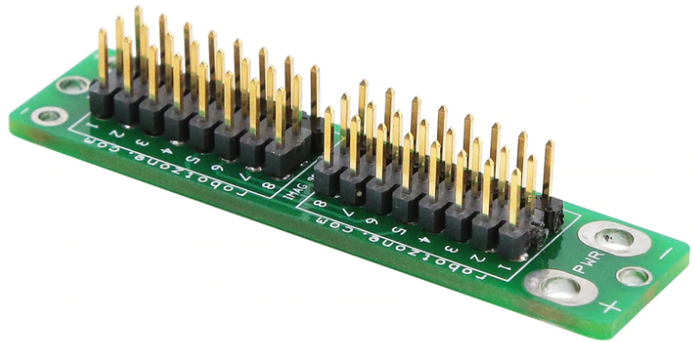


Hooking up Servos and Qwiic devices is then as simple as plugging them in:



Servo Power Distribution Board

In order to isolate the noisy high voltage power required by the servos and rover drive motor we will use a ServoCity **Servo Power Board**, details at : https://www.servocity.com/servo-power-board-assembled-with-full-pins/

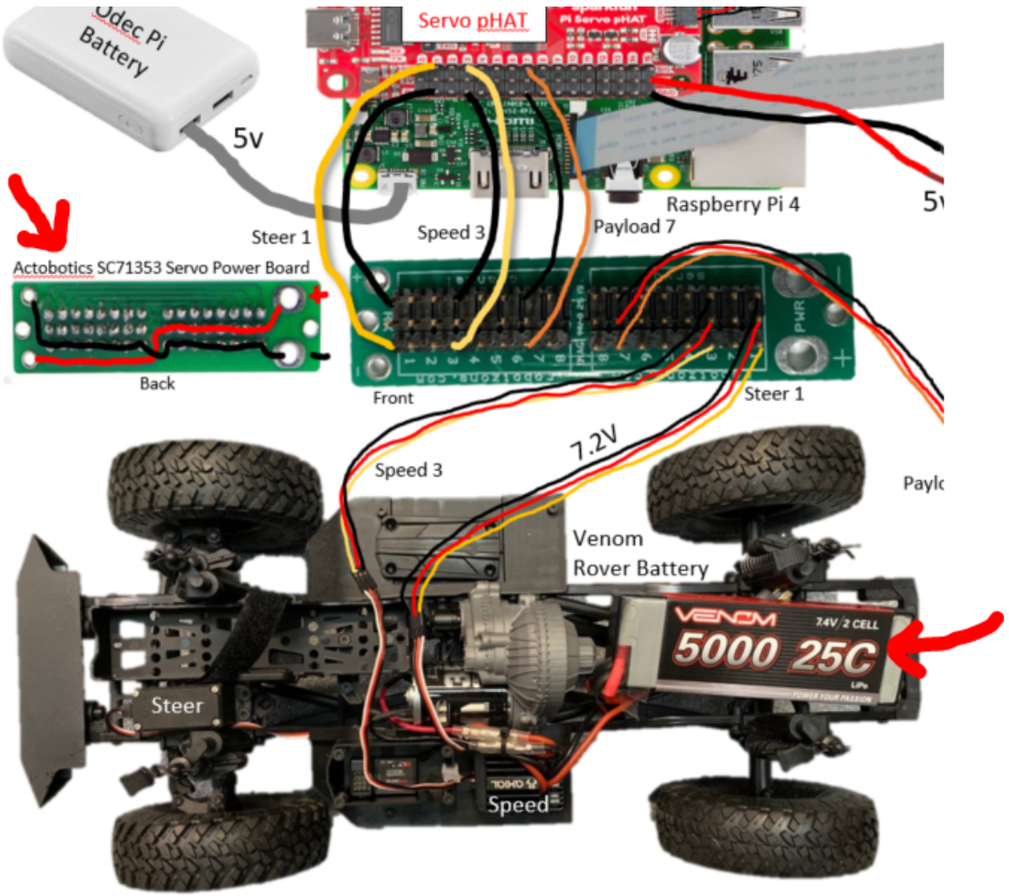


The Servo Power Board allows you to cleanly isolate high current and voltage from your controller to protect the tiny traces within from burning out. One side of the board shares a direct connection with your servo controller while the other allows servos (up to 8) to be plugged in. The positive trace is split between the servo controller side and the servo side so that you can supply power to the solder pads without sending that power on to the servo controller. This is advantageous in setups where the servos require a different voltage than the robot controller. This setup will protect the controller from the current drawn by the servos as the current will travel between the battery and the servos and not through the controller.

On your Rover the Raspberry Pi is running off the 5VDC Odec battery pack, you will want to run your Rover drive motor, steering servo and payload servo off of the more powerful 7VDC Venom power pack mounted on the rover. The **Servo Power Board** lets you safely connect the two grounds and signals from these separate power busses while keeping the power seperate.

ENGR3390: Fundamentals of Robotics Final Project Rev A

Please see diagram below:

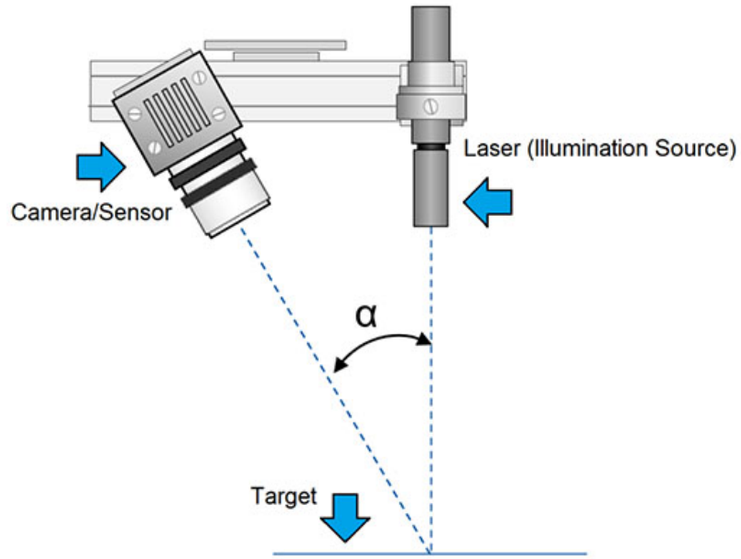


Using separate power busses keeps your Pi safe from over-voltages and power spikes and allows you to run larger motors at higher power levels. Note we will use port 1 for steering servo, port 3 for drive servo and port 7 for payload dump servo.

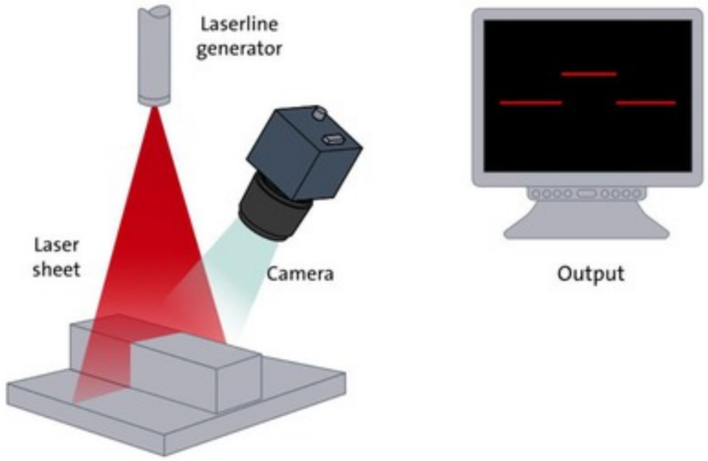
Pi Cam and Red Line Laser

To provide your Rover with machine vision, we will add a Raspberry Pi-Camera V2 to it. As this sensor was well covered in the foundation SENSE lab, no further technical data will be provided here. Please consult the Sense lab tutorial as needed. In addition to the Pi-Cam, your team will also be given a 2mW Quarton red line laser: https://www.amazon.com/Quarton-VLM-650-28-LPT-Generator-ECONOMICAL/dp/B00ARBPI5Y/ref=asc_df_B00ARBPI5Y/?tag=hyprod-20&linkCode=df0&hvadid=198105196047&hvpos=&hvnetw=g&hvrand=4848305228083500688&hvpone=&hvptwo=&hvqmt=&hvdev=c&hvdvcmdl=&hvlocint=&hvlocphy=9002080&hvtargid=pla-320331401471&psc=1

This laser can be used with the Pi-Cam to do very precise distance ranging across its full field of view using structured light. In a structured light setup, the laser is mounted at a slight angle to the camera:



The camera's center axis intersects the laser line's projection plane at a fixed standoff distance, say 1m, any object closer than 1m will shift the laser line down in the field of view, any further will shift it up. And the shift is linearly proportional to the distance the object is from the fixed standoff distance.

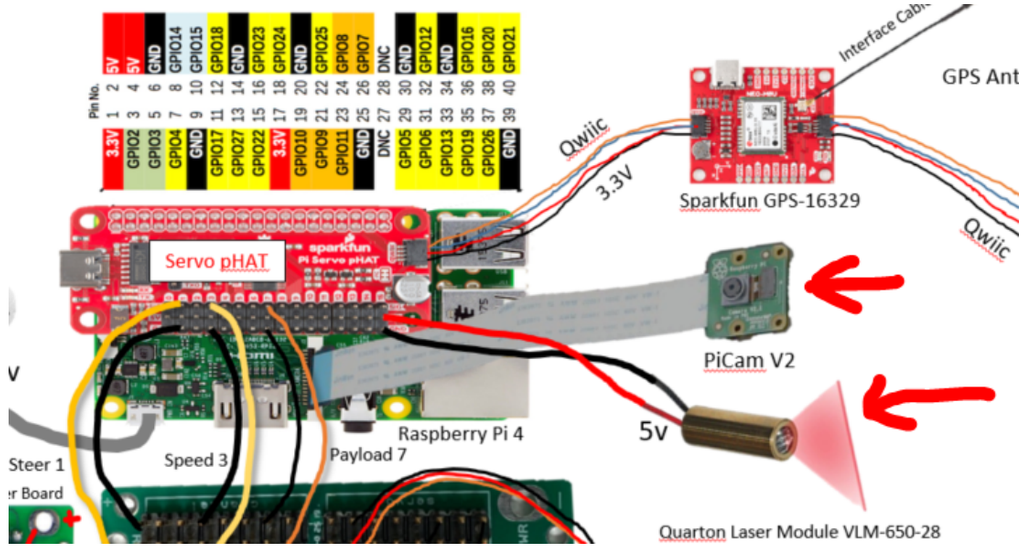


You can use this to get a precise 2d distance map of what is in front of your rover across the camera's full field of view. It will give you much more detailed and much more useful information than the array of fixed IRs can.

If you mount the laser in your pan-tilt, you can build up a full 3d range map of what is around your rover by scanning the laser light sheet over the terrain in front of your Rover. Either or both would give your Rover a big sensor advantage in the race. Your Raspberry Pi 4 is quite a powerful small computer and we are using it on the Rover mainly to give you the possibility of exploring more advanced vision based ranging systems like this one.

Please see the provided structured light papers in the Canvas folder for more details.

The Pi-Camera and Quarton laser are connected to the Pi and pHAT as shown below:



GPS Qwiic Board

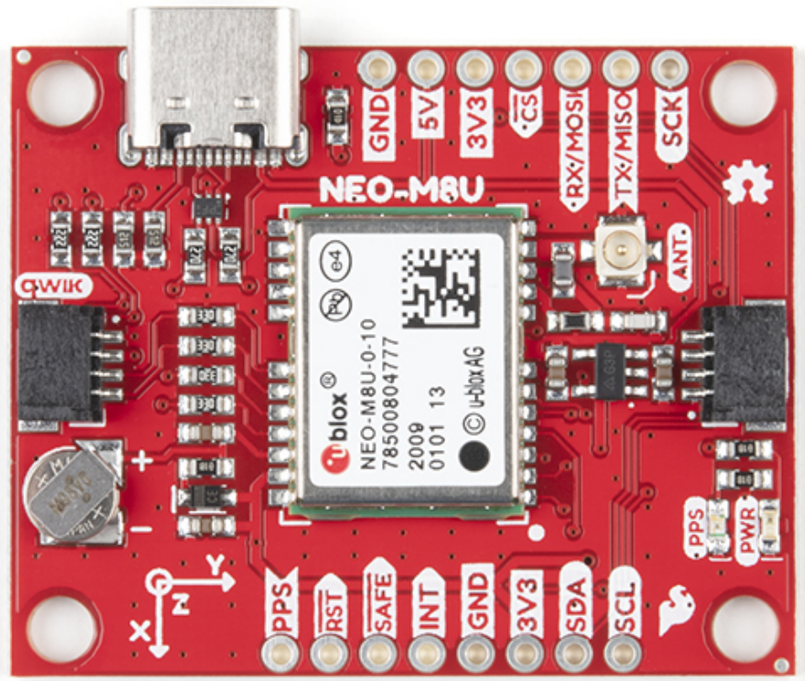
While the Pi-Cam, Sharp RangeFinders and Sonar can help your robot determine what is around it, it will also need to be able to determine where it is on the outdoor test track. This is traditionally done via GPS, with the potential for augmentation with an Inertial Measurement Unit.

For your Rover, your team will be supplied with a quite capable SparkFun **GPS Dead Reckoning Breakout - NEO-M8U (Qwiic).**

See for details: **https://www.sparkfun.com/products/16329**

The SparkFun NEO-M8U GPS Breakout is a high quality, GPS board with equally impressive configuration options. The NEO-M8U takes advantage of u-blox's

Untethered Dead Reckoning (UDR) technology. The module provides continuous navigation without needing to make any electrical connection to a vehicle, thus reducing cost of installation for after-market dead reckoning applications.



The NEO-M8U module is a 72-channel u-blox M8 engine GNSS receiver, meaning it can receive signals from the GPS, GLONASS, Galileo, and BeiDou constellations with ~2.5 meter accuracy. The module supports concurrent reception of three GNSS systems. The combination of GNSS and integrated 3D sensor measurements on the NEO-M8U provide accurate, real-time positioning rates of up to 30Hz.

Compared to other GPS modules, this breakout maximizes position accuracy in dense cities or covered areas. Even under poor signal conditions, continuous positioning is provided in urban environments and is also available during complete signal loss (e.g. short tunnels and parking garages). With UDR, position begins as soon as power is applied to the board even before the first GNSS fix is available! Lock time is further reduced with on-board rechargeable battery; you'll have backup power enabling the GPS to get a hot lock within seconds!

Additionally, this u-blox receiver supports I2C (u-blox calls this Display Data Channel) which made it perfect for the Qwiic compatibility so we don't have to use up our precious UART ports. Utilizing our handy Qwiic system, no soldering is required to connect it to the rest of your system. However, we still have broken out 0.1"-spaced pins in case you prefer to use a breadboard.

U-blox based GPS products are configurable using the popular, but dense, windows program called u-center. Plenty of different functions can be configured on the NEO-M8U: baud rates, update rates, geofencing, spoofing detection, external interrupts, SBAS/D-GPS, etc. All of this can be done within the SparkFun Arduino Library!

The SparkFun NEO-M8U GPS Breakout is also equipped with an on-board rechargeable battery that provides power to the RTC on the NEO-M8U. This reduces the time-to-first fix from a cold start (~26s) to a hot start (~1.5s). The battery will maintain RTC and GNSS orbit data without being connected to power for plenty of time.

The u-blox NEO-M8U is a powerful GPS units that takes advantage of untethered dead reckoning (UDR) technology for navigation. The module provides continuous positioning for vehicles in urban environments and during complete signal loss (e.g. short tunnels and parking garages). We will quickly get you set up using the Qwiic ecosystem and Arduino so that you can start reading the output!

Dead Reckoning is the process of determining current position by combining previously determined positional data with speed and heading. This process can also be applied to determine future positions as well! The NEO-M8U uses what is called Untethered Dead Reckoning (UDR) which calculates speed and heading (amongst many other points of data) through the use of an internal inertial measurement unit

(IMU). The addition of an IMU allows the M8U to produce more accurate readings in between GNSS data refreshes!

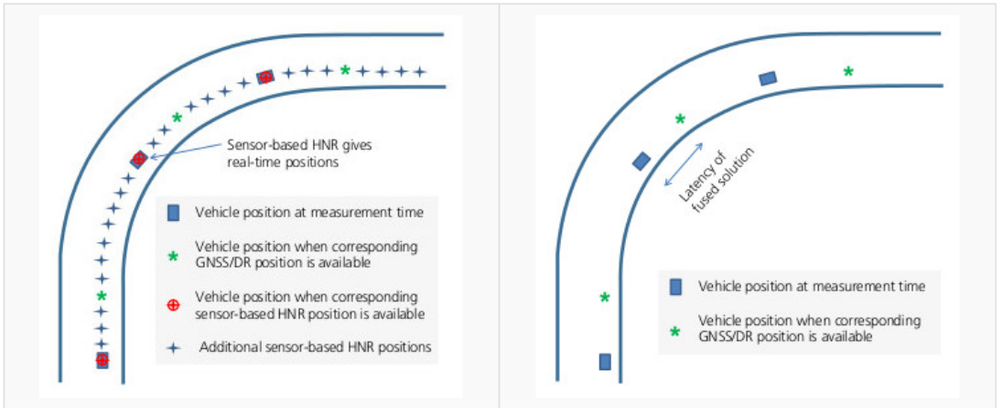
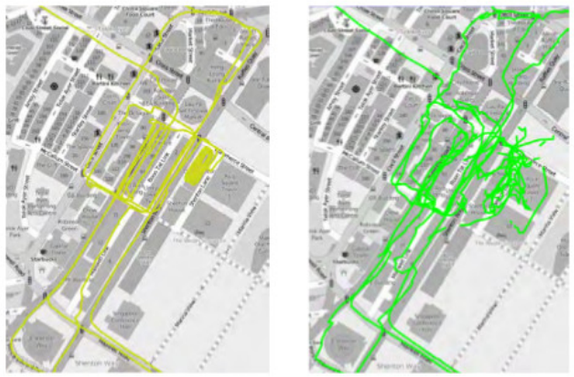


Illustration of Using an IMU with GNSS Data | *Illustration of Using only GNSS Data Only*

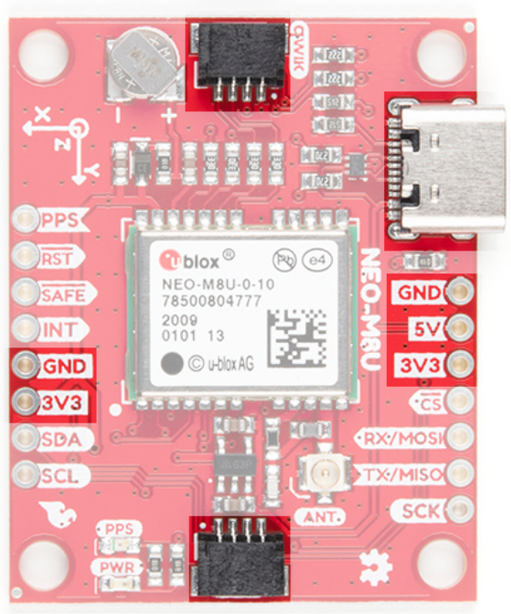
In addition, the module can also give accurate and useful GNSS data in areas where satellite connections are difficult to maintain: areas like the dense urban environments of major cities, long tunnels, parking garages, any large UFO's that may descend from the sky, etc.



UDR | GNSS only

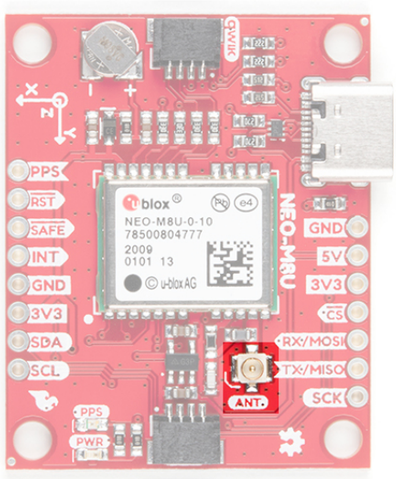
Untethered Dead Reckoning vs GNSS Only Comparison in an Urban Canyon. Image Courtesy of u-blox from the UDR Whitepaper.

Power for this board is 3.3V and we have provided multiple power options. This first and most obvious is the USB-C connector. Secondly, are the Qwiic Connectors on the top and bottom of the board. Thirdly, there is a 5V pin on the PTH header along the side of the board that is regulated down to 3.3V. Make sure that power your provide to this pin does not exceed 6 volts. Finally, just below the 5V pin is a 3.3V pin that should only be provided a clean 3.3V power signal.

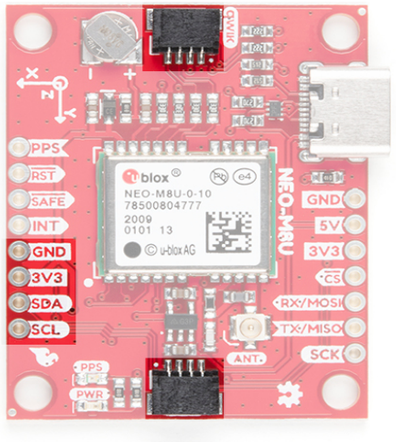


There's is a red power LED just to the left of the bottom Qwiic connector and near the board's edge to indicate that the board is powered. There is another LED just above the power LED labeled PPS that is connected to the Pulse Per Second line. When connected to a satellite, this line generates a pulse that is synchronized with a GPS or UTC time grid. By default, you'll see one pulse a second.

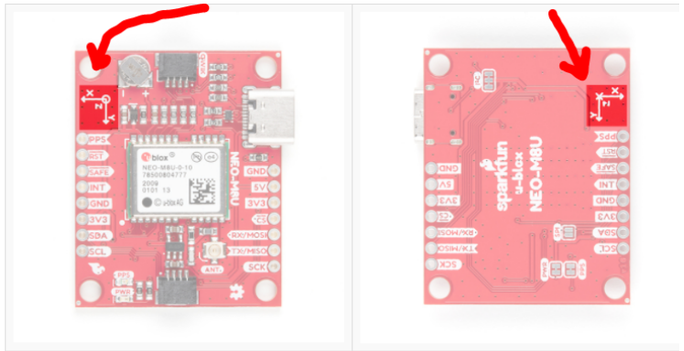
The SparkFun GPS NEO-M8U has a u.FL connector in which you can connect a patch antenna. Be really really careful plugging this in. Can destroy a board if you do it wrong.



There are two pins labeled SDA and SCL which indicates the I2C data lines. Similarly, you can use either of the Qwiic connectors to provide power and utilize I2C. The Qwiic ecosystem is made for fast prototyping by removing the need for soldering. All you need to do is plug a Qwiic cable into the Qwiic connector and voila!



For easy reference, we've documented the IMU's vectors with 3D Cartesian coordinate axes on the top and bottom side of the board. Make sure to orient and mount the board correctly so that the NEO-M8U can accurately calculate navigation information. This is explained in detail in the Dead Reckoning Overview. Remember, it's all relative.



Top View with the the Axis for Reference | *Bottom View with the the Axis for Reference*

GPS Capabilities

The SparkFun NEO-M8U is able to connect to up to three different GNSS constellations at a time. Below are the listed capabilities of the GPS unit taken from the datasheet when connecting to different GNSS constellations.

Constellations		GPS+GLO	GPSL	GLO	BDS	GAL
Horizontal Position Accuracy	Autonomus	2.5m	2.5m	4.0m	3.0m	To Be Confirmed
	with SBAS	1.5m	1.5m			
Max Navigation Update Rate	PVT	25Hz	25Hz	25Hz	25Hz	25Hz
Time-To-First-Fix	Cold Start	24s	25s	26s	28s	29s
	Hot Start	2s	2s	2s	2s	2s
Sensitivity	Tracking and Navigation	-160dBm	-160dBm	-160dBm	-160dBm	-154dBm
	Reacquisition	-160dBm	-159dBm	-156dBm	-155dBm	-152dBm
	Cold Start	-148dBm	-147dBm	-145dBm	-143dBm	-133dBm
	Hot Start	-157dBm	-156dBm	-155dBm	-155dBm	-151dBm
Velocity Accuracy		0.05m/s	0.05m/s	0.05m/s	0.05m/s	0.05m/s
Heading Accuracy		1deg	1deg	1deg	1deg	1deg

Dead Reckoning Overview

As mentioned in the "What is Dead Reckoning?" section, the u-blox M8U module has an internal inertial measurement unit or IMU for short. The IMU calculates position based on the last GNSS refresh and its own movement data points. To use the SparkFun GPS Dead Reckoning Board, there are a few guidelines to orienting and mounting the module to a vehicle outlined in the u-blox ReceiverDescrProtSpec Datasheet.

Orientation for the SparkFun Dead Reckoning

The SparkFun Dead Reckoning adheres to two particular frames of reference: one frame of reference for the car and the second a geodetic frame of reference anchoring it to the globe. The latter, known as the local level frame uses the following as its' axes:

- X-axis points to the North
- Y-axis points to the East
- Z-axis uses the right hand system by pointing down.

This frame will be referred to by its acronym NED (North-East-Down) in the image below.

The second frame of references is the Body-Frame reference and uses the following as its' axes.

- X-axis points to the front of the vehicle
- Y-axis points to the right of the vehicle
- Z-axis uses the right hand system by pointing down.

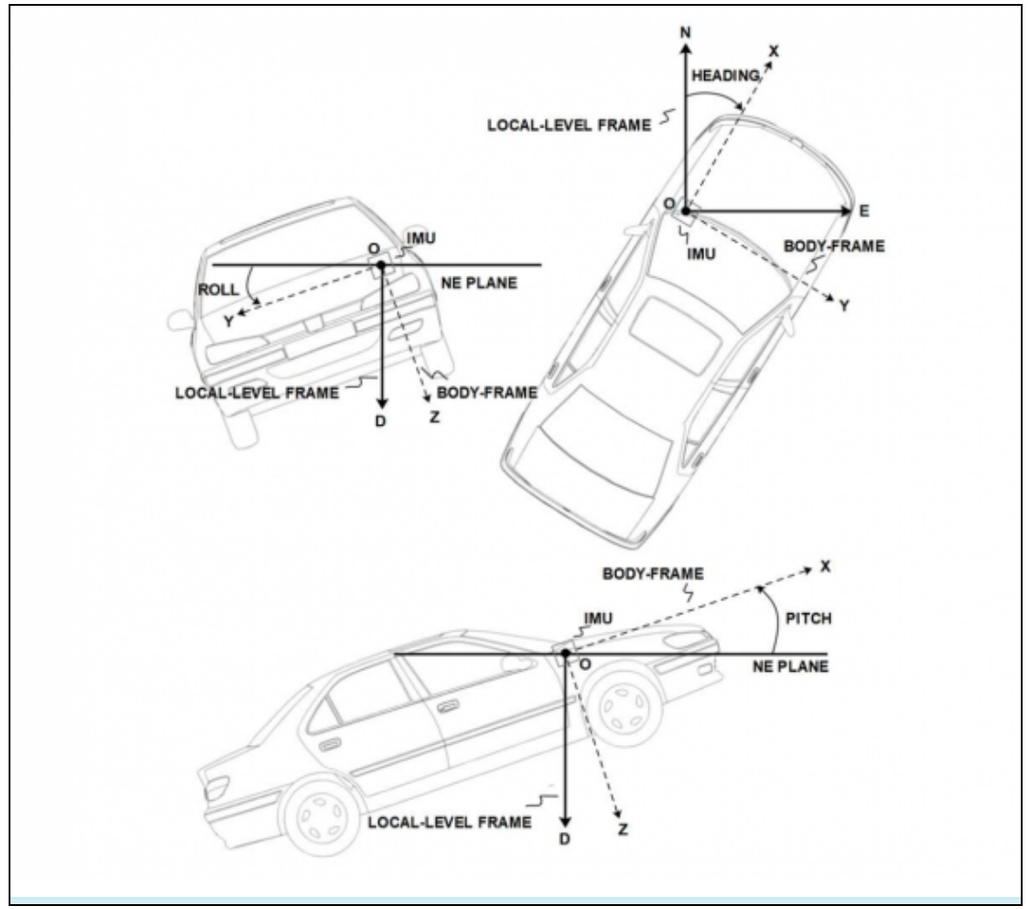
The transformation of the vehicle within these two frames are given as heading, pitch, and roll. In the datasheet these three angles are called the vehicle's attitude. Below is an image that illustrates how all of these elements fit together.

Calibration

After you've mounted the SparkFun Dead Reckoning M8U, there is still a calibration phase to complete that must satisfy the following movements:

- First, the car needs to be stopped with the engine turned on.
- Secondly, the car must do left and right hand turns.
- Lastly, the car must reach a speed over 30 km/h.

In SparkFun's u-blox Arduino library, SparkFun has included an Example (shown below), that prints out the module's calibration status.

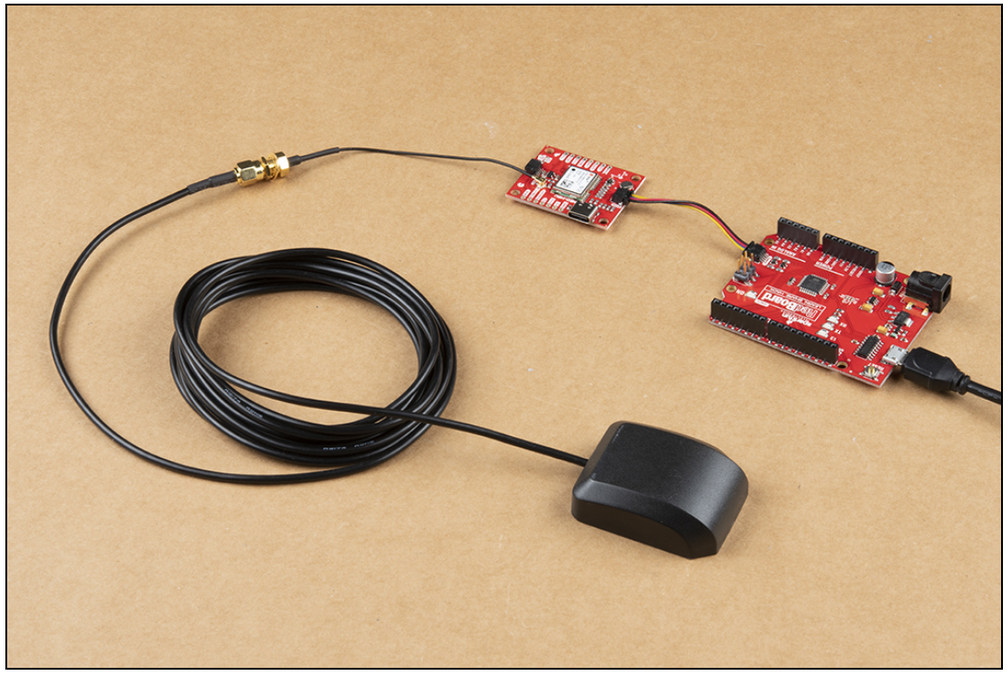


The only guideline here is that the SparkFun Dead Reckoning is stable within 2 degrees, and of course that the X-axis points towards the front of the car as mentioned above.

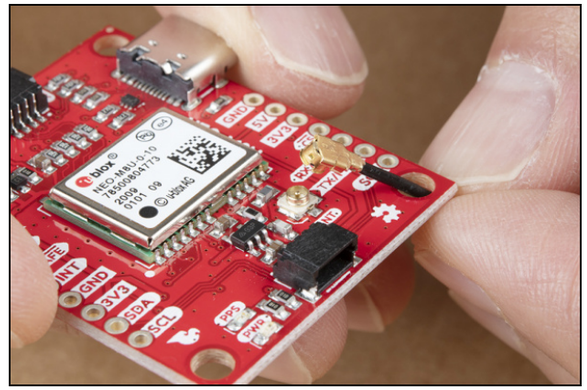
ENGR3390: Fundamentals of Robotics Final Project Rev A

Hardware Assembly

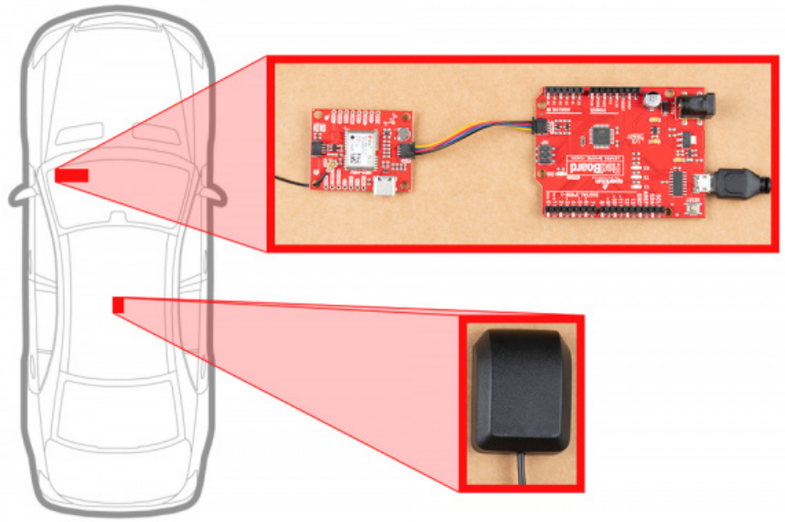
For this example, I used a RedBoard Qwiic and associated USB cable. Connecting the boards with Qwiic cable, the assembly is very simple. Plug a Qwiic cable between the RedBoard and SparkFun NEO-M9U. Then plugged in one of our patch antennas to the u.FL connector. If you need tips on plugging in the U.FL connector, then check out our U.FL tutorial. If you're going to be soldering to the through hole pins for I2C functionality, then just attach lines to power, ground, and the I2C data lines to a microcontroller of your choice. Your setup should look similar to the image below.



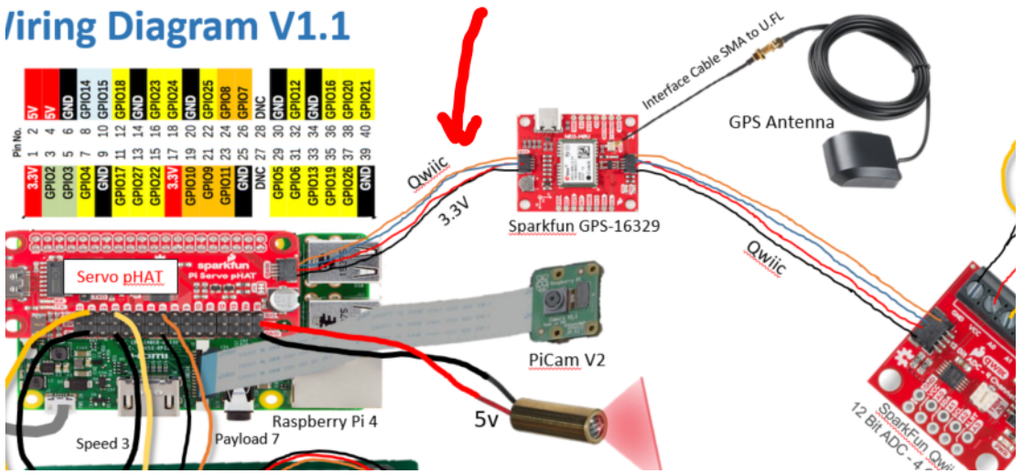
For secure connections, you may want to thread the U.FL cable through a mounting hole before connecting. Adding tape or some hot glue will provide some strain relief to prevent the cable from disconnecting.



Make sure to secure the board above a vehicle's dashboard using some tape or sticky tack when prototyping and testing. For best signal reception, it is suggested to guide the antenna from the inside of the car and through a window before attaching the GPS on top of a car.

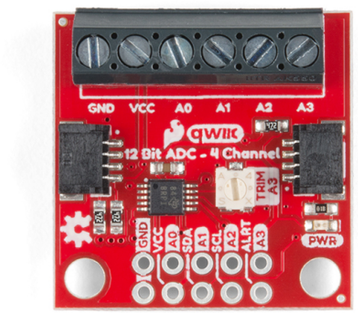


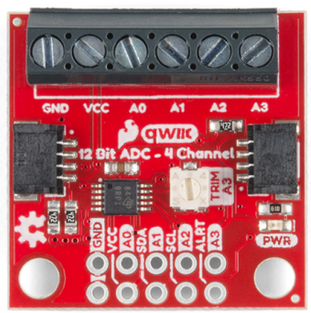
Finally, to add this GPS to yoru rover, plug it into the Qwiic bus connector on the pHAT as shown below:



Qwiic Analog Input Board

In order to connect to the analog outputs of the Sharp IR Range Sensors and Maxbotix Sonars your team will be given, we need to add an analog input capability to the Raspberry Pi. We will do this by using two **SparkFun Qwiic 12 Bit ADC - 4 Channel (ADS1015)** breakout boards.





https://www.sparkfun.com/products/15334

A lot of the time you just need to add more analog inputs to you Pi. The SparkFun Qwiic 12 Bit ADC can provide four channels of I2C controlled ADC input to your Qwiic enabled project. These channels can be used as single-ended inputs, or in pairs for differential inputs. What makes this even more powerful is that it has a programmable gain amplifier that lets you "zoom in" on a very small change in analog voltage (but will still affect your input range and resolution). Utilizing the handy Qwiic system, no soldering is required to connect it to the rest of your system. However, we still have broken out 0.1"-spaced pins in case you prefer to use a breadboard.

The ADS1015 uses its own internal voltage reference for measurements, but a ground and 3.3V reference are also available on the pinouts for users. This ADC board includes screw pin terminals on the four channels of input, allowing for solderless connection to voltage sources in your setup. It also has an address jumper that lets you choose one of four unique addresses (0x48, 0x49, 0x4A, 0x4B). **You can use one stock address but need to set the second board to the next sequential address to avoid bus conflicts.** With this, you can connect up to four of these on the same I2C bus and have sixteen channels of ADC. The maximum resolution of the converter is 12-bits in differential mode and 11-bits for single-ended inputs. Step sizes range from 125µV per count to 3mV per count depending on the full-scale range (FSR) setting.

SparkFun has included an onboard 10K trimpot connected to channel A3. This is handy for initial setup testing and can be used as a simple variable input to your project. But don't worry, they added an isolation jumper so you can use channel A3 however you'd like.

NOTE: The I2C address of the ADS1015 is 0x48 and is jumper selectable to 0x49, 0x4A, 0x4B. A multiplexer/Mux is required to communicate to multiple ADS1015 sensors on a single bus. If you need to use more than one ADS1015 sensor consider using the Qwiic Mux Breakout.

ADS1015

- Operating Voltage (VDD): 2.0V-5.5V
- (Note: When powering with a Qwiic cable, the input range is only 3.3v)
- Operating Temperature: -40°C to 125°C
- Operation Modes: Single-Shot, Continuous-Conversion (Default), and Duty Cycling
- Analog Inputs:
- Measurement Type: Single-Ended (Default)
- Input Voltage Range: GND to VDD
- Full Scale Range (FSR): ±.256V to ±6.114V (Default: 2.048V)
- Resolution:
- 12-bit (Differential) or 11-bit (Single-Ended)
- LSB size: 0.125mV - 3mV (Default: 1 mV)
- Sample Rate: 128 Hz to 3.3 kHz (Default: 1600SPS)
- Current Consumption (Typical): 150µA-200µA
- I2C Address: 0x48 (Default), 0x49, 0x4A, or 0x4B
- Screw pin terminals for solderless connection to voltage sources
- Four unique I2C addresses:
- 0x48
- 0x49
- 0x4A
- 0x4B
- 2x Qwiic connection ports
- Onboard 10K trimpot

An analog to digital converter (ADC) is a very useful tool for converting an analog voltage to a digital signal that can be read by a microcontroller. The ability to convert from analog to digital interfaces allows users to use electronics to interface to interact with the physical world.

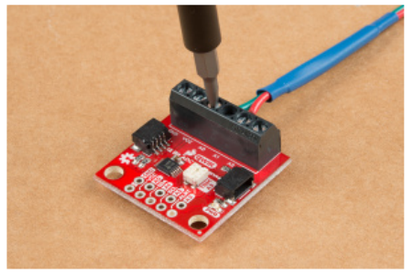
Power: There is a power status LED to help make sure that your Qwiic ADC is getting power. You can power the board either through the polarized Qwiic connector system or the breakout pins (3.3V and GND) provided. This Qwiic system is meant to use 3.3V, be sure that you are NOT using another voltage when using the Qwiic system.

Annotated image of power LED along with VCC and GND connections.

Caution: The analog input voltage range is **(GND - .3V)** to **(V_{DD} + .3V)**; anything slightly higher or lower and you will damage the ADC chip. If you are using the Qwiic system, this is approximately **-.3V to 3.6V** in reference to the GND pin. If the voltages on the input pins can potentially violate these conditions, use external Schottky diodes and series resistors to limit the input current to safe values (as mentioned in the datasheet).

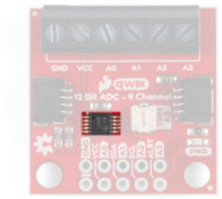
There are four input measurement channels for the ADS1015, labeled AIN0-3, accessible through the screw pin terminals shown below. With the Qwiic system, the absolute minimum and maximum voltage for these inputs is -.3V and 3.6V, respectively.

Annotated image of input connections on board. Click to enlarge.

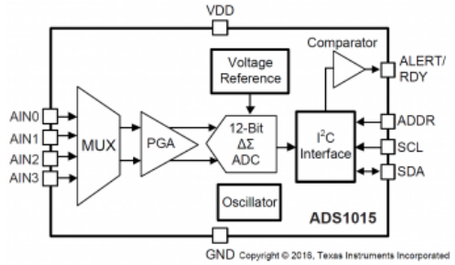


Example of attaching inputs through screw terminals. Click to enlarge.

ADS1015 ADC: The ADS1015 ADC is a low-power 12-bit analog-to-digital converter (ADC), which includes a built-in integrated voltage reference and oscillator. At the core of its operation, the ADC uses a switched-capactior input stage and a delta-sigma (ΔΣ) modulator to determine the differential between AINP (positive analog input) and AINN (negative analog input). Once the conversion is completed, the digital output is accessible over the I2C bus from the internal conversion register.



Annotated image of ADS1015 on board. Click to enlarge.



Functional block diagram from datasheet. Click for more details.

Characteristic	Description
Operating Voltage (V_{DD})	2.0V to 5.5V (**Default on Qwiic System: 3.3V**)
Operating Temperature	-40°C to 125°C
Operation Modes	Single-Shot (**Default**), Continuous-Conversion, and Duty Cycling
Analog Inputs	Measurement Type: Single-Ended or Differential Input Voltage Range: **GND** to **V_{DD}** (*see **Caution** note, below*) Maximum Voltage Measurement: *Smallest of* **V_{DD}** or **FSR** Full Scale Range (FSR): ±.256V to ±6.114V (**Default: 2.048V**)
Resolution	12-bit (Differential) or 11-bit (Single-Ended) LSB size: 0.125mV - 3mV (**Default: 1 mV** *based on FSR*)
Sample Rate	128 Hz to 3.3 kHz (**Default: 1600SPS**)
Current Consumption (*Typical*)	Operating: 150µA to 200µA Power-Down: 0.5µA to 2µA
I^2C Address	**0x48** (**Default**), 0x49, 0x4A, or 0x4B

Caution: The absolute, analog input voltage range is **(GND - .3V) to (V_{DD} + .3V)**; anything slightly higher/lower may damage the ADC chip. If the voltages on the input pins can potentially violate these conditions, as specified by the datasheet, use external Schottky diodes and series resistors to limit the input current to safe values.

The ADS1015 has 2 different conversion modes: single-shot and continuous-conversion with the ability to support duty cycling. Through these modes, the ADS1015 is able to optimize its performance between low power consumption and high data rates.

Single-Shot: By default, the ADS1015 operates in single-shot mode. In single-shot mode, the ADC only powers up for ~25µs to convert and store the analog voltage measurement in the conversion register before powering down. The ADS1015 only powers up again for data retrieval. The power consumption in this configuration is the lowest, but it is dependent on the frequency at which data is converted and read.

Duty Cycling: In single-shot mode, the ADS1015 can be duty cycled to periodically request high data rate readings. This emulates an intermediary configuration between the low power consumption of the single-shot mode and the high data rates of the continuous-conversion mode.

Continuous-Conversion: In this mode, the ADS1015 continuously performs conversions on analog voltage measurements and places the data in the conversion register. If the configuration settings are changed in the middle of the conversion process, the settings take effect once the current process is completed.

Analog-to-Digital Conversion: Although, it is listed as a 12-bit ADC, the ADS1015 operates as an 11-bit ADC when used with single-ended (individual) inputs. The 12th bit only comes into play in differential mode, as a sign (+ or -) indicator for the digital output. This allows the digital output to represent the full positive and negative range of the FSR (see table and figure below).

Table 3. Input Signal Versus Ideal Output Code

INPUT SIGNAL $V_{IN} = (V_{AINP} - V_{AINN})$	IDEAL OUTPUT CODE(1)(1)
≥ +FS $(2^{11} - 1)/2^{11}$	7FF0h
$+FS/2^{11}$	0010h
0	0000h
$-FS/2^{11}$	FFF0h
≤ –FS	8000h

(1) Excludes the effects of noise, INL, offset, and gain errors.

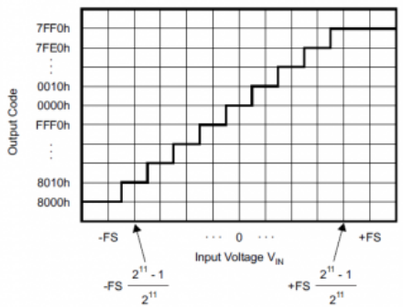


Figure 18. Code Transition Diagram

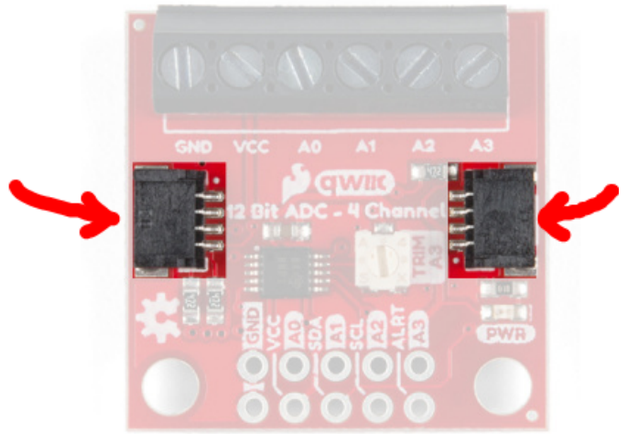
NOTE

Single-ended signal measurements, where V_{AINN} = 0 V and V_{AINP} = 0 V to +FS, only use the positive code range from 0000h to 7FF0h. However, because of device offset, the ADS101x can still output negative codes in case V_{AINP} is close to 0 V.

I2C Address: The ADS1015 has 4 available I2C addresses, which are set by the address pin, ADDR. On the Qwiic ADC, the default slave address of the ADS1015 is 0x48 (HEX) of 7-bit addressing, following I2C protocol. The ADS1015 does have an additional general call address that can be used to reset all internal registers and power down the ADS1015 (see datasheet).

Default I2C Slave Address: 0x48

Connections: The simplest way to use the Qwiic ADC is through the Qwiic connect system. The connectors are polarized for the I2C connection and power. (*They are tied to their corresponding breakout pins.)



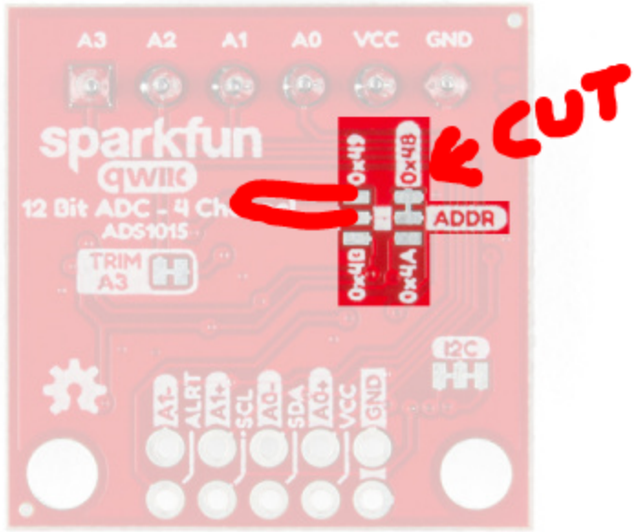
Annotated image of the Qwiic connectors.

The ADS1015 has four available I2C addresses, which can be configured by the jumpers on the back of the board. The address selection pin is connected to the center pad of the jumpers, the below table shows the addresses available when the address selection pin is tied to each of the 4 available pads.

Pin	GND	VCC	SDA	SCL
Address	**0x48 (Default)**	0x49	0x4A	0x4B

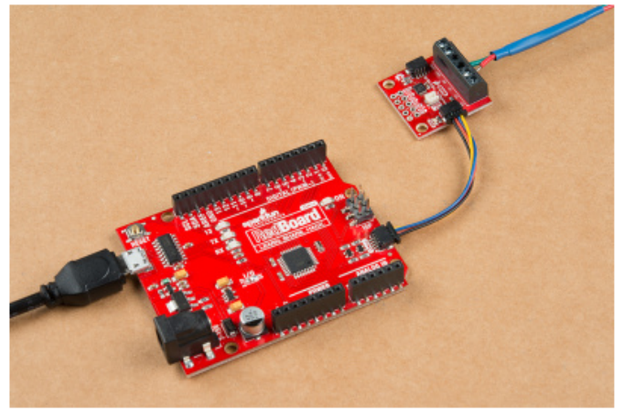
Note: Make sure this jumper is only shorted on one of the four available pads. There is a pullup jumper on the 3.3V pad, but to help prevent a shorts if multiple pads are accidentally bridged.

The address selection pin, by default, is tied to GND on the PCB. Cutting the trace and bridging the I2C address jumper to another pad changes the slave address from I2C Jumper Default: 0x48. The location of the jumpers is shown in the image below.
Note: You will need to do this to one of your boards (carefully)



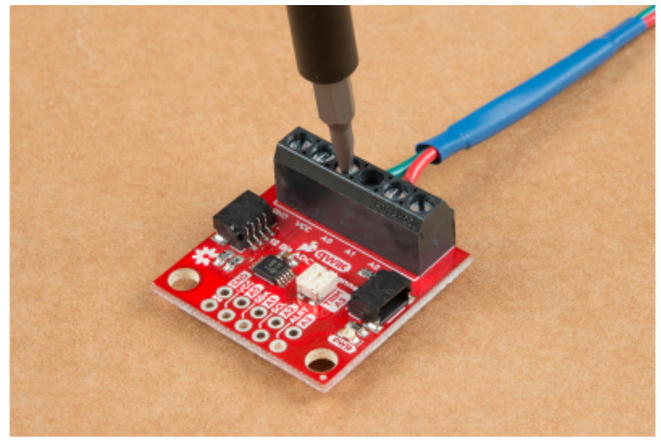
Annotated image of the I²C address jumper.

With the Qwiic connector system, assembling the hardware is fairly simple. For the examples below, all you need to do is connect your Qwiic ADC to Qwiic enabled microcontroller with a Qwiic cable. Otherwise, you can use the I2C pins, if you don't have a Qwiic connector on your microcontroller board. Just be aware of your input voltage and any logic level.



Example setup with RedBoard Qwiic.

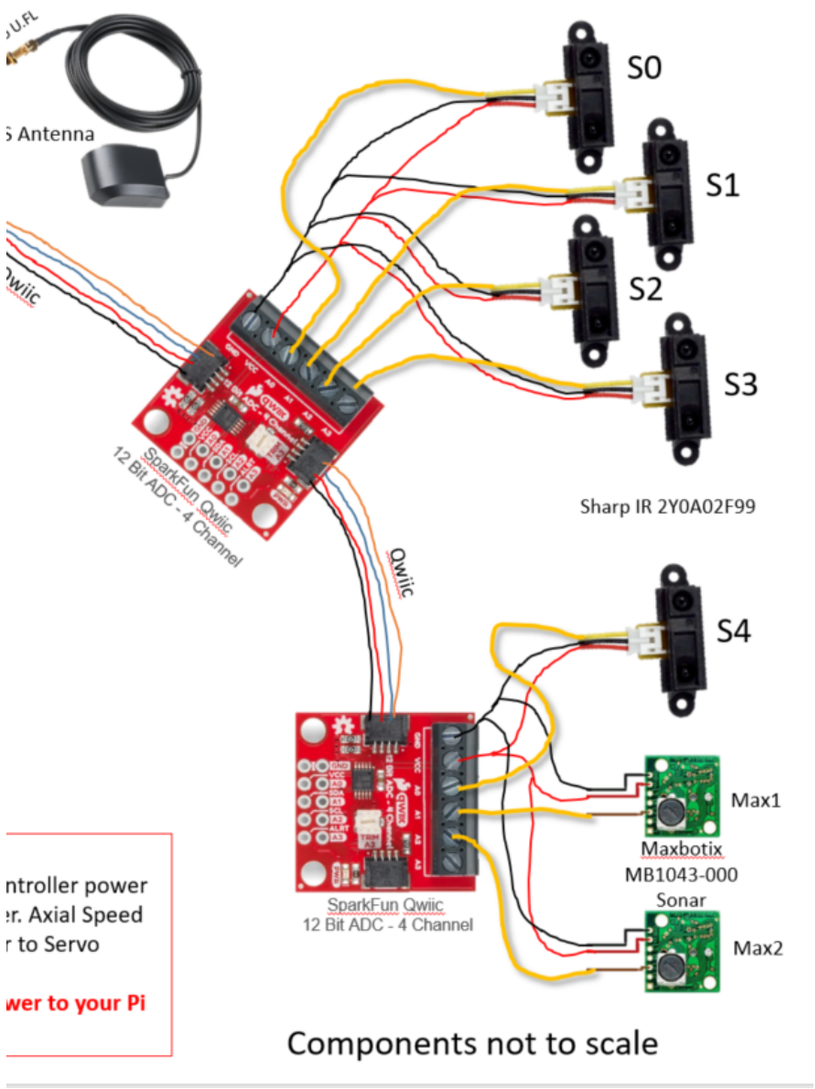
Additionally, you can connect your input voltages to the available inputs on the screw terminals and or use the breakout pins on the board. Make sure the connections are fully inserted and that you are ground looping your inputs. Ground looping can be done by connecting the ground of your input to the ground of the ADC.



Example of attaching inputs through screw terminals.

ENGR3390: Fundamentals of Robotics Final Project Rev A

Finally, connect your Qwiic boards with provided cables and then wire in Sharp IR Range Finders and Maxbotic Sonars as shown below:



Software Design

On first pass, it might look like there is an overwhelming amount of code to be written for this final project, but take heart, you have already written most of what you will need in the previous tutorials and labs and can just refactor your code and reuse it here in your actual rover. In your very first Matlab for Robots tutorial, you simulated a rover driving between waypoints in the oval, you have already used the Pi-Cam, Sharps and Sonars to find targets or holes, written whale tracking behaviors and you have already mastered Pan-Tilt servos and driven drive wheels. This final project code will have your team integrate all of your existing work together into one robot controller for a smoothly functioning robot planetary rover.

Overall Code Architecture

To maximize your team's code development efficiency, you will want to think seriously about reusing the first tutorial's **Oval-Rover code** as your team's main architectural structure, add the **Think lab's** behavior engine to it and then break down all of the remaining code into well defined functions that individual students can write. Good team code is modular. Good team code uses one small main program and lots of modular individual functions that can be developed independently.

If you put a simple main robot control structure in place first, then break up the remaining function writing evenly among your team, you will succeed. If you do almost anything else, your team will fail. *Specifically, if you listen to, blindly follow the loudest E:C super-programming wizard on your team, your project will almost surely go down in flames.* Professional code creation is team creation; simple, clean, written by all and understandable by all.

Let's talk a little about the robot control structure your team should consider using. You have in fact been using it for the whole first part of this course. **The first part of your structure is a documentation paragraph** that describes what your main code does, in enough detail that a cogent fellow programmer can follow it. To make field editing code easy, it should also include a wiring diagram of your robot and a clear indication of what external Matlab code parts (functions, files, objects, etc.) need to be in the same directory folder and on the same path as your robot's control code.

Next should be a clearly delineated section that contains all of **the code that is run, just once**, to set up your Rover's Raspberry PI, data-structures, variables, etc. This would also be a good place to download the current mission parameters and set up the main mission data (like waypoints, active behaviors, etc).

Following the code that runs once, comes **the main Sense-Think-Act control loop that runs over and over.** This is where the behavior engine consisting of a set of mission specific behaviors and an arbiter for the vehicle and perhaps a separate one for the pan-tilt head are situated.

After the body of text that holds themain code in the loop that runs over and over, your team should have at least **three function repositories**, one for Sense functions (i.e. ReadSharpIR, etc.), one for Think functions (GoToWaypoint etc.) and one for Act functions (PanTilt(angle, angle).

Please note, your team can lay out the full structure of your robot rovers control code in a single meeting and then put in place a set of empty Sense, Think and Act functions with jointly agreed on input and output arguments. Having done this you can break into subteams and asynchronously write, debug and test those functions using a common team Matlab-Drive code repository.

By design, you and your programming partners have been doing this process all semester. There is a common robot control template beneath all of the labs and you can now go back and harvest the Sense, Think, and Act functions you created during those labs for reuse here in your final project rover.

Let's take a brief look at the more familiar functions you should think about reusing and a more indepth look at the newer ones we haven't touched much yet.

Sense: Sharp IR Range Sensor Array

Your Rover can have up to 5 long range Sharp IR range sensors. The ones we are using are similar to the ones on the Sense test stand, but have a much longer range. You will want to independently calibrate one of them and then either linearize its output or fit a polynomial to its voltage to distance curve. Once you have a solid calibration curve you can write a single function that the main control program can call, once each loop cycle, to see what is around your Rover. Your team can design this function any way you'd like, but a classic way to do this is to pass a vehicle centric simple occupancy grid (matrix) to the function as an input parameter, use the body of the function to add the Sharp range data to that grid, then have function return the updated occupancy grid to the main program control loop. The more computation you can do locally in a function, the more reliable your main code will be. Your Sharp IR function could look something like this:

```
function  [roverOcupancyGrid] = SharpIRRange (roverOcupancyGrid, other, other)
% SharpIRRange adds the current set of Sharp Range measurements to the control
% loop cycle's Rover centric occupancy grid. 0=open space 1-10= obstacles
% written by Alexi and Bumi 3-30-21 Rev A
    Body of code
end
```

If you need a detailed refresher on how to code for the Qwiic 12bit ADC board, please go back and read through the Raspberry-Pi-Sharp IR Range sensor part of the Sense lab. In brief summary of that write up, please download a copy of the **ads1015.m** function from Canvas. It is needed for the Pi to talk with the 12bit ADC over the i2c bus from canvas. Typing **help ads1015** at the command line will bring up a detailed description of the function:

```
>> help ads1015
 ads1015 Analog-to-Digital converter.

  adc = ads1015(rpi, bus) creates a ads1015 ADC object attached to the
  specified I2C bus. The first parameter, rpi, is a raspi object. The
  I2C address of the ads1015 ADC defaults to '0x48'.

  adc = ads1015(rpi, bus, address) creates a ads1015 ADC object
  attached to the specified I2C bus and I2C address. Use this form if
  the ADDR pin is used to change the I2C address of the ads1015 from
  the default '0x48' to something else.

  [voltage, rawData] = readVoltage(adc, AINp) reads the single-ended voltage
measurement from AINp input port. Also returns the contents of conversion
  register, optionally in rawData
  Accepted values for AINp are 0,1,2,3.

  [voltage, rawData] = readVoltage(adc, AINp, AINn) reads the differential
voltage measurement between AINp and AINn input ports. Also returns the
  contents of conversion register, optionally in rawData
  Supported differential measurement pairs: (0, 1), (0, 3), (1, 3), (2, 3).
  Customizable properties of ads1015 given below:
  The "OperatingMode" property of the ads1015 ADC object determines
power consumption, speed and accuracy. The default OperatingMode is
  'single-shot' meaning that the ads1015 performs a single analog to
  digital conversion upon request and goes to power save mode. In
  continuous mode, the device performs continuous conversions.

  The "SamplesPerSecond" property sets the conversion rate.

  The "VoltageScale" property of the ads1015 ADC object determines the
  setting of the Programmable Gain Amplifier (PGA) value applied before
  analog to digital conversion. See table below to correlate the
```

input voltage scale with the PGA value:

VoltageScale	PGA Value
6.144	2/3
4.096	1
2.048	2
1.024	4
0.512	8
0.256	16

NOTE: Do not apply voltages exceeding VDD+0.3V to any input pin.

There's a lot of information here, don't panic, it's mostly for background, the actual code and hook up is surprisingly easy.

For quick reference you need to create a **adcDevice** as shown:

Set up robot control system (code that runs once)

```
% Create a global raspberry PI object so that it can be used in functions
% Create a *raspi* object. You must be on Olin-Robots WIFI network and use your
% pis correct IP address, replace address in code below with yours

robotPi = raspi('192.168.10.141')

% create SparkFun Qwiic Analog Input adc device that is present on i2c bus 1 at address 0x49

adcDevice = ads1015(robotPi,'i2c-1','0x48')

disp('Warning! Data Collection Active! ');
```

And then create a Sense function that will use it to read channels 0-4 as shown below:

Robot Functions (store this codes local functions here)

In practice for modularity, readability and longitevity, your main robot code should be as brief as possible and the bulk of the work should be done by functions

```
%place genratl functions here
```

Sense Functions (store all Sense related local functions here)

```
function rangeData = SENSE(adcDevice)
    disp('Sense');
    rangeData = adcDevice.readVoltage(0)  % read ADC Pin 0 voltage
end
```

Sense: Maxbotix Sonar Array

Your Rover can have up to 2 very long range Maxbotix sonar sensors. The ones you are using on your Rover are the same as the ones on the Sense test stand. You will want to independently calibrate one of them and then either linearize its output or fit a polynomial to its voltage to distance curve. Once you have a solid calibration curve you can write a single function that the main control program can call, once each loop cycle, to see what is around your Rover. Your team can design this function any way you'd like, but a classic way to do this is to pass a vehicle centric simple occupancy grid (matrix) to the function as an input parameter, use the body of the function to add the Sonar range data to that grid, then have function return the updated occupancy grid to the main program. The more computation you can do locally in a function, the more reliable your main code will be. Your Sonar function could look something like this:

```
function  [roverOcupancyGrid] = SonarRange (roverOcupancyGrid, other, other)
% SonarRange adds the current set of Sonar Range measurements to the control
% loop cycle's Rover centric occupancy grid. 0=open space 1-10= obstacles
% written by Scott and Sara 3-30-21 Rev A
    Body of code
end
```

ENGR3390: Fundamentals of Robotics Final Project Rev A

If you need a detailed refresher on how to code for the Qwiic 12bit ADC board, please go back and read through the Raspberry-Pi-Sonar sensor part of the Sense lab. In brief summary of that, please download a copy of the **ads1015.m** function from Canvas, needed for the Pi to talk with the 12bit ADC over the i2c bus from canvas. Typing **help ads1015** at the command line will bring up a detailed description of the function:

```
>> help ads1015
 ads1015 Analog-to-Digital converter.

  adc = ads1015(rpi, bus) creates a ads1015 ADC object attached to the
  specified I2C bus. The first parameter, rpi, is a raspi object. The
  I2C address of the ads1015 ADC defaults to '0x48'.

  adc = ads1015(rpi, bus, address) creates a ads1015 ADC object
  attached to the specified I2C bus and I2C address. Use this form if
  the ADDR pin is used to change the I2C address of the ads1015 from
  the default '0x48' to something else.

  [voltage, rawData] = readVoltage(adc, AINp) reads the single-ended voltage
measurement from AINp input port. Also returns the contents of conversion
  register, optionally in rawData
  Accepted values for AINp are 0,1,2,3.

  [voltage, rawData] = readVoltage(adc, AINp, AINn) reads the differential
voltage measurement between AINp and AINn input ports. Also returns the
  contents of conversion register, optionally in rawData
  Supported differential measurement pairs: (0, 1), (0, 3), (1, 3), (2, 3).
  Customizable properties of ads1015 given below:

  The "SamplesPerSecond" property sets the conversion rate.
```

NOTE: Do not apply voltages exceeding VDD+0.3V to any input pin.

There's a lot of information here, don't panic, it's mostly for background, the actual code and hook up is surprisingly easy.

For quick reference you need to create a **adcDevice** as shown:

Set up robot control system (code that runs once)

```
% Create a global raspberry PI object so that it can be used in functions
% Create a *raspi* object. You must be on Olin-Robots WIFI network and use your
% pis correct IP address, replace address in code below with yours

robotPi = raspi('192.168.10.141')

% create SparkFun Qwiic Analog Input adc device that is present on i2c bus 1 at address 0x49

adcDevice = ads1015(robotPi,'i2c-1','0x48')

disp('Warning! Data Collection Active! ');
```

And then create a Sense function that will use it to read channels 0-4 as shown below:

Robot Functions (store this codes local functions here)

In practice for modularity, readability and longitevity, your main robot code should be as brief as possible and the bulk of the work should be done by functions

```
%place genratl functions here
```

Sense Functions (store all Sense related local functions here)

```
function rangeData = SENSE(adcDevice)
    disp('Sense');
    rangeData = adcDevice.readVoltage(0)  % read ADC Pin 0 voltage
end
```

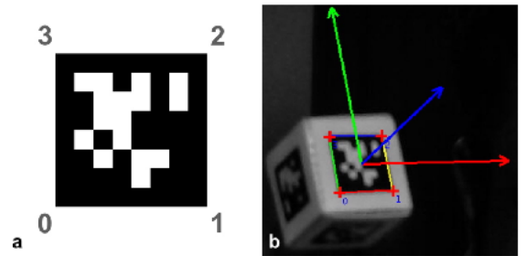
Sense: PiCam Red Circle Dock Detection

Your Rover will have one Raspberry Pi Camera V2, most probably mounted on its Pan-Tilt head. The Pi-Cam you are using is the same as the one on the Sense Lab test stand. Its main function is to find the big red circles on docks and payload drop points. You will need to frequently calibrate the color mask used to do so (based on lighting just before an experimental run). You could even put a small red circle on your rover and build in an auto-calibration procedure to save yourself some work. Just have your Rover auto-calibrate red before each run. Once you have a solid calibration you can write a single function that the main control program can call, once each loop cycle, to see if the dock or payload drop is within sight of Rover. Your team can design this function any way you'd like, but a classic way to do this is to pass a vehicle centric simple occupancy grid (matrix) to the function as an input parameter, use the body of the function to add the dock to that grid, then have function return the updated occupancy grid to the main program. The more computation you can do locally in a function, the more reliable your main code will be. Your PiCamDock function could look something like this:

```
function [roverOcupancyGrid, dockBearing, dockRange] = PiCamDockFind (roverOcupancyGrid, panAngle, tiltAngle)
% PiCamDockFind adds the current PiCam dock sensing to the control
% loop cycle's Rover centric occupancy grid. 0=open space 1-10= obstacles
% 20 = dock
% written by Mary and Beth 3-30-21 Rev A
    Body of code
end
```

Sense: PiCam April Tage Range and Direction (optional)

Your Rover will have one Raspberry Pi Camera V2, most probably mounted on the Pan-Tilt head. The Pi-Cam you are using is the same as the one on the Sense test stand. Its main function is to find the big red circles on docks and payload drop points. But you can also use it to get more precise range and bearing data by looking for April Tags:



You will need to frequently calibrate the color mask used to do color blob tracking outdoors in highly variable real-world lighting; this can pose a significant challenge. To overcome this, April tags were invented to provide a relatively lighting invariant way of getting accurate range and bearing to a visual fiducial target. April tags are a little complex, but fortunately Matlab has built in April tag support in its Vision toolbox. We will provide you with some April tag starter code (see Canvas) and strongly recommend that your team explore using them. Once you have a solid tag finding code in place you can write a single function that the main control program can call, once each loop cycle, to accurately see if the dock or payload drop is within sight of Rover. Your team can design this function any way you'd like, but a simple way to do this is to pass a vehicle centric simple occupancy grid (matrix) to the function as an input parameter, use the body of the function to add the dock position to that grid, then have function return the updated occupancy grid to the main program. The more computation you can do locally in a function, the more reliable your main code will be. Your PiCamDock function could look something like this:

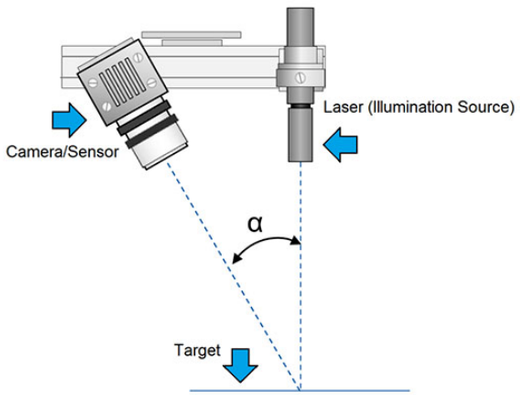
```
function [roverOcupancyGrid, dockBearing, dockRange] = PiCamAprilTagFind (roverOcupancyGrid, panAngle, tiltAngle)
% PiCamAprilTagFind adds the current PiCam dock sensing to the control
% loop cycle's Rover centric occupancy grid. 0=open space 1-10= obstacles
% 20 = dock, 30 = April Tag
% written by Agash and Fred 3-30-21 Rev A
    Body of code
end
```

Sense: PiCam Laser Obstacle Detection (Optional)

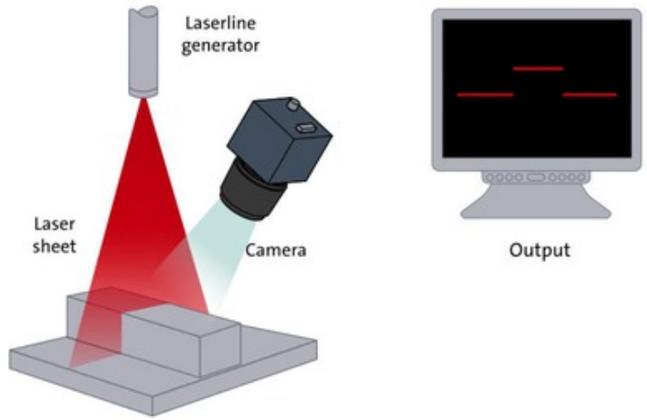
Your Rover will have one Raspberry Pi Camera V2, most probably mounted on the Pan-Tilt head. The Pi-Cam you are using is the same as the one on the Sense test stand. Its main function is to find the big red circles on docks and payload drop points. But it can also serve as a far better obstacle detection sensor, than your Sharp-IRs if used in conjunction with a sheet of red laser light. Your team will be given a red line laser and probably should mount it in your Rover's camera mount. If it's mounted at a slight angle to your camera you will have all of the pieces in place to build an advanced structured light obstacle detection system.

Your SharpIR range sensor will give you very sparse range data at 5 discrete points off the front of your Rover. The two Sonars will give you longer range, but fuzzier coverage 360 around Rover if you mount them on a pan-tilt head. But neither will give you really detailed data about what is immediately in front of the rover, which may be critical to both docking and payload deployment.

The red laser light sheet can be used with the Pi-Cam to do very precise distance ranging across its full field of view using structured light. In a structured light setup, the laser is mounted at a slight angle to the camera:



The camera's center axis intersects the laser line's projection plane at a fixed standoff distance, say 1m, any object closer than 1m will shift the laser line linearly down in the field of view, any further will shift it up. And the shift is linearly proportional to the distance the object is from the fixed standoff distance.



You can use this to get a precise 2d distance map of what is in front of your rover across the camera's full field of view. It will give you much more detailed and much more useful information than the array of fixed IRs can. It would give your rover a decided advantage in the final race, especially in the area of precise positioning. If you mount the laser in your pan-tilt, you can also build up a full 3d range map of what is around your rover by vertically scanning the laser light sheet over the terrain in front of your Rover. Either or both would give your Rover a big sensor advantage in the race. Your Raspberry Pi 4 is quite a powerful small computer and we are using it on the Rover mainly to give you the possibility of exploring more advanced vision based ranging systems like this one.

Please see the provided structured light papers in the Canvas folder for more details.

Once you have a solid set of structured light code you can write a single function that the main control program can call, once each loop cycle, to get really detailed data on what is in front of the rover. Your team can design this function any way you'd like,

but a simple way to do this is to pass a vehicle centric simple occupancy grid (matrix) to the function as an input parameter, use the body of the function to add the laser range data to that grid, then have function return the updated occupancy grid to the main program. The more computation you can do locally in a function, the more reliable your main code will be. Your PiCamDock function could look something like this:

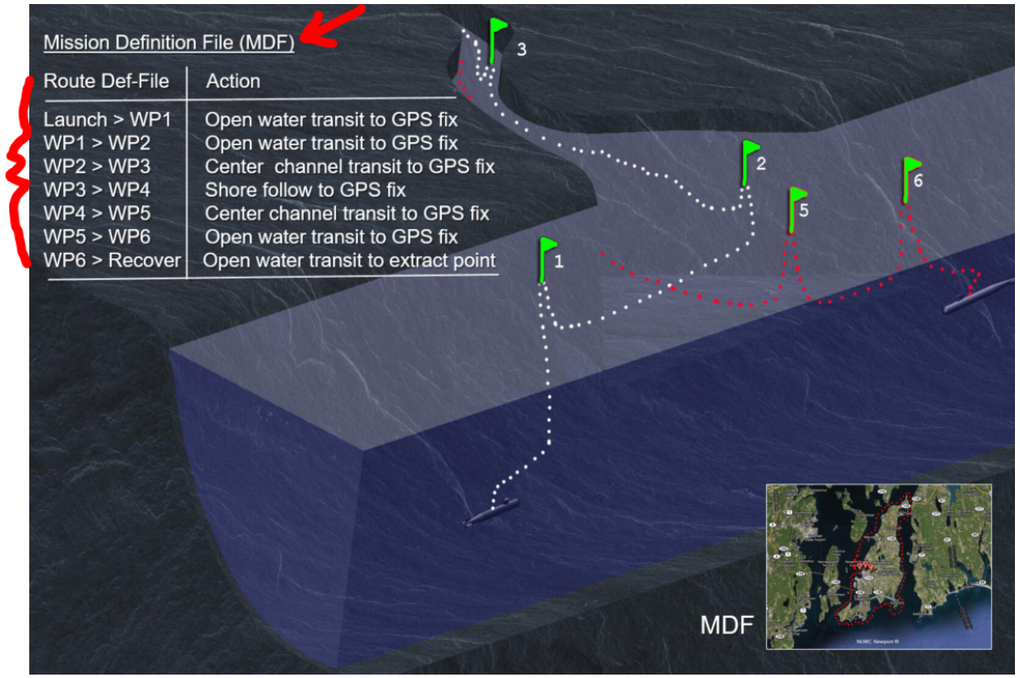
```
function [roverOcupancyGrid] = PiCamLaser (roverOcupancyGrid)
% PiCamLaser adds PiCam structures laser light sensing to
% loop cycle's Rover centric occupancy grid. 0=open space 1-10= obstacles
% 1-3 drive over, 4-10 drive around
% 20 = dock
% written by Mary and Beth 3-30-21 Rev A
    Body of code
end
```

Sense: GPS and IMU Position and Orientation

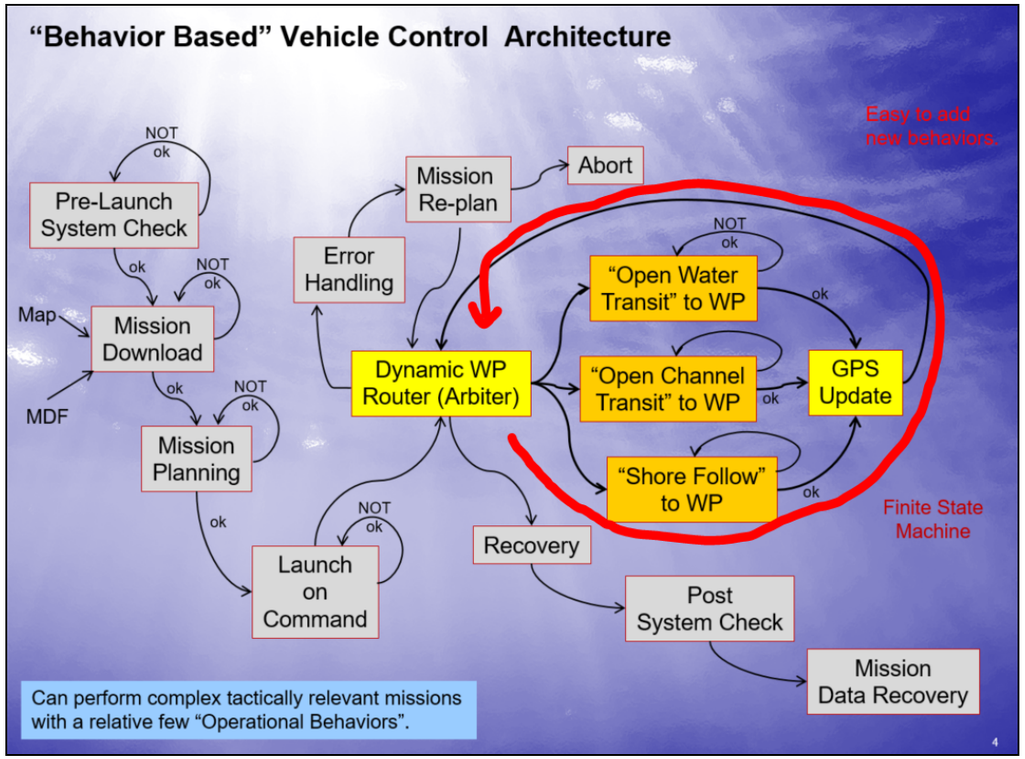
Your Rover will have a very sophisticated Qwiic based GPS-Inertial Measurement Unit sensor to give you both the Rover's location and orientation. Given its complexity, we will supply you with a function and a separate short tutorial on how to use it. Please download both from Canvas.

Think: Mission Planner and Executor

Your final project Rover is going to need a slightly more sophisticated controller than the one you built in your **Think lab**. Your Think tugboat was largely reactive and driven by external sensor readings as to heading and what behavior to do. Your Rover will still need that medium level of control, but you will also need to incorporate a more procedural higher level of control to cycle your robot through a complete real-world mission. Going back to the basics, your team will want to create an extended finite state machine to carefully cycle through all of the components of a complete mission; consider oceanographic sampling mission to swim up a river and collect water samples as shown below:

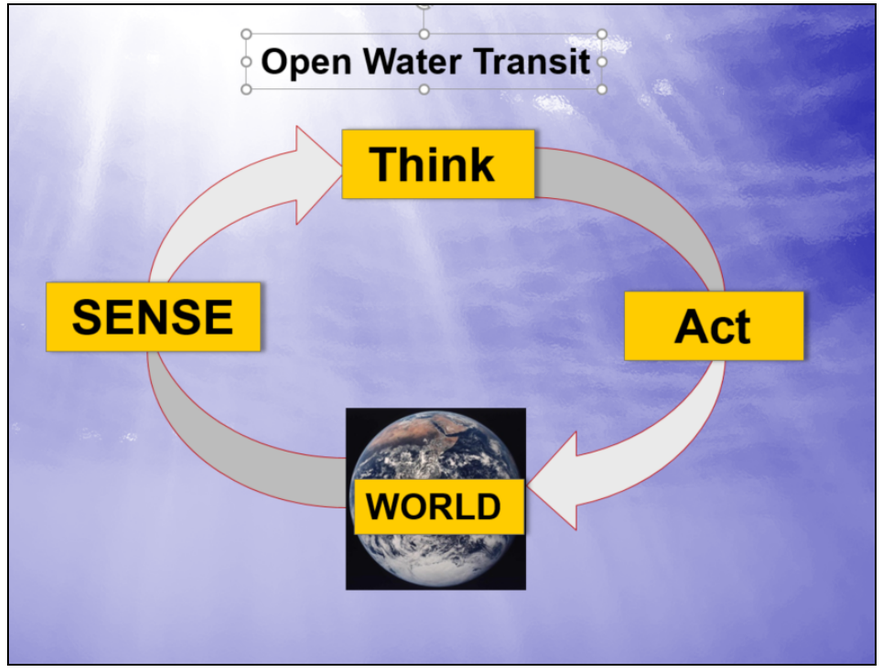


In order to command this mission a **Mission Definition File** must be created that describes what the robot is doing on each leg of the mission and what behaviors it would express during each leg. For example, if sampling water quality up a river, the robot would switch between Open Water Transit and Channel Transit. And there is no point in turning on the sensors until you get to the test site. A Mission Definition File can be turned into a Finite State Machine Graph (which can in turn be converted into Matlab code with a **switch** statement) that looks like this:



The gray states in the above diagram are **code that runs once,** the colored states are meta-behaviors that run over and over, or **code that loops**, until a waypoint or time limit is achieved.

Each yellow meta-behavior is really just a **Sense-Think-Act** loop that runs until it gets your robot close enough to its desired waypoint that it can cycle to the next waypoint and its corresponding meta-behavior (i.e. sub-mission) in the **Mission Definition File**:

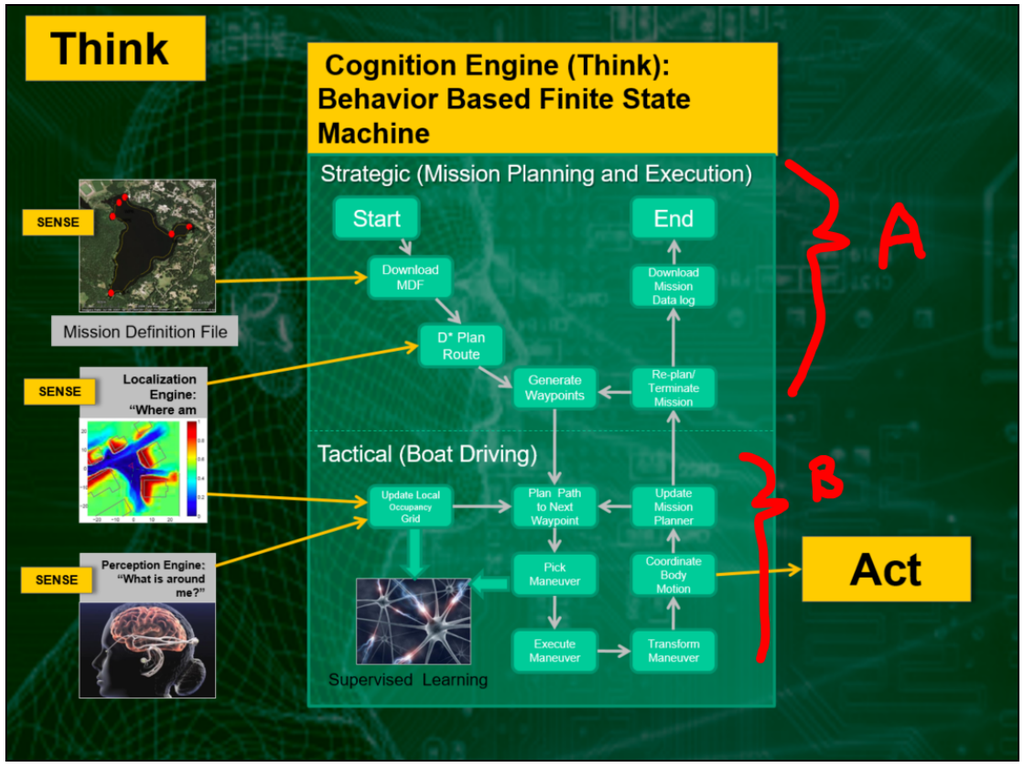


It's important to note that each meta-behavior, like **Open Water Transit** or **Open Oval Transit** is actually potentially made up of several discrete smaller robot behaviors like **Go to Waypoint, Obstacle Avoidance,** and **Look for BigRedDot.**

Each of these discrete behaviors will typically output a **brain wave** of desired robot headings and speeds that will feed into some form of **weighted polling arbiter** to fuse them into a single robot heading and speed that can be passed by ACT functions to drive the robot.

In some very real sense, your team now needs to write a two level controller, the top level is a big **Matlab switch based Finite State Machine** cycling through the desired legs of a **Mission Definition File**. The lower level is a **behavior engine based controller** that will run the two to four discrete behaviors your robot will need

to successfully execute each leg of the mision. Graphically this type of multi-level robot controller looks like this:



We strongly recommend that your full team sits down with a whiteboard and plans out your full Rover mission in the form of a **mutually agreed on Finite State Machine Diagram**, two folks co-write big switch statement that drives mission execution in this diagram and then you break into sub coding teams to create the behaviors, functions and code that goes in each finite state block. Please note, pre-flight mission checking and on mission error handling are just as critical as the big main behaviors like **going to a waypoint** to be able to create a robust Rover that can deal with a dynamic real world environment. A robust controller would be able to deal with sensor malfunctions, sun-blinding, cloudy skies and **annoying Ninja-Aliens**!

Before proceeding please note; it is perfectly ok to stop your Rover dead in its tracks, take a detailed sensor scan by pivoting camera-sonar head 360 degrees and then running some optimization path planning algorithm to plot out what to do next. Just like the Mars rovers, there is no a priori need to keep your Rover moving at high speed continuously. It can stop, look and think. It probably should. Finally a bit of advice:

Carpenters have a solid guideline; **Measure twice , cut once**. The entire robotics industry could benefit from the coding guideline, **Diagram twice, code once** !

As a suggestion, not command direction, your team will need to create a set of Think code blocks (or functions) that fall into the following areas:

Think: Behavior Engine and Waypoint Arbiter

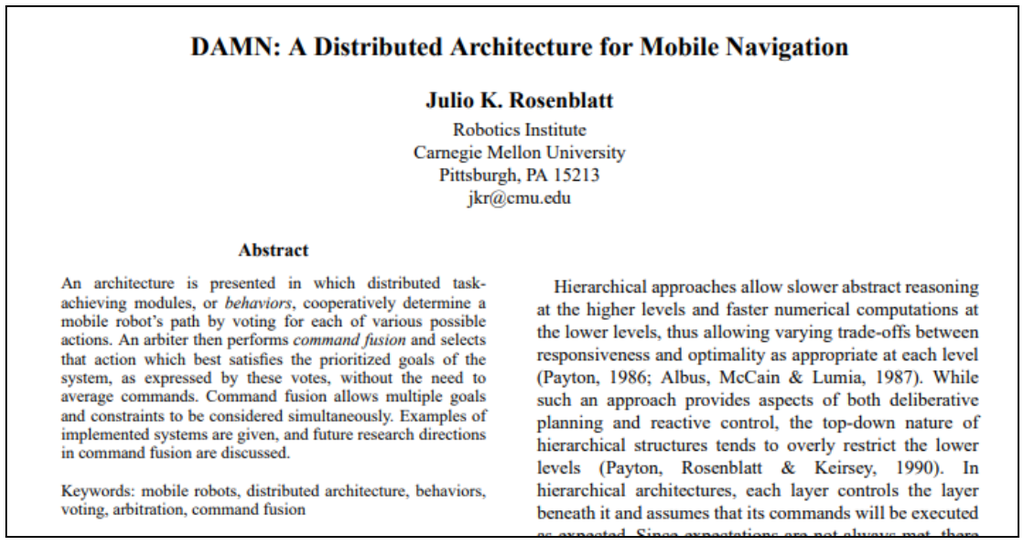
At the core of your Rover controller, you will need a good behavior engine and a behavior arbiter. You have already created a simple version of this via your **ThinkLab Tugboat** experience, a behavior engine usually contains 2 or more competing robot behaviors that all seek to command your robot's actuators.

A complex robot will have more than one arbiter, for example your Rover could have one arbiter merging competing behaviors to command the steering angle and the drive motor, while another arbiter collects and arbitrates behavior commands for where to point the pan-tilt sensor head.

Before diving into your behavior engine arbiter design, it would be good to absorb some seminal background material on one of the first ones.

A classic is the Rosenblatt DAMN arbiter presented in:

ENGR3390: Fundamentals of Robotics Final Project Rev A



DAMN: A Distributed Architecture for Mobile Navigation

Julio K. Rosenblatt
Robotics Institute
Carnegie Mellon University
Pittsburgh, PA 15213
jkr@cmu.edu

Abstract

An architecture is presented in which distributed task-achieving modules, or *behaviors*, cooperatively determine a mobile robot's path by voting for each of various possible actions. An arbiter then performs *command fusion* and selects that action which best satisfies the prioritized goals of the system, as expressed by these votes, without the need to average commands. Command fusion allows multiple goals and constraints to be considered simultaneously. Examples of implemented systems are given, and future research directions in command fusion are discussed.

Keywords: mobile robots, distributed architecture, behaviors, voting, arbitration, command fusion

Hierarchical approaches allow slower abstract reasoning at the higher levels and faster numerical computations at the lower levels, thus allowing varying trade-offs between responsiveness and optimality as appropriate at each level (Payton, 1986; Albus, McCain & Lumia, 1987). While such an approach provides aspects of both deliberative planning and reactive control, the top-down nature of hierarchical structures tends to overly restrict the lower levels (Payton, Rosenblatt & Keirsey, 1990). In hierarchical architectures, each layer controls the layer beneath it and assumes that its commands will be executed

Please download from Canvas and read through. It is a ground floor view of what goes into a robust arbiter:

Abstract: *An architecture is presented in which distributed task achieving modules, or behaviors, cooperatively determine a mobile robot's path by voting for each of various possible actions. An arbiter then performs command fusion and selects that action which best satisfies the prioritized goals of the system, as expressed by these votes, without the need to average commands. Command fusion allows multiple goals and constraints to be considered simultaneously. Examples of implemented systems are given, and future research directions in command fusion are discussed.*

The Distributed Architecture for Mobile Navigation: Deliberative planning and reactive control are equally important for mobile robot navigation; when used appropriately, each complements the other and compensates for the other's deficiencies. Reactive components provide the basic capabilities which enable the robot to achieve low-level tasks without injury to itself DAMN: A Distributed Architecture for Mobile Navigation or its environment, while deliberative components provide the ability to achieve higher-level goals and to avoid mistakes which could lead to inefficiencies or even mission failure. But rather than imposing an hierarchical structure to achieve this symbiosis, the Distributed Architecture for Mobile Navigation (DAMN) takes an approach where multiple modules concurrently share control of the robot. In order to achieve this, a scheme is used where each module votes for or against various alternatives in the command space based on geometric reasoning, without regard for the level of planning involved.

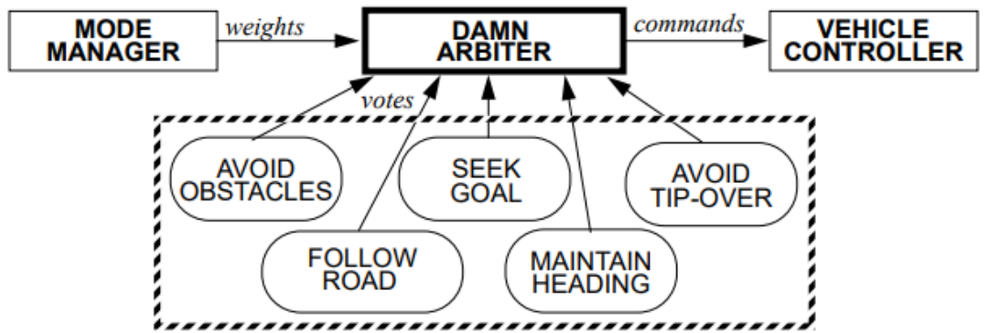


Figure 1: Overall structure of DAMN.

The arbitration process used in DAMN is illustrated in Figure 2, where: (a & b) the votes from behaviors are received, (c) a weighted sum of those votes is computed, nd (d), the summed votes are smoothed and interpolated to produce the resulting command sent to the vehicle controller.

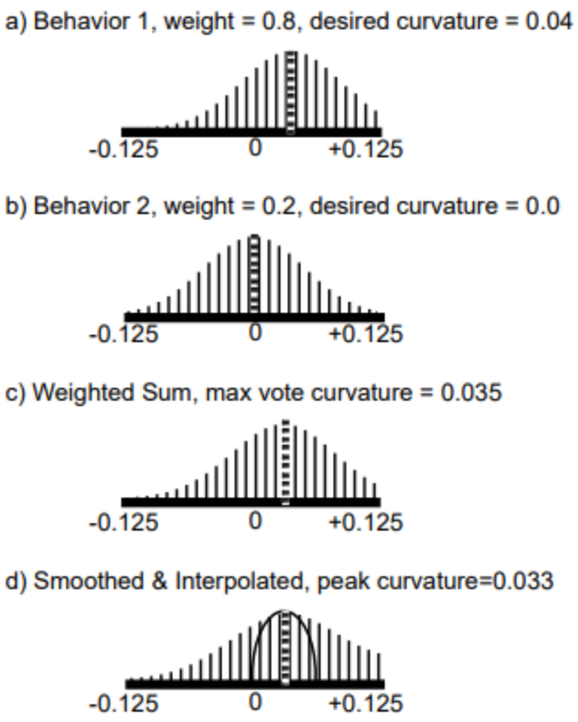
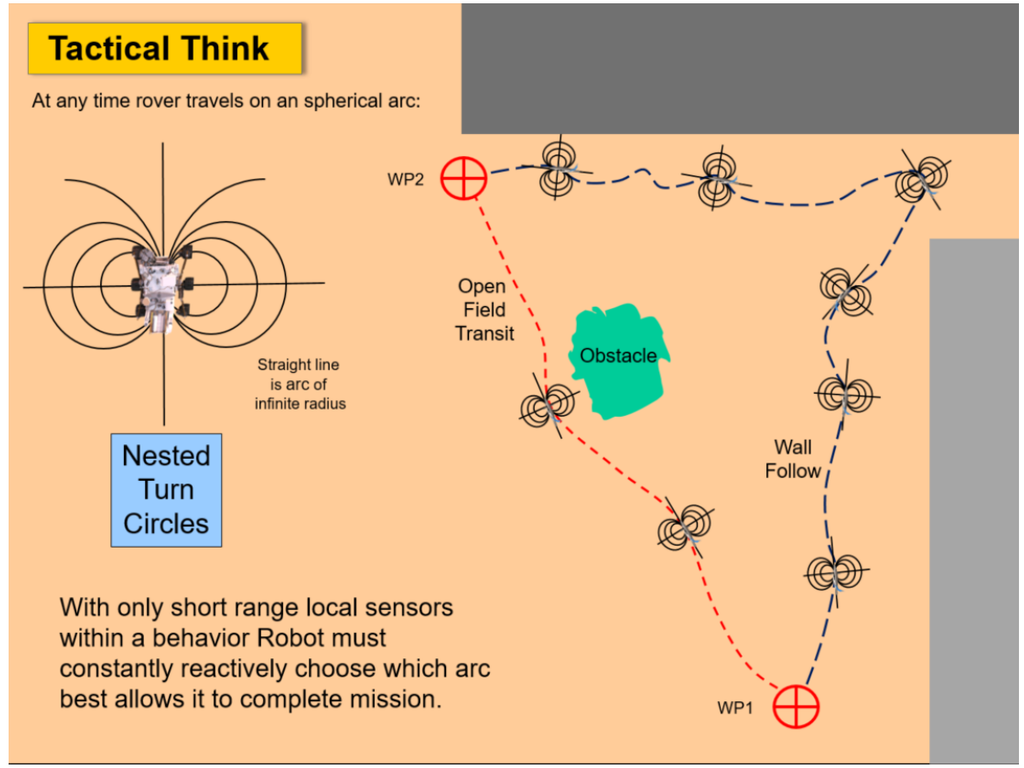


Figure 2: Command fusion process

You have already used a simplified version of this on your ThinkLab tugboat.

Your behavior engine's task is simple. At any instant of time a wheeled ground vehicle in motion is travelling on one of a nested set of **bumble-bee circles** (see diagram). The purpose of your behavior engine and arbiter is to pick the best instantaneous circle to be on to achieve its current main behaviors mission. The DAMN arbiter is a tried and true way to pick this circle, but it is not the only way. If you search behavior arbiters you will find many other equally interesting options.



Once your team has chosen an arbitration path, it is time to move on to creating discrete Rover behaviors. Your team's arbiter could look like this:

```
function [direction, speed] = roverArbiter (behavior1, behavior2, behavior3,
behavior4)
% roverArbiter takes the weighted steering vectors from multiple behaviors and
% merges them down into a single Rover steering angle and speed.
% written by Vision and Wanda 3-30-21 Rev A
    Body of code
end
```

Think: Go to Waypoint Behavior

One powerful concept that allows a human operator to define and control a complex robot mission is the idea of a geographically located waypoint. A waypoint is a physical location in the mission area that has some form of significance. It might be a dock, the edge of a building, a precise GPS latitude and longitude or the location of a scientific payload.

There is a whole sub-genre of people that run through the woods from GPS waypoint to waypoint (often called hikers) and the skill to do so at speed is called orienteering. One really useful behavior (that you already used in your second week Robot Toolbox training) that you could create is to use Matlab's built in **Pure-Pursuit** functions to create a path stored as physical waypoints in a **Mission Definition File**.

Depending on how your team structures your code, a GotToWaypoint behavior function could look something like this:

```
function [bumbleBeeSteeringVector] =GoToWayPoint(roverOcupancyGrid,
targetWaypoint)
% GoToWayPoint takes the current occupancy grid and next target waypoint and %
% generates a weighted steering angle brainwave that can be used by an arbiter
% to steer Rover. Rover centric occupancy grid. 0=open space 1-10= obstacles
% 20 = dock, 30 = April Tag
% written by Tom and Wanda 3-30-21 Rev A
    Body of code
end
```

Think: Obstacle (and Alien) Avoidance Behavior

A well designed Rover control system doesn't drive into either stationary or moving obstacles. Using the combined data from Sharp IRs, Sonar and your camera system; fused into a **Rover-centric Occupancy Grid** your rovers obstacle avoid behavior should produce a weighted steering vector that minimizes the potential for collison. It could look something like this:

```
function [bumbleBeeSteeringVector] =ObstacleAvoid(roverOcupancyGrid)
% Obstacle Avoid takes the current occupancy grid , the obstacles on it and
% generates a weighted steering angle brainwave that can be used by an arbiter
% to steer avoid Rover hitting obstacles. Rover centric occupancy grid. 0=open
% space 1-10= obstacles
% 20 = dock, 30 = April Tag
% written by Tom and Wanda 3-30-21 Rev A
    Body of code
end
```

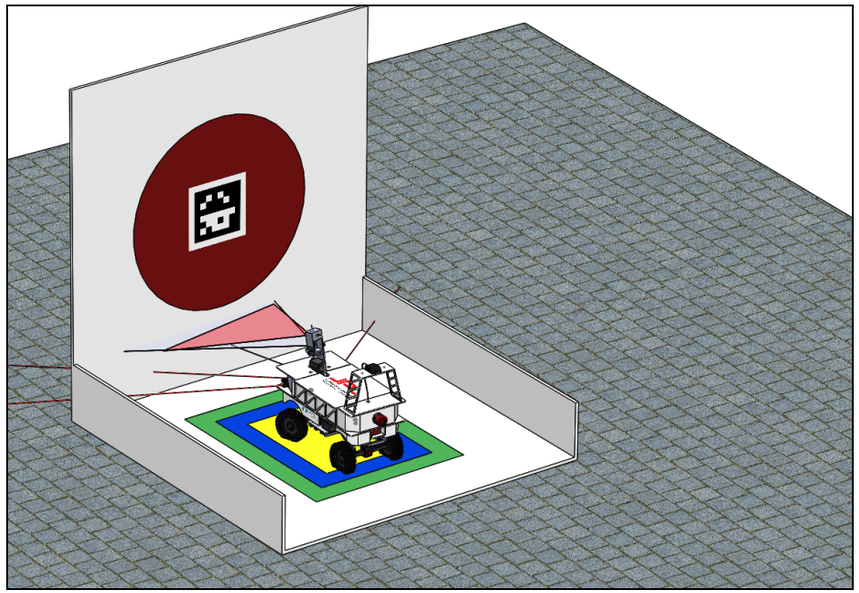
Think: Wall or Curb Follow Behavior

GPS has its advantages and disadvantages. First off, it can only tell you where your Rover is within a meter or two. Second, it tends to drop out near buildings at the most awkward of times. You can compensate for this by using fixed observable features in your environment, like walls to follow, that in conjunction with **a good digital map** (ground based occupancy grid) will let your Rover smoothly get from waypoint A to waypoint B. Luckily the Rover test track in the Oval has lots and lots of curbs, walls and features your Rover can follow to get around in the absence of reliable GPS. It's highly recommended that you consider creating a right and left-hand wall following behavior. A prototype for this would look like:

```
function [bumbleBeeSteeringVector] =WallFollow(roverOcupancyGrid, right-left)
% WallFollow takes the current occupancy grid , the obstacles on it and
% generates a weighted steering angle brainwave that can be used by an arbiter
% to steer to follow the wall at a fixed offset. Rover centric occupancy grid. 0=open
% space 1-10= obstacles
% 20 = dock, 30 = April Tag
% written by Tom and Wanda 3-30-21 Rev A
    Body of code
end
```

Think: Dock Rover Behavior

At the start and potentially end of each Rover mission, your robot has to leave a well-marked dock and find and park at another well marked dock.



Although your team will be provided with the GPS Latitude and Longitude of the docks physical location, that position fix will only be good to 1-2 m and you will need to use your vision system, pan-tilt sensor head, Red-dot finder and potentially April Tag finder to precisely navigate onto the docks parking station. In addition to the Dock Rover Behavior, you might also want to create a Find Dock behavior that stops the rover and slowly scans a 360 degree horizon to look for the dock.

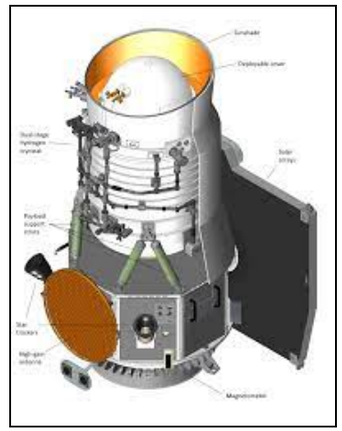
Once you have a good idea of where it is and are close enough to it to have a good visual lock on the big red dot and April Tag, then you can visually steer your rover onto the landing pad. You might need a behavior function that looks like this:

```
function [bumbleBeeSteeringVector] =DockRover(roverOcupancyGrid,
RedDotCoords)
% DockRover takes the current occupancy grid , the obstacles on it and
% generates a weighted steering angle brainwave that can be used by an arbiter
% to steer Rover toward dock. Rover centric occupancy grid. 0=open
% space 1-10= obstacles
% 20 = dock, 30 = April Tag
% written by Tom and Wanda 3-30-21 Rev A
    Body of code
end
```

One very interesting wrinkle is how to best stop a rover on a landing pad a fixed distance from Big Red Dot.

Think: Deploy Scientific Payload Behavior

As part of the overall mission goals, your rover will need to go to certain GPS located waypoints and drop a scientific payload at them, the “scientific payload” will be a small 3D printed object that’s (hopefully) designed to be easy to grip.



The drop boxes will be a short (~2in tall) tupperware box with a red dot and apriltag. Using similar code to that which you use to find the dock, you will also want to create a behavior that will let you drop your payload squarely in the payload box. Consider creating something like this:

```
function [bumbleBeeSteeringVector] =DeployPayload(roverOcupancyGrid,
RedDotCoords)
% DeployPayload takes the current occupancy grid , the obstacles on it and
% generates a weighted steering angle brainwave that can be used by an arbiter
% to steer Rover toward the payload drop site. Rover centric occupancy grid.
% 0=open
% space 1-10= obstacles
% 20 = dock, 30 = April Tag
% written by Sam and Jackie 3-30-21 Rev A
    Body of code
end
```

The challenge here will be to get cleanly right over the drop bin.

Think: Go Home Behavior

If and when something goes wrong, your Rover gets lost, it starts to rain or you complete the mission successfully, you just don't know what else to do; it will always be useful to have a **GoHome** behavior (just like your cars GPS), that will safely and autonomously bring your Rover back to its starting point and home dock.

```
function [bumbleBeeSteeringVector] =GoHome(roverOcupancyGrid, HomeCoords)
% GoHome takes the current occupancy grid , the obstacles on it and
% generates a weighted steering angle brainwave that can be used by an arbiter
% to steer Rover towards its home dock. Rover centric occupancy grid.
% 0=open
% space 1-10= obstacles
% 20 = dock, 30 = April Tag
% written by Sam and Jackie 3-30-21 Rev A
    Body of code
end
```

Act: Drive Rover on Curved Path R at Speed S

Compared with the potential complexity of both the Sense and Think functions needed by your Rover, the Act functions should be relatively straight forward. Your team may want to create three; one for the Rover, one for its Pan-Tilt Head and one for the payload deployment system. You have already successfully controlled RC type servo motors in the **ACTLAB,** here the one new wrinkle for the final project is that we will use a hardware extension of the Raspberry Pi in the form of a **Spark-Fun Servo pHAT** to actually drive the PWM signals to the 5 servo motors. This will accomplish three positive outcomes. It separates the need for the Pi to deliver precise, non-jittery, PWM signals from its GPIO pins, the pHAT has its own embedded processor to do that hard real-time task for up to 16 RC motors. In doing so, it frees up enormous Pi compute cycles to work on your machine vision (find Red Dot, or Find April Tag) code. And this particular board also includes the base port of an easy to use Qwicc IO device expansion system that will let you daisy chain your Sharp and Sonar sensor back to the Pi.

For the first act function, we will provide you with this code, see pHat tutorial and download code from Canvas:

```
function [ ] =DriveRover(steerAngle, speed)
% DriveRover takes in the desired steering angle and speed for the Rover and
% commands the steering Servo and embedded drive motor speed control to those
% setpoints.
% written by Brady and Amy 3-30-21 Rev A
    Body of code
end
```

Act: Point Pan-Tilt at Angles {P. T}

If you are focused on conserving energy on your Rover mission (and you should be!) It makes a lot more sense to scan the horizon with sensors mounted on a movable Pan-Tilt Head than to rotate the entire vehicle to collect data. For this and a host of other good reasons, you will probably need a simple Act Pan-Tilt command that will let Think behaviors point the head sensors where they should be pointed. You probably need something like this:

```
function [ ] =DrivePanTilt(panAngle, tiltAngle)
% DrivePanTilt takes in the desired pan angle and tilt angle for the sensor head and
% commands the pan and tilt Servos to those setpoints.
% written by Brady and Amy 3-30-21 Rev A
    Body of code
end
```

We will provide you with the Qwiic pHAT driver for this. Download from Canvas.

Act: Deploy Payload

Last but not least, you will need to control the position of the RC Servomotor that drives your payload deployment system. We will provide the base Qwiic drive code for you from Canvas too, but you will probably need a function like this:

```
 function [ ] =DeployPayload(payloadNumber)
% DeployPayload drops the number of Payloads given as an input parameter.
% written by Brady and Amy 3-30-21 Rev A
    Body of code
end
```

Suggested Team Work Breakdown

There is a reasonable, but significant amount of work to do between the final project kickoff and the final (wildly successful) demo. In order to complete it in time, we recommend that you organize your student team to use individual strengths to the maximum possible level, while still giving all teammates a balanced and fair access to contributing to the overall design. Your team can structure themselves any way you are comfortable, but please consider the guideline below while doing so. It may make sense to break up into four sub-teams at the start and end with one team where everyone is coding and field testing at the end for the final demo. In discrete terms your team will need to cover the following tasks

Project Management

Your team needs a **Keeper of the Main Finite State control diagram**, who will both constantly update it and will keep it aligned with the project mission. Your team will also need a **Work in Progress task manager** whose goal is to make sure all tasks are getting done and will be completed in time. Please see the final section of this document for a brief overview of creating a work in progress tracking system.

ME

This course has been pretty heavily skewed to code generations and robot component wiring up to this point. To give our hard-core MEs something substantial to do, your team will need to quickly design, manufacture and assemble your Rover's Sintra Hull as well as design and print its sensor mounts and sensor head. Probably only need 2-3 students on this, preferably with previous strong ME design, fabrication and assembly skills.

ECE

Using the provided wiring diagram as a guide and promising to never hook up any hardware hot, the EE work to be done includes making a quick plug-it-all-together and get it to work on a table top phase, followed quickly by a careful and robustly install all components into the Rover's hull phase. Bearing in mind that we've overspent the courses budget at this point and there are no spares, the teammates selected to do this work should be patient, methodical, careful and have reasonably good EE assembly skills and **NEVER EVER PLUG IN PARTS HOT !** Please note: we don't have spares, if the ME's break an ME part, they can just rebuild it. If your EE team burns out, shorts, frys or loses a critical piece, I can't replace it and you might have to pass the hat around the team to fund ($$) a new replacement part.

CODE

The initial code work would be in the form of extending the course robot code template to include all of the new mission functionality required and then once that is in place, generating, field testing and debugging the many functions required to do the full mission. It may be advisable to have a few teammates work on the main structure of the code first,while the ME's are design-building the hull and the EE's are hooking up the table top wiring, then have the entire team shift into writing functions.

This all being said, please strive for inclusion and don't shift teammates from tasks they passionately want to do , just because they are relatively inexperienced at them. This is Olin, we learn by doing.

Suggested Project Timeline

Your team will have just about 4 weeks plus a few days to design, build, test, debug, test, improve, test ,debug and demo your Rover. This is a tight, but realistic schedule. You Rover is, by design, a fusion of all of the basic fundamental robotic components you have already gained technical skills with in the Sense-Think-Act labs. To stay on track and on schedule, you will want to consider meeting these project milestones:

Project kickoff: Monday April 12

Rover built and ready for code: Wednesday April 21 (a week and 2 days from kickoff)

Achieve stable waypoint navigation: Wednesday April 28 (2 weeks from kickoff)

Perform full mission (unoptimized): Wednesday May 5 (3 weeks from kickoff)

Optimize mission for final demo and Final demo: Tuesday May 11 !

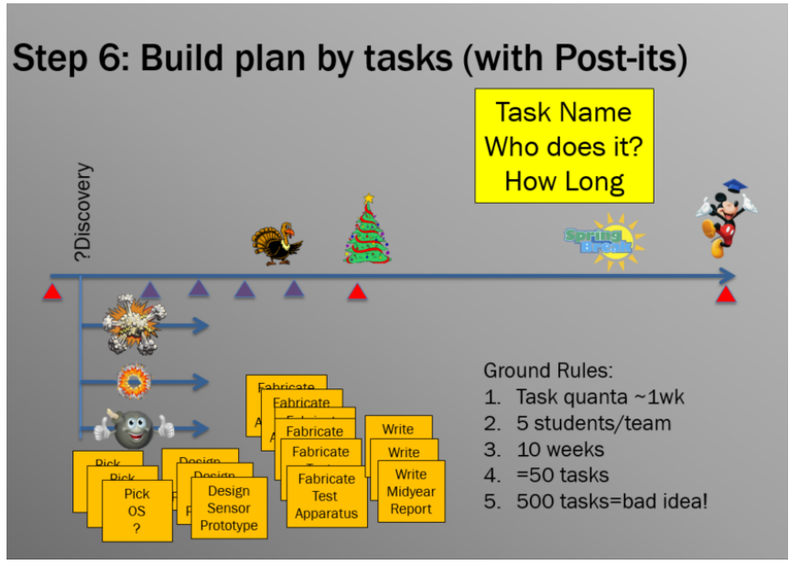
Required Work Management System

By design and by intention this final project is not as heavily scaffolded with respect to workflow as the three preceding Robot labs were. This is to give your team a substantial experience in self-organizing your own workflow to optimize project results with a small team within the given time frame.

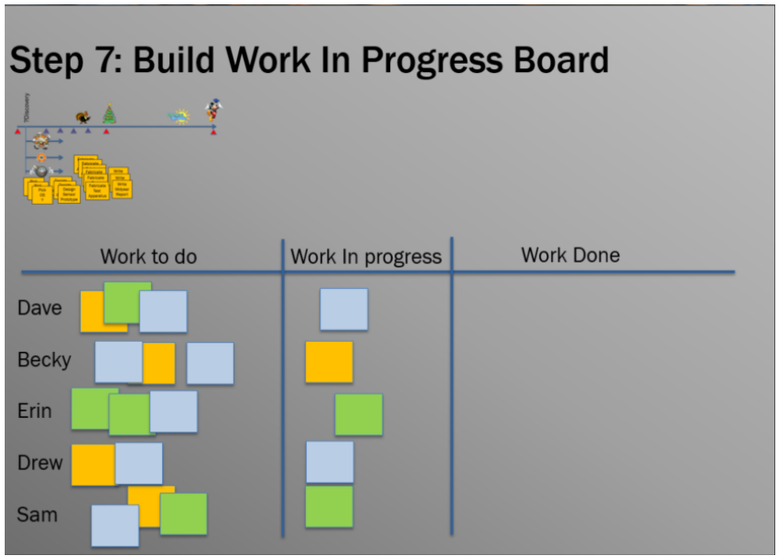
You are free to self-organize in any way you like, but that being said, to insure fairness to all, to achieve individual student contribution transparency and to give you some exposure to a low-overhead small-team self-management system; we will use a commonly used, lightweight post-it based **Work In Progress** (WIP) workflow guidance system.

To set up and use this system, first create your team's finite State Machine Diagram to facilitate the planning conversation. Then sit down as a team (we will do this in class) with a set of post-it notes and write out each half week long task needed to get to the final demo. Each student gets roughly 8 post-it notes and half week tasks.

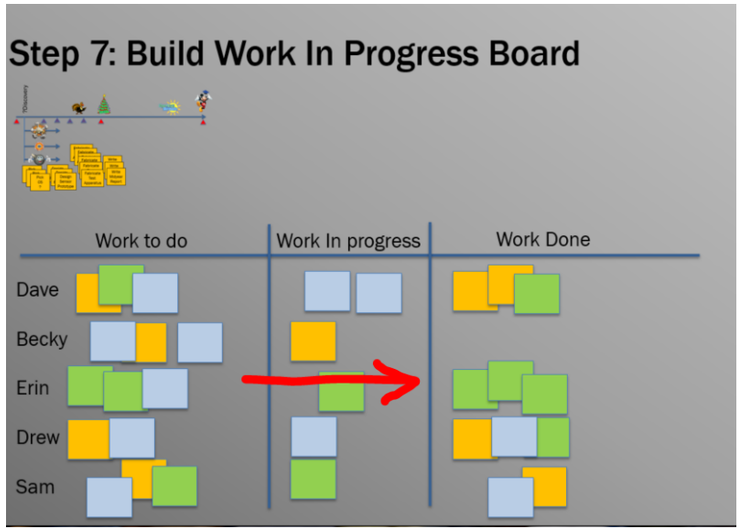
ENGR3390: Fundamentals of Robotics Final Project Rev A



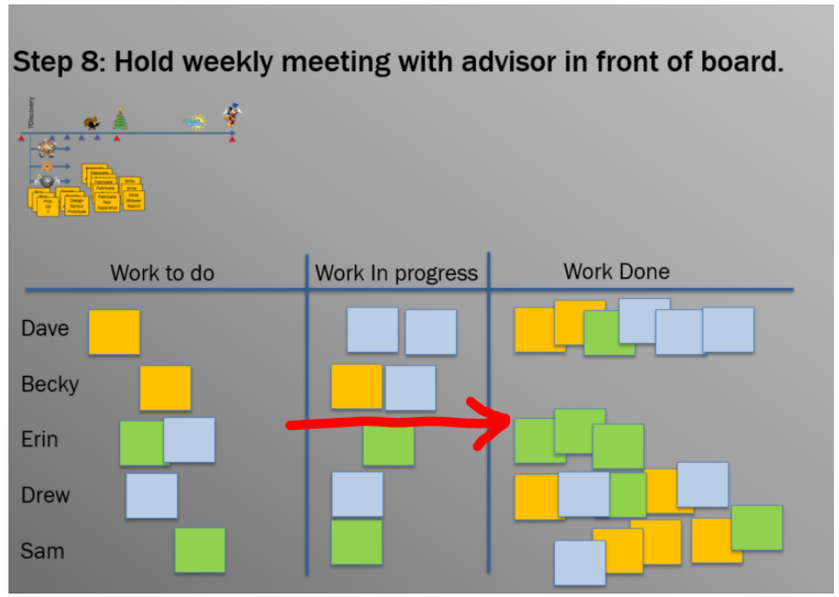
In your team space create a three column WIP board and place the tasks to be done by each student next to their names:



As work progresses, move the post-it notes from the left to the right:



On a bi-weekly basis update the WIP board:



A good common WIP board lets all team members see progress, see what work remains to be done and allows the team to quickly update their supervisor (and course instructors and Ninjas) on progress and setbacks. Most importantly it lets your team work in a highly efficient, asynchronous manner, see where you stand and see what needs to get done next. WIP boards like this are very common in start-up companies, aerospace companies and consumer product companies, really any place with a fast project cycle time. Get good at this and your team will succeed.

Conclusion

You have a great team, a full supply of hardware and a well-defined mission. Your team has all of the pieces needed to succeed in this final Planetary Rover contest. Don’t forget to seek out help from the course instructor and Ninjas as needed and now dive right in and go for it !

May the best Rover win.