Team 11 – NASA/RASC-AL Robo-Ops

FAMU/FSU College of Engineering

Final Report

Faculty Advisors:

Jonathan Clark – Mechanical Engineering
Uwe H. Meyer-Baese – Electrical Engineering

Team Members:

Boris Barreto – Electrical and Computer Engineering
Jason Brown – Mechanical Engineering
Justin Hundeshell – Mechanical Engineering
Linus Nandati – Electrical Engineering
Tsung Lun Yang – Mechanical Engineering

Date: 04/17/2014

Contents

Table of Figures

1.0 Abstract

This document describes the FAMU/FSU College of Engineering's rover design for the 2013/2014 NASA RASC-AL Robo-Ops competition. The team consists of five undergraduate engineering students each with interests in space exploration and robotics. The team has experience in fields pertinent to remote robotic systems such as wireless communications, object-oriented programming, materials science, and mechatronics. Professional guidance and working facilities have been provided to the team by their main advisor, Dr. Jonathan Clark, and the STRIDe Lab, which operates under his direction. Additional guidance has been provided by the Electrical Coordinator, Dr. Michael Frank, and the Mechanical Coordinator, Dr. Kamal Amin.

The goal of this year's team was two-fold. For one, the team was to build upon the successes of last year's competition team while developing cutting-edge designs to overcome the shortcomings of the previous year's rover. Furthermore, the team had the goal of entering the 2014 NASA RASC-AL Robo-Ops competition. The team made many improvements upon the design, namely improving the controls, enhancing the locomotion, implementing a new gripper, and utilizing the Verizon 4G network for communication with the rover. However, the team was unable to meet all their goals as NASA denied their proposal to enter the competition based on the locomotion of the platform. This was quite a blow to the team, especially since this event drastically affected the economics of the project and the locomotion was seen by NASA as a strong point in last year's competition. The change in opinion by NASA was quite a setback, however, the team recovered and modified the goal with a view to next year's competition.

With respect to the rover itself, the proposed design features hexapedal locomotion which provides the rover with key features to be rover successful. For one, the legs can operate over terrain that wheels cannot. On top of the platform is a four degree of freedom robotic arm designed with a complementary unique compliant gripper. Finally, this year's design features a new control interface will allow for real time control and dynamically improved communication throughput paired with iterations of redundancies.

2.0 Team Members and Facilities

Jason Brown, Project Lead, Lead Arm Programmer – Jason Brown is a senior at The Florida State University pursuing his BS in Mechanical Engineering. He spent this past summer working in the Center for Intelligent Systems, Controls and Robotics on the development of an autonomous quadrotor. The work focused on the integration of an autonomous quadrotor with an autonomous ATV.

Linus Nandati, Communication Specialist – Linus Nandati is a Senior at The Florida State University and is pursuing a BS in Electrical Engineering. Currently, he is employed by Tech One IT Consulting as a Networking Intern and will be promoted to a full-time professional upon graduation. His area of expertise is communication systems and network security.

Boris Barreto, Chief Programmer – Boris is not only bi-lingual in English and Spanish, but is also a dual-major in Electrical and Computer engineering. His specializations include hardware and software engineering and has experience in communications, electronics, and power. He is eager to implement these skills in engineering our Robot.

Tsung-Lun 'Chris' Yang, Compliant Gripper Designer – Chris is a citizen of both U.S and Taiwan, he fluently speaks both English and Mandarin Chinese. Chris joined the STRIDe Lab in 2013 as a research assistant to focus on specialized attachment mechanisms for dynamic climbing on natural surfaces. This experience should transfer towards our claw mechanism.

Justin Houdeshell, Robotic Arm Designer – Justin has held many leadership positions and understands dedication. Multiple years' experience as the Robotics Captain in HS, has led his team to nationals on several occasions. Justin is very familiar with the implementation of cameras and sensors to develop autonomous robots. Justin lives an active well-rounded lifestyle with academia as first priority.

STRIDe Lab, Working Facilities – Scansorial and Terrestrial Robotics and Integrated Design Lab was founded in 2007 by its director, team advisor Dr. Jonathan Clark, with the aim of developing robotic platforms which can challenge the agility and versatility of animalsand insects. STRIDe Lab has worked extensively on the design and control of legged platforms and is well equipped for the task of developing a legged rover. The lab boasts several tools to aid in the manufacture of a rover including a laser cutter, composite material construction tools, extensive analysis and testing devices, and a capable machine shop located adjacent to the FAMU/FSU College of Engineering.

3.0 Lessons Learned

Last year the FAMU-FSU team competed with a legged rover with a low degree of freedom robotic arm, utilized a 3G/4G Verizon Wireless Dongle inserted into a type G router which communicated commands sent from the home base GUI system via TP-Link MR3420 router LAN connections to 2 Raspberry Pi's. The Raspberry Pi's connected sent commands to a Xula2-LX25 FPGA via SPI communication, which sent commands to each of the six individually actuated legs.

The rover consisted of 6 independently actuated C shaped passively compliant legs utilizing Buehler's Algorithm to command the position of each leg. The legged motion provided a unique and capable means of handling the various obstacles at the NASA rock yard. Experience from both last year's team in the competition and experiences in the Lunabotics Competition in 2012 has shown a hexapedal legged the platform is capable of handling sandy terrain (including fine Regolith sand), steep inclines, and obstacles larger than the ground clearance. This further supported the ability of legged locomotion and thus we plan to implement the hexapedal locomotion as will be discussed later.

While the locomotion platform performed well against any terrain in a straight line, the control algorithms struggled with navigating around obstacles and handling minor adjustments. The legged locomotion had a fixed minimum distance for forward motion with standard stepping motion. Additionally, turning motions are more challenging to accomplish compared to wheeled platforms. The

rover last year had 3 different gaits: moving forward/reverse, turning in place, and a hill climbing gait. The team experienced challenges in moving to a rock once it was identified. Some of the limitations of the control are inherent to the locomotion system, which include the standard step size which influences the minimum in place turn which can be achieved. This made clear the need for more control schemes to effectively maneuver the course. Some of the limitations were overcome with new gaits and control schemes such as turn-while walking, which will be discussed below.

The Sample Extraction Module from last year was designed to take advantage of the unique locomotion platform, having only 2 linear degrees of freedom in the x and y directions, with the rover itself providing the third degree of freedom. The mechanism was relatively slow from the point of identifying a sample to acquiring the sample. The system's stored configuration was far away from the extraction area. Next, the system required the operator to make many minor adjustments once the system was deployed. Limited visibility and perspective from the main camera then amplified the difficulty in making the small adjustment. The limited workspace hindered the modules collection speed which was then amplified by the limited movement of the rover. This year, the team completely revamped the arm, creating a three degree arm and gripper. With increased mobility and intuitive controls, the modified arm is a huge improvement and it will be discussed below as well.

The user control from last year was a GUI system which used command inputs to control the rover. The command inputs were constructed through the GUI in the form of arrows which designated direction and a window where the user entered the desired number of steps. The system would then execute the command and a new command could not be entered until the previous command had been completed. This method of user input slowed the team in handling the rover, for it required several seconds to enter each command. Since hundreds of commands were entered during competition, the seconds added up and the added lag in communications hindered user command speed. This made clear a new interface need to be developed which was more intuitive and allowed for the rover to respond in real time. There were several ideal features from the GUI such as a large display for the video feeds and relatively intuitive controls. This year, the GUI will implement more advanced commands coming from an Xbox controller.

Last year's communications system utilized Verizon Wirelesses 3G network to send commands to the rover and send the video feed to the mission control. The Verizon network was slower than expected on the day of competition, which resulted in the connection being dropped multiple time during competition. This made clear a more advanced communication system would be necessary to maintain connection and transmit the necessary video feed. This year, the team upgraded to the Verizon 4G network and used the services of no-ip.com to secure IP addresses. The improved communications design will be discussed below.

The table below summarizes the lessons learned from last year's team and the improvements made over the past year:

Table 1 -- Lessons Learned

	Performance During Competition	Lessons Learned
Locomotion	Straight Line handled all obstacles Navigation was challenging	Showed legged motion's ability to handle obstacles Showed need for additional gate types
User Interface	Required multiple commands to execute step based control	System slowed teams ability to control the rover and real time interface would be desirable for competition
Communication	Connection dropped multiple times and lagged continuously during competition	Communication dependent on single network is vulnerable to network fluctuations.
Sample Extraction Module	Able to pick up single rock, but struggled for several minutes and failed to acquire an relatively easy sample	Slow mechanism Poor vision Limited reach/workspace Poor terrain adaptation

4.0 SEM System

4.1 –Robotic Arm

4.1.1 – Robotic Arm Construction

A goal for this year was to develop a new robotic arm with a higher degree of freedom and lighter weight. We wanted to reduce the weight of the arm as it weighed around 10 kg, the new arm weighs no more than 3 kg, approximately 1/3rd of the weight. The old arm was capable of 2 degrees of freedom, this has also been improved, doubled to 4 degrees of freedom. If we examine the design of the current model arm, we started with a target reach and strength. Over engineering to minimize error, we ran calculations with a large budget expecting to receive such funding. The design was to use square aluminum beams, stainless steel plates, and large high efficiency motors.

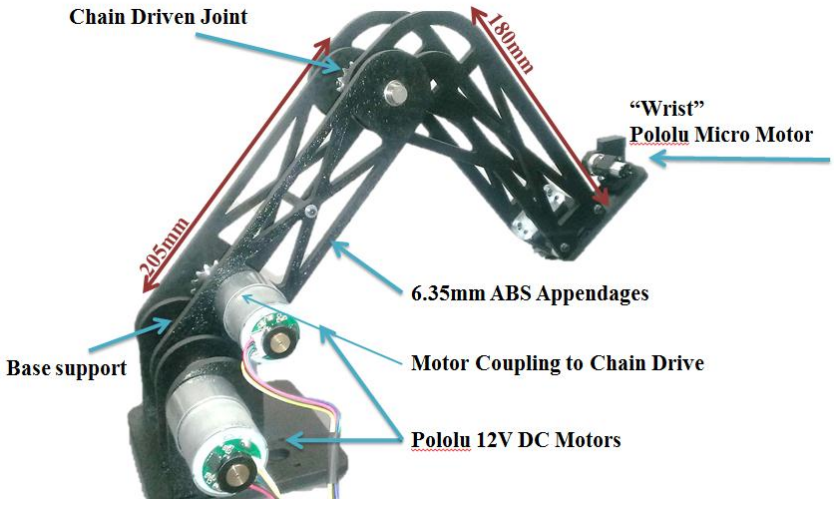


Figure 1 -- Sample Extraction Module Arm

With this basic design, materials, and motors selected, simulations were ran to determine the torque loads on each motor as well as the stresses along the ligaments of the arm. This ensured that all

our components would work and gave us the ability to extrude material from the beams to reduce weights and torques. Creo Parametric simulations provided all necessary data elements. When the budget had been reduced significantly, the same general design was created with the available parts.

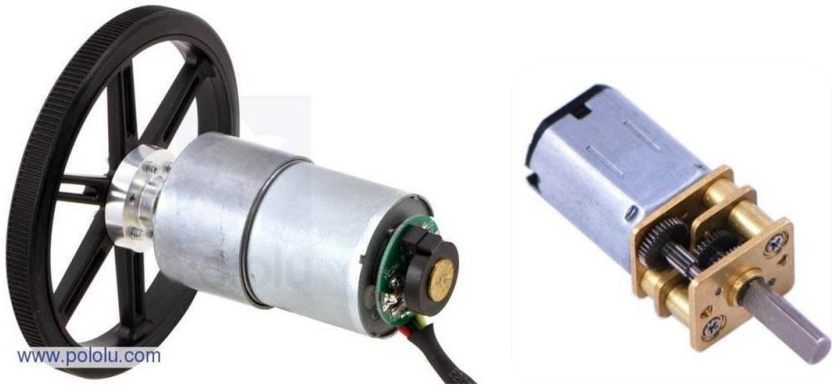


Figure 2 -- Motors Selected for Arm

The STRIDe Lab provided us with an array of tools as well as a few materials. To overcome the unexpected change in budget, we shorted the desired length of the arm by about 30%, which would still allow plenty range of motion. Instead of square aluminum beams, parallel ABS sheets used. Now with different distances and materials, calculations and more simulations told us the cheapest motors we could now use were 1/8th the cost of the original design and much lighter. We kept the arm slightly over engineered to allow for heavier object collection and for future project work.

In this current and final design, sufficient reach was achieved and the weight had been reduced significantly. It is comprised of 2 large Pololu DC motors, one directly at the base with a high torque shaft coupling agent. The second drives a chain belt to the elbow joint and is attached as near to the base as possible to reduce strain on the first motor. A Pololu “micro-motor” is in retrospect weightless and paired with a small gearbox powerful too, this is found at the wrist which attaches to the innovative robotic extraction module.

The arm was designed to be constructed using the tool available in the STRIDe Lab which include a drill press, hack saw, and laser cutter. The laser cutter allow for 2D cuts of most plastics up to a 1/4”. The laser cut parts were then drilled to allow screws to support 3D structures. Several parts were ordered to increase the precision of the manufacturing. The shafts, bearings, shaft couplers, and shaft clamps were all purchased from Misumi, while the motors and encoders were purchased from Pololu.

All in all, the robotic arm was physically a success as it meets each of our specifications. The unfortunate part is the programming as the DC motor and motor controller are different makes, they didn’t work as easily as plug-and-play.

4.1.2 – SEM Programming

With the chosen design of a 3 degree of freedom (DOF) arm with a 1 DOF wrist and a potentiometer model, the programming had to provide precise position control and had to be able to read the positions from the potentiometers. In order to provide the precise position control, the

RoboClaw 2x30A motor controller with built in PID control command was chosen to read the encoders and internally make decisions on the direction and speed the motors would move. PID control uses the error in the position and desired velocity to scale the motor power. For our purposes, the motors desired position would be set by the potentiometer model and the desired speed would be set to zero since the arm needed to hold its desired position.

Figure 3 -- RoboClaw Motor Controller

The RoboClaw motor controller would be connected to one of the Raspberry Pi control modules and communicate using UART communication protocols. UART is a 2 wire communication protocol that uses a transmission and receive terminal, such that the transmit terminal from one device goes to the receive terminal on the second device and vice versa. The RoboClaw then would read the information sent across the UART lines and compare the input to a bank of set commands. Each command has a set order of inputs it is expecting with an example provided in **Fig. X** . The address of the RoboClaw is used to distinguish between several RoboClaws which like the team used. As part of the communication protocol, a value called checksum is used to ensure all the values sent are correct. Checksum is the sum of all the previous values masked to 7 bits.

38 - Drive M1 With Signed Speed And Acceleration

Drive M1 with a signed speed and acceleration value. The sign indicates which direction the motor will run. The acceleration values are not signed. This command is used to drive the motor by quad pulses per second and using an acceleration value for ramping. Different quadrature encoders will have different rates at which they generate the incoming pulses. The values used will differ from one encoder to another. Once a value is sent the motor will begin to accelerate incrementally until the rate defined is reached. The command syntax:

```
Send: [Address, CMD, Accel(4 Bytes), Qspeed(4 Bytes), Checksum]
```

4 Bytes (long) are used to express the pulses per second. Quadrature encoders send 4 pulses per tick. So 1000 ticks would be counted as 4000 pulses. The acceleration is measured in speed per second. An acceleration value of 12,000 QPPS with a speed of 12,000 QPPS would accelerate a motor from 0 to 12,000 QPPS in 1 second. Another example would be an acceleration value of 24,000 QPPS and a speed value of 12,000 QPPS would accelerate the motor to 12,000 QPPS in 0.5 seconds.

Figure 4 -- Example Command for RoboClaw

The RoboClaw built in PID command however would not function. When the command was sent to the RoboClaw, no output would occur. This generally means the command was sent incorrectly, but the values sent were following the user manual. The team is still in communication with the manufacturer to ensure the command is being sent correctly and to determine any other possible bugs with the programming. Since the built in PID commands did not function as expected, the secondary option was to create the PID control on the Raspberry Pi. This is inherently an inferior solution since all decisions are made at the speed of the communication protocols can support rather than the speed of a microprocessor which operates at speed 100 to 1000 faster than the communication protocols can support. Additionally, it places a much larger processor load on the Raspberry Pi. While the solution was not ideal, the PID control through the Raspberry Pi does work with moderate success. Its operates at around 25-50 Hz, which is about the speed the motor can react, which is not ideal. Since the solution is not ideal, the team is still attempting to work with the manufacturer to debug the onboard PID code.

4.1.3 – Potentiometer Arm Model

As part of our goal to make a more intuitive and user friendly interface, the user input for the robotic arm is a potentiometer model the user can physically move. The potentiometer model, shown in **Fig. X** , has a high resolution potentiometer at each joint which changes resistance as the model is moved. The potentiometers are independently connected to a DragonBoard microprocessor which has multiple analog to digital converters (ADC) pins. This is necessary to convert the potentiometer to a digital signal which can be sent from the command station to the rover and be understood by the PID control. The ADC on the DragonBoard is 10 bits, meaning that the potentiometer which will be supplied 5V from the microprocessor, can have a digital value ranging from 0-1024 corresponding to 0-5V through the potentiometer. This digital value is then mapped to the encoder position of the motor, and the desired encoder position is sent to the PID control. The DragonBoard also has serial communication commands and can send the potentiometer values through a USB terminal to the command computer which reads the values in the GUI and transmits the values.

4.2 – Gripper Design

4.2.1 – Overview

The sample extraction system needs a mechanical component to grasp rock samples at the end effector of the robotic manipulator. According to the competition guidelines, rock samples will vary in sizes and masses ranging from 2 - 8 cm in diameter and 20 - 150 g mass. The mechanism must be capable of acquiring largest rock sample discretely as points will be awarded for the selection of specific rocks. The component must be versatile in rock acquisition, and strong enough to endure competition environment.

4.2.2 – Core performance

The core system of this gripper is based on two actuated four bar mechanism. This mechanism will effectively consist of grounded crank and rocker links connected by a coupler link. The coupler's motion is used for actual grasping, and this motion made up the pincher component of the design which provides the precise capturing motion.

The end of the gripper is a wide area arc attachment lined with elastic webbings designed to conform to the orientation and shape of the sample to provide increased tractions between the gripper and the sample without the need of increased torque.

4.2.3 – Preliminary design

A model of the gripper design is pictured in the figure below. The four bar mechanisms are planned to comprise of cardboard links for the process of rapid prototyping. The coupler-arc are connected to a base enclosure via mirrored mechanisms on each side of the enclosure. The Driving Four bar mechanisms are driven by one servos, power transmitted via the spur gears connected to the crank.

In search for the elastic material for the gripper design, several important parameters were established to evaluate the material. First, the material must have great elasticity but also durable. This is to ensure the reliability and repeatability of sample extraction process. Second, the material must exhibit strong cold temperature tolerance. This is due to the extreme cold climate of the surface of Mars. First aid tapes were first equipped on the initial prototype as shown in figure.

4.2.4 – Final design

The final gripper design material is chosen to be ABS plastic instead of aluminum since the specific strength of the plastic was tested to be satisfactory. The design is powered by a Pololu 6V gear micromotor coupled to a spur gear train to transmit the output torque throughout. Two 1:1 spur gear arrangement couples the two drive cranks together to achieve the desired counter-rotating motion. The pincher two four bar mechanisms were improved to ensure the force angle of the crank to coupler remains sufficient when the gripper is fully closed.

The compliant material for the end effector attachments have been carefully considered due to the potentially extreme cold climate of outer planets like Mars. The proposed material, polydimethylsiloxane (silicone rubber), is known for its elasticity and ability to withstand large temperature variation ranging from -120C to 300C while maintaining the essential elastic properties. It is also important to note that the manufacturing process in which the silicone rubber polymer is synthesized and shaped determines the various temperature ranges the material can endure. Based on the low cost and abundant availability, the team will be using low grade silicone rubber sheets; however, selection of higher grade material would make the design suitable for extreme climates.

4.3 – Gripper Concepts

4.3.1 – Pitcher

The common pincer style claws attempt to mimic the way relatively small objects are most typically secured by the human hand. These claws generally consist of prongs or fingers which move towards each other to capture an object and prevent further motion through continuous application of force. This style of claw is good at picking up discreet objects but requires a relatively high level of precision from the manipulator arm module due to the small end surface area of the pincher tip.

4.3.2 – Scooper

The scooper design employs the term 'scoop' quite literately in its name. Most scooper design are based on the idea of using the geometry (generally a concave surface) and the direction of gravity to capture and retain an object or substance. Scooper are generally used to pick up large quantities of a material and are not ideal for acquisition of discreet objects for the reason that it often picks up unwanted materials along. Although Scooper can be operated successfully with much less precision than pincer style claws, it lacks the precision the pincher possesses.

4.3.3 – Universal jamming gripper

This gripper concept utilizes not so common technique of picking up objects. Instead of having rigid moving members which grasp or scoop and object, this universal gripper conforms to the object in which it is grasping. The gripper consists of an ordinary latex party balloon filled with ground coffee. When the coffee-filled balloon is pressed onto the desired object to be picked up, the balloon and coffee conform to the object. At this point, a vacuum pump evacuates air from the balloon, solidifying the balloon, and thus gripping the object. This solidification is due to a "jamming transition" experienced by the coffee. When the air is vacated from the coffee filled balloon, the particulates of the coffee are pressed against each other causing them to resist slipping by one another or causing "jamming."

This concept is very beneficial in that the gripper will not have to orient itself to the object being picked up, but rather simply press against it. Conventional grippers require the target object to be oriented a certain way between the contact points to be picked up. This concept has a major flaws when it comes to implementation in the competition, as it requires strong suctions from an air compressor and additional powers from the battery making it almost impossible to implement on a rover platform with limited battery power and weight restriction.

5.0 System Components

5.1 – Communication

The figure below is a schematic illustrating the overall communications system. Mission Control, consisting of a computer with internet connection, communicates with the rover over the Verizon 4G Network. Mission Control makes up the first Local Area Network (LAN). The rover has a router that communicates with the Raspberry Pi mini computers that activate the robots legs and arm, and two IP (Internet Protocol) cameras that send a video feed, making the rover a second LAN. The router also has a Verizon 4G modem attached to its USB port. The Mission Control communicates with the on-board rover over the Verizon 4G Network using a service called no-ip.com. The no-ip.com service offers a url that access a pool of IP addresses. This allows the router to have dynamic IP addressing, without having the user to lose connection due to IP address changes, or having to constantly "sniff" or re-connect.

ROVER COMUNICATION DESIGN OVERVIEW

On board the rover, a router with a Verizon 4G LTE USB Card installed will link the rover's components to mission control. This router will be linked with the mission control rover over a commercial WAN. Last year's design relied solely on the Verizon 3G Network, however this lead to interruptions in

communication as the network faced problems on the day of the competition. This year, the team upgraded to the 4G network, which is up to 8 times faster than the 3G network. This will create less lag in the design and improve performance.

The vision system will include an IP camera that sits atop the camera mast which is able to pan and tilt. A second IP camera is mounted to the Sample Extraction Module to ensure an optimal viewing angle for the user controlling the arm. The cameras are ideal for the design as they have specific IP addresses, making links to the router easier to implement. Hypertext Transfer Protocol (HTTP) will be used to feed the videos to the on-rover router.

Secure Shell (SSH) protocol will be used to communicate with the Raspberry Pi mini-computers. The Raspberry Pis are programmed with the C programming language and have a Linux-based Operating System installed. The SSH server and client are readily available on many Linux distributions making SSH protocol a viable solution. Last year's team discovered most wireless broadband carriers prevent incoming SSH connections, but do permit outgoing SSH connections. Thus, a script was written that, upon booting the system, connects the Raspberry Pi to a Mission Control port, allowing the server to reverse-SSH back into the Raspberry Pi allowing the user to communicate with the mini-computer.

To communicate between different devices on the rover (i.e. between the leg motors and the mini computers) Serial Peripheral Interface (SPI) wires were integrated into the design. The SPI protocol transmits information in full Duplex, meaning it can transfer a byte and read another byte being sent all simultaneously. This property greatly improves the speed of the design.

6.0 Locomotion and Control

6.1 – Overview

Figure 5 -- C Shaped Leg Design

In order for the rover to be useful in collecting samples, it must contain a fluid locomotion system as well as a user-friendly control interface. The legged design which is being used for this rover requires precise control of each leg, which is established through extensive programming. While moving forward is simple with a wheeled platform, it requires very precise algorithms, specifically the Buehler Clock, in order to achieve the same effect in the legged design. This extra work will prove to be worthwhile when the benefits of the legged platform are exposed on the rock field.

6.2 – Project Scope

The goal for this year's group is to improve on the locomotion system and controls system which was established previously. The rover was able to traverse different terrains fine, but it was very slow and limited in its range. The rover previously required a command from a complicated GUI for each input. The command would instruct the robot on which direction to move, how fast to move, and how many steps to take. After each command, the rover would return to standing position and wait for the next input.

Although this was sufficient for a moving platform, it was by no means user friendly or fast. This year, three main focuses were established and were implemented. First, user interface will be improved to be more forgiving for the controller of the robot. Hopefully, a complete stranger will be able to take the machine and control it with minimal instruction. This was established through an XBOX controller and will be explained thoroughly below. Secondly, multiple gaits will be added to the rover to create more options for locomotion. Examples are Turn While Walking and Stair Climbing. Finally, existing commands will be improved in multiple ways. Two key points here are to ensure that the command can be run continuously so that the rover will continue to move until instructed to stop and to enable dynamic switching between gaits. These changes will allow for the rover to be easier to control and faster in its operation.

6.2 – New Locomotion Gaits

6.2.1 – Existing Locomotion Gaits

The platform which was created last year was able to move in a very limited range of motion. The functions which were available included calibrate, forward walk, backward walk, turn-in-place, lay down, stand, and hill climb. Although these gaits combined to allow for movement to any general point, it struggled with specific locations. The c-shape legs are too big to do precise movements with these functions. For example, whenever one step is taken in the left or right direction, the rover moves approximately 30°. If a desired turn angle of 15° is desired, the rover would not easily be able to accomplish the task. Also, the need to stop moving forward while changing direction made for slow travel times. These problems are the motivation for the new turn-while-walking function.

The locomotion functions each had individual functions and thus had different parameters that must be entered into the function to run. The lay down and stand functions were void and thus had no parameters. The Hill Gait function had a number of steps parameter and an RPM parameter. This two told the rover how many leg rotations to make and how fast to make them up a hill. Finally, the locomotion functions will all take in similar parameters. The parameter list contains RPM, direction (Forward, Backward, Left, and Right), and number of steps.

6.2.2 Buehler Clock

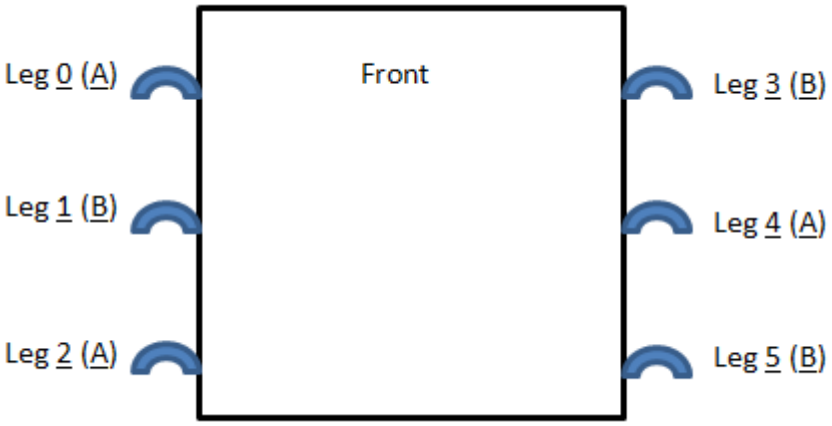


Figure 6 -- Aerial View of the Robot to Display the Leg's Labels and Their Respective Groups

When the robot walks in a straight line, the 0, 2, and 4 legs will be coupled together (call them set A), sharing the same movements. The 1, 3, and 5 legs will also be coupled (set B), and they will move at exactly 180° phase difference from set A. To be precise, this means that while one set is pointing directly downward, in its peak contact with the ground, the other set will be directly upright, at its highest point.

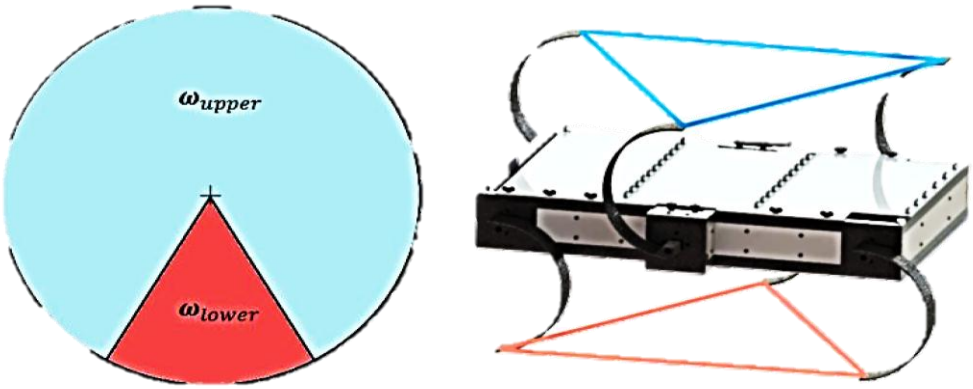


Figure 7 -- Buehler Clock Graph for both Set of Legs (Red = A, Blue = B)

To understand the locomotion further, one must understand the Buehler clock. The Buehler clock describes the relationship between the speed of the leg and its location in its rotation. When any given set of legs are on the ground, they must move slower than when they are in the air, so that the other legs can “catch” the robot right as they are leaving the ground stage. Figure 13 shows this relationship. The slope of the lines describes the speed of the legs rotation, the y axis describes the location in the legs rotation, and the x axis describes time. Notice that the legs change speeds at T/4 and (3T)/4. Notice that in this image, both sets of legs start and end at 0 and 2π respectively.

6.2.3 Turn While Walking

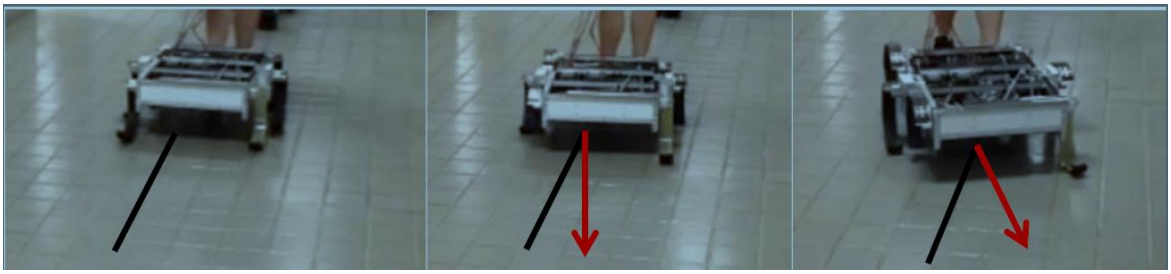


Figure 8 -- Turn While Walking

Now that walk is understood, turn while walking must be implemented. One's immediate response to implementing turn while walking is to increase the speed of one side of the legs and thus create a turn. This design was considered but quickly failed when it was hypothesized and proven that the rover would simply fall over, since the legs would lose their coupling over time. The next idea was to adjust the phase at which the left legs differ from the right legs. For example, put leg 1 20° ahead of legs 3 and 5, while simultaneously putting leg 4 20° behind legs 0 and 2. This will cause the left legs (the ones that are ahead) to hit the ground slightly before the right legs leave the ground. For the second that the legs are together on the floor, there will be a slight turning motion to the right, and then the robot will continue to move forward once the left legs catch up (at which point the other set of legs will have lifted into the air).

In order to ensure that even the most subtle angles could be reached through the turn-while-walking function, a separate input parameter was introduced, called "angles". This variable controls the phase between the legs, and ranges on a scale from 0 – 10, where 10 is the highest allowable phase difference and 0 is the lowest. In practical terms, 10 makes the difference between the legs approximately 5000 motor encoder values while 0 represents the normal forward walking function. When this value is increased, the time that the set of legs are on the ground together will increase and thus make for a wider turn.

6.2.4 Stair Climbing

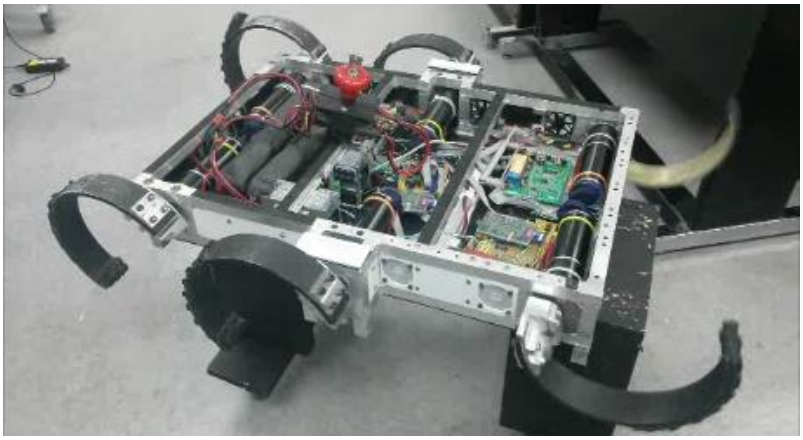


Figure 9 -- Starting Position for Stair Climb Gait

Although there are no stairs on Mars (yet), a stair climbing gait was implemented in order to show the power of a legged platform over a wheeled platform. In order to implement the stair climb, the legs had to be grouped into pairs. The groupings were the “Back” pair, the “Middle” pair, and the “Front” pair. These three groupings were controlled in three phases, where each leg moved to a new position at a new speed.

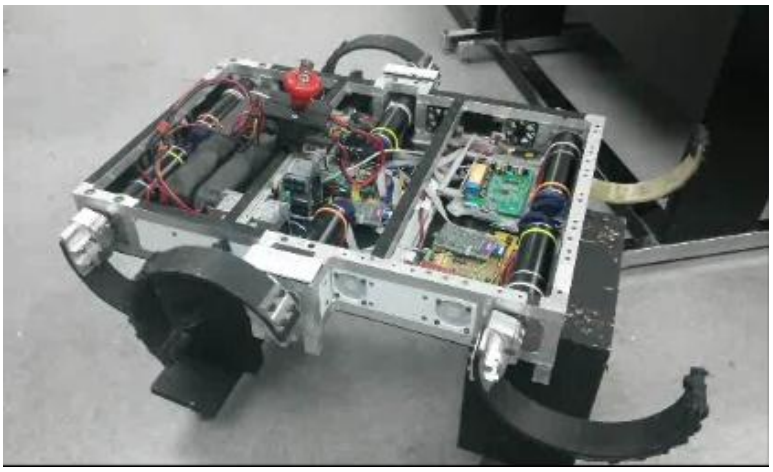


Figure 10 -- First Phase of Stair Climb Gait

In the first stage, the rover starts in a new position on the stairs. Note that the “Back” and “Middle” pairs are in the same position, whereas the “Front” pair is in its own position. The “Back” legs then begin movement to rest on top of the next step in the stair case. While these legs are in the air, moving toward the next step, the other two sets hold the rover in place.

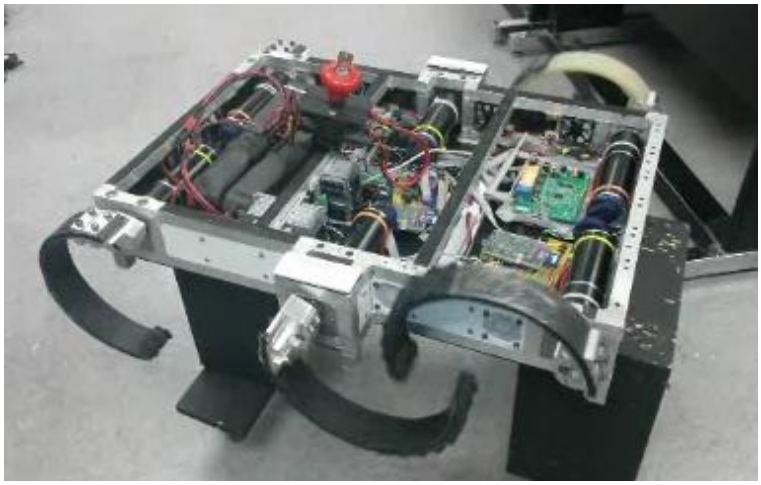


Figure 11 -- Second Phase

In the second stage, the “Back” pair begins to push the rover up the stairs slowly. This holds the rover in place for the “Middle” set to move to the next step. The “Front” legs are also still on the ground. They will begin to move slowly, pushing the rover up with the “Back” set.

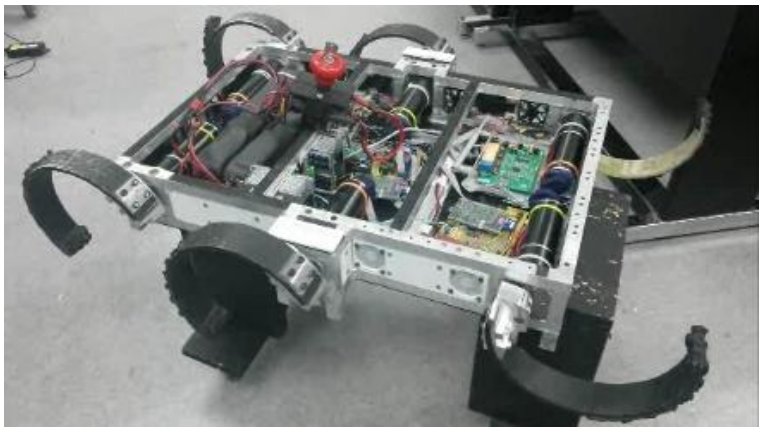


Figure 12 -- Final Phase Returning to Start Position

In the third and final stage, the “Back legs continue to slowly push the rover up the stairs. The “Middle” legs have reached the next step and now begin to slowly push the rover up the stairs with the “Back” legs. These two sets will hold the rover while the “Front” set quickly rotates and catches on to the next step. At the end of this stage, the legs will be in the same position that they were in before the step climb, ready for the next step. Notice that at all points, there are four legs holding the rover on the stairwell to prevent slipping or falling, making the function extremely consistent.

6.3 User Interface Improvements

6.3.1 Existing User Interface

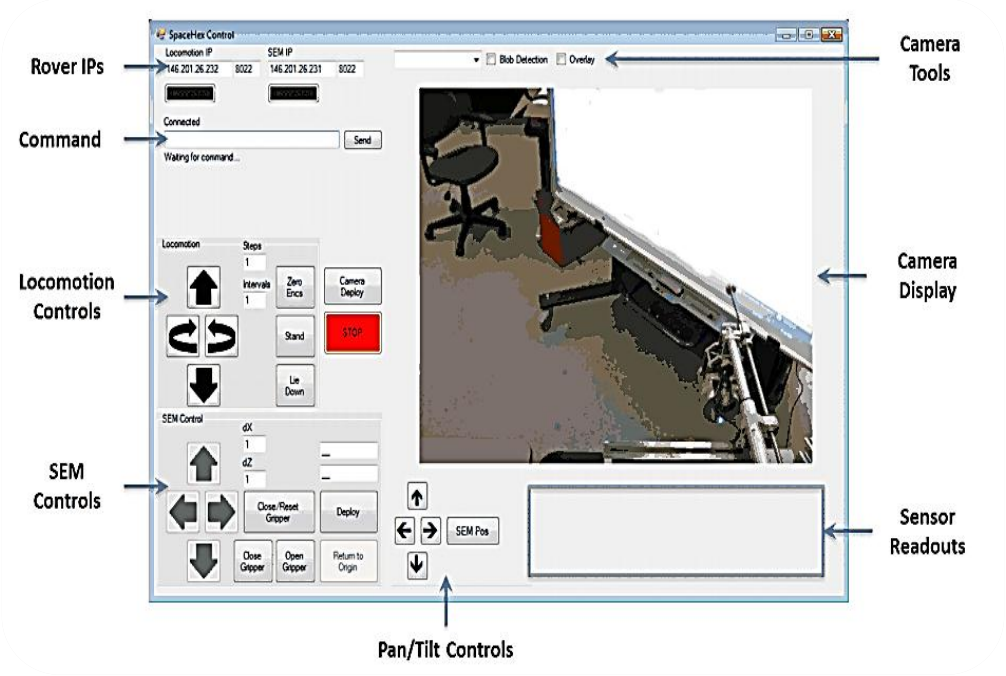


Figure 13 -- GUI

The existing user interface took the form of a Graphical User Interface. The GUI came equipped with the video feed, the temperature sensor readings for the motors, status indicators for the network connection, arm controls functions, and the necessary locomotion controls. To send a command, the number of steps and speed must be typed into their appropriate boxes. Then the button for the direction must be pressed. The command will appear in the command text box. If the send button is pressed, the command is relayed over the network and then the robot will begin motion.

Figure 14 -- Madcatz XBOX360 Controller

The GUI provided a lot of benefits, but the locomotion control was very slow and not user friendly. The large camera feed and the status boxes which came with the GUI were preserved and will be re-used in correspondence with a new control interface. The new interface will be an XBOX

controller, which will allow for quick, consistent locomotion control, all on a very comfortable and user friendly environment.

6.3.2 – XPadder

Figure 15 -- XPadder Window

XPadder is a program which was necessary to eliminate the GUI and transfer the control to an XBOX controller. It simulates the keyboard and mouse using a controller which allows for key mappings from the keyboard to the controller. A first prototype was created and tested. In this prototype, hardcoded commands were sent to the command line by pushing a button on the XBOX controller. While this allowed for control through a more user friendly device, it was not a sufficient design. The functions were limited by the number of commands which could be mapped to the controller, approximately 22. This means that if one button was used for “Forward Walk, 5 steps, 20 RPM” then it could not be altered in any way without accessing the software. In addition, the function inputs from the XBOX controller were slow and did not provide much of an increase in performance. XPadder is still used in the final design in order to operate the rover through the XBOX controller.

6.3.3 – SDL & Curses Library

In order to allow for fluid movement which depended on individual key presses as opposed to commands, the SDL library was explored and implemented. SDL is a library designed to provide low level access to a keyboard. It is a software used in many computer games to control individual models in movement and interaction with their environments. This library would allow for individual key presses to be read into the program directly and interact with the code accordingly. The SDL library was downloaded and installed onto the locomotion Raspberry Pi and tested with sample programs. After multiple tests, the SDL environment proved to be counterintuitive and ineffective.

Curses is a terminal control library for Unix-like systems. The library enables the construction of text user interface, meaning that keyboard inputs could be read by a program and affect a function directly. The library provides necessary functions, specifically cbreak(void), halfdelay(int), flushinp(void), and getch(void).The cbreak(void) function allows for input without pressing the enter key, halfdelay(int) tells the getch(void) function how long to wait before reading a default value for its return value,

flushinp(void) clears the input buffer which allows for the latest command to be read, and getch(void) waits for a user input and returns that input to a character value.

6.3.4 – XBOX Controller Mapping

Figure 16 -- Xpadder Window Used

With the library instantiated, the Madcatz XBOX360 controller could now be mapped to individual keyboard buttons. When the joystick is pushed forward, a string of 'w's appear on the command line until the joystick is released. This concept applies to every button on the joystick and is mapped by the following table.

Table 2 -- Controller to Function Mapping

Button Pressed On Controller	Function
Left Joystick-Up	Forward Walk
Left Joystick-Down	Backwards Walk
Left Joystick-Left	Turn In Place - Left
Left Joystick-Right	Turn In Place - Right
Right Joystick-Left	Turn While Walking - Left
Right Joystick-Right	Turn While Walking - Right
'A'	Sit/Stand
'Y'	Calibrate Legs
Left Bumper	Decrease 'angles' by 1
Right Bumper	Increase 'angles' by 1
Left Trigger	Decrease RPM by 5
Right Trigger	Increase RPM by 5

6.4 – Function Improvements

6.4.1 – Motivation

A large problem with the walk functions incorporated in the rover's previous design is the need to guess the correct distance to an object so that the correct number of steps could be input. There was little room for error and thus, to be precise, each command took up to 30 second each to be input to the

rover. This caused for extremely slow movement and a high amount of time wasted trying to get into position to pick up objects. A locomotion system which does not require step inputs but simply moves the rover continuously until told to stop would reduce this guessing and increase productivity.

6.4.2 – Continuous Movement

The first issue which must be tackled in the programming is the slow nature of a robot which takes a predetermined number of steps. The goal is to make a machine which continues to move forward until the joystick is released, much like a wheeled machine would.

The first programming prototype was simple. The number of steps was reduced to one in each function and thrown in a continuous loop which ended when the appropriate button was released on the joystick. The theory was correct, but the implementation was faltering. The walk function would not exit upon release of the joystick. It was realized that there was an input buffer which was holding values for the character which ended the while loop. This buffer continued to fill and provided values to the while loop one by one. This resulted in unpredictable 'walk' distances and proved even more difficult to predict than the step-counting method used prior.

In order to correct this, a new curses method was discovered. flushinp(void) is a library function which clears this input upon command. This made it so the while loop only read in the most recent character put in to check, which ensured that no fault commands were being input to the loop. This method worked for very specific situations. As planned, the legs would continue to rotate until the button is released. There were errors though. When the step ended and another command was input, the legs would sometimes have incorrect timing variables and result in an error. It was discovered that when the steps ended in an odd step, the legs ended at 180° off phase of the original start position for the walk command. This phase difference threw off the necessary timing variables which controlled where the legs should be and consistently resulted in an error. Even steps also had errors, though they were inconsistent and sometimes did not occur.

6.4.3 – Dynamic Gait Switching

The Buehler clock functions which executed the walk commands were further examined. There were two timing variables which created the variable 'BuehlerPhase' which kept track of where all six legs should be in their rotations. These two timing variables were re-instantiated in each function execution, which solved the inconsistent errors in even steps. Although the even steps were working, in order to allow for the rover to stop and continue a walking motion, the BuehlerPhase had to be changed to account for odd steps and even steps accordingly. First the main code was changed. An oddsteps variable was introduced and kept track of whether a given step was odd or even. This oddsteps variable was then input as a new parameter into the locomotion gaits. Inside the Buehler function, an if statement was introduced to check for oddsteps. If the function was executing an even step no further changes were necessary. The even steps were correct as is. If the function was executing an odd step, the BuehlerPhase was changed to $(1.5 * BuehlerPhase)$. This resulted in the BuehlerPhase properly representing to the legs that they should be at 180° off of their original starting position and thus resulted in a smooth walking function for odd steps.

This code was inserted into all locomotion functions and worked across each function. In order to implement dynamic switching between them, a new main file was written. This new file worked with an infinite loop which contained a large switch statement. The switch statement contained case statements for the following buttons and their corresponding commands.

Table 3 -- Case Conditions and Their Corresponding Commands

Case Statement	Function
'w'	Forward Walk
's'	Backwards Walk
'a'	Turn In Place - Left
'd'	Turn In Place - Right
'q'	Turn While Walking - Left
'e'	Turn While Walking - Right
'l'	Sit
'c'	Calibrate Legs
'i'	Decrease 'angles' by 1
'k'	Increase 'angles' by 1
'u'	Decrease RPM by 5
'j'	Increase RPM by 5
default	Stand And Hold

This switch statement allowed internally for dynamic switching. Whenever no buttons were being pressed, the rover would return its legs back to the standing position and enter a while loop which held the rover in place until a button was pressed. In this state, the rover would do nothing. Whenever a button was pressed, the legs would move into the starting walk position and enter their appropriate switch statement. The legs would then enter a loop which continued until the button was released. This loop would contain the walk function with the corresponding direction, oddsteps value, angles value, and RPM. When another button is pressed, the loop will exit and the switch statement will check the value. If the value is one of the possible case conditions, it will enter that condition and enter that while loop, flowing with little delay from one function to the other. This loop will continue until that button is released. When all buttons are released, the rover goes into the standing position and holds until a new command is entered.

Appendix A – Competition Overview

A. Competition Rules and Requirements

Competition Summary

The NASA Rascal Robo-Ops competition's goal is to create new and innovative solutions toward the development of rover's capable of traversing mars which is simulated at the NASA Space Center. The rover must be controlled remotely over a commercially available wireless network, be able to navigate obstacles, and be able to identify and acquire brightly colored rock samples. Teams also must engage in public outreach to foster interest in space exploration and robotic development, utilizing social media and community events.

Requirements for 2013

The process to become a participant is as follows. First, a notice of intent form is submitted to the competition stewards. Next, an proposal documents is submitted (due December 8, 2013) which must be no more than 8 pages. From the proposals submitted, 8 teams will be selected (notified December 20, 2013), which nets the team a $5,000 grant to construct the proposed rover.

Rover specs for competition trim

In the rover's "Stowed configuration", meaning with all peripherals retracted, the rover must not exceed dimensions 1m x 1m x0.5m. The maximum mass (without payload) must not exceed 45kg, or else points will be deducted. No internal combustion engines are allowed, and the rover must be water-proof.

Rover performance and capability required

The rover must be capable of traversing obstacles at least 10cm tall, negotiate +/- 33% grades, and traverse level sand surfaces for at least 20 feet of distance. The areas of the JSC Rock Yard to be included in the competition are the Rock Field, Lunar craters, Sand Dunes, and the Mars Hill. The rover must selectively acquire at least five irregularly shaped rocks while traversing the JSC Rock Yard. The rocks are outlined as having diameters from 2 - 8 cm, masses from 20 - 150 gm, and be of different colors each corresponding to a point value. The rover must store and carry these rocks throughout the course. The JSC Rock Yard and the rocks of interest can be seen in Figure 8, below.

Controls and Communications Requirement

As stated before, the rover must be remotely controlled from the team's home campus over a commercial cellular data network (ie. via wireless broadband card). Rover data must be sent from the rover itself to operators and spectators online. This data is required to consist of live video feed and some rover telemetry. The video feed must be capable of distinguishing color (rock samples), and must be recorded and posted on the team's website.

Requirements for 2014

After being selected to compete in the 2014 RASC-AL Robo-Ops competition, the team will be required to continuously document and broadcast rover development progress. These reports are required to outline how the team has met the aforementioned "milestones" of the competition. Next, each report will be introduced along with the milestones the team is expected to cover in that report.

Mid-Project Review Report + Video – due March 15, 2014

The purpose of this report is to display to the competition stewards that a team is on schedule to completing a rover capable of satisfying all design and performance requirements. This report consists of a five-page written portion and a YouTube video. The whole report must demonstrate the rover's present functionality and chronicle what is yet to come. Team must outline where they are with the project and how confident they are that their rover will be completed. If the stewards feel that a team's report does not show this, they will be required to do a live follow-up web chat with the stewards to redeem themselves. Only after the stewards are satisfied with a team's progress will be awarded an additional $5,000 grant money be awarded to a team.

Appendix B – Design Concepts

B1.0 -- Proposed Designs

B1.1 – Arm Concepts

B1.1.1 – 2 DOF Arm

The first arm concept is to improve upon last year's design, a planar two degree of freedom arm. The design would need to be reduced and a more advanced wrist would be developed with the arm. The rover can have very precise control over it Z position making this arm simple and easy.

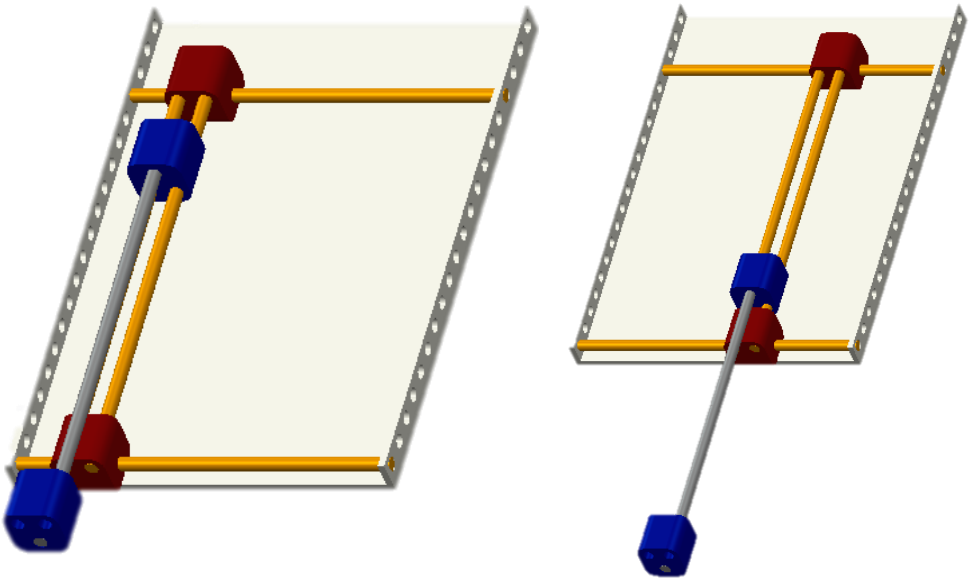


Figure 17 -- 2 DOF Arm Design

The advantages of this design were described by last year's team, but will be explored again for comparison. The system requires the control of only 2 motors, and the control over the rover itself. The thought was also that by having fewer motors and systems, the overall weight would be reduced, and would impact the cost and reliability of the system. The final thought was that the arm remaining close to the body would keep the platform more stable.

The desired advantages turned out to be some of the arms shortcomings. The control did turn out to be significantly easier, and allowed the team to develop a click to grab routine where the user could click on the GUI and the robot would move the arm into position to pick up the sample. This routine was not tested in competition because an encoder failed the night before competition.

The from the hardware desire, the arm could not fit into the competition dimensions with the arm remaining planar. A innovative rail follower was devised, but forced the arm into the air, which then hurt the center of gravity and the deployment speed. Finally, to achieve the reach desired, a large linear actuator was needed and then forced the rest of the system to be bigger and heavier than originally thought.

B.1.2 – 3 DOF Arm with 1 Planar Joint

Three degree of freedom arm concepts were then explored. The first of these was this arm design which has 2 revolute joints and a planar joint to extend the arm. This design would be similar to WPI's design with the addition of a linear actuator instead of a rigid arm.

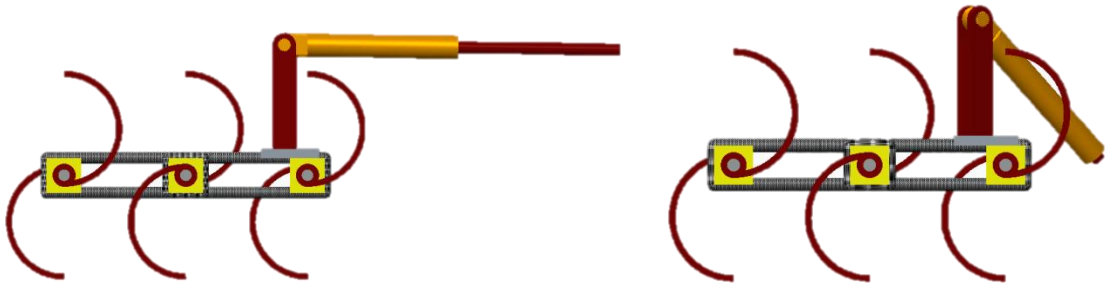


Figure 18 -- 3 Degree of Freedom with 1 Planar Joint

The advantages to this design is it requires less control than an arm like Caltech's or Maryland's, but still give the necessary degrees of freedom to reach most rocks without the rover needing to move. The design could still stay close to the body and keep the center of gravity low.

The disadvantages for this design include the weight of the system which would likely by higher to include an linear actuator. The revolute joint would need to be designed to withstand the weight and the motors controlling the degrees of freedom would need to be stronger.

B.1.3 – 3 DOF Arm with All Revolute Joints

The final design explored is a three degree of freedom arm with all revolute joints. The design would consist of a 2 degree of freedom "shoulder" joint, and a 1 degree of freedom "elbow. The current concept is to include a 1 degree of freedom wrist, but this would depend on the gripper concept chosen.

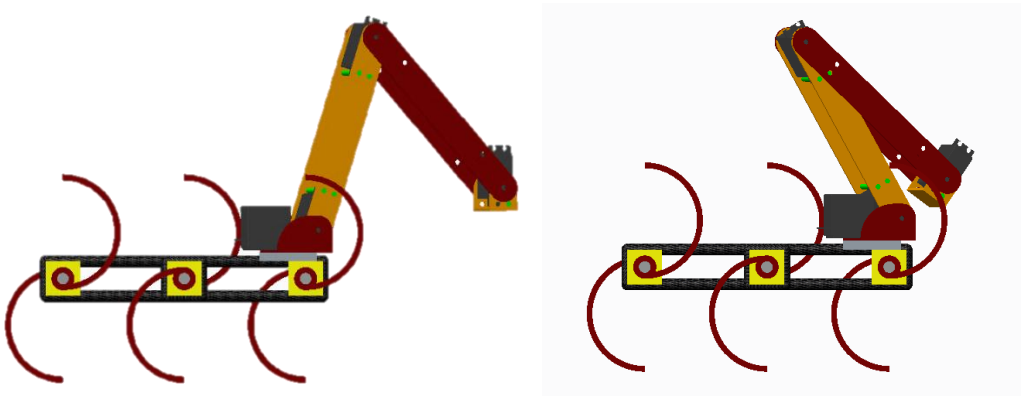


Figure 19 -- 3 Degree of Freedom Arm with all Revolute Joints

The advantages of this concept is the flexibility in it use and in its flexibility in placement on the rover. It could be placed on top of or in front of the rover without any difference in functionality. Additionally, the storage compartment could be placed in any location which is convenient. Finally, it can be very compact and utilize very lightweight materials and still be strong enough and have the desired range.

The disadvantage is the advanced control necessary to utilize such a design. For it to be user friendly to use, the arm would need to have some form of automation, or at least control mapped from the x y z frame to the motor.

B.2 Gripper Concepts

B.2.1 – Scooper Design

The first design discussed was a scooper designed gripper. Scoop is defined by Merriam Webster as "something that is shaped like a bowl or bucket and used to pick up and move things". Essentially, it uses a distinctive shape and gravity to collect large quantities of material. It does not require as much precision as the pincer design which we will talk through next.

Figure 20 -- Scooper Design in Action

B.2.2 – Pincer/ Finger Designs

Pincer/finger grippers use the same technique as the human had to grab and secure small objects. The pincer normally has two fingers which can move in toward an object or release away from the object, and uses a constant force on the object to keep it secured. It is very good at picking up discrete objects, but needs to be placed precisely for the object to be secured.

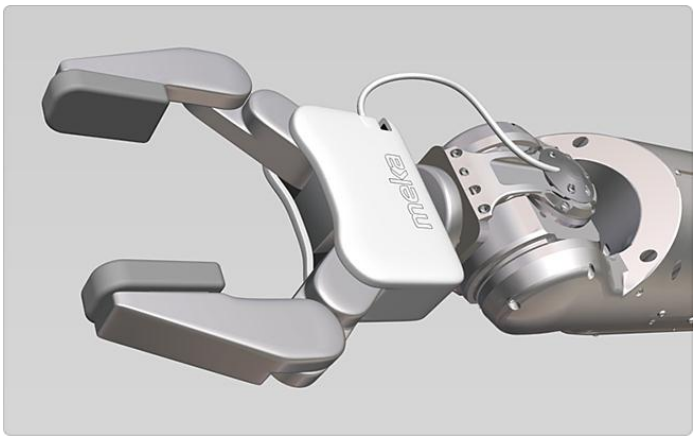


Figure 21 -- CAD Model of a Pincer Design

B.2.3 – Universal Gripper

Some more recent gripper research has gone into attempting to develop a gripper which has the ability to easily grab a discrete object. This has led to the development of universal grippers which utilize a conformable material to grasp an object. The gripper shown below in **Figure #** consists of a balloon filled with ground coffee. The balloon is pressed onto the object desired, and then a vacuum pump evacuates air from the balloon, causing the coffee grounds to jam against each other, forming a 'rigid' gripper.

Figure 22 -- Universal Gripper Design

B.2.4 – Compliant Gripper Mechanism

The combinations of the previous designs lead to very interesting concepts. The first of these is compliant fingers which combines the universal gripper with the pincer gripper. Compliant finger grippers need to be precise in their implementation, but require less precision than rigid fingers. They can reach and grab discrete objects in confined spaces, however the precision required to use them is still very high. There are a few finger ideas on the board, 2 pronged, which is small and can reach most everything but needs to squeeze the rock and get a good grip as it's only touching the rock in two places, or 3 pronged, which would hold the rock very stable but may not be able to get access to 3 different sides of the rock. The FESTO Fin Adaptive Finger (right) has gained our curiosity as its shape conforms to object it is grabbing and is delicate enough to pick up an egg. The second idea for fingers are rake-like, skinny tendrils on the fingers allow the fingers to close around the rocks shape to have greater contact area.

Figure 23 -- FESTO Fin Adaptive Fingers

By combining the universal gripper, with a scoop design, along with some elements of the pincer, you can come up with a mesh gripper. The mesh gripper consists of two clamps that have a mesh screen in their center, then became an elastic mesh grip which will be more versatile and have a higher friction coefficient. With the bottom support was removed to create an upside-down U structure so we can get the mesh as close to the ground as possible. This mesh gripper clamps onto the rock and it conforms to the unique shape of the rock.

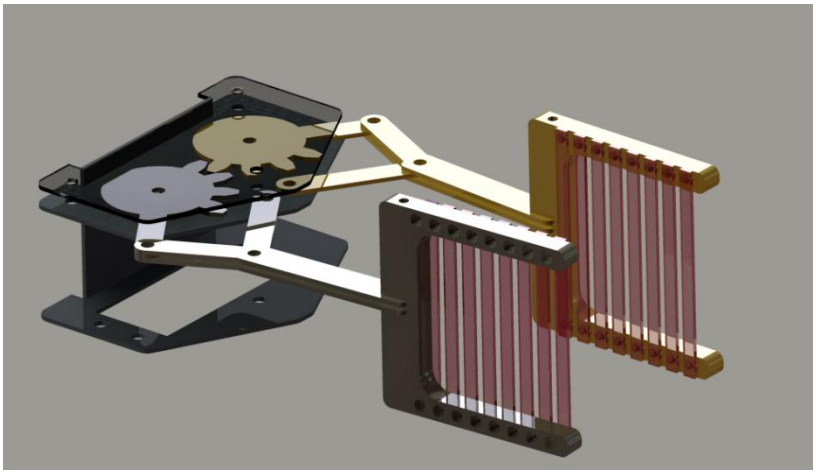


Figure 24 -- Cad of Mesh Gripper Design

B.3 Communications Concepts

Last year's design utilized Verizon 3G coverage to communicate with the platform. This was advantageous considering the fact that a tower is very near to the competition site. A "mission control" center was established in Tallahassee where the users controlled the rover. The design was simple. Mission control consisted of a user working with the GUI to operate the robot. The GUI would be on a laptop. Using a 3G USB Card, the laptop would communicate with a router on board the rover. The router has a USB port, which is helpful in communications. Last year's operators plugged in a Verizon 3G card in the router as well. The on-rover router would communicate with the Raspberry Pi computers, thus linking the user to the rover.

B.3.1 – Graphical User Interface

The Graphical User Interface (GUI) is a custom computer application which aims to greatly simplify the operation of the rover through integration of information display, in the form of video feeds and sensor data, and rover control. In essence, it gives the user a tool for controlling the rover.

In last year's design, the GUI was written in the C# program language. Below is an image of the objective of the design:

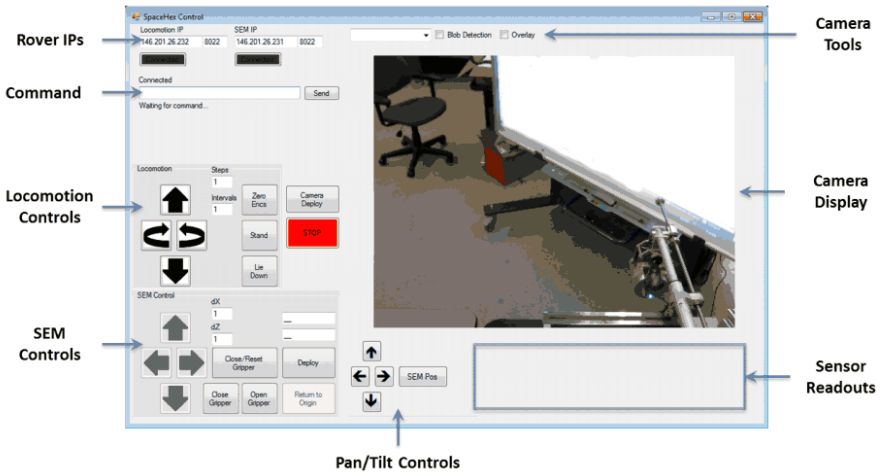


Figure 25 -- Previous Year's GUI

The GUI was operational, but many aspects will be changed in order to make the GUI more user friendly. For one thing, the user would have to input the number of steps and the direction that the rover should proceed. The process was very cumbersome, especially if the user needed the rover to move to a specific spot. As the rover will be competing with other rovers to pick up the most rocks, creating a GUI that allows the user to interact more freely with the rover would be much more efficient. There were also locomotion concerns, as was discussed earlier, as the rover could not turn while walking. So the GUI only has the controls Forward, Reverse, turn-Left, and turn-Right. Our goal is to implement an Xbox or PlayStation controller allowing the user 360 degrees of control, with the ability to change direction while moving. We wish to eliminate the need to enter the number of steps prior to moving. A simple push of the joystick will command the rover to move.

B.3.2 – Communications and Networking

To establish communication between the cameras and computing systems on the rover and the Mission Control server located at the college detailed networking protocol is desired. The figure below displays the design of the network. The blocks on the right represent (top to bottom) the rover arm, locomotion and cameras.

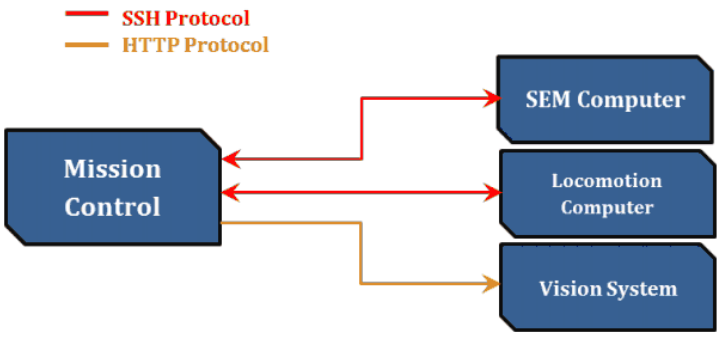


Figure 26 -- Communication Block Diagram

As the above figure shows, communications via SSH (Secure Shell) were established between the on-rover computers and the mission control computer; communication via HTTP (Hypertext Transfer Protocol) was used to link the cameras to mission control. In last year's case, both the mission control computer and networked hardware on the rover are behind NAT (Network Address Translation) firewalls. The NAT firewall prevents all incoming connections to all the devices.

B.3.3 – SEM and Locomotion Computers

In last year's design, the communications system was put together in more haste than what would have been ideal. For one, the mission control operated from a student's apartment. Also, the on-rover router used was a G-type router leading to limited bandwidth. Looking at last year's issues, a lack of bandwidth may have contributed to the issues of last year's team, such as lagging and dropped communications. Additionally, the video feed would be impaired by a low-resolution, which normally would be used in cases where the bandwidth was limited. To counteract these issues, a higher grade router will be used. Last year's router, the TP-Link TL-MR3430 (pictured below) was a fine router for home usage, but a higher grade router would do the project well.

Figure 27 -- Communication Between User and Raspberry Pi

Figure 28 -- Left: Type G Router Right: Type N Router

The router's function was so that the user could communicate with the Raspberry Pi computers. Raspberry Pi's are now 3G compatible, but using a router makes the connection between the user and the Pi computers more secure. As will be discussed below, we plan on using a 4G network for this year's design. The TP-LINK SafeStream TL-ER6020 Gigabit Dual-WAN VPN Router (pictured below) is an ideal router to use with a 4G card. It is a next generation, the N-type. It creates a VPN (Virtual Private Network) thus adding more security by securing an IP address, and preventing interference from other addresses. Additionally, the router is much more powerful, with enough bandwidth to spare.

B.3.4 – Networks

In order to further improve the design, some other minor modifications are necessary. This year's team will make the mission control router the DNS-enabled router. Last year, the team did not take care to make sure only one router was DNS-enabled. Also, some issues arose that were out of the control of the team. The team relied on Verizon's 3G network as there was a tower near the site. Ironically, the 3G network had issues on the day of the competition. This year's team plans to incorporate 4G. While some 3G networks are faster than 4G networks, within a carrier, 4G always trumps 3G. For instance, Verizon 3G is faster than MetroPCS 4G, but Verizon's 4G is faster than its 3G. Verizon's 3G network is actually relatively poor when compared to other network speeds with download and upload speeds 1.05 and 0.75 Megabits per second (Mbps) respectively However, Verizon's 4G network showcases a vast improvement over its predecessor with download and upload speeds of 7.35 and 5.86 Mbps respectively. These speeds are bested only by AT&T's network. Verizon's network is advantageous in part due to the tower nearby the competition site. We plan on using 2 Verizon 4G USB sticks, 1 on the rover, and 1 at mission control.

Figure 29 -- Verizon 4G USB Stick (left) AT&T 4G USB Stick(right)

We are going to strive for as much redundancy with the platform due to some issues that arose last year. The Verizon Network was down that day, much to the team's dismay. Using AT&T's network is an option we are strongly considering in case Verizon's network fails this year. We will use the same communications model as with the 4G, but we will not utilize it unless Verizon's network fails. This practice ensures we are not sending conflicting commands to the

rover, which may cause serious ramifications, such as the robot's malfunction. Since Houston is a major city, using AT&T may very well be the way to go.

B.4 Controls Concept Development

Last year's six legged design had the necessary tools for traversing the competition grounds, but there is still work to be done to allow the rover to move more freely and efficiently through the different terrains. The rover was only able to turn while stationary, and walk directly forwards/backwards.

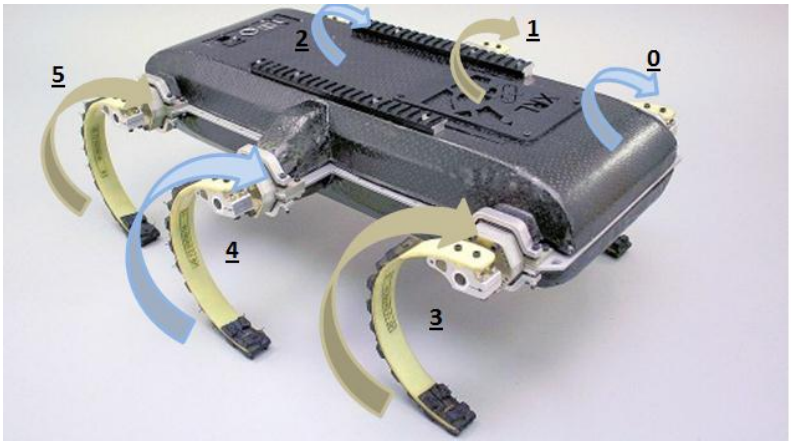


Figure 30 – Proposed six legged device. The legs are labeled (and will be referenced as) 0 through 5.

This year, the team will be attempting to implement a turn while walking function, a turn while climbing function, a more precise turning function, and a "lay-down-nudge" function. These will all be controlled by a wireless controller instead of the GUI interface which was used last year.

B.4.1 – Turn While Walking

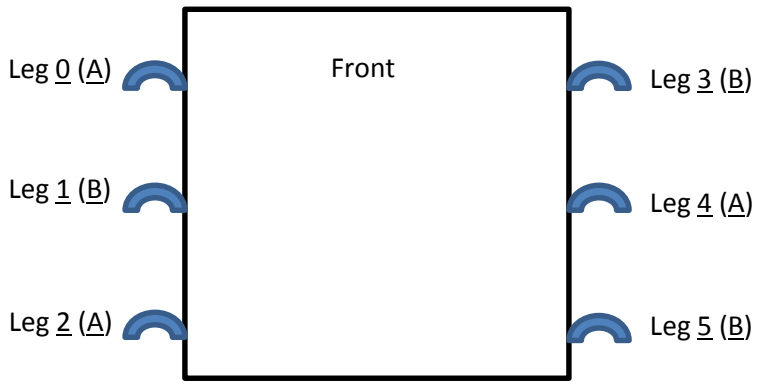


Figure 31 -- Aerial view of the robot to display the leg's labels and their respective groups

When the robot walks in a straight line, the 0, 2, and 4 legs will be coupled together (call them set A), sharing the same movements. The 1, 3, and 5 legs will also be coupled (set B), and they will move at exactly 180 °phase difference from set A. To be precise, this means that while one

set is pointing directly downward, in its peak contact with the ground, the other set will be directly upright, at its highest point.

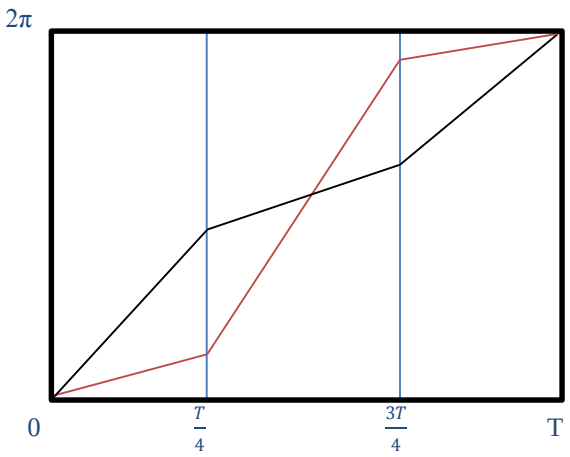


Figure 32 -- Buehler Clock graph for both sets of legs (Red = A, Black = B)

To understand the locomotion further, one must understand the Buehler clock. The Buehler clock describes the relationship between the speed of the leg and its location in its rotation. When any given set of legs are on the ground, they must move slower than when they are in the air, so that the other legs can "catch" the robot right as they are leaving the ground stage. Figure 13 shows this relationship. The slope of the lines describes the speed of the legs rotation, the y axis describes the location in the legs rotation, and the x axis describes time. Notice that the legs change speeds at T/4 and (3T)/4. Notice that in this image, both sets of legs start and end at 0 and 2π respectively.

Now that walk is understood, turn while walking must be implemented. One's immediate response to implementing turn while walking is to increase the speed of one side of the legs and thus create a turn. This design was considered but quickly failed when it was hypothesized and proven that the rover would simply fall over, since the legs would lose their coupling over time. The next idea was to adjust the phase at which the left legs differ from the right legs. For example, put leg 1 20 °ahead of legs 3 and 5, while simultaneously putting leg 4 20 °behind legs 0 and 2. This will cause the left legs (the ones that are ahead) to hit the ground slightly before the right legs leave the ground. For the second that the legs are together on the floor, there will be a slight turning motion to the right, and then the robot will continue to move forward once the left legs catch up (at which point the other set of legs will have lifted into the air).

B.4.2 – Turn While Climbing

Turning while climbing is very similar to turning while walking, but with an extra hurdle. Walking on flat land is simple, if the legs are in phase, they will move forward with no problem. However, on a small hill, the rover has a tendency to turn with the hill as it climbs. To adjust for this, a separate hill climbing function was created and is currently functioning on the rover.

This function must now be added to. Just as with the turn while walking, the team wants to make the robot more agile when on a hill. It seems likely that adjusting the phase just as was done in

the turn while walking will resolve this issue and become extremely helpful in climbing hills quickly.

B.4.3 – Precision Turns

Currently, precision turns are working, but not for slight turns. The reason for this is that the robot is defined to work in "steps". Every time the precision turn function is called, a number of steps must be input to the rover. The robot then takes this many "steps" to that direction, without moving forwards or backwards.

Currently, it takes the robot six steps to completely turn around an approximate 180 °. This means that for each step that the robot is instructed to take; it is currently turning roughly 30 °. This is great for a machine which wants to turn quickly, but extremely non-ideal for one which wants to pick up rocks, and precisely position a gripper to easily pick up those rocks. The turn must be worked on so that it can be more precise for angles lower than 30 °.

To do this, the robot will have to be programmed to be able to take a "half step", or maybe even a "quarter step". Currently, a step is counted every single time a set of legs gets off of the floor, so every time a set of legs makes a full rotation, it is two steps. This means that part of the problem comes from the fact that the legs are long. Downsizing to the smaller machines should serve as a partial solution to the problem, but it might not be enough. On a more core level, however, there are two options to create a precise precision turn. Steps will either be redefined in the current function or a new function will have to be written which can input fractional steps, and thus allow the rover to stop its rotation mid-step.

B.4.4 – "Lay-Down-Nudge" Function

A new idea which is going to be attempted this year is to implement a nudge while laid down function. Last year's team discovered that the most efficient way for the rover to pick up objects is to lay it down and then operate the arm and gripper. This causes a problem, however, because if the robot lies down and is slightly out of position, a complete repositioning of the machine is required. This means it has to completely stand up and relocate to a hopefully better position.

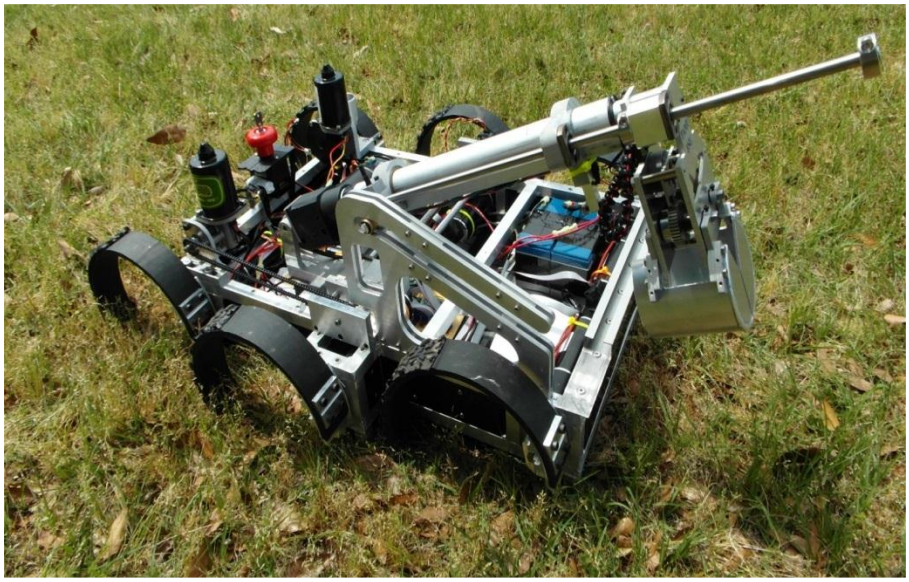


Figure 33 -- SpaceHex laying down

To combat this, a "nudge" function will be implemented. The rover will very quickly push the legs into the ground, creating lift and hopefully pushing the robot backwards. This could also be implemented to just the left or right legs, which will allow the robot to turn slightly even though it is lying down.

The advantages to this could be incredibly evident, since the team which collects the most rocks gets the most points. Last year's team was only able to collect one rock because of how hard it was to correctly position the rover over a rock.

B.4.5 – Control through Gaming Controller

Using a GUI (graphical user interface) was reasonable for last year's machine, but this year a more user-friendly interface is going to be implemented. All options are being considered, so long as it is a wireless controller. Some ideas have been discussed, but the most common ones are gaming controllers.

Figure 34 -- Common Gaming Controllers

The advantage to these types of controllers is extremely evident. There are so many ways which these controllers could be of use to the rover. First and foremost, there must be a way to control the locomotion of the machine, while simultaneously controlling the arm of the machine.

The current machine is designed to operate in locomotion until a rock is spotted and gone after. The machine lies down before the arm is activated. This is a good design, and allows for the machine to perform both tasks. On startup, the controller can be in "locomotion mode." In this mode, the left analog sticks will be used to control the robots forward and backward motion, while the right analog stick is going to control the left and right motion. This will allow the controller to control speed, direction, and intensity of every motion the machine makes. When the robot enters "lie down mode" (i.e. after pressing 'X'), the robot can use the joysticks to control the arm. The vast numbers of buttons can allow the robot to perform different tasks such as "drop arm" and "nudge backwards".

The biggest problem with this design is that the current code for the rover isn't dynamic enough for this control mechanism. The robot moves with each command, and does not allow any commands to be input until the command finishes its execution. A controller is constantly changing commands (with a joystick). This can be worked around by making the code more dynamic and allowing commands to change throughout. This is usually easily accomplished by enabling interrupts in the code, which will be attempted.

Appendix C – Decision Matrices

C.0 Decision Matrices

After considering the designs above, the team created the following decision matrices to make our selection.

C.1 Arm Selection

Table 4 -- Arm Design Decision Matrix

Robotic Arm								
	Rank	Weight	2 DOF		3 DOF w/ 1 planar		3 DOF all revolute	
			Value	Score	Value	Score	Value	Score
Weight	1	0.25	7	0.175	8	0.200	9	0.225
Size	8	0.02	5	0.010	7	0.014	8	0.016
Controllability	6	0.06	10	0.060	8	0.048	6	0.036
Speed	4	0.15	7	0.105	7	0.105	8	0.120
Reliability	3	0.17	9	0.153	7	0.119	6	0.102
Autonomous	7	0.04	9	0.036	7	0.028	6	0.024
Reach	2	0.21	5	0.105	7	0.147	8	0.168
Cost	5	0.10	8	0.080	7	0.07	6	0.06
				0.724		0.731		0.751
Total			**0.724**		**0.731**		**0.751**	

Description of Design Factors

Weight – The weight of the overall rover design greatly affects the score teams receive at competition. The weight of the arm mechanism therefore has the most weight in our decision

Reach – Once the rover has gotten close to a rock, the amount of reach it has becomes important. Being able to grab a rock that is far away from the rover will reduce the time needed to collect rock samples.

Reliability – The reliability of the arm is also very important. Several teams, who were selected to compete in the competition, could not do so because some part of their system failed the day before of the day of competition.

Speed – The rate at which the arm goes from stowed position to the position of the sample and is important to improve the overall speed of sample acquisition.

Cost – As a school project, cost is a factor. The more expensive the design, the harder it will be to receive the necessary funds to construct the design.

Controllability – The difficulty to move the arm from one point to a new point. The difficulty of mapping from robots joints frames to the x y z coordinate frame.

The difficulty in making the system autonomous – The difficulty in making the system almost completely autonomous. With autonomous control, less work will be required to command the arm and acquire the rock samples, saving time on collection.

Size – The overall size of the design needs to fit within certain size requirements. The robotic arm cannot exceed these specifications, but as long as the design does, the arm's size is not critical.

C.2 Gripper Selection

Table 5 -- Gripper Design Decision Matrix

Gripper Design										
	Rank	Weight	Scooper		Pincer		Complaint Finger		Complaint Mesh	
			Value	Score	Value	Score	Value	Score	Value	Score
Weight	6	0.05	3	0.015	7	0.035	5	0.025	3	0.015
Size	4	0.10	5	0.050	9	0.090	7	0.070	5	0.050
Speed	7	0.03	7	0.021	3	0.009	4	0.012	5	0.015
Reliability	3	0.20	7	0.140	3	0.060	3	0.060	5	0.100
Tolerance	1	0.30	9	0.270	1	0.030	3	0.090	8	0.240
Precision	2	0.25	1	0.025	9	0.225	9	0.225	7	0.158
Cost	5	0.07	9	0.063	7	0.049	5	0.035	7	0.049
Total			**0.584**		**0.498**		**0.517**		**0.627**	

Description of Design Factors

Tolerance – Tolerance is the grippers ability to pick up the same rock from multiple different positions and orientations

Precision – Precision is the gripper's ability to selectively pick up a single rock without picking up any other material.

Reliability – The Reliability of the gripper is its consistence in working for the same rock and for no component on the gripper to fail.

Size – The size of the gripper affects the size of the arm and the motors needed for the arm. However, the larger the size, the more area the gripper has to use to grab samples.

Cost – Cost is the difference in cost for the components of the grippers

Weight – Weight is similar to size and affects the size and motors needed for the arm mechanism.

Speed – Speed is the amount of time it takes the grippers to close onto a rock and acquire it.

Appendix D – Detailed Design and Design for Manufacturing

D1.0 – SEM Prototyping

One of the key goals of the project was to develop designs quickly and prototype the most viable design concepts. The prototype could then be tested to establish the capabilities and weakness of the initial design, which will allow further prototypes to correct for the weakness encountered in the initial design without significant investment into a sign design.

The initial designs show in Figure 19 uses 4 servos to control the various joints of the arms, and has each servo placed at the joint it controls. This design was created to reduce the difficulty of the control needed to program and control the robotic arm. Servo's are actuated by providing a desired position in the form of a pulse width, then the servo has its own built in control to maintain the position. Placing the motors on the joint it will actuate translates to the position of the motor being the same as the position of the arm.

Our initial prototype, shown in Figure 19 utilized servo controlled joints and was built to a relative ½ scale of our expected design. The prototype was constructed using ABS plastic which was laser cut for rapid manufacturing. The design showed some flaws with the initial concept, which were the servos preprogrammed control could not accurately maintain a position when it was strained toward the limit of its torque capability. It was evident the torque capability of the servos would be insufficient for the full scale robotic arm.

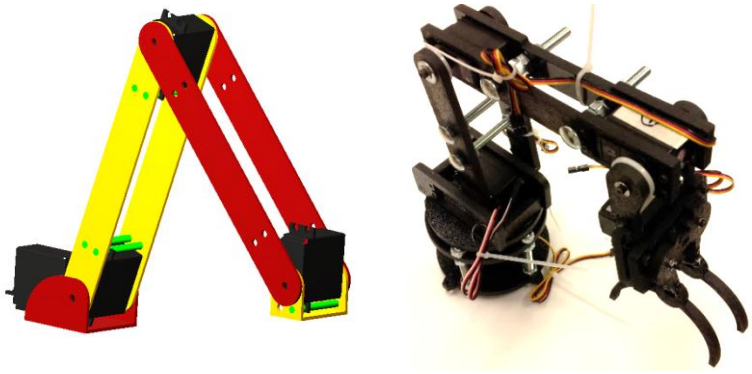


Figure 35 -- Initial Design CAD and Prototype

The arm was then redesigned to rectify the issues of the initial design. The solutions were to use DC motors, which require more complex control algorithms but can produce significantly more torque. By using DC motors, the control algorithm can be tuned so the arm will maintain the exact desired position in situation which are below the torque limit of the motor. Selecting a DC motor for the various joints require analysis to determine the amount of torque necessary for each joint while keeping the weight of the arm light. The motor selection is described in section D.2.2. The second solution was to move the motor on the elbow joint to the base and use some form of linkage or chain drive to actuate the joint. These design changes led to the second generation design.

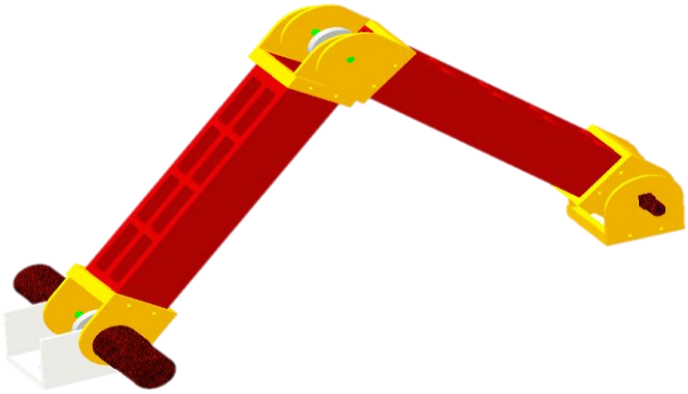


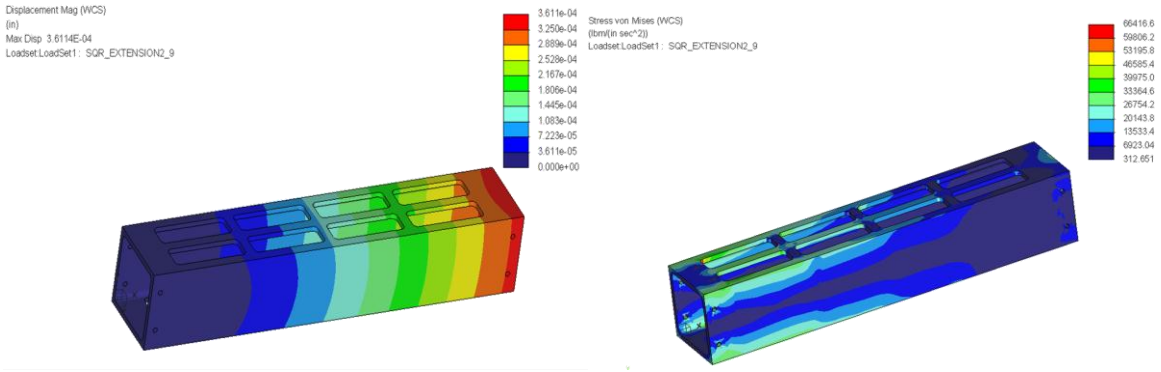
Figure 36 -- Second Generation Design

Once DC motors were selected and the motor placement was determined, the second-generation design was created. The scale model showed we needed to reduce the amount of weight the motors need to actuate to allow readily available motors to be utilized. The initial design used a 3 feet reach, which was designed to require minimal precision of the rover to acquire samples, however, the design was shortened to 2 feet to reduce the torque requires at the base joint to support and accelerate the arm through its range of motion.

The design has a chain drive to drive the second joint. A chain drive was selected to provide the closest simulation of direct drive. Only the relative sizes of the sprockets, which are going to be the same, are needed to be added the position of the motor. A chain drive was selected over a belt drive for durability and for the reliability in cold temperatures expected on mars.

D1.1 – FEM Analysis

FEM analysis was used to determine the amount of deflection and the stresses which the beams would experience during the competition. The deflection is important in maintaining a mapping from the base of the arm to the location of the gripper based on just the positions of the motors which will be directly measured using encoders. The FEM analysis is shown below. The deflection measured was 0.0036in which is small enough to not affect the ability of the gripper to move to a set location. The stress analysis showed the max stress was 309.38 psi which is below the modulus of elasticity of Aluminum 6063 which is 10000 ksi.



D1.2 – Motor Analysis

The amount of torque necessary was determined by assuming the arm was in its worst case scenario and determining the forces on the arm. The arms were represented by a distributed mass, the motors were evaluated as point masses at their location. The equations for the torque at any point were then calculated in Matlab and plotted to visualize the torque requirements through the entire range of motion.

The base motors selected were a RE 40 Ø40 mm, Graphite Brushes, 150 Watt with a Planetary Gearhead GP 42 C Ø42 mm, 3–15 Nm. This combination of a 170 mNm motor with a 113:1 planetary gearbox provides allows the motor to provide 15 Nm of torque nominally. The expected load determined in the worst case scenario is 10 Nm. Since these motors are going to be installed at the base, their weight will not affect the torque requirements.

The motors for the wrist and the gripper do not require significant torque, the requirement determined was 3 kg-cm, but the motor weight will affect the torque of the base motors. Therefore, the Pololu 298:1 Micro Metal Gear motor was selected because it was the lightest weight motor that could provide the necessary torque.

D2.0 – Gripper Prototyping

The work done to prototype the gripper has produced several working prototypes. The first generation was a proof of concept which was actuated by hand. It gave us a sense of the amount of effort a motor would need to provide and some of the manufacturing issues we would face with producing a gripper with compliant materials. The material for the elastic gripper was initial chosen to be rubber bands for simplicity.

Figure 37 -- First Generation Gripper Prototype

The second generation model was produced using rapid prototyping techniques. Cardboard was used as the construction material, for it is free from the local hardware store and when paired with hot glue can produce a reasonably strong structure. The prototype used a servo mechanism to actuate the gripper and an elastic first aid tape for the elastic material. With this gripper, we were able to test the design and see the design pick up rock samples. While the prototype was able to pick up rocks, the cardboard caused some issues which were resolved in the current prototype.

Figure 38 -- Second Generation Gripper Prototype

The current gripper prototype, which appears to work as our final version used ABS plastic for the frame and linkages. The ABS was used to make gear which keep the linkages at the same location through the range of motion. The elastic material was changed to silicon rubber, which will remain elastic at the temperatures expected on mars. The design also utilize the pololu motor and was shown to prove powerful enough to pick up rocks larger than expected at the competition.

Figure 39 -- Third Generation Prototype

Appendix E – Programming and Control Implementation

E1.0 –Xpadder

Once the SDL is imported onto the SD card and implemented, the next step is transferring those commands from the keyboard to the actual XBOX controller. Using a free program called Xpadder, this is not only possible, but it is easy to implement.

Figure 40 -- Example of Xpadder Interface

Xpadder is a software application which allows a controller to emulate a keyboard. The user must upload a picture of the controller to the program, and then map the buttons to whichever keyboard button they wish. The computer will then read the button presses as key presses, and thus the computer will act normally as if that button was pressed on the keyboard.

With the combination of Xpadder and SDL, the rover will be able to take commands directly from the XBOX controller. By simply mapping the joystick to the forward, backward, left and right buttons on the keyboard, the rover will be able to walk continuously until the joystick is released. This will allow for real-time control and a dynamic rover, as intended.

E2.0 – SDL Library

Currently, the rover is being controlled through the command prompt on a laptop. While this is a functioning design, it is not ideal since there are huge delays between inputs and they are not intuitive. To improve the rover's locomotion and control, an XBOX controller will be implemented in order to reduce delays between commands, essentially controlling the rover in real-time.

In order to achieve this, low level keyboard access and event handling are necessary. Simple DirectMedia Layer (SDL) is a library written in C which enables both keyboard access and event handing to the user. With this, the user will be able to control the rover by simply pressing buttons on the keyboard (i.e. holding w will move the rover forward) as opposed to typing entire commands to the command prompt. The code will then have to be re-written in order to allow for real-time control, which means the implementation of a dynamic function with either interrupts or continuous looping of the walk function.

Appendix F – Source Code

F1.0 – Static Switch Main.c

```
 1/*************************************************************/
 2 #include <stdlib.h>
 3 #include <stdio.h>
 4 #include <unistd.h>
 5 #include <math.h>
 6 #include "motor.h"
 7 #include "buehler.h"
 8
 9 //void printHelp();
10 void applyHold(int holdPos)
11 {
12   FILE * fp;
13   fp = fopen("/root/src/holdTxt.txt" ,"w");
14   fprintf(fp,"1 %d", holdPos);
15   fclose(fp);
16 }
17
18 void rmHold()
19 {
20   FILE * fp;
21   fp = fopen("/root/src/holdTxt.txt","w");
22   fprintf(fp, "0 0");
23   fclose(fp);
24   delay(1000);
25 }
26
27 int main(int argc, char* argv[])
28 {
29   struct timeval tv = {.tv_sec = 0, .tv_usec = 0};
30   int ipos[6] = {0}, mpos[6] = {0}, duty[6] = {0x00};
31   int rpm,steps,angles,phase_offset,stepPhase_time,i;
32   char dir,turn_dir;
33
34   //Initialize SPI,UART,DecoderRst pin,EmergencyStop pin
35   Init(0,8000000,6,5);
36
37   if(argc == 1)
38   {
39   //  printHelp();
40     return(0);
```

```
41   }
42
43   else
44   {
45     rmHold();
46     switch(*(++argv[1]))
47     {
48       case 'v': printf("Hill Gait\n");
49                 sscanf(argv[2],"%d",&rpm);
50                 sscanf(argv[3],"%d",&steps);
51                 moove(10,'F','A',OFFSET2);
52                 hillGait(rpm,'F',steps);
53                 moove(10,'F','A',OFFSET2);
54                 applyHold(OFFSET2);
55                 break;
56
57       case 's': printf("LETS CLIMB STAIRS\n");
58                 sscanf(argv[2],"%d",&rpm);
59                 sscanf(argv[3],"%d",&steps);
60                 //move(10,'F','F',OFFSET3);
61                 printf("Finished F\n");
62                 //move(10,'F','M',OFFSET1);
63                 printf("Finished M\n");
64                 //move(10,'F','B',OFFSET4);
65                 printf("Finished B\n");
66                 stair(rpm,steps);
67                 break;
68
69       case 'q': printf("Turn during Locomtion\n");
70                 sscanf(argv[2],"%d",&rpm);
71                 sscanf(argv[3],"%c",&dir);
72                 sscanf(argv[4],"%d",&steps);
73                 sscanf(argv[5],"%d",&angles);
74                 moove(10,'H',1,OFFSET2);
75                 walk_turn(rpm,dir,steps,angles,0);
76                 moove(10,'F','A',OFFSET1);
77                 applyHold(OFFSET1);
78                 break;
79
80       case 'z': printf("Precision Turn\n");
81                 sscanf(argv[2],"%d",&rpm);
82                 sscanf(argv[3],"%c",&dir);
83                 sscanf(argv[4],"%d",&steps);
84
```

```
 85                 if(dir == 'R')
 86                   moove(10,'H',1, OFFSET2);
 87                 else
 88                   moove(10,'H',0, OFFSET2);
 89
 90                 turn(rpm,dir,steps);
 91                 moove(10,'H','A',OFFSET1);
 92                 applyHold(OFFSET1);
 93                 break;
 94
 95       case 'w': printf("WALK\n");
 96                 sscanf(argv[2],"%d",&rpm);
 97                 sscanf(argv[3],"%c",&dir);
 98                 sscanf(argv[4],"%d",&steps);
 99                 moove(10,'H',1,OFFSET2);
100                 walk(rpm,dir,steps,0);
101                 moove(10,'F','A',OFFSET1);
102
103                 applyHold(OFFSET1);
104                 break;
105
106       case 't': printf("TURN\n");
107                 sscanf(argv[2],"%d",&rpm);
108                 sscanf(argv[3],"%c",&dir);
109                 sscanf(argv[4],"%d",&steps);
110                 moove(10,'H',1,OFFSET2);
111                 walk(rpm, dir, steps,0);
112                 moove(10,'F','A',OFFSET1);
113                 applyHold(OFFSET1);
114                 break;
115
116       case 'c': printf("CALIBRATE\n");
117                 duty[0] =
duty[1]=duty[2]=duty[3]=duty[4]=duty[5] = -3;
118                 driveAllMotors('B',duty);
119                 delay(6000);
120                 stopMotors();
121                 break;
122
123       case 'u': printf("STAND\n");
124                 moove(10,'F','A',OFFSET1);
125                 applyHold(OFFSET1);
126                 break;
127
```

```
128       case 'l': printf("LIE\n");
129                 moove(10,'F','A',OFFSET1);
130                 moove(5,'F','A',2000);
131                 break;
132
133 //      case 's': printf("ESTOP\n");
134 //                stopMotors();
135 //                break;
136
137       case 'r': printf("RESETDEC\n");
138                 resetAllDecoders();
139                 break;
140
141       case 'h': printf("HOLD\n");
142                 applyHold(0);
143                 break;
144
145       case 'b': printf("REMOVE HOLD\n");
146                 rmHold();
147                 break;
148
149       case 'm': printf("POSITION\n");
150                 readAllMotorPos(mpos);
151                 printf("mpos0: %d, mpos1: %d, mpos2: %d,
mpos3: %d. mpo    s4: %d, mpos5: %d\n",
mpos[0],mpos[1],mpos[2],mpos[3],mpos[4],mpos[5]);
152                 break;
153
154       case 'x': printf("TEST\n");
155                 /*walk(10,'F',3);
156                 duty[0] = duty[1]=duty[2]=duty[3]=duty[5] =
0;
157                 duty[3] = 40;
158                 driveAllMotors('F',duty);
159                 delay(4000);
160
161                 duty[0] = duty[1]=duty[2]=duty[3]=duty[4] =
0;
162                 duty[5] = 40;
163
164                 driveAllMotors('F',duty);
165                 delay(4000);
166                 duty[0] =
duty[1]=duty[2]=duty[3]=duty[4]=duty[5] = 0;
```

```
167                 driveAllMotors('F',duty);
168                 break;
169                 */
170       default : printf("**Invalid option**\n");
171                 break;
172     }
173   }
174   return 0;
175 }
```

F2.0 – Dynamic Switching Main.c (GET_CH_Test.c)

```
 1 #include <stdlib.h>
 2 #include <stdio.h>
 3 #include <unistd.h>
 4 #include "motor.h"
 5 #include "buehler.h"
 6 #include <curses.h>
 7
 8  void applyHold(int holdPos)
 9  {
10    FILE * fp;
11    fp = fopen("/root/src/holdTxt.txt" ,"w");
12    fprintf(fp,"1 %d", holdPos);
13    fclose(fp);
14  }
15
16  void rmHold()
17  {
18    FILE * fp;
19    fp = fopen("/root/src/holdTxt.txt","w");
20    fprintf(fp, "0 0");
21    fclose(fp);
22    delay(1000);
23  }
24
25 void preprocess()
26 {
27   struct timeval tv = {.tv_sec = 0, .tv_usec = 0};
28   int ipos[6] = {0}, mpos[6] = {0}, duty[6] = {0x00};
29   Init(0,8000000,6,5);
30   rmHold();
31 }
32
33 int stepcount = 1;
34 int oddstep = 0;
35 int INIT_STEP = 0;
36
37 int main()
38 {
39   initscr();
40   refresh();
41   cbreak();
42   noecho();
43   halfdelay(5);
```

```
44   keypad(stdscr,TRUE);
45   refresh();
46   endwin();
47   int fish;
48   char ch = 'm';
49   int angles = 5;
50   int rpm = 20;
51   char dir = 'F';
52   int steps = 1;
53   int duty[6] = {0x00};
54
55   preprocess();
56   resetAllDecoders();
57
58   preprocess();
59   while(ch != 27)
60   {
61      preprocess();
62      initscr();
63      refresh();
64      flushinp();
65      ch = wgetch(stdscr);
66      refresh();
67      endwin();
68     switch(ch)
69     {
70       case 'w':
71         // Forward
72            if (stepcount % 2 == 0)
73             oddstep = 0;
74            else
75             oddstep = 1;
76            initscr();
77            refresh();
78            flushinp();
79            ch = wgetch(stdscr);
80            refresh();
81            endwin();
82            //printf("BEFORE RESET");
83            walk(rpm,'F',0, oddstep);
84            //printf("AFTER RESET");
85            while (ch == 'w')
86            {
87              //printf("BEFORE: Oddstep is: %d\n",oddstep);
```

```
 88              //if (oddstep == 0)
 89               walk(rpm,'F',steps/*1*/,oddstep);
 90              //else
 91               //walk(rpm,'F',steps,1);
 92              ++stepcount;
 93              //printf("Step count is %d\n", stepcount);
 94              printf("Inside the w while loop\n");
 95              initscr();
 96              refresh();
 97              flushinp();
 98              ch = wgetch(stdscr);
 99              refresh();
100              endwin();
101          }
102          //printf("Outside the w while loop\n");
103       break;
104
105       case 's':
106         // Backwards
107         if (stepcount % 2 == 0)
108           oddstep = 0;
109         else
110           oddstep = 1;
111         //printf("Oddsteps is %d\n",oddstep);
112         initscr();
113         refresh();
114         flushinp();
115         ch = wgetch(stdscr);
116         refresh();
117         endwin();
118         walk(rpm,'F',0,oddstep);
119         while(ch == 's')
120         {
121           //if (oddstep == 0)
122             walk(rpm,'B',steps,oddstep);
123           //else
124             //walk(rpm,'B',steps,1);
125           ++stepcount;
126           //printf("Step count is: %d\n",stepcount);
127           printf("Inside the s while loop\n");
128           initscr();
129           refresh();
130           flushinp();
131           ch = wgetch(stdscr);
```

```
132           refresh();
133           endwin();
134         }
135       break;
136
137       case 'a':
138       //left
139         if (stepcount % 2 == 0)
140           oddstep = 0;
141         else
142           oddstep = 1;
143         initscr();
144         refresh();
145         flushinp();
146         ch = wgetch(stdscr);
147         refresh();
148         endwin();
149         walk(rpm,'F',0,0);
150         while (ch == 'a')
151         {
152           walk(rpm,'L',steps,oddstep);
153           ++stepcount;
154           initscr();
155           refresh();
156           printw("Inside the a while loop\n");
157           flushinp();
158           ch = wgetch(stdscr);
159           refresh();
160           endwin();
161         }
162         break;
163
164       case 'd':
165       //right
166         if (stepcount % 2 == 0)
167           oddstep = 0;
168         else
169           oddstep = 1;
170         initscr();
171         refresh();
172         flushinp();
173         ch = wgetch(stdscr);
174         refresh();
175         endwin();
```

```
176
177         walk(rpm,'F',0,0);
178         while (ch == 'd')
179         {
180           walk(rpm,'R',steps,oddstep);
181           ++stepcount;
182           initscr();
183           refresh();
184           printw("Inside the d while loop\n");
185           flushinp();
186           ch = wgetch(stdscr);
187           refresh();
188           endwin();
189         }
190       break;
191
192       case 'u':
193         // Speed UP
194         printf("Previous RPM = %d\n", rpm);
195         if(rpm < 26)
196           rpm += 5;
197         else
198           rpm = 30;
199         printf("Current RPM = %d\n", rpm);
200       break;
201
202       case 'j':
203         // Speed DOWN
204         printf("Previous RPM = %d\n", rpm);
205         if(rpm > 14)
206           rpm -= 5;
207         else
208           rpm = 10;
209         printf("Current RPM = %d\n",rpm);
210       break;
211
212       case 'i':
213         // Angles up
214         printf("Previous angles = %d\n", angles);
215         if (angles < 10)
216           ++angles;
217         else
218           angles = 10;
219         printf("Current Angles = %d\n",angles);
```

```
220         break;
221
222       case 'k':
223         // Angles Down
224         printf("Previous angles = %d\n",angles);
225         if (angles > 0)
226           --angles;
227         else
228           angles = 0;
229         printf("Current Angles = %d\n",angles);
230         break;
231
232       case 'q':
233         // Turn while walking - Left
234         if (stepcount % 2 == 0)
235           oddstep = 0;
236         else
237           oddstep = 1;
238         //printf("Oddsteps is %d\n",oddstep);
239         initscr();
240         refresh();
241         flushinp();
242         ch = wgetch(stdscr);
243         refresh();
244         endwin();
245         walk_turn (rpm,dir,0,0,oddstep);
246         while (ch == 'q')
247         {
248           walk_turn(rpm,'L',steps,angles,oddstep);
249           ++stepcount;
250           initscr();
251           refresh();
252           printw("Inside the q while loop\n");
253           flushinp();
254           ch = wgetch(stdscr);
255           refresh();
256           endwin();
257         }
258         if (stepcount % 2 == 0)
259           oddstep = 0;
260         else
261           oddstep = 1;
262         if (oddstep == 0)
263         {
```

```
264           moove(10,'F',1,OFFSET2);
265           moove(10,'H',0,OFFSET1);
266         }
267         else
268         {
269           moove(10,'F',1,OFFSET1);
270           moove(10,'H',0,OFFSET2);
271         }
272       break;
273
274       case 'e':
275         // Turn while walking - Right
276         if (stepcount % 2 == 0)
277           oddstep = 0;
278         else
279           oddstep = 1;
280         //printf("Oddsteps is %d\n",oddstep);
281         initscr();
282         refresh();
283         flushinp();
284         ch = wgetch(stdscr);
285         refresh();
286         endwin();
287         walk_turn(rpm,dir,0,0,oddstep);
288         while (ch == 'e')
289         {
290           walk_turn(rpm,'R',steps,angles,oddstep);
291           ++stepcount;
292           initscr();
293           refresh();
294           printw("inside the e while loop\n");
295           flushinp();
296           ch = wgetch(stdscr);
297           refresh();
298           endwin();
299         }
300         if (stepcount % 2 == 0)
301           oddstep = 0;
302         else
303           oddstep = 1;
304         if (oddstep == 0)
305         {
306           moove(10,'H',1,OFFSET2);
307           moove(10,'F',0,OFFSET1);
```

```
308         }
309         else
310         {
311           moove(10,'H',1,OFFSET1);
312           moove(10,'F',0,OFFSET2);
313         }
314       break;
315
316       case 'c':
317         // EZ Calibrate
318         duty[0] = duty[1] = duty[2] = duty[3] = duty[4] =
duty[5] = -3;
319         while (ch == 'c')
320         {
321           driveAllMotors('B',duty);
322           delay(2000);
323           stopMotors();
324           initscr();
325           refresh();
326           printw("Inside the c while loop\n");
327           flushinp();
328           ch = wgetch(stdscr);
329           refresh();
330           endwin();
331         }
332         resetAllDecoders();
333       break;
334
335       case 'l':
336         // Lie Down
337         moove(10,'F','A',OFFSET1);
338         moove(5,'F','A',2000);
339         initscr();
340         refresh();
341         flushinp();
342         ch = wgetch(stdscr);
343         refresh();
344         endwin();
345         while((ch != 'l') && (ch != 27))
346         {
347           printf("Inside the l while loop, this is ch:
%c\n", ch);
348           initscr();
349           refresh();
```

```
350           flushinp();
351           ch = wgetch(stdscr);
352           refresh();
353           endwin();
354         }
355       break;
356
357       default:
358         // Move back to stand
359         if (stepcount % 2 == 0)
360           oddstep = 0;
361         else
362           oddstep = 1;
363
364         moove(10,'F','A',OFFSET1);
365         stepcount = 0;
366         while (ch != 'l' && ch != 'c' && ch != 'w' && ch !=
's' && ch !    = 27 && ch != 'a' && ch != 'd' && ch != 'u' && ch
!= 'j' && ch != 'i' &    & ch != 'k' && ch != 'q' && ch != 'e')
367         {
368           hold(OFFSET1);
369           initscr();
370           refresh();
371           flushinp();
372           ch = wgetch(stdscr);
373           refresh();
374           endwin();
375         }
376         if (ch != 27 && ch != 'u' && ch != 'j' && ch != 'i'
&& ch != 'k    ' && ch != 'c' && ch != 'l')
377         {
378           moove(10,'H',1,OFFSET2);
379         }
380       break;
381     }
382   }
383   refresh();
384   endwin();
385   printf("Exiting the program peacefully human\n");
386   return 0;
387
388 }
389
```

F3.0 – Buehler.h

```
  1
/*************************************************************/
  2 #ifndef BUEHLER_H
  3 #define BUEHLER_H
  4
  5 #include <stdio.h>
  6 #include <stdlib.h>
  7 #include <sys/time.h>
  8 #include <curses.h>
  9
 10 #define OFFSET1 33852       //Offset for even triplet
 11 #define OFFSET3 22852       //OFFSET TEST
 12 #define OFFSET4 40852       //OFFSET TEST
 13 #define OFFSET2 12415       //Offset for odd triplet
 14 #define BOUND 50
 15
 16 struct timeval walks_turn_buehler(int rpm, char dir, int*
pos, int angl    es, int stepcount);
 17 struct timeval stair_buehler(int rpm, int* pos);
 18 //Caculates the ideal Position
 19 struct timeval buehler(int rpm, char dir, int* pos, int
oddstepss)
 20 {
 21 //  printf("IN BUEHLER = %d\n",oddstepss);
 22   static struct timeval startTime = {.tv_sec = -1, .tv_usec
= -1};
 23   struct timeval currTime;
 24   static int buehlerPeriod;
 25   static int T1,T2;
 26   int P1,P2;
 27
 28   //Initialize function if its the first call in the loop
 29   if(startTime.tv_sec == -1)
 30   {
 31     //Get start time
 32     gettimeofday(&startTime,NULL);
 33
 34     //Calculate buehler period
 35     buehlerPeriod = (int) (((float)(60)/rpm)*1000000);
 36
 37     //Calculate transition points
 38     T1 = (int) ((float)(buehlerPeriod)/4);
 39     T2 = buehlerPeriod - T1;
```

```
 40   }
 41
 42 /*if (dir == 'U')
 43   {
 44     startTime.tv_sec = currTime.tv_sec;
 45     startTime.tv_usec = currTime.tv_usec;
 46   }
 47 */
 48   //Get current time
 49   gettimeofday(&currTime, NULL);
 50
 51
 52
 53   if (dir == 'U')
 54   {
 55     startTime = currTime;
 56     startTime.tv_sec = currTime.tv_sec;
 57     startTime.tv_usec = currTime.tv_usec;
 58   }
 59
 60
 61   //Caculate the current buehler phasor
 62   unsigned long long buehlerPhase = (unsigned long long)
((currTime.tv_    sec - startTime.tv_sec)*1000000 +
 63                                      (currTime.tv_usec -
startTime.tv_us    ec)) % buehlerPeriod;
 64   //printf("buehlerPeriod : %d, beuhlerPhase %d startTime %f
currTime %    f\n", buehlerPeriod, buehlerPhase,
startTime.tv_usec, currTime.tv_usec/    *Int*/);
 65   //printf("before IF %d\n", oddstepss);
 66   if (oddstepss == 1 && dir != 'U')
 67   {
 68     //printf("You wrong%d\n", oddstepss);
 69     buehlerPhase = (buehlerPhase + (buehlerPeriod/2)) %
buehlerPeriod;
 70   }
 71 //  printf("Buehler/2 = %d, buehlerPhase = %d, oddstep =
%d\n", (buehle    rPeriod/2), buehlerPhase, oddstepss);
 72 //  printf("buehlerPeriod : %d, beuhlerPhase %d startTime %f
currTime %    f\n", buehlerPeriod, buehlerPhase,
startTime.tv_usec, currTime.tv_usec/    *Int*/);
 73   //Get positions of both buehler cycles
 74   //printf("after IF %d\n", oddstepss);
 75   /*if (oddstep == 0)
```

```
 76   {*/
 77     if(buehlerPhase <= T1)
 78     { P1 = ((int)
(((float)(1)*NUM_POS*buehlerPhase)/(3*buehlerPeriod))    ) %
NUM_POS;
 79       P2 = ((int)
(((float)(5)*NUM_POS*buehlerPhase)/(3*buehlerPeriod))    ) %
NUM_POS; }
 80     else if (buehlerPhase < T2)
 81     { P1 = ((int) ((((float)(5)*NUM_POS*(buehlerPhase-
T1))/(3*buehlerPe    riod)) + ((float)(1)*NUM_POS)/12)) %
NUM_POS;
 82       P2 = ((int) ((((float)(1)*NUM_POS*(buehlerPhase-
T1))/(3*buehlerPe    riod)) + ((float)(5)*NUM_POS)/12)) %
NUM_POS; }
 83     else
 84     { P1 = ((int) ((((float)(1)*NUM_POS*(buehlerPhase-
T2))/(3*buehlerPe    riod)) + ((float)(11)*NUM_POS)/12)) %
NUM_POS;
 85       P2 = ((int) ((((float)(5)*NUM_POS*(buehlerPhase-
T2))/(3*buehlerPe    riod)) + ((float)(07)*NUM_POS)/12)) %
NUM_POS; }
 86   //}
 87   /*else
 88   {
 89     if(buehlerPhase <= T1)
 90       { P2 = ((int)
(((float)(1)*NUM_POS*buehlerPhase)/(3*buehlerPeriod    ))) %
NUM_POS;
 91         P1 = ((int)
(((float)(5)*NUM_POS*buehlerPhase)/(3*buehlerPeriod    ))) %
NUM_POS; }
 92       else if (buehlerPhase < T2)
 93       { P2 = ((int) ((((float)(5)*NUM_POS*(buehlerPhase-
T1))/(3*buehler    Period)) + ((float)(1)*NUM_POS)/12)) %
NUM_POS;
 94         P1 = ((int) ((((float)(1)*NUM_POS*(buehlerPhase-
T1))/(3*buehler    Period)) + ((float)(5)*NUM_POS)/12)) %
NUM_POS; }
 95       else
 96       { P2 = ((int) ((((float)(1)*NUM_POS*(buehlerPhase-
T2))/(3*buehler    Period)) + ((float)(11)*NUM_POS)/12)) %
NUM_POS;
```

```
 97         P1 = ((int) ((((float)(5)*NUM_POS*(buehlerPhase-
T2))/(3*buehler    Period)) + ((float)(07)*NUM_POS)/12)) %
NUM_POS; }
 98   }*/
 99   //printf("buehlerPeriod: %d, buehlerPhase: %llu, T1: %d,
T2: %d, iPos    [0]: %d, iPos[1]: %d\n",buehlerPeriod,
buehlerPhase, T1, T2, pos[0], po    s[1]);
100
101   //Adjust position with offset and direction
102   if(dir == 'B')
103   { pos[0] = pos[2] = pos[4] = NUM_POS - (P1 + (NUM_POS -
OFFSET1)) % N    UM_POS;
104     pos[1] = pos[3] = pos[5] = NUM_POS - (P2 + (NUM_POS -
OFFSET2)) % N    UM_POS; }
105   else if(dir == 'R')
106   { pos[0] = pos[2] = (P1 + OFFSET1) % NUM_POS;
107     pos[1] = (P2 + OFFSET2) % NUM_POS;
108     pos[3] = pos[5] = NUM_POS - (P2 + (NUM_POS - OFFSET2)) %
NUM_POS;
109     pos[4] = NUM_POS - (P1 + (NUM_POS - OFFSET1)) % NUM_POS;
}
110   else if(dir == 'L')
111   { pos[0] = pos[2] = NUM_POS - (P1 + (NUM_POS - OFFSET1)) %
NUM_POS;
112     pos[1] = NUM_POS - (P2 + (NUM_POS - OFFSET2)) % NUM_POS;
113     pos[3] = pos[5] = (P2 + OFFSET2) % NUM_POS;
114     pos[4] = (P1 + OFFSET1) % NUM_POS; }
115   else
116   { pos[0] = pos[2] = pos[4] = (P1 + OFFSET1) % NUM_POS;
117     pos[1] = pos[3] = pos[5] = (P2 + OFFSET2) % NUM_POS;
118   }
119
120   return(currTime);
121 }
122
123 //Walking Algorithm
124 void walk(int rpm, char dir, int numSteps, int oddsteps)
125 {
126   //printf("AFTER: Oddsteps is %d\n",oddsteps);
127   struct timeval tv = {.tv_sec = 0, .tv_usec = 0};
128   int ipos[6] = {0}, mpos[6] = {0}, duty[6] = {0},
done[6]={0};
129   int stepNum = 0, lpos = 0;
130
```

```
131   if (numSteps == 0)
132       dir = 'U';
133
134   if((dir=='L') || (dir=='B'))
135     numSteps = numSteps;
136   else if (dir == 'U')
137     tv = buehler(rpm,dir,ipos,oddsteps);
138   else
139     numSteps = numSteps+1;
140
141   //Walk numSteps
142   while(stepNum < numSteps)
143   {
144   // if (numSteps == 1)
145   //    dir = 'U';
146     tv = buehler(rpm,dir,ipos,oddsteps);
147     readAllMotorPos(mpos);
148
149     //printf("Init Pos = %d\n", *ipos);
150     //printf("Motor Pos = %d\n", *mpos);
151
152
153
154     PD(&tv,ipos,mpos,duty);
155     driveAllMotors('H',duty);
156
157
158     if((dir=='L') || (dir=='B'))
159     {
160       if((((OFFSET1 > (*ipos)) && (lpos >= OFFSET1))) ||
((OFFSET2 > (*    ipos)) && (lpos >= OFFSET2)))
161         stepNum++;
162       lpos = *ipos;
163     }
164     else
165     {
166       if((((OFFSET1 < (*ipos)) && (lpos <= OFFSET1))) ||
((OFFSET2 < (*    ipos)) && (lpos <= OFFSET2)))
167         stepNum++;
168       lpos = *ipos;
169     }
170 //    printf("Init Pos = %d\n", *ipos);
171 //    printf("Motor Pos = %d\n", *mpos);
172   //  printf("NumSteps = %d\n",numSteps);
```

```
173   //  printf("stepNum = %d\n",stepNum);
174   }
175
176   //Ensure motors are stopped
177   stopMotors();
178   return;
179 }
180
181 //Ideal follower for moving legs angles less than 2*pi
182 struct timeval follower(int rpm, char dir, char legs, int
flag, int* sp    os, int* pos)
183 {
184   static struct timeval startTime = {.tv_sec = -1, .tv_usec
= -1};
185   struct timeval currTime;
186   static int period;
187   int cpos;
188
189   //Get start time if not initialized
190   if((startTime.tv_sec == -1) || (flag == 1))
191   {
192     gettimeofday(&startTime,NULL);
193     period = (int) (((float)(60)/rpm)*1000000);
194   }
195
196   //Get current time and set all positions to ideal start
197   gettimeofday(&currTime, NULL);
198
199   //Calculate current phase position
200   unsigned long long phase = (unsigned long long)
((currTime.tv_sec - s    tartTime.tv_sec)*1000000 +
201
(currTime.tv_usec - s    tartTime.tv_usec)) % period;
202   cpos = (int) ((((float)(1)*NUM_POS*phase)/(period))) %
NUM_POS;
203
204   //Calculate position with start offset and proper
direction for all l    egs
205   int i;
206   for(i = 0; i < NUM_MOTORS; i++)
207     if(dir == 'F')
208       pos[i] = (spos[i] + cpos) % NUM_POS;
209     else
```

```
210       pos[i] = NUM_POS - (cpos + (NUM_POS - spos[i])) %
NUM_POS;
211
212   //If only 1 triplet is being driven zero the other triplet
213   if(legs == 0)
214     i = 1;
215   else
216     i = 0;
217
218   if (legs == 'F')
219   {
220     pos[1] = spos[1];
221     pos[2] = spos[2];
222     pos[4] = spos[4];
223     pos[5] = spos[5];
224   }
225   else if (legs == 'M')
226   {
227     pos[0] = spos[0];
228     pos[2] = spos[2];
229     pos[3] = spos[3];
230     pos[5] = spos[5];
231   }
232   else if (legs == 'B')
233   {
234     pos[1] = spos[1];
235     pos[3] = spos[3];
236     pos[4] = spos[4];
237     pos[0] = spos[0];
238   }
239   else if(legs != 'A')
240     for(i; i < NUM_MOTORS; i+=2)
241       pos[i] = spos[i];
242
243   return(currTime);
244 }
245
246
247 void hillGait(int rpm, char dir, int numSteps)
248 {
249   struct timeval tv = {.tv_sec = 0, .tv_usec = 0};
250   int ipos[6] = {0}, mpos[6] = {0}, spos[6] = {0}, duty[6] =
{0}, done[    6] = {0};
251   int stepNum = 0, lpos = 0;
```

```
252
253   readAllMotorPos(spos);
254   follower(rpm,dir,'A',1,spos,ipos);
255
256   //Walk numSteps
257   while(stepNum < numSteps)
258   {
259     tv = follower(rpm,dir,'A',0,spos,ipos);
260     ipos[1] = spos[1];
261     ipos[4] = spos[4];
262     readAllMotorPos(mpos);
263     PD(&tv,ipos,mpos,duty);
264     driveAllMotors('H',duty);
265
266     if((((OFFSET1 < (*ipos)) && (lpos < OFFSET1)))) //||
((OFFSET2 < (*    ipos)) && (lpos <= OFFSET2)))
267       stepNum++;
268     lpos = *ipos;
269   }
270
271   return;
272 }
273
274 void moove(int rpm, char dir, char legs, int epos)
275 {
276   int ipos[6] = {0}, mpos[6] = {0}, spos[6] = {0}, duty[6] =
{0}, done[    6] = {0};
277   struct timeval tv = {.tv_sec = 0, .tv_usec = 0};
278   int numMotors = 0, numDone = 0, i = 0;
279
280   if(legs == 'A')
281     numMotors = 6;
282   else if((legs == 'F') || (legs == 'M') || (legs == 'B'))
283     numMotors = 2;
284   else if((legs == 1) || (legs == 0))
285     numMotors = 3;
286
287   readAllMotorPos(spos);
288   follower(rpm,dir,legs,1,spos,ipos);
289   while(numDone < numMotors)
290   {
291     tv = follower(rpm,dir,legs,0,spos,ipos);
292     readAllMotorPos(mpos);
293
```

```
294     //Check  to see which motors have finished
295     for(i = 0; i < NUM_MOTORS; i++)
296     {
297       if((mpos[i] > epos - 1500 /* 500*/) && (mpos[i] < epos
+ 1500 /*     500*/))
298         if(done[i] == 0)
299         {
300           done[i] = 1;
301           numDone++;
302         }
303       if(done[i] == 1)
304         ipos[i] = epos;
305
306 //      printf("Motor Pos(%d) = %d\n", i, *(mpos+i));
307     }
308     //printf("Init Pos = %d\n", *(ipos+1));
309     //printf("Motor Pos(%d) = %d\n", i, *(mpos+i));
310
311     //Drive Motors that havent finished
312     PD(&tv,ipos,mpos,duty);
313 /*    int ol;
314     for (ol = 0; ol < 6; ol++)
315       printf("duty: %d\n", duty[ol]);
316 */
317     driveAllMotors('H',duty);
318   }
319   //Ensure motors are stopped
320   stopMotors();
321 }
322
323
/*************************************************************
*******    ***************
324                                 HOLD
325
*************************************************************
*******    **************/
326
327 void hold(int hpos)
328 {
329   int iPos[6] = {hpos,hpos,hpos,hpos,hpos,hpos};
330   struct timeval tv = {.tv_sec = 0, .tv_usec = 0};
331   int mpos[6] = {0};
332   int duty[6] = {0};
```

```
333   int i = 0;
334
335   while(i < 100)
336   {
337     readAllMotorPos(mpos);
338     PD(&tv,iPos,mpos,duty);
339     driveAllMotors('H',duty);
340     delay(20); // leave delay 20 not having it does not
improve the hol    d function
341     i++;
342   }
343 }
344
345 void turn(int rpm,char dir, int numSteps)
346 {
347   int ipos[6] = {0}, mpos[6] = {0}, spos[6] = {0}, duty[6] =
{0}, done[    6] = {0};
348   struct timeval tv = {.tv_sec = 0, .tv_usec = 0};
349   int numMotors = 3, numDone = 0, i = 0, tripod =0;
350
351   int stepNum = 0, lpos = 0;
352
353   if(dir == 'R')
354   {
355     tripod = 1;
356     i = 3;
357   }
358
359   readAllMotorPos(spos);
360   follower(rpm,dir,tripod,1,spos,ipos);
361
362   //Walk numSteps
363   while(stepNum < numSteps)
364   {
365     tv = follower(rpm,dir, tripod,0,spos,ipos);
366     readAllMotorPos(mpos);
367
368     if(dir == 'R')
369       ipos[1] = OFFSET2;//mpos[1];
370     else
371       ipos[4] = OFFSET2;//mpos[4];
372
373     PD(&tv,ipos,mpos,duty);
374     driveAllMotors('H',duty);
```

```
375
376     if((OFFSET1 < (ipos[i])) && (lpos < OFFSET1))
377       stepNum++;
378     lpos = ipos[i];
379   }
380
381   return;
382 }
383
384
385 struct timeval turn_buehler(int rpm, char dir, int* pos, int
phase_off,     int stepPhaseTime)
386 {
387   static struct timeval startTime = {.tv_sec = -1, .tv_usec
= -1};
388   struct timeval currTime;
389   static int buehlerPeriod;
390   static int T1_1,T1_2,T2_1,T2_2;
391   int P1,P2;
392
393   //Initialize function if its the first call in the loop
394   if(startTime.tv_sec == -1)
395   {
396     //Get start time
397     gettimeofday(&startTime,NULL);
398
399     //Calculate buehler period
400     buehlerPeriod = (int) (((float)(60)/rpm)*1000000);
401
402     //Calculate transition points
403     T1_1 = (int) ((float)(buehlerPeriod)/4) + stepPhaseTime;
404     T2_1 = buehlerPeriod - T1_1;
405
406     T1_2 = (int) ((float)(buehlerPeriod)/4) - stepPhaseTime;
407     T2_2 = buehlerPeriod - T1_2;
408
409   }
410
411   //Get current time
412   gettimeofday(&currTime, NULL);
413
414   //Caculate the current buehler phasor
415   unsigned long long buehlerPhase = (unsigned long long)
((currTime.tv_    sec - startTime.tv_sec)*1000000 +
```

```
416                                     (currTime.tv_usec -
startTime.tv_us    ec)) % buehlerPeriod;
417   //printf("buehlerPeriod : %d, beuhlerPhase %d startTime %d
currTime %    d\n", buehlerPeriod, buehlerPhase, startTime,
currTimeInt);
418
419   //Get positions of both buehler cycles
420   if(buehlerPhase <= T1_1)
421   { P1 = ((int)
(((float)(1)*NUM_POS*buehlerPhase)/(3*buehlerPeriod)))     %
NUM_POS;
422     P2 = ((int)
(((float)(5)*NUM_POS*buehlerPhase)/(3*buehlerPeriod)))     %
NUM_POS; }
423   else if (buehlerPhase < T2_1)
424   { P1 = ((int) ((((float)(5)*NUM_POS*(buehlerPhase-
T1_1))/(3*buehlerPe    riod)) + ((float)(1)*NUM_POS)/12)) %
NUM_POS;
425     P2 = ((int) ((((float)(1)*NUM_POS*(buehlerPhase-
T1_1))/(3*buehlerPe    riod)) + ((float)(5)*NUM_POS)/12)) %
NUM_POS; }
426   else
427   { P1 = ((int) ((((float)(1)*NUM_POS*(buehlerPhase-
T2_1))/(3*buehlerPe    riod)) + ((float)(11)*NUM_POS)/12)) %
NUM_POS;
428     P2 = ((int) ((((float)(5)*NUM_POS*(buehlerPhase-
T2_1))/(3*buehlerPe    riod)) + ((float)(07)*NUM_POS)/12)) %
NUM_POS; }
429
430   //printf("buehlerPeriod: %d, buehlerPhase: %llu, T1: %d,
T2: %d, iPos    [0]: %d, iPos[1]: %d\n",buehlerPeriod,
buehlerPhase, T1, T2, pos[0], po    s[1]);
431
432   //Adjust position with offset and direction
433   if(dir == 'B')
434   { pos[0] = pos[2] = pos[4] = NUM_POS - (P1 + (NUM_POS -
OFFSET1)) % N    UM_POS;
435     pos[1] = pos[3] = pos[5] = NUM_POS - (P2 + (NUM_POS -
OFFSET2)) % N    UM_POS; }
436   else
437   { pos[0] = pos[2] = pos[4] = (P1 + OFFSET1) % NUM_POS;
438     pos[1] = pos[3] = pos[5] = (P2 + OFFSET2) % NUM_POS; }
439
440   return(currTime);
```

```
441 }
442
443 void walk_turn(int rpm, char dir, int numSteps, int angles,
int oddstep    s/* int phase_off, int stepPhaseTime, char
turn_dir*/)
444 {
445   struct timeval tv = {.tv_sec = 0, .tv_usec = 0};
446   int ipos[6] = {0}, mpos[6] = {0}, duty[6] = {0},
done[6]={0};
447   int stepNum = 0, lpos = 0;
448
449   //if (dir != 'L' && dir != 'R')
450   //{
451   //  printf("INCORRECT DIRECTION L OR R");
452   //  return;
453   //}
454
455   if (numSteps == 0)
456     dir = 'U';
457
458   //if (dir == 'R')
459   if (dir == 'U')
460     tv = walks_turn_buehler(rpm,dir,ipos,angles,oddsteps);
461   else
462     numSteps = numSteps + 1;
463   //else
464     //numSteps = numSteps;
465
466   while(stepNum < numSteps)
467   {
468     tv = walks_turn_buehler(rpm,dir,ipos,angles,oddsteps);
469     readAllMotorPos(mpos);
470     PD(&tv,ipos,mpos,duty);
471     driveAllMotors('H',duty);
472
473    //printf("Desired = %d, Acutual = %d\n",ipos[4],
mpos[4]);
474     if(dir=='L')
475     {
476       if((((OFFSET1 < (*(ipos+3))) && (lpos <= OFFSET1))) ||
((OFFSET2     < (*(ipos+3))) && (lpos <= OFFSET2)))
477         stepNum++;
478       lpos = *(ipos+3);
479     }
```

```
480     else
481     {
482       if((((OFFSET1 /*>*/< (*(ipos/* + 3*/))) && (lpos
/*>*/<= OFFSET1)    )) || ((OFFSET2 /*>*/< (*(ipos/* + 3*/))) &&
(lpos /*>*/<= OFFSET2)))
483         stepNum++/* = stepNum + 2*/;
484       lpos = *(ipos/* + 3*/);
485     }
486
487 /**************
488     else
489     {
490       if((((OFFSET1 < (*ipos)) && (lpos <= OFFSET1))) ||
((OFFSET2 < (*    ipos)) && (lpos <= OFFSET2)))
491         stepNum++;
492      lpos = *ipos;
493     }
494 *************/
495   }
496   //printf("Desired = %d, Acutual = %d\n",ipos[0], mpos[0]);
497   //Ensure motors are stopped
498   stopMotors();
499   return;
500 }
501
502 struct timeval walks_turn_buehler(int rpm, char dir, int*
pos, int angl    es,  int oddstepss)
503 {
504 //  printf("EnteringBuehler");
505   static struct timeval startTime = {.tv_sec = -1, .tv_usec
= -1};
506   struct timeval currTime;
507   static int buehlerPeriod;
508   static int T1, T2;
509   int P1, P2;
510
511   if (angles < 0)
512     angles = 0;
513   else if (angles > 10)
514     angles = 10;
515   else
516   {}
517
518   if(startTime.tv_sec == -1)
```

```
519   {
520     gettimeofday(&startTime,NULL);
521     buehlerPeriod = (int) (((float)(60)/rpm)*1000000);
522     T1 = (int) ((float)(buehlerPeriod)/4);
523     T2 = buehlerPeriod - T1;
524   }
525
526   gettimeofday(&currTime, NULL);
527
528   if (dir == 'U')
529   {
530     startTime = currTime;
531     startTime.tv_sec = currTime.tv_sec;
532     startTime.tv_usec = currTime.tv_usec;
533   }
534
535   unsigned long long buehlerPhase = (unsigned long long)
((currTime.tv_    sec - startTime.tv_sec)*1000000 +
536                                       (currTime.tv_usec -
startTime.tv_us    ec)) % buehlerPeriod;
537
538   if (oddstepss == 1 && dir != 'U')
539   {
540     buehlerPhase = (buehlerPhase + (buehlerPeriod/2)) %
buehlerPeriod;
541   }
542
543   //printf("buehlerPeriod : %d, beuhlerPhase %d startTime %d
currTime %    d\n", buehlerPeriod, buehlerPhase, startTime,
currTime/*Int*/);
544
545   if (buehlerPhase <= T1)
546   { P1 = ((int)
(((float)(1)*NUM_POS*buehlerPhase)/(3*buehlerPeriod)))     %
NUM_POS;
547     P2 = ((int)
(((float)(5)*NUM_POS*buehlerPhase)/(3*buehlerPeriod)))     %
NUM_POS;}
548   else if (buehlerPhase < T2)
549   { P1 = ((int) ((((float)(5)*NUM_POS*(buehlerPhase-
T1))/(3*buehlerPeri    od)) + ((float)(1)*NUM_POS)/12)) %
NUM_POS;
```

```
550     P2 = ((int) ((((float)(1)*NUM_POS*(buehlerPhase-
T1))/(3*buehlerPeri    od)) + ((float)(5)*NUM_POS)/12)) %
NUM_POS;}
551   else
552   { P1 = ((int) ((((float)(1)*NUM_POS*(buehlerPhase-
T2))/(3*buehlerPeri    od)) + ((float)(11)*NUM_POS)/12)) %
NUM_POS;
553     P2 = ((int) ((((float)(5)*NUM_POS*(buehlerPhase-
T2))/(3*buehlerPeri    od)) + ((float)(07)*NUM_POS)/12)) %
NUM_POS;}
554
555
556   // printf("buehlerPeriod: %d, buehlerPhase: %llu, T1: %d,
T2: %d, iPo    s[0]: %d, iPos[1]: %d\n",buehlerPeriod,
buehlerPhase, T1, T2, pos[0], p    os[1]);
557
558
559   if (dir == 'L')
560   {
561     pos[0] = pos[2] = (P1 + OFFSET1) % NUM_POS;
562     pos[4] = ((P1 + OFFSET1) + (500*angles)) % NUM_POS;
563     pos[3] = pos[5] = (P2 + OFFSET2) % NUM_POS;
564     pos[1] = ((P2 + OFFSET2) - (500*angles)) % NUM_POS;
565   }
566   else if (dir == 'R')
567   {
568     pos[0] = pos[2] = (P1 + OFFSET1) % NUM_POS;
569     pos[4] = ((P1 + OFFSET1) - (500*angles)) % NUM_POS;
570     pos[3] = pos[5] = (P2 + OFFSET2) % NUM_POS;
571     pos[1] = ((P2 + OFFSET2) + (500*angles)) % NUM_POS;
572   }
573   else
574   {};
575 //  printf("exitingBuehler");
576   return (currTime);
577 }
578
579
580
581 void stair(int rpm, int numSteps/* int phase_off, int
stepPhaseTime, ch    ar turn_dir*/)
582 {
583   struct timeval tv = {.tv_sec = 0, .tv_usec = 0};
```

```
584   int ipos[6] = {0}, mpos[6] = {0}, duty[6] = {0},
done[6]={0};
585   int stepNum = 0, lpos = 0;
586   int off_R, off_M, off_F;
587
588   off_R = ((int) ((float)(1)*NUM_POS)/6)%NUM_POS;
589   off_M = ((int) ((float)(1)*NUM_POS)/5)%NUM_POS;
590   off_F = ((int) ((float)(3)*NUM_POS)/4)%NUM_POS;
591
592   printf("%d\n, %d\n, %d\n", off_R, off_M, ((OFFSET1 +
off_F)%NUM_POS))    ;
593
594   moove(10,'F','B',(off_R+OFFSET1)%NUM_POS);
595   moove(10,'F','M',(off_M+OFFSET1)%NUM_POS);
596   moove(10,'F','F',(off_F+OFFSET1)%NUM_POS);
597
598   while(stepNum < numSteps)
599   {
600     tv = stair_buehler(rpm,ipos);
601     readAllMotorPos(mpos);
602     PD(&tv,ipos,mpos,duty);
603     driveAllMotors('H',duty);
604    //printf("Desired = %d, Acutual = %d\n",ipos[4],
mpos[4]);
605     {
606       if(((((((off_F+OFFSET1)%NUM_POS)-1000) < (*ipos)) &&
(lpos <= (((    off_F+OFFSET1)%NUM_POS)-1000)))))// || ((OFFSET2
> (*ipos)) && (lpos >=     OFFSET2)))
607       {
608         stepNum++;
609         printf("Forward Motor: %d\n",(*ipos));
610         printf("Forward Motor(L): %d\n",(lpos));
611         printf("%d\n", stepNum);
612       }
613       lpos = *ipos;
614     }
615   }
616
617   //printf("Desired = %d, Acutual = %d\n",ipos[0], mpos[0]);
618   //Ensure motors are stopped
619   stopMotors();
620   return;
621 }
622
```

```
623 struct timeval stair_buehler(int rpm, int* pos)
624 {
625 //  printf("EnteringBuehler");
626   static struct timeval startTime = {.tv_sec = -1, .tv_usec
= -1};
627   struct timeval currTime;
628   static int buehlerPeriod;
629   static int T1, T2;
630   int P1, P2, P3;
631
632   if(startTime.tv_sec == -1)
633   {
634     gettimeofday(&startTime,NULL);
635     buehlerPeriod = (int) (((float)(60)/rpm)*1000000);
636     T1 = (int) ((float)(buehlerPeriod)/4);
637     T2 =  buehlerPeriod - T1;
638   }
639
640   gettimeofday(&currTime, NULL);
641
642   unsigned long long buehlerPhase = (unsigned long long)
((currTime.tv_    sec - startTime.tv_sec)*1000000 +
643                                      (currTime.tv_usec -
startTime.tv_us    ec)) % buehlerPeriod;
644
645   //printf("buehlerPeriod : %d, beuhlerPhase %d startTime %d
currTime %    d\n", buehlerPeriod, buehlerPhase, startTime,
currTime/*Int*/);
646
647   if (buehlerPhase <= T1)
648   { P1 = ((int)
((((float)(7)*NUM_POS*(buehlerPhase))/(3*buehlerPeriod)    ) +
((float) (1)*NUM_POS)/6)) % NUM_POS;
649     P2 = ((int) ((float)(1)*NUM_POS)/6)%NUM_POS;
650     P3 = ((int) ((float)(3)*NUM_POS)/4)%NUM_POS;
651     //P3 = ((int)
((((float)(11)*(NUM_POS*buehlerPhase)/3*buehlerPeriod    )) +
((float) (1)*NUM_POS)/24))%NUM_POS;
652   }
653   else if (buehlerPhase < T2)
654   { P1 = ((int) ((((float) (1)*(NUM_POS*(buehlerPhase-
T1))/(2*buehlerPe    riod)) + ((float)(3)*NUM_POS)/4))) %
NUM_POS;
```

```
655     P2 = ((int) ((((float) (7)*(NUM_POS*(buehlerPhase-
T1))/(6*buehlerPe    riod)) + ((float)(1)*NUM_POS)/6))) %
NUM_POS;
656     P3 = ((int) ((((float) (5)*(NUM_POS*(buehlerPhase-
T1))/(6*buehlerPe    riod)) + ((float)(3)*NUM_POS)/4)))%NUM_POS;
657   }
658   else
659   {
660     P1 = ((int) (((float)  (2)*(NUM_POS*(buehlerPhase-
T2))/(3*buehlerPe    riod))))%NUM_POS;
661     P2 = ((int) ((((float) (5)*(NUM_POS*(buehlerPhase-
T2))/(3*buehlerPe    riod)) + ((float)(3)*NUM_POS)/4)))%NUM_POS;
662     P3 = ((int) ((((float) (7)*(NUM_POS*(buehlerPhase-
T2))/(3*buehlerPe    riod)) + ((float)
(1)*NUM_POS)/6)))%NUM_POS;
663   }
664
665   // printf("buehlerPeriod: %d, buehlerPhase: %llu, T1: %d,
T2: %d, iPo    s[0]: %d, iPos[1]: %d\n",buehlerPeriod,
buehlerPhase, T1, T2, pos[0], p    os[1]);
666
667   {
668     pos[0] = pos[3] = (P3 + OFFSET1) % NUM_POS;
669     pos[1] = pos[4] = (P2 + OFFSET1) % NUM_POS;
670     pos[2] = pos[5] = (P1 + OFFSET1) % NUM_POS;
671   }
672 //  printf("exitingBuehler");
673   return (currTime);
674 }
675
676
677
678
679 #endif
```