# Table of Contents

Abstract ........................................................................................................................................... 4
Keywords ........................................................................................................................................... 4
Introduction ...................................................................................................................................... 4
Method to Obtain Object ................................................................................................................ 4
Method of Ceiling Mobility ............................................................................................................. 25
Operating system – ROS ................................................................................................................ 32
Computer Vision ............................................................................................................................ 40
Graphical User Interface ............................................................................................................... 44
Kinect Control ............................................................................................................................... 47
System Level State Machine ......................................................................................................... 47
User Friendly Launch – The Launch File ....................................................................................... 49
Communication Protocol .............................................................................................................. 54
Conclusions and Future Work ...................................................................................................... 66
Acknowledgements ....................................................................................................................... 66
Appendix A – Gantt Charts ........................................................................................................... 71
Appendix B – Budget ..................................................................................................................... 73
Appendix C – Team Charters ........................................................................................................ 78
# Team Members

<table>
<thead>
<tr>
<th>Name</th>
<th>Email</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nelson Chu</td>
<td><a href="mailto:nchu1@ucsc.edu">nchu1@ucsc.edu</a></td>
</tr>
<tr>
<td>Thomas Gilbert</td>
<td><a href="mailto:thgilber@ucsc.edu">thgilber@ucsc.edu</a></td>
</tr>
<tr>
<td>Tony He</td>
<td><a href="mailto:tche@ucsc.edu">tche@ucsc.edu</a></td>
</tr>
<tr>
<td>Kyle Huey</td>
<td><a href="mailto:kyahuey@ucsc.edu">kyahuey@ucsc.edu</a></td>
</tr>
<tr>
<td>Henry Lau</td>
<td><a href="mailto:hlau3@ucsc.edu">hlau3@ucsc.edu</a></td>
</tr>
<tr>
<td>Joel Thompson</td>
<td><a href="mailto:jjthomps@ucsc.edu">jjthomps@ucsc.edu</a></td>
</tr>
<tr>
<td>Phillip Wong</td>
<td><a href="mailto:pwong6@ucsc.edu">pwong6@ucsc.edu</a></td>
</tr>
<tr>
<td>Behnam Zohoor</td>
<td><a href="mailto:bzohoSor@ucsc.edu">bzohoSor@ucsc.edu</a></td>
</tr>
</tbody>
</table>
Abstract:

We have developed a robotic infrastructure to aid the elderly and persons with disabilities. Our approach utilizes the ceiling as a platform for robotic mobility which greatly reduces the amount of obstacles which is the main complication that traditional robots face. This paper focuses on the technical aspects of how we constructed our robot, E.R.I.C.A. (Enhanced Robotic Integrated Ceiling Arm).


Introduction:

The objective of this project is to design and construct a semi-autonomous system that can aid the elderly and persons with disabilities with daily chores as well as removing their ‘reach’ limitations. This system will allow for human-robot interactions while minimizing the need for human-like navigation. By introducing the method of robot mobility on the ceiling, we are able to focus less on collision control, and more on the true objective of an assistive robot, helping people. The key insight underlying this proposal is that while the floor environment is challenging, the ceiling is clear, clean, and static. A robot moving on the ceiling would mitigate many of the difficulties that have typically been encountered by those on the floor. Our current project is mainly a proof of concept that we are able to locate an object, find its location, calculate a best path, and pick up the object. We approached this project by first identifying the necessary components needed to achieve our overall objective. We are then able to break down the project into sections that can be completed and integrated together. The project was broken down into three groups, a group in charge of the robotic arm, robot ceiling mobility, and computer vision and communication.

Method to obtain object:

The method we choose to accomplish the task of picking up an object is by using a robotic arm. The arm has four degrees a freedom. The top segment, middle segment, bottom segment, and a rotation. Attached to the bottom segment is a universal jammer gripper which allows us to be able to pick up theoretically any shaped item.
Design

Originally, we designed the arm based off an elephant trunk. The reason we initially decided on an elephant trunk was the near infinite degrees of freedom. It allows us to theoretically reach any spot location within the radial hemisphere. However, after doing a trade study, we changed the arm design into a more conventional arm with four degrees of freedom. The elephant trunk required more torque per motor with higher cost. The conventional arm allows less torque required as well as less motor usage.
Flow chart

The flowchart for the arm is shown in the figure above. The microcontroller will get coordinates from R.O.S. and compute the duty cycle needed to reach a certain position and send it to the H-bridges to drive the motors. The motors shaft overall position will be tracked with optical encoders. That data will go into a feedback control system and the microcontroller will adjust the duty cycle to keep its position. The motor will also drive the gears to move the arm and the gripper. The arm bars position will be tracked using magnetic encoders. That data will be sent to the Feedback control which adjusts the duty cycle to the motors. Sonar sensors will be triggered if an object is too close to the arm. Due to the way the rack was built, we decided to remove this feature because of false positives read from the posts. The pressure sensor will read the pressure the gripper is applying to the object. Once an object applies enough pressure, the gripper will turn on allowing a good grip on the object. The arm will constantly talk back to R.O.S. reporting its status.

Components/Hardware

To find the materials for the arm, we use the stress formula:

\[
\frac{\text{Stress}}{\text{Area}}
\]

In parallel with the modulus of elasticity. We were able to find a table of modulus of elasticity for different elements using google search and were able to find which material would be able to hold the amount of force with the a small amount of bend. We found that aluminum was strong enough to support the way as well as have a small amount of bend.

Figure 4. Stress diagram on Arm Channels
In our project we use unpolished aluminum because of having less flexibility. So, increasing the weight on the arm it would not bend the metal. We have different way of bending. In our project because we are using the longer length of aluminum U channel for the length of the arm, we have vertical bending. By looking at the figure below we can see that the most force when the arm will start moving up it will be the end of the length. Therefore, each join must be really hard and less flexible. In the first segment because all the weight it is on the platform which motors attached to it, we decided to make those plates out of aluminum including arm material. For the arm we used U channel so easily we can make the wire throw the hole to reach the motors and power supply on the top. Our first segment needed the most torque and strongest dc motors but with less weight therefore, we came up with putting two big DC motor which they both connecting to the metal gears to attach to the arm and double the torque. Therefore, this way we have two motor putting the whole arm up. To even increasing the torque more we decide to by using gears reduce the speed and increase the torque. The figure above shows the stress on the channels with an applied force. Red is the higher stress while blue is the area with the least amount of stress.

![Figure 5. Lazy Susan](image)

Rotating section is done by a DC brushless motor and a Lazy Susan shown in the figure 5 above. The Lazy Susan is one the tools which we can attach two different platform and be able to turn easily from 0 to 360 degree. However, after calculating the amount of stress the U channels can handle before they bend, we concluded that the amount of torque caused by an object misaligned with the center of gravity can only successfully lift up an object of 5lbs within 23.6 degrees away from the origin on both sides. Since the rotational section’s sole purpose was to fine tune, we did not have a problem with over rotating.

Position Sensors

Feedback is a very important aspect in controlling the arm. Knowing where our arm is crucial in pinpointing where our arm must go and how it moves towards target location to accommodate this feature, encoders are required to acquire data. There were two types of encoders we looked at, optical rotary encoders and magnetic rotary encoders.
Optical encoders uses an LED, a light sensor either next to the LED the receives reflected light or on the other side of a spinning disk with pre-defined holes that allow the light to pass through the disk to detect that the motor is spinning as shown in figure 6 below.

![Figure 6. Basic model of a Rotary encoder](image)

Every time the sensor goes from no light to light shining on it, the sensor would send a high to the micro-controller. The benefit that optical encoders gave us is that they are simple to use. Plug them into the back-shaft, power the encoder and the sensor output can then be directly read through our micro-controller. The precision of these encoders will then depend upon the back-shaft of our motor. The more geared our motor, the more precise our encoders get due to having more clicks per revolution of the motor. A down side to optical encoders are that they do not record position but record distance. They tell us how far our motor has rotated but do not give us our original location. Another downside is finding the right size encoder that fits the back-shaft of our motor. We've found a few that can suffice for our motors however, they were priced around 40$. We threw out the idea of buying encoders and thought to make our own optical encoders since they only consisted of a LED, a plate, and a light sensor. This was our last resort as we did more research.

Aside from optical encoders, we also looked into magnetic rotary encoders specifically the as5040 by Austria Micro-Systems shown in Figure 7.
The AS5040 is a 10-bit contactless magnetic rotary encoder for accurate angular measurement over a full turn of 360°. It is a system-on-chip, combining integrated Hall elements, analog front end and digital signal processing in a single device. The angle can be measured by rotating a two-pole magnet over the center of the chip. The magnet can be on above or below the chip. The angular resolution that this chip provides is 0.35° per click thus gives us a total of 1024 positions or 10-bits per full revolution. The position of the magnet can be read digitally in two ways: one is to read it as a serial bit stream through pin 9 or to read it as a PWM signal through pin 12 as shown in figure 8.
The data output of pin 9 is of a Synchronous Serial Interface which the chipKit32 does not support so we decided to go with the PWM output of pin 12. The position is encoded into a pulse width modulated signal with 1μs pulse per step up to 1024μs for the full turn as shown in Figure 9. The pwm is outputted at a period of about 1ms.

![Figure 9. Pwm output](image)

As shown in the figure, the angle is in respect to the pulse width; the angle is higher when the pulse width is wider and the angle is lower when the pulse width is smaller. Since we know how to derive the angle from the signal, now we have to measure the distance between the rising and falling edge to determine the angle the current rotation is at.

One way to measure the pulse width was through software, specifically the input capture module that is part of the peripheral libraries. The input capture module works similar to the external interrupts however it is asynchronous compared to the external interrupt.

![Figure 10. Use of the input capture module](image)

As shown in Figure 10, we wrote the interrupt to initially fetch a rising edge and record time1. The program then looks for a falling edge and records the time when that has occurred, then it finally looks for another rising edge and records the third and final time. At this point it updates the length of the PWM when it was high by computing the difference between the falling edge time and the initial rising
edge time divided by the overall time (time3 \(\text{−}\) time1) which should be around 1ms. This should give us the fraction of the time when the PWM output was high and thus giving us the angle by multiplying this length by 360 degrees. At the same time it sets the final time3 to be the initial time1 and begins to wait for a falling edge and yet another rising edge to compute a new length. Because the period of the PWM is around 1ms, the angle should be updated pretty quickly and we can always get an accurate angle. However, at the time, the readings we got from using the input capture was decent at best. It would give us the correct angles however the angle would fluctuate greatly from time to time. For example, the magnet is rotated 180 degrees above the chip and kept there for a time. The terminal would output 180 degrees however it would jump to above 200 degrees or below 100 degrees for no reason. This was a problem I was unable to fix however I found the mistake that I have made and fixed it. But before I found my mistake, I had already scrapped the Input Module and decided to measure the pulse width another way.

The second was to convert the PWM signal into an analog signal which can then be read through the analog to digital converter pin. However the precision of the angle would then depend upon the resolution of the ADC. Ironically the resolution we worked with is 10bits thus we have 1024 positions to work with and 0.35 degrees per click. To convert the PWM signal, I ran the output through two low-pass filters as shown in Figure 11.

![Figure 11. PWM to Analog conversion](image)

By causing a delay in the signal, the PWM output would begin to change through the low-pass filters. Because the frequency of the PWM output is around 1kHz, or a period of 1ms, I didn't have to worry too much about adding a huge time constant. The recommended start values were 4.7k and 1μF,
however the output barely changed from its original square wave. With the limited parts, I replaced the 1μF with 10μF capacitors the output changed to exactly what I wanted as shown in Figure 12.

The analog output shown as channel 2 can now be read by the Analog to Digital pin of the pic32. As we rotate the magnet, the analog value changes as well depending on the current duty cycle or position of the magnet relative to the as5040. If the magnet is rotated to about 20% duty cycle, then the pic32 would convert the analog value and output the position to be around 72 degrees. Similar for 52% duty cycle, the terminal would output around 187 degrees and at 95% duty cycle, the position would be at 342 degrees. There would be a fluctuation of a plus or minus one degree due to noise however that is considered negligible in our calculations because it would not affect the position due to our control algorithm.
Sonar Sensors

To add some safety to our robotic arm, we incorporated the use of sonar sensors, specifically the XL-MaxSonar-Ez Series: MB1220.

![MB1220 Sonar Sensor](image)

Figure 12. MB1220 Sonar Sensor

The benefits of these Sonar sensors is that they have a range of up to almost 23 feet or 700 centimeters which is more than enough. The sensor is also designed to operate in the presence of noise such that the readings will not fluctuate or read a false positive even in the presence of periodic noise from motors. It also comes with an analog output that can also be read by our analog to digital converter. However with a supply of 3.3V and a scaling factor of (Vcc/1024) per centimeter, we have a resolution of 3.2mV/cm with a maximum range of 600cm or 19.68 feet. One negative aspect to note is that anything within 20cm is a no sensor deadzone. Anything within that region would be reported to be 20cm. However, what we are interested in is anyone within 1 to 3 feet of the sensor. Because we only have a resolution of about 6 bits from the sonar sensor, we wanted take full advantage of the 10 bits from the ADC. We decided to amplify the signal such that 3 feet would be the maximum value because we only care about people who wander too close to the arm and decided that 20cm to 3 feet is a good range to work in.

![Determines the gain to set 3ft as new maximum value(right) account for new minimum value due to amplification(left)](image)

Figure 13. Determines the gain to set 3ft as new maximum value(right) account for new minimum value due to amplification(left)

After some simple algebra, we found that a gain of 11 would decrease our maximum distance to about 3 feet, however when a signal is amplified, the minimum value is also amplified which we calculated to be 704mV as 20 cm rather than 64mV.
After passing the sonar sensor through the amplifier, we are able to get 10bit resolution of a distance of 20cm to 3 feet (91.5cm). Because we are setting a hysteresis bound to determine whether someone has gotten too close to the arm, we do not need to set a virtual ground and can leave the minimum distance to be 704mV which is read as the value 218 through the ADC.

Pressure Sensor

To determine whether we have pushed down far enough on the object to turn on the gripper, we decided to add a Force Resistive Sensor shown in Figure 15.
According to the graph, the FSR acts as a variable resistor such that the more force is applied, the less resistance is shown. To be able to get accurate data, we need to choose a resistor such that the current is limited and the sensitivity be as linear as possible.

We chose resistor Rm to be 10kΩ as it seems to linearize our output for the range we want to work in which is between 400 to 800g of force.
Torque Calculations

![Diagram of the force relative to the direction of the arm path](image)

Figure 17. Diagram of the force relative to the direction of the arm path

There is a constant force going downward from the pull of gravity, however the path of which the arm follows is not the same as the force applied. We can solve for the amount of gravity that is affecting the arm segment by creating a triangle based on the angle which the arm has moved in relative to the 90 degree position.

![Diagram of force vectors from gravity affecting the arm path](image)

Figure 18. Force vectors from the gravity affect the arm path

From figure 18 above, angle ‘a’ is the angle which the arm as moved in relative to 90 degrees. Angle ‘b’ is 90 degrees and angle ‘c’ is solved by simply summing up the total degrees equal to 180 degrees. Force of gravity is 9.8, by solving for the path of Arm, we are able to find how much gravity affects the arm depending on the angle it’s at. From this we are able to find how much torque is needed to move the arm depending where the arm needs to move. This portion of the math is mainly used in the software to determine the duty cycle needed to supply the torque required. When choosing the motors, we calculated the maximum torque needed, which is when the force from gravity fully affects the arm segment. From this, we use the torque formula:

\[ \tau = r \ (radius) \times F \ (force \ in \ newtons) \]

We took this number and put it through an online torque converter which allowed us to change the units to match the units the motors are listed in. The amount of maximum amount of torque needed
to lift itself as well as 5lbs was 8.7 Nm for the top segment and 36Nm for the middle segment. The torque that the top segment could provide was 9Nm at 24 volts, each motor supplying 4.5Nm worth of torque. We didn’t have a 24volt power supply so we were not able to reach the expected torque. Also due to the change in design and material, the amount of torque needed increased from the expected value.

The middle segment required 4Nm which was enough for only one motor to drive. The gear ratio was 1:1.25. There was an unexpected force that was constantly increasing the reverse torque going back into the pinion. This caused the gear that the pinion was driving to have a diagonal offset causing it to crack the motor housing. If we had more time and realized this problem earlier, we could have redesigned the arm to have only one stabilizer bar. This would’ve had made the middle segment much more robust. The figure below shows the unexpected force from the parallel arm operating the middle segment.

![Figure 19. Force diagram of the arm bars](image)

The left arm caused more damage than benefits. Since the gripper was connected to that arm, the weight gets distributed and the center of mass has an offset away from the upward force of the motor. We know from Newton’s third law, that there is always an equal and opposite force. The figure below shows the opposite forces acting back into the gears.

![Figure 20. Force acting back on the motor drive](image)
Vector ‘a’ is the force from gravity, vector ‘b’ is the force from gravity from the gripper and the vector ‘c’ is the force from gravity from the parallel arm. With two of the three forces going back in being diagonal, it pushes the gear at an angle as more the motor produces more torque to lift the object. As the gear is more misaligned, it pushes against the motor housing which causes the housing to break as shown in the figure below. A better design would have been to only use one arm to lift objects up, this way the center of gravity will always be aligned with the upward force of the motor. It also mitigates any forces going back into the gear to cause it to misalign and ultimately increase the usage of torque.

The gear ration used on the top segment is 1:3. The smaller value being the value of the pinion. This allowed us to have a more effect torque usage on the way up, however the downside of this was the decrease in speed as well as a higher reverse torque going back into the pinion. This caused the top segment to fall once the power gets turned off. This isn’t much of a problem since we have a PID controller constantly checking for the error of its current position as well as the desired position. Having a smaller pinion gear ratio also allowed us to have more resolution per tick from the motors. We were able to get pin point on the coordinates before the offset of the gripper.

In the original design, there was a bottom segment. The bottom segment required 25.32Nm of torque. The main purpose of this segment is to fine tune the position of the arm incase we overshoot or undershoot our position. However, since we were able to get +/- 3mm away from the desired coordinate, we decided to get rid of the bottom segment entirely. This allowed less torque usage from the top segment and middle segment. It also allowed us to have a smaller power budget and decreases the cost for the arm by almost $150.

The last segment is considered the “wrist” of the arm. It consists of a single degree of rotation for the gripper to hang from to allow a greater reach for the arm as well as keeping it on the gripper facing downward. With a target weight of 5 lbs we aimed to allow the gripper to rotate downward

The main goal for the wrist was to keep the gripper pointing downward. The first component of this design required that we use an accelerometer to know the orientation on the gripper at any time. With this Information we could control a DC motor that could move the wrist using a gear attached to the arm. We originally had the wrist set up to use a stepper motor because it had a significant amount of torque and would allow the wrist to move the gripper out of the way when necessary. The wrist was over a pound heavier than the target of 5 pounds with this motor. To reduce the weight we compromised torque for the smaller DC motor and decided that the wrist only needed to remain pointing down at all times. We also cut out pieces of the wrist wall to minimize the material weight from the acrylic.

To mount the gripper to the wrist, we slid the shaft of the gripper through the two-layered base. Even though this added a bit more weight, it was worth the cost to keep the gripper stable and ensure that the writ would not break under the weight of the gripper and the added weight of the target object (up to 5 pounds). With this relatively high weight requirement, we included holes and slots so that we could easily screw the walls of the wrist together for a secure hold without using heavy amounts of glue.
which would have made it difficult to take apart when inevitable changes had to be made. For ease of rotation, we used a bolt through each side of the wrist that passed through two walls (outer and inner), each of which had ball bearings. With the bolts held in place by the arm, we attached a gear to one of them so that a gear attached to the DC motor could rotate the wrist about the bolt’s axis.

Forward Kinematics

To derive the inverse kinematics, we first solve the forward kinematics by assigning our link frames and finding the Denavit–Hartenberg parameters (DH).

diagram 1: Link Frame assignment

After assigning the link frames, the DH parameters should be as follows:

<table>
<thead>
<tr>
<th>i-1</th>
<th>i</th>
<th>αi-1</th>
<th>ai-1</th>
<th>di</th>
<th>Θi</th>
<th>Joint limits</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Θ1</td>
<td>(-90, 90)</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>0</td>
<td>L1</td>
<td>0</td>
<td>Θ2</td>
<td>(-120, 120)</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>0</td>
<td>L2</td>
<td>0</td>
<td>Θ3</td>
<td>(-90, 90)</td>
</tr>
</tbody>
</table>

By finding the DH table we can solve the transformation matrices shown in diagram 2.
By solving the forward kinematics, we can now solve the inverse kinematics in two ways.

The first method is to solve the thetas geometrically and the second method is to use the transformation matrices and solving it algebraically.

### Diagram 2: Forward Kinematics Transformation Matrices

\[
\begin{align*}
T_{01} &= \begin{bmatrix}
\cos \theta_1 & -\sin \theta_1 & 0 & 0 \\
\sin \theta_1 & \cos \theta_1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix} \quad T_{12} &= \begin{bmatrix}
\cos \theta_2 & -\sin \theta_2 & 0 & L_1 \\
\sin \theta_2 & \cos \theta_2 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix} \\
T_{23} &= \begin{bmatrix}
\cos \theta_3 & -\sin \theta_3 & 0 & L_2 \\
\sin \theta_3 & \cos \theta_3 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\end{align*}
\]

\[
T_{03} = \begin{bmatrix}
\cos(\theta_1+\theta_2+\theta_3) & -\sin(\theta_1+\theta_2+\theta_3) & 0 & L_2\cos(\theta_1+\theta_2)+L_1\cos(\theta_1) \\
\sin(\theta_1+\theta_2+\theta_3) & \cos(\theta_1+\theta_2+\theta_3) & 0 & L_2\sin(\theta_1+\theta_2)+L_1\sin(\theta_1) \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

### Diagram 3: Geometrically showing the two solutions
To solve the inverse kinematics geometrically, we can either start with \( \theta_1 \) or \( \theta_2 \). Due to the limitations on our top segment, we decided to solve for \( \theta_2 \) first and solve \( \theta_1 \) relative to \( \theta_2 \). We are given a coordinate \((y,x)\) thus we would know the distance from the origin to the object.

\[
r = \sqrt{y^2 + x^2}
\]

**Solve for \( \theta_2 \):**

\[
r^2 = L_1^2 + L_2^2 - 2L_1L_2\cos(\Phi)
\]

\[
\cos(\Phi) = (L_1^2 + L_2^2 - r^2) / 2L_1L_2
\]

\[
\Phi = 180 - \theta_2
\]

\[
\cos(\theta_2) = (r^2 - L_1^2 - L_2^2) / 2L_1L_2
\]

let \( X_2 = \cos(\theta_2) \)

\[
Y_2 = \sin(\theta_2) = \pm \sqrt{1 - \cos^2(\theta_2)}
\]

\[
\theta_2 = -1 \times \text{atan2}(Y_2, X_2)
\]

diagram 4. Solves for theta 2

To solve for \( \theta_2 \), we use the fact that \( r \) is given to us, then by using the law of cosines, we solve for \( \Phi \). Because we know \( \Phi = 180 - \theta_2 \), we can solve the x-component and thus solving the y-component. The + and – in \( Y_2 \) refers to the two possible solutions that \( \theta_2 \) can be. This solution is based off our robotic arm facing upwards however it is actually facing downwards, and our y-axis is actually reversed by 180 degrees therefore we must multiply the solution by -1 for it to yield the correct solution or else it will be reversed.

diagram 5. Solving for Theta 1

\[
\theta_1 = \beta \mp a (+ \text{if } \theta_2 < 0) (- \text{if } \theta_2 > 0) \]

\[
\beta = \text{atan2}(y, x)
\]

\[
\cos(\alpha) = (r^2 + L_1^2 - L_2^2) / 2rL_1
\]

\[
a = \text{atan2}(\sqrt{1 - \cos^2(\alpha)}), \cos(\alpha))
\]

\[
\theta_1 = -1 \times \text{atan2}(y, x) \mp \text{atan2}(\sqrt{1 - \cos^2(\alpha)}), \cos(\alpha))
\]

Or

\[
a = \text{atan2}(L_2 \sqrt{1 - \cos^2(\theta_2)}, L_1 + L_2\cos(\theta_2))
\]

\[
\theta_1 = -1 \times \text{atan2}(y, x) \mp \text{atan2}(L_2 \sqrt{1 - \cos^2(\theta_2)}, L_1 + L_2\cos(\theta_2))
\]

To solve for \( \theta_1 \), there are two ways, both of which are shown in diagram 5. The first way is to find \( \alpha \) using the cosine rule and the second way is to complete the right triangle as shown in diagram 3 with the
red dashed lines and solve for $\alpha$ that way. The solution for both yields the same answer. Note that $\theta_1$, like $\theta_2$, also needs to be multiplied by -1 to account for the phase shift of 180 degrees by the y-axis.

**End Effector**

For the gripper at the end of the arm, we decided to use a universal jamming gripper mechanism. The “universal” comes from the gripper’s ability to pick a wide range of objects that other grippers would have to spend a long time observing in order to calculate the best way to approach the object or in some cases would not be possible by other grippers. The jamming gripper mechanism derives from the jamming characteristics of granular materials. A balloon membrane holds the granular material in a loose almost liquid like state as it is placed on the target object. Negative pressure removes the air from the balloon causing the granular material to transition to the jammed state. This causes the granular material to lock into place against all the other grains and hold on to the target object so that it can be picked up. Our original design included a 10” latex balloon encased by a funnel. We wrapped the end of the balloon around the edge and cheese cloth placed over it to keep the coffee grounds in. We then slipped a tube over the cheese cloth to keep it together. After a number of tests we found it to be fairly effective but it needed to be better sealed off to hold a grip on objects. The small size of the balloon also limited the sizes of objects that it could pick up. Our second iteration of the gripper used a larger balloon, a larger funnel, and a PVC fitting sealed with liquid silicone to better hold the negative pressure. This proved to be able to pick up larger and heavier objects. It did still consist of a funnel as the outer covering which looked unprofessional. We had also filled the balloon to the top with coffee grounds which caused the balloon to hold the grounds in place. We solved these problems by reducing the amount of grounds in the balloon and using ABS tubing to make the gripper look more professional as well as allowing more movement of the coffee grounds which optimized the gripping capabilities. Our Final gripper was able to pick up a five pound dumbbell. The only change from the previous versions of the gripper was a tube glued on the inside to keep the balloon from being pulled out of the housing when it was gripping heavy objects such as the dumbbell. The connector from the balloon housing to the air tube required a PVC fitting to hold cheese cloth over the balloon outlet and seal the system by filling the bottom side with liquid silicone. We were able to screw an air tube nipple onto the end of the PVC fitting where the gripper connected to the air valve system. To cover and protect this piece of the gripper, we used a two inch ABS tube. This tube was able to fit into the wrist and be the connection between the gripper and the wrist.

**Air Valve system**

In order for us to generate both positive and negative pressure inside the gripper, we used a 12V DC air pump and four solenoid air valves. In the diagram of the air valve system below, you can see that when the pump is on (signal 1), air flows into the right valve of the pump and out of the left pump. The valves however are normally closed so when only the pump is on, no air flows because there is no path. When one of the valve pairs (2 & 3) opens up, the system will either remove air from the gripper (Signals 1 & 2) or fill up the gripper with air (Signals 1 & 3).
In order to control the power to the valves and the pump, we designed a switching transistor circuit. There were three transistors which could sync current through the pump or one of two pairs of valves. The transistors required a forward voltage of only 2.4V while the UNO32 supplied an output voltage of 3.3V. To minimize the current flow from the microcontroller, we put a voltage divider between the output and the base. The circuit diagram (FIGURE) shows that the transistor board contains the three transistor, the three voltage dividers, the input voltage source (12V), and three output terminals for the components’ power.

Figure 21. Flow chart for the gripper

Figure 22. State machine for the arm
State Machine

Idle State:

The program stays in the idle state until it receives the coordinates from R.O.S. twice. When it has received the coordinates it will run through the loop to check the coordinates to itself. If it is the same, meaning that it isn’t garbage data, it will exit the state.

Retract State:

If the program received a command to retract, it will control the arm to move into a retracted position. This was not implemented because the sole purpose of this state is to go through doorways into the next room.

Reset State:

This state is to reset the Arm back to its original position. This is used when after the arm has successfully picked up its item, on user command, or an unexpected sonar trigger.

Math State:

In the math state, the program takes the received coordinates and calculates the angle each segment has to be in. Once the math has been complete it will go into the Select path state.

Select Path State:

Since two possible paths are generated, this state choose which is the better path given our physical constraints. When the path is chosen, it will leave this state and move onto the traverse state.

Traverse:

This is the state where the arm starts to move to the desired position. Since the table acts as an obstacle, a certain set of motions needs to be completed before the arm can reach its desired position. The program will compare the position that the arm wants to go and determine which set of motions to complete before going to the desired position without hitting the table. The program will leave the traverse state in two conditions: if the sonar sensor is triggered or if it has successfully reached its destination. If the sonar sensor is triggered it will go into the sonar state, if the arm successfully accomplished its task, it will go to the push down state.

Sonar State:

This state is meant for unexpected sonar triggers. This portion of the state machine isn’t implemented due to constant false readings from the support poles. In this state, the program will reach the encoders and return those values to the motors. It will then go into the stop state.
Stop State:

In the stop state, the motors will read the encoder values and keep its position. After a certain time frame the program will go into the reset state.

Push Down State:

In the push down state, the program will slowly lower the angle of both the top and middle segment until the pressure sensor threshold is exceeded. It will then keep its position and enter the activate gripper state.

Activate Gripper:

In this state, the arm will turn the gripper on. If for any reason the pressure is below the threshold set, it will continue to press down until the pressure is exceeded again. Once a certain timer is exceeded the program will leave this state and proceed to the pick-up state.

Pick Up state:

The pick-up state will move the arm back up to the original desired position and perform the opposite maneuvers that it took to reach that position without hitting the table. Once that is complete it will then go back to the reset state.

Method to Ceiling Mobility:

The method to carry the arm around the room includes two motorized vehicles driving on an elevated track. These vehicles will receive will go to a given coordinate where the arm can reach out to pick up the selected object. The system consists of a cruciform rail on which the robotic arm travels around the room. Two motor housings roll along the rail and attach to a bogey holding the system’s power, and various electrical components.

Design

The rail is a cruciform design, and extends out three feet on the longer ends and one foot on the shorter ends. The portion the robotic arm travels on is one-half inch thick aluminum L-channels, which are three inches on each end. The design choice for aluminum was so the rails could support the weight of the system without cracking or sagging significantly. The center pieces are also one-half inch thick aluminum and contain rounded edges to prevent the system from getting caught if it were to go off-center. Grease was also added to the sides of the center pieces to increase the slickness of the sides of the center and prevent grinding and scraping. The space between the rails is approximately 6 inches and .6-.7 inches in the center. The frame around the rails is two inch by six inch pieces of wood and the rail is raised up eight feet using four inch by four inch wooden posts. Since we could not install the rails on the ceiling, the wooden posts were used and wood pieces were nailed across the top to prevent sagging in the center of the rail. The bottom of the rail system contains the tracking system, which
allows the bogey to localize when it is started and track its position with accuracy of up to a few centimeters.

The tracking system is located on the bottom of the rails. The tracking consists of one rail containing a single color which is read by three tracking sensors on one side of the bogey. The other rail contains a white color with two black strips which denote when the bogey has reached the center of the rail system, such that one of the motor housings is aligned with the center of the other rail.

The original design was rather than two black stripes to denote the center, the bottom of the rail contained various shades of gray and alternated gray shades with white. This allowed for improved tracking and quicker localization, however, upon testing, the box rocked side to side and caused the tracking sensor values to vary too much, so localization would be unreliable.

Possible improvements to the tracking system would be to scrap the tracking sensors and implement RFID tags along the rail instead of black stripes. This would decrease the time for localizing and improve the reliability of tracking because the lighting conditions and distances affected the tracking sensor readings.

The two motor housings are one-quarter inch aluminum rectangular boxes, which hold two twelve-volt brushed motors with encoders, which count have an accuracy of 1,802 counts per revolution. The top of the box has an acrylic top with four wheels angled out thirty degrees to correct for the bogeys from travelling in a not straight direction. Each of the motors attached to a 54-millimeter inline skate wheel. The bottom of each of the motor housings contain two rectangular holes for wiring of the motors to go to the microcontroller and the H-bridges and a circular hole in the center for a three-eighths inch bolt to extend from the motor housings and attach to the bogey.

The original design of the motor housings was a hexagonal shape and accounted four additional free-spinning wheels. The four free-spinning wheels were removed because they caused the housings to drag and caused the motors to drive in a not straight direction. The shape was changed because there was no benefit to the hexagonal shape after removing the free-spinning wheels and to save money when purchasing the steel pieces. The wheels assisting with the straight driving were originally placed to push against the L-channeled rails; however, they were later flipped to above the motor housings to push against the wood frame because when pushing against the rail, the wheels would get caught against the metal rectangular boxes.

A three-eighths inch carriage bolt that was eight inches long was placed through each of the motor housings and attached to the bogey at each end. On the bolt, we attached a copper ring to the top so if the bolt bumped into the center portion of the rail, the copper ring would rotate, but the motor housings would not. The wires from the motors within the motor housings were also hot glued to the bolt, so they would not rub against the rail when the bogey is travelling around the rail.

An improvement for the motor housings would be to decrease the size of the rectangular boxes to lower the cost to build them, and to weld or design the hole so the carriage bolt would not move around when the bogey is moving around the rail. Also the acrylic top of the box could be designed to
be added and removed more easily because the tab-and-slot design did not sit flush because of imperfections in cutting the metal pieces of the rectangular box.

In order to determine the location of the bogey on the rail we implemented a grayscale tracking system. We placed grayscale colors on the bottom of the rail and six grayscale sensors on the top of the box to read the colors above the box. The position of the sensors can be seen in the diagram below where the red boxes represent the location of the sensors:

![Figure 23. Location of tracking sensors](image1)

Originally we were going to use these values to determine absolute position by resetting the encoder values each time a certain color was found. We found this was not possible because as the box swayed even a little the output of the sensor would change drastically. Instead we used the colors to determine which rail the bogey was on and to determine whether or not it was on the center of the rail. To do this we needed two different strips of colors on each side of the rail as shown in the picture below:

![Figure 24. Tracking system](image2)

You can see in this picture that there are two different patterns on the rail. The bottom pattern is a light grey which is used to determine which rail the bogey was on. In this picture the light grey color
indicates that the bogey is on rail x, rail y has a dark black color. The top pattern is used to determine when the bogey is in the center of the rail. When two of the grayscale sensors are on the black strips the bogey is in the middle of the rail with the wheels locked into the gap. This was extremely helpful for the state machine because we were able to check the sensors for a threshold value to determine when the bogey was exactly in the center.

**State Machine:**

In order to make the bogey run reliably along the rail all four of the wheels had to run at the exact same speed. This caused us many problems before we put the side wheels on it. In order to make the motors run at the same speed we used PID (proportional integral derivative) controller with logic. This allowed us to give the bogey a speed value and the controller would make the motors go the appropriate speed and the whole system would move straight at the same speed.

We used a hierarchical state machine for the bogey to perform its task. Below are state diagrams with descriptions of each state and variables used:

Outer states:

![State Diagram](image)

**Figure 25. Outer State**

**INIT** - This state performs initialization when the system is turned on. The bogey can be anywhere on the rail and at the end of initialization it will be on rail y at position 17 (center of the rail with the wheels in the gap).

**WAIT** - This state waits for new commands from the end effector, when a new location is detected the state machine moves to the move state.
MOVE - While in this state the goal is for the bogey to get to the desired position given by the end effector.

currRail - The current rail that the bogey is on.

currPos - The current position of the bogey.

goToPos - The position given by the end effector.

newLocation - A flag that is set high when a new instruction is received.

Needtoinit - A flag that is set high when the system needs to initialize again for any reason.

Inner state machine INIT:
WHICHRAIL - Checks the value of the tracking sensors to determine which rail the bogey is currently on.

GOTOCENTER_X - The bogey is on rail x and needs to go to the center.

GOTOCENTER_Y - The bogey is on rail y and needs to go to the center.

ONCENTER_X - The bogey is on the center and on rail x.

ONCENTER_Y - The bogey is on the center and on rail y.

FRONT90_X_INIT - The front bogey on rail x does a 90 degree turn.

TANTURN_X_INIT - The bogey does a tangent turn from rail x to rail y.

REAR90_X_INIT - The rear bogey on rail x does a 90 degree turn.

currRail - The current rail that the bogey is on.

TS[number] - Tracking sensor #[number]

Inner state machine WAIT:

![Diagram of WAIT state machine]

Figure 27. Inner State WAIT

This inner state just waits for newInstruction to go high and then switches the outerState to MOVE.
Inner state machine MOVE:

FROM_WAIT - The decision state; chooses which state to go to after coming from WAIT or GOTO.

FRONT90_Y - The front bogey on rail y does a 90 degree turn.

FRONT90_X - The front bogey on rail x does a 90 degree turn.

TANTURN_Y - The bogey does a tangent turn from rail y to rail x.

TANTURN_X - The bogey does a tangent turn from rail x to rail y.

REAR90_Y - The rear bogey on rail y does a 90 degree turn.

REAR90_X - The rear bogey on rail x does a 90 degree turn.

GOTO_X - The bogey traverses rail x until it reaches the desired position.

GOTO_Y - The bogey traverses rail y until it reaches the desired position.

currRail - The current rail that the bogey is on.

currPos - The current position of the bogey.

goToPos - The position given by the end effector.

Needtoinit - A flag that is set high when the system needs to initialize again for any reason.
Operating System - ROS:

When working on our project, one of the more difficult portions to approach and develop is writing the software. We had a wide range of options that we could have worked with ranging from the programming language to write the software into the graphical user interface (GUI). Aside from choosing the method of implementation, the software code itself was huge and it was expected to require multiple header files with different functions to develop a good control of our robot with a specific purpose. In order to assist in programming a working robot easier, a company called Willow Garage created and continues to work on the open-source Robot Operating System (ROS). ROS was created in order to give hobbyists and researchers the ability to program any robotics project using predefined functions and methods to control it. ROS offers us a choice of working in either C++ or Python programming languages to use their software's libraries and tools. It is continuously being developed for a variety of manufactured robots but also offers many drivers that can be applied to certain components. Examples of these manufactured robots and components include the PR2, an out of box research and development platform that requires limited hardware and software work, and the Xbox 360 Kinect Sensor, an motion and object sensing input device. These devices do have their own processing data types, but since ROS is open-source, Willow Garage and users are able to continuously develop ROS compatible libraries that access the components and utilize their functionalities. Thus, in order to gain insight and understand how ROS will be useful in its application on our project, we will go over some of the key features of ROS that we applied.

The sole purpose of ROS is to provide hobbyists and researchers an easier method for robot development in a familiar language. Given that we have the proper background in programming, working in ROS will simplify the time it takes to debug problems and to complete our robot. ROS comes with many beneficial features for us such as language independency and simple integration to other robot development software. The communication infrastructure is designed such that we can use different ROS compatible languages to achieve the same goal. For example, if two members of a project each know different programming languages, one knows Python and the other knows C++, they can still split the software work and have each of their programs interact with each other to complete the project. ROS is also set up to easily integrate any code written for ROS to other robot software frameworks. If we are working in multiple software frameworks, we would be able to integrate their files over to work as we designed it in ROS.

In order to utilize ROS and its tools, we must be willing to work on a Linux operating system and in C++ or Python. Although ROS has the term “operating system” in it, it does not necessarily operate as a replacement for a Windows or Linux operating system; instead it only runs on a Linux operating system through the use of its terminal. To be specific, the recommended Linux system to operate ROS on is the Ubuntu operating system and its terminal. Since ROS is an open-source library and is continuously being developed, Willow Garage has created and released multiple versions per year since 2010 that are able to run on the latest version of Ubuntu. Since it runs on the Linux operating system, ROS will be controlled through the Linux terminal using both Linux and ROS commands. For example,
Normally to compile and run a program in Linux, we would use “cmake” and “./”, respectively. But once ROS is installed, we can use “rosmake” and “rosrun” for those operations.

Once ROS is fully installed and operating on its respective Ubuntu system, we had to choose whether we want to develop the project in C++ or Python. There are no immediate positives or negatives to choosing one over the other because of the language independency feature. We can also choose to use both C++ and Python for their projects depending on the application of the program they’re developing. For example, we may want to gain access and process data in C++ as data is sent and received from a device by using predefined C++ functions. However, we may also want the data or any other information to be displayed in a user friendly manner by using a GUI, which can be developed easier in Python. Once the ideal language was chosen, we can better understand ROS.

To begin working with ROS, we have to first install both Ubuntu and a corresponding ROS version. In order to determine the version of Ubuntu that is required, we should first visit the ROS at http://www.ros.org/wiki/ and click on Install to go to the installation page to choose the ROS version they desire. We will be directed to install the most recent version of ROS, ROS Groovy, which runs on Ubuntu 12.10, 12.04, and 11.10. Near the top of the page, a link labeled Fuerte Installation is available to install the previous version of ROS, ROS Fuerte. In a similar fashion to the top of the ROS Groovy page, the ROS Fuerte page will give options to versions before and after it. Figure 29 summarizes all the ROS versions with their Ubuntu versions.

<table>
<thead>
<tr>
<th>ROS</th>
<th>Ubuntu</th>
</tr>
</thead>
<tbody>
<tr>
<td>Diamondback</td>
<td>10.04, 10.10, 11.04</td>
</tr>
<tr>
<td>Electric</td>
<td>10.04, 10.10, 11.04, 11.10</td>
</tr>
<tr>
<td>Fuerte</td>
<td>10.04, 11.04, 12.04</td>
</tr>
<tr>
<td>Groovy</td>
<td>11.04, 12.04, 12.10</td>
</tr>
</tbody>
</table>

Figure 29. Corresponding ROS and Ubuntu versions

Once we choose our ROS version, we can then install one of the corresponding Ubuntu versions. To install Ubuntu, visit http://www.ubuntu.com/, click on Ubuntu Desktop, and navigate to the correct Ubuntu version. Note, it is possible to install Ubuntu within Microsoft Windows, but it is advised to install a full Ubuntu desktop by replacing the entire Windows operating system or dual booting the two to avoid computer hardware limitations such as speed and speed. For our purpose and throughout this paper, we will work with ROS Fuerte and Ubuntu 12.04.

With Ubuntu 12.04 installed, we will now be able to install ROS Fuerte. To install ROS, return to the ROS Fuerte installation page and click on Ubuntu as the platform. Open a terminal in Ubuntu and run these command lines successively:

```
sudo -c 'echo "deb http://packages.ros.org/ros/ubuntu precise main" > /etc/apt/sources.list.d/ros-latest.list'
wget http://packages.ros.org/ros.key -O - | sudo apt-key add -
sudo apt-get update
```
```bash
sudo apt-get install ros-fuerte-desktop-full

echo "source /opt/ros/fuerte/setup.bash" >> ~/.bashrc
.

source /opt/ros/fuerte/setup.bash

sudo apt-get install python-rosinstall python-rosdep
```

The first three command lines will set the Ubuntu source list to be able to find ROS, set up the keys, update the Ubuntu source list to ensure it connects to the ROS server. The next command line will actually install the desktop version of ROS Fuerte. The last four command lines will setup ROS such that every terminal opened is connected to it and install two commonly used ROS tools (or commands), rosinstall and rosdep. For details and fixing errors during the installation process, we can refer to the installation page. Once completed, ROS is fully installed on Ubuntu and we can begin looking at specific components and features through their tutorial at [http://www.ros.org/wiki/ROS/Tutorials](http://www.ros.org/wiki/ROS/Tutorials).

To begin working with ROS in detail with their command lines, we have to first give a general idea of how the whole operating system works. One of the first things to describe is that ROS will only operate within a specific directory that has been previously initialized to work with ROS and its commands. That is, we have to create a directory and initialize it with ROS by using the Ubuntu (or Linux) terminal. If this directory is not initialized with ROS, it can only be accessed in the terminal with basic Linux commands such as “cd” and “ls”, but not “roscd” and “rosls”. Once initialized, the directory will be known as our ROS environment (or ROS workspace), in which all work within it can utilize ROS functionalities and its commands. Looking at Figure 30, we can see a high level visual of this concept. The outer box is the general computer system, Ubuntu (or another Linux system), with Directory 1 and Directory 2 created within it as in Figure 30a. If we try to run any ROS commands for an item within either directory, an ROS error saying it cannot locate the file will occur. After we initialize Directory 2, as in Figure 30b, and run any ROS command for an item within Directory 2, the stated ROS command will operate as expected. If we try to run it for an item within Directory 1 in Figure 30b, an ROS error will still occur. This initialized workspace essentially creates one operating system within another i.e., ROS within Ubuntu.

![Figure 30. (a) Ubuntu with two non-ROS directories (b) Ubuntu with one non-ROS directory and one ROS directory](image-url)
In order to create a workspace for ROS, we will have to make a directory in which ROS can be initialized to. On the tutorial page, click on **Installing and Configuring Your ROS Environment** to create a working ROS environment. To create an environment, follow the website or these commands:

```
source /opt/ros/fuerte/setup.bash
roswsinit ~/fuerte_workspace /opt/ros/fuerte
mkdir ~/fuerte_workspace/sandbox
rosws set ~/fuerte_workspace/sandbox
source ~/fuerte_workspace/setup.bash
```

The first command is used just to ensure our terminal is connected to an ROS environment for the moment to create a user workspace. Note that this command should not be used after creating a workspace because these files (in directory `/opt/ros/fuerte`) are meant to hold important ROS files and its tools. The next command will initialize and create a directory named “fuerte_workspace” with the ROS files needed to allow us to use ROS tools outside of the `/opt/ros/fuerte` environment. The fourth and fifth lines create a sandbox, which is a program testing directory within the workspace, and sets the sandbox to be a part of the “fuerte_workspace” environment. The last command updates the current terminal if new workspaces have been made. Similar to the first command, the last command can be used when a new terminal is created to connect the terminal to the “fuerte_workspace” environment.

With a clear understanding of the ROS workspace, we can now introduce a few important structural components used within the workspace: packages, stacks, and manifests. A visual and conceptual representation of packages, stacks, and manifests is provided in Figure 3. ROS packages are uniquely defined units for organizing software when working with ROS. They are the lowest level of organization that ROS uses to hold important files from the C++ or Python source code to the program make files created when they are built. In most cases, we will want to create an ROS package and place source code into the package before compiling or going further with the software development.

![Figure 3. Concept of a package and stack within ROS](image.png)
ROS stacks are very similar to ROS packages except that they are uniquely defined directories that contain multiple packages with similar functionality. Stacks essentially organize multiple packages so we can find and run packages easier. They are not required throughout software development but are very useful when we want to submit our ROS programs to Willow Garage because they can test the user’s program by running the entire stack rather than running each package individually. Thus, they are the most common way for Willow Garage and ROS users to share their code for completed applications such as complete simulations or data processing programs.

As mentioned, ROS packages and stacks are uniquely defined directories that are created through ROS commands. When they are each created using their respective ROS commands, they each receive a unique file that define whether the directory is a package or stack. These unique files are called manifests, which are .xml files that contain descriptions of the packages and stacks. For packages, the manifest file is labeled “manifest.xml” and for stacks, the manifest file is labeled “stack.xml” as seen in Figure 30. The manifests of each component will contain all the dependencies of an ROS program to ensure they are linked correctly. Dependencies are ROS defined programs that allow us to work with certain features such as allow for ROS to contain and process all the C++ or Python libraries. These are different from header files, which are predefined libraries that must be included at the start of programs, because we can’t access dependencies. A very high level way to understand dependencies verse header files is dependencies are more like drivers required to run ROS properly while header files are user accessible functions for their program.

In order to create packages, stacks, and manifests, we need to navigate their terminal into the ROS workspace. In our case, we would want to navigate to “sandbox” because it is our testing space within the “fuerte_workspace” environment. The command lines to create them have the following format:

\[ \text{roscreate-pkg [package_name] [depend1] [depend2] [depend...]} \]
\[ \text{roscreate-stack} \]

The package command line states that we want to create a package with its name and all the specified dependencies included in it. The stack command line will create a stack of all the packages within the current directory. When ROS executes this command, the manifests with their description and other important compiling files, such as a make file, are created within each one.

Now that we know how ROS organizes user programs and files, we can begin to understand a few important concepts that make its communication infrastructure possible. The first concept we will want to understand is nodes. Nodes are processes that perform computation. They are essentially the executable files that are made after compiling ROS packages. The purpose of a node is for small scaled operation within a robot system and to communicate with other nodes. Since nodes were created for small scaled operation, they are meant to represent specific sections and components of robots in order to have better control of it. An example is having a node control a laser sensor and another node to control the wheel motors. Therefore, with the sole purpose of working at such a small scale, our project used many nodes that communicate with each other in order to fully utilize ROS.
To make a node in our code, we have to first include the ROS header file “ros/ros.h”. This header file contains everything we need to work in the ROS environment and set up our nodes for publishing and subscribing to topics. It contains multiple class members that are functions to operate the entire ROS computations. In our main, we have to initialize ROS with the ros::init function, in which we were also able to name our nodes to simplify debugging, and declare a node with the ros::NodeHandle namespace. This essentially declares a node for the ROS system to recognize during its computation and all publishing and subscribing can operate on a lower level through it. With these set up, our overall node and ready for the ROS environment, but it cannot do anything significant with what ROS has to offer. We have yet to dive into topics and how to send data between them. Note that to close nodes after executing them, we can use Ctrl+c to exit, which tells ROS to activate the ros::shutdown() exit function.

By creating a project that has multiple nodes and making them operate at such a small scale, the ROS communication infrastructure is being set. When nodes (or executable files) run, they get placed on an internal graph that allows nodes to communicate with other nodes on the graph. This graph that holds nodes is a very unique structure because it allows different programming languages to communicate with each other, such as a C++ programming sending and receiving messages to a Python program. In order to better understand this setup, ROS provides a GUI so we can visualize how their nodes are communicating to one another and debug any errors that may occur. A list of the currently running nodes and the GUI can be accessed by entering the following command lines:

```
sudo apt-get install ros-fuerte-rqt
rosnode list
rosrun rqt_graph rqt_graph
```

The first command line will install the program that contains the GUI, called rqt, into an ROS directory. The second line will display all the current nodes running in a list format within the terminal. The third line will run the GUI program and, assuming nodes are running, display nodes similar to Figure 32. In Figure 32, we see three different nodes, which means three programs are currently running with an attempt to communicate with another node.

![Graph of nodes](image)

**Figure 32. Graph of nodes**

The second concept that we need to understand to complete the communication infrastructure is topics. Topics are named channels in which nodes are able to send and receive messages through. To
send or receive a message, nodes have to publish or subscribe, respectively, to a topic. When we say publish we mean send out or output data to other nodes. When we say subscribe we mean to receive or look for data from other nodes. Topics do not have specific nodes that correlate to them; instead, nodes that publish a topic are anonymous to other nodes subscribed to that topic so that any interested in them can subscribe to them. If more than one node wants to subscribe to a topic’s message, it can without any issues in determining who is publishing it. For one node to be able to publish a topic and many nodes subscribe to one means that there can only be one node publishing that topic name. Figure 33 shows the nodes from Figure 32 publishing and subscribing topics. As shown, a topic is simply a line going from the publisher to the subscriber with the name (or topic) of it displayed above it. The node at the end of the arrow is publishing the message to the topic and the nodes at the arrow head are subscribed to all messages of the topic.

![Graph of nodes and topics](image)

Figure 33. Graph of nodes and topics

An important note with this communication setup is that the topics must have similar topic types, or message types. If a publisher published to a topic multiple messages with different message types, there would be no concerns or noticeable errors. But a subscriber must be able to receive the message type or it will not establish a message channel. This will prevent random messages from being accepted with all the communication going on throughout the graph and through each topic. To set up the node to send, or publish, messages onto topics, we must include a header file for the standard messages (or basic data types) that ROS can send and receive between nodes. This header file is labeled “std_msgs/(message type)”. Based on what type of data type we wanted, we would include that header file. For the most part, we worked with “std_msgs/Int32MultiArray” because we would be able to publish and subscribe arrays onto topics and thus, throughout our system. This became the most significant message type for us because we would be able to control multiple nodes since we have a very high level state machine that controls the state machines within each of the nodes. Once we included the right standard message header file, we had to declare a publisher object with the function ros::Publisher, which is a function within the ros/ros.h header file, and declare an object of the standard message, which is Int32MultiArray for us. Here we set the node with the standard message type that it transmits on topics, with the parameters being the name of the topic and how large the buffer is. Using basic C++ or Python methods to store data, we can then call the function to actually publish a topic; this function is called as (declared object name).publish(std_msg).

To set up the node to receive, or subscribe to topics, we have to understand what spinning message threads do. As other nodes are publishing out messages to topics, nodes that want to read messages on those topics have to receive it. ROS designed the method of spinning threads, which is the
reading an entire queue of the topic message, to allow nodes to receive all the published messages at a
moment. That is, every time a node spins, it will read in all the published messages and the node will do
its computations based on the conditions set in it. Once a message is received from spinning, the node
will enter a designated callback, in which adjust or computations can be made before returning to the
main function. To declare the subscription in a node, we declare an object using ros::Subscriber, in
which the parameters are the topic name, the size of the queue, and the callback that will be entered
whenever a message is received from the queue. Once the subscription is set, the callback needs to be
set. The callback is simply another function, usually placed before to main function to differentiate it
from other types of functions, with its parameter being a pointer to the received message type. The
parameter is a point because we want to read what the message is within callbacks and save or compute
anything necessary before we leave the callback. Anything from basic mathematical computations, state machines, or even notifications can be done within callbacks. To call for a spin of
threads, there are two main ways of doing it, ros::spin() and ros::spinOnce(). The first function, spin(), is
effectively the second, spinOnce(), in an infinite loop. The first function is very restricting in that it does
not allow us to stop spinning threads unless we shut down the node. The function ros::spin() does not
need to be in a while loop and is only use in unique cases of a node. The latter is more frequently used
since we need to use it within a while loop if we want to continuously check topic messages and work
other computations or updates. At this point, we had enough background in ROS to understand how
everything is structured and flowed, which allowed us to start coding our overall high level state
machine. With this in mind, just going over the basic examples of each topic allowed us to get better
acquainted with writing code for ROS and debug any issues that occurred.

As mentioned earlier, our entire ROS program worked mainly with the standard message
Int32MultiArray, which are messages with 32-bit integers being sent at as many dimensions of an array
that we would want. Once the header file, std_msgs/Int32MultiArray, is included in the ROS program,
we can declare an object for the Int32MultiArray object. After we declare an object, we have to set a
few settings in the event we need to identify the message received. As an example, if we declare an
Int32MultiArray called rotate, we have to first push the array back the number of dimensions that we
want it to be. Since we only want to send messages with one array, we call the function
rotate.layout.dim.push_back() once so that rotate is a message array of one. Once the dimension is set,
the settings we have to set next are rotate.layout.dim[0].label, rotate.layout.dim[0].size, and
rotate.layout.dim[0].stride. Label is the name of the message sent out to others for identification, size is
the length of the array, and stride is the size of an image if it was a multi array. Once the settings are set,
we have to allocate the actual elements of the message array by using the function
rotate.data.push_back(). Every time this function is called, the array is pushed backed once and is stored
with whatever is set as the one parameter it takes in. Publishing this message is exactly the same as
mentioned earlier, but we have to ensure we fill in the array data correctly. To fill in the array of any
data element of the array, we can use rotate.data[index number] and set it to any data we want. When
a node receives the message, the callback will store the message into arrays for us to access.

We now have gone over all of the basic components and tools of ROS that we worked with in
order to develop a state machine to control our robot. Although we went over the majority of what ROS
has to offers in making programming a robot a lot easier, it was not everything that ROS had to offer. There are a lot more features that can be useful for other purposes such as service and clients, in which a client node (a customer) asks a serving node for a service (or task) to be done. But for what we needed to work with, everything we’ve gone through on ROS was more than enough to control our system.

Computer Vision:

Theoretical Primer - Point Cloud Library:

The basis of the entire computer vision portion of our project was based on point clouds and using the point cloud library. A point cloud is basically a 2D image, broken up into thousands of points. The points have fields such as x,y,z and r, g, b values associated to it. We can access each of these points and get the data associated with it. We can select each one of the data points and input it into our state machine to find objects and get its coordinates.

The drivers developed by ROS have the Xbox 360 Kinect publish a new point cloud every second. There are multiple formats published, which allows for flexibility and optimization in the program. In our program, we are able to subscribe to our desired point cloud topic and incorporated into our program. The format of point cloud we used was pcl::PointXYZRGB. Other formats available to use are rgb image, xyz image, rectified image etc... We chose point cloud xyzrgb because we needed to create an image for the user to see, as well as needing xyz so we can know how far a desired item is away from us.

Figure 34. Point Cloud of Scene
In Figure 34, we have an image of how a point cloud looks like. This point cloud consists of 307200 points. Each point has an xyz and rgb value associated, as we can see in the image above. The program we used to display the point cloud was called pcd_viewer. This program creates an empty Cartesian coordinate plane and places each point from the point cloud into that plane. This also displays the color each point has allowing us to see an image in 3D. The black area on the point cloud doesn't have any points. The black areas are around the border of the picture. Due to the position of the camera, it isn't able to get an xyz of a point cloud around the borders, but it can still record an rgb value. Some of the black areas include areas under the table and behind the boxes.

![Figure 35. RGB Image of Scene](image)

Figure 35 shows the point cloud image transformed into a 2D image. Basically, we took each point from the point cloud, and place them in a 2D grid. With its 307200 points, it was saved into a 640x480 rgb image, each point was saved as a pixel. It was important for us to be able to save the point cloud with xyz and rgb because our graphic user interface allows the user to click on the 2D image, and we need a direct translation between the corresponding point in the point cloud. Our timing with our project didn't allow for us to have a prewritten function which saves a point cloud turned into a picture format (ie: jpg, png etc... ).

The latest version of Ubuntu which works with ROS, is 12.04, and this came with the point cloud library (pcl) 1.6. In pcl2.0, the developers of PCL wrote a library of functions to convert pcd files to png. PCL2.0 also changed the format of a point cloud, so we edited to make it work for our current version of the pcl. The basis of the function was that we needed to pass in an array of the points, followed by the dimension of the picture you want it to be, and the channel. The channel being how many successive
elements in the array belong to the same element. In our case, the channel is 3, because we need 3 values associated with each pixel, being the rgb values.

**Point Cloud Algorithms:**

Just being able to retrieve the point cloud from the Kinect is enough to make the computer vision part of our project complete. However, it doesn’t show case all of the functionalities of the PCL. We used several algorithms from PCL to add more functionality to our program in order for greater usability and efficiency. We wanted to be able to find objects and increase the calculations of our overall system, so we used PCL’s Euclidean Cluster Extraction and a downsampling algorithm.

The downsampling algorithm takes a point cloud and reduces the total points in the cloud so that it will only have the vital points which retains the shape of an object. For example, if there is a point cloud of a perfect square table, the algorithm will remove 90% of the points but when it is displayed through pcd_viewer, it still retains points for the user to tell it is a table, but reducing 90% of the points will greatly increase the speed of calculations for any future manipulations of the cloud.

![Figure 36. Left is a regular point cloud, right is the downsampled cloud](image)

In our case, we can use the downsampled point cloud and through it into the Euclidean cluster extraction algorithm. This algorithm identifies any of the background planes and removes it from the point cloud. Then it groups together points in the new cloud which it calculates it be associated with an object by parameters such as the distance between each cloud, and the density of points within a certain area.
Figure 37. Point cloud is segmented and displayed in pcd_viewer.

Figure 37 shows a table and an assortment of a few items. Although the points are all in the same image, one may think that it is still all within one array of points. Displaying each segmented point cloud is only a feature of pcd_viewer. Each different color is its own point cloud file(.pcd file).

Problems with the point cloud algorithms:

We were able to downsample the original point cloud of 307200 points in to a point cloud of roughly 30,000 points. However, downsampling came with problems for us. This came down to the design of the pcd to png algorithm. A point cloud has a field called, organized. This tells other algorithms which works with point cloud whether or not something is organized. Being organized means determining if the point cloud is formatted such that it resembles an image and clearly defined borders. If we take a look at the Figure 36, the image on the right is downsampled, but the user does not get to choose which points are removed. If there is a clear table in the point cloud, the algorithm may remove more points since it can still resemble a table. However, if there is a very specific shape like a figurine, the algorithm will likely remove fewer points from it to keep at the details associated with the figurine. Since the algorithm removes points to optimally downsample a point cloud, it doesn’t care for the organization of the point cloud. Now if we send a downsample point cloud into the pcd to png algorithm, the algorithm would not be able to figure out where on the point cloud defines a row and a column. Our problem is this, since we need to interact with the user, we must need to show the user any object segmented or any scene downsampled. However, running any of these algorithms on it will let us convert the point cloud for us to show to the user.

Our limitations became very obvious, we were no longer able to downsample or use segmentation directly. We figured out a method to work around this issue. By segmenting an object, and comparing the original point cloud, we were able to selectively pick points which were segmented out. We then can directly change the rgb values with each point and thus give the illusion of
highlighting, and then saving the new altered pcd into a png. We can save many of these pictures and allow for the user to scroll through them. Another problem which came out of this was just how computationally heavy this is.

In a messy room, there are over two dozen objects, and if our segmentation algorithm finds each one of the two dozen objects and compares it to the point cloud, we would have at least 48 * 307200 comparisons each run though, and saving each of those as images. All this is happening while we still have to constantly live stream both the kinects, and it is constantly publishing all the topics associated to each one. On run time, this limited us heavily, a run through would take up to 4 minutes, just on the computer vision portion of the system flow. This doesn’t include the time it takes for rest of the system to communicate, usb communication and protocol, the time it takes for the arm and the bogey to move. This is highly impractical and could take the user up to 7 minutes to actually get an item.

Our solution to cut down on the computation came down to designing software because we cannot get a more powerful computer to do the processing. What we did was iterate through the original point cloud and find each point the user is allowed to select, and highlight the point. We cannot just allow the user to click anywhere they desire because there some points in the rgb image which doesn’t have an xyz point. Some of the main places this happens is if an object is too close to the lens of the Kinect, or a thick border of the point cloud doesn’t have an xyz.

Graphical User Interface:

The Graphic User Interface(GUI) consists of buttons, slider bars, radio buttons, images, and tabs for the user to interact with. In terms with the robotic arm and bogey, we only send it commands and they do the task they are assigned. The input methods are only to allow the user to choose the item they desire.

Using PyGTK Library

The GUI was written using python because the ROS libraries allowed programs to be written in python and c++. Scripts allow for more compact code and easier to implement and thus we choose to write the GUI in python. The library we used was pygtk. This library has functions which allows for us to input widgets. Widgets in pygtk are anything that the user can interact with, or see. For the script, everything you see is a widget. The tabs which are labeled ‘page1-page5’, the buttons, radio buttons and both the sliders are buttons. Widgets, by programming, want to occupy as much space as possible, so we cannot add widgets without containers giving assigned space for each widget. This script uses a table to organize the widgets. Tables are simple, each table has a row and column field, and we get to choose which row and column to insert each widget. In order for the buttons to do anything, we need to associate each button with a signal. When a button is connected to a signal, the script executes a function connected to the button. The ‘Update Zones’ button allows for me to delete all of the current tabs and recreate new tabs with new images, so it acts as a fresh button for the images. The ‘Previous/Next Object’ buttons does something similar but it only updates page5 and moves on to the next image of highlights. The ‘Custom Screen’ and ‘Move Kinect’ button does the same thing, it sets a
flag high, and publishes that flag to the high level state machine. After it finishes publishing, it changes the flag down to inactive.

Figure 38. Graphic User Interface

One strange way the script is different from the programs in terms of ROS publishing and subscribing, is that the script only publishes and does not subscribes to anything. So regardless of what state we are currently in, the script will publish if buttons are pressed. It belongs to the job of all the other c++ nodes to only subscribe depending on the state and flags set. There was a few reasons the team decides to do it this way. The main reason is that we were unable to make the script subscribe like the nodes in c++. The function to make the script subscribe once (rosspin, as described previously), was not available in python. So do the time constraint, we decided to this was the best way to work around, instead of rewriting the whole script in c++. We also wanted to allow the user to move the Kinects anytime they desired. So we never want the script to be inactive.

Using the User Interface

Using the system isn’t very difficult. There are 4 separate components the user needs to see in order to use the GUI. In Figure 39, we have a the GUI as described in the previous section. On the top right, we have two video streams of the Kinect, and the bottom right, we have a terminal which guides the user and instructs the user which buttons to press and when.
The first phase of the state machine starts with initializing. During this phase, the user waits as the bogey moves to the center of the rail, and the Kinects rotate around to map out the room. On Figure 40, we see the different initialization positions of the Kinect, and with the correct angles tilted downwards, we are able to see every area of the room. The rotate joins are programmed to going into each configuration (as the arrow points on Figure 40), and the Kinects take a zoning picture of each zone of the room.

Once that is complete, the user has to hit the ‘Update Zones’ button on the GUI, and the image displayed will go from gray images, to the images the Kinect took. The user then has to select between page1 - page4 and click on an object of choice. Then the user will have to wait as the bogey takes the whole system to the location the user chose.
After the system arrives to the location, the user will have to move either of the Kinects around by adjusting the slider bars labeled, ‘Horizontal’ and ‘Vertical’, selecting ‘Kinect1’ or ‘Kinect2’ and hit the ‘Move Kinect’ button. When the object is in sight of the camera, the user will click ‘Custom Screen’. Make sure before the user hits ‘Custom Screen’ the correct radio button is selected for the Kinect number. After processing is done, the user needs to hit ‘Update Zone’ again, and it’ll display the image the user took. From page 5, the user now much click on the desired object. At this point, the bogey will align itself directly infront of the object, and the arm will pick up the object, retract, and the bogey will take the system back to the origin. After this, the user can repeat the process from selecting an object from page 1–page 4.

**Kinect Control:**

The Kinects were mounted on opposite corners of the box. We decided to do this so that we could cover the most area in the room without having to use a third Kinect. With pan and tilt capabilities, we would be able to see most of the room around the box. This saved us from having to burden the arm’s motors with extra weight if we had decided to mount a Kinect on the arm. This also reduced unnecessary movement of the arm by allowing the arm to remain motionless while the Kinect’s servos pointed to the four walls of the room.

The actual movement of the servos was simple using the RCServo.h library for servo control along with the Mavlink.h library for receiving the user’s desired orientation from the GUI. The RCServo library consisted of function to initialize, set, and end the servo output. With these functions we easily took the values received from ROS through the UART and the Mavlink (see Mavlink section) to set the position of either of the two Kinects. The Kinects were powered by a 5V rail and the position signal was sent from one of the three UNO32 microcontrollers.

**System Level State Machine:**

The state machine controls all of components of system, including the servos to move the Kinects, microcontroller to move the arm, microcontroller to move to the bogey, the graphic user interface. Unlike the embedded state machines from the arm and bogey team, our state machine only controls when they are active, and not any of the lower level functions. The nodes of the state machine include any of the calculations required from the tower, which isn’t possible with a microcontroller. This includes all of the point cloud processing, handling the point cloud files, the math calculations required for determining the shortest distance for rail, and the math to translate all of the coordinate systems from the Kinect to the arm and bogey.

Due to the nature of ROS, our state machine consists of nodes which are also its own program. Splitting up all of the states into programs also decreases the complexity overall. Another design choice in this state machine was to centralize the control of data through one node. By creating a centralized state machine, we are able to always have access to the data from all nodes at all times. If state 6 need data from state 1, we don’t have to transfer the data through states 2 to state 5. Instead we can store the data in the centralized node and send it the appropriate node when we change the appropriate state.
In figure 41, it shows we have 9 programs and 1 script, but we have 6 states; each state with a number after its name and a description. The central node being ‘State Machine’ is used as an intermediary to the next state. The bottom nodes are the nodes which aren’t a state, but separated from the main nodes because the primary use is to communicate with the microcontroller with the MAVLink overhead implemented. Also, in Figure 41, each bidirectional arrow means the nodes subscribe to each other, and the unidirectional arrows mean the receiving end subscribes but the other end doesn’t subscribe.

In prog1, we initialize, the program. We send coordinates to RailMCU so it knows to move to the origin. Next, we send positions to the servos so the Kinects get a nice area of the room. When the servo moves to there, the Kinect takes a picture. This is repeated 4 times until all of the zoning pictures are taken. The each of the images has a pcd and a png file associated with it.
Then we move onto script2, which we have to wait for the user to click on an image in page1-page4. After a successful click, the script publishes, the data for each widget associated with the script. This includes the page number the user selected, the xy of the image the user clicked on, the flag for ‘Custom Screen’. This information is sent to the state machine, and the state machine readies the array to be sent out.

The data from the script is sent to state machine node, and passed along to program 3. Here, we make sure to not advance states unless the user clicks on somewhere valid. As mentioned in the ‘Problems with Point Cloud’ section, the user may select an area where there isn’t an xyz field associated with the point. We do this by opening up if this is the case, this node will publish to the state machine that an invalid point was chosen and change state back to 2. If the user did select a valid point, we would convert that pixel xy to find a point on the previously saved pcd file. Since the conversion between the pixels on the xy coordinate are different from the way the points are stored in the array for the point cloud. The top left of the pixel value is (0,0) and the bottom right is (640, 480). The point cloud array is stored linearly. A formula was created to directly convert the xy pixel value to the array index. Once the xyz coordinate of the file was retrieved, we sent it to the state machine.

The state machine took the xyz coordinate of the selected point and send it to program 4. Here we use the coordinate transformation function from the Kinect to the center of the bogey, and a function written by the rail and bogey team. This converts it into a rail number and position number for the bogey to move to. Once program 4 receives an acknowledgement from railMCU saying the bogey arrived, we move onto state 5. Here, we ask the user to move the Kinect so they can see the object, and click ‘Custom Screen’. Program 5 will do the algorithms to save that custom pcd and png file. After it finishes the processing, it moves onto state 6.

As shown on Figure 41, we see there is a unidirectional arrow which points from script 2, to program 6. This insures that program 6 always has the most up to date numbers from the move Kinect because it needs those values to run the function which translate the coordinates of the Kinect to the center of the box. This program also uses the function which translates the box’s xyz coordinate into rail number and position. We used the function to move the bogey again to make sure we line up directly in front of the object because we assume the arm only moves in a 2 dimensional plane. After we get an acknowledgement back from the bogey saying it is in front of the object, we send the arm coordinates of the position of object, and assumes it gets there. During this time, the arm will move to the object, pick it up and retract. When the arm gives us an acknowledgement saying the item was picked up and arm is retracted, we move on to talk to the bogey again. We tell the bogey to move back to the origin. When we get back to the origin, we move back to state 2 and repeat the process to use the robot again.

User Friendly Launch – The Launch File:

Since our overall system may be designed for someone with no technology or mechanical background, we chose to create a more user friendly method of executing the entire program. Rather than being required to open up multiple terminals to execute terminal commands to run our programs, we chose to make it such that the user will be able to execute one terminal command in order to run
the entire program. To do this, we researched into how ROS integrates many programs to operate on with one command. We discovered that they termed executing multiple programs at once (use one terminal command) as “launching”, which are identified with .launch extensions and use XML formatting. Thus, the idea of understanding and using XML formatting in our case is to design it such that it will execute all our programs in a specific order. A few of our programs includes turning on both the Kinects such that data can be read from them, our project's GUI, the state machine, and other computational components(or nodes). Since we will be introducing XML files in terms of ROS, a good resource to refer to for more detailed documentation throughout this section is http://www.ros.org/wiki/roslaunch/XML.

XML Format and Tags

Although XML files have a general format, ROS have their own unique tags that users are able to take advantage of since these are specifically used to launch all the files within it. The XML format used for ROS has specific tags, each with their own attributes and elements specific to them. Attributes are declared within the same line of its tag and elements are used within tags to get more specific. To denote the start and end of XML tags, the programmer will have to begin the line with <tag name> and end it with </tag name>. An example would be using the node or launch tags, which in the XML files will be <include> and <node> to denote the start of and </include> and </node> to denote the end of those tags. If a tag is used for one line, such as for an argument tag(arg), the start and end of it, along with its attributes, can be denoted with < at the beginning and /> at the end of the line. An example using the arg tag is <arg/>. If tags are designed to be multiple lines, between the start and end of the tags are any of the tag's specific elements. A few commonly used tags to use are the launch, include, node, group, and arg tags. To see a list of all available tags and their details, visit the roslaunch/XML page mentioned earlier. Thus, we now have a basic understanding of XML files within ROS, but it is more than enough to work towards our goal for the week.

System Launch File

With basic knowledge of formatting XML files in ROS, we were able to create a launch file to turn on both the Kinects, display a live stream of the Kinects, and execute all the programs required for our system to run. We’ve done previous tests on sample programs to verify that any program that is designed to operate in ROS, whether it’s the C++ state machine, the Python GUI, or any other ROS package, can start together in one launch file. To start making our launch file, we had to create a directory within an ROS package called “launch”. Within this launch directory, we had to create a file with a “.launch” extension. For convenience, the name for the file is usually the name of the ROS package but in our case we chose to name it “statemachine.launch” to signify what this launch file is meant for. In order for ROS to identify that it is a launch file, the very first line of the file must be the <launch> tag; the last line of the file must be </launch>. All of the other tags are elements within the launch tag, thus everything must go between the start and end of the launch tags to make sense. Refer to Figure 42 to see our final launch file. Lines 1 and 57 represent the launch tag, which indicates that we want everything within those tags to be launched.
Since the current ROS Kinect drivers that we are working with have their own roslaunch command, “roslaunchopenni.launchopenni.launch”, we had to include the Kinect drivers' launch file within our launch file. To do so, we used the include tag and its file attribute, which will specify which launch file we wanted to include within our launch file. We thought of these include tags as including header files to any programming language such as the .h extension in C/C++, which is why it is best to include them early in the event that other nodes will need them to be active. As shown in Figure 42 in lines 3 and 11, the file attribute must be very specific in order for ROS to locate the launch file. Had we just put file="openni.launch", we would have to execute the launch file within the directory openni_launch. In most cases, the file pathway should be very similar to the one shown in the figure. Next, we needed to include the arguments of openni.launch so that we could use two different Kinects at once. All Kinect driver argument names and variables required can be found by going into the

```xml
<launch>
  <include file="/opt/ros/foxtrot/stacks/openni_launch/openni.launch">
    <arg name="camera" default="kinect1" />
    <arg name="device_id" default="100" />
  </include>
  <include file="/opt/ros/foxtrot/stacks/openni_launch/openni.launch">
    <arg name="camera" default="kinect2" />
    <arg name="device_id" default="200" />
  </include>
  
  <node name="image_viewkinect1" pkg="image_view" type="image_view" args="image:kinect1" rgb/image_color
  
  <group ns="stateMachine">
    <node name="controller" pkg="my_pcl_tutorial" type="statemachine" launch-prefix="gnome-terminal --working-directory=/home/ender/effector/Pictures -e" output="screen" />
  </group>

  <group ns="stateMachine">
    <node name="SavePNGCD" pkg="my_pcl_tutorial" type="program3" />
  </group>

  <group ns="stateMachine">
    <node name="GUI" pkg="my_pcl_tutorial" type="script2.py" />
  </group>

  <group ns="stateMachine">
    <node name="PixelToXYZ" pkg="my_pcl_tutorial" type="program3" />
  </group>

  <group ns="stateMachine">
    <node name="TalkARMRAIL" pkg="my_pcl_tutorial" type="program4" />
  </group>

  <group ns="stateMachine">
    <node name="Save Zones" pkg="my_pcl_tutorial" type="program5" />
  </group>

  <group ns="stateMachine">
    <node name="PixelToXYZ Zones" pkg="my_pcl_tutorial" type="program6" />
  </group>

  <group ns="stateMachine">
    <node name="Rotate MAVLink" pkg="MAVLnKODES" type="rotate_msg" />
  </group>

  <group ns="stateMachine">
    <node name="Rall MAVLink" pkg="MAVLnKODES" type="rall_msg" />
  </group>

  <group ns="stateMachine">
    <node name="Arm MAVLink" pkg="MAVLnKODES" type="arm_msg" />
  </group>
</launch>
```

Figure 42. Code for system's launch file

Since the current ROS Kinect drivers that we are working with have their own roslaunch command, “roslaunchopenni.launchopenni.launch”, we had to include the Kinect drivers' launch file within our launch file. To do so, we used the include tag and its file attribute, which will specify which launch file we wanted to include within our launch file. We thought of these include tags as including header files to any programming language such as the .h extension in C/C++, which is why it is best to include them early in the event that other nodes will need them to be active. As shown in Figure 42 in lines 3 and 11, the file attribute must be very specific in order for ROS to locate the launch file. Had we just put file="openni.launch", we would have to execute the launch file within the directory openni_launch. In most cases, the file pathway should be very similar to the one shown in the figure. Next, we needed to include the arguments of openni.launch so that we could use two different Kinects at once. All Kinect driver argument names and variables required can be found by going into the
openni.launch file and studying the code. For our purposes, we needed to only use two arguments, the name of the camera and the device number. The name of the camera is used to help identify which Kinect we are using. Thus, as shown in lines 5 and 10, we called one Kinect1 and the other one Kinect2. The second argument we needed to work with is the device_id, which is used to identify any Kinect connected to the USB bus port. Since the Kinects will be continuously streaming data to the USB bus ports at the same time, we are required to connect the two Kinects to different bus ports on the motherboard of the computer. Thus, we connect one Kinect to BUS001 and another to BUS002 on the computer; in our case, the ports on front of computer are BUS001 and two of the ports on the back are BUS002. For any reason other ports are used or a different computer is used, enter “lsusb” within the Ubuntu terminal with both Kinects plugged in to determine which USB bus each is connected to. For the device_id, the Kinect driver’s launch files were designed to look at the ports for any device connected. For Kinect1, we use 1@0 as the device_id, which essentially says “look for any Kinect device connected to BUS001”; and for Kinect2, we use 2@0, which says “look for any Kinect device connected to BUS002”. Note that if the Kinects were connected to BUS003 or BUS004, the id would be 3@0 or 4@0, respectively. At this point, if we removed all the lines after line 11, we would be able to turn on two Kinects at once. To verify this, we can look at Figure 43, in which we view some of the nodes created by both the Kinects being connected to their respective ports and launch. Notice the /Kinect1/ and /Kinect2/ before each node, which correlates with the device connected to BUS001 and BUS002. For future reference, the main nodes in this image that we will be working with will be /Kinect1/depth_registered/points and /Kinect2/depth_registered/points.

![Figure 43. ROS rqt_graph showing nodes of both Kinects running](image-url)
With the two Kinects working with one initial roslaunch call, we can now go through how to include any ROS package (nodes) within the launch file such that we can start them all at once. The first ROS packet that we include after initializing both the Kinects is the image_view package. This packet allows us to open two small windows for the user to see a live stream of the camera, which is used so that he or she could see what they want to point the cameras at before continuing with the program. Since opening two live stream cameras have no effect on our program, we do not need to group them in the any namespace with the <group>tag. The argument that we use for the image_view package is image:=kinect#/rgb/image_color, with the kinect# corresponding to the camera initialized with the Kinect name before it when we launched the two Kinects. At this point, when we run the launch file without any of the programs in the state machine group namespace, the two Kinects will launch and two windows will pop up with live streams of the cameras.

Next we begin to put our own programs (or nodes) into the launch file so that they can all be executed together. The first thing we do is decide on a group namespace that all of our programs will be under so that they can communicate with each other by publishing and subscribing. This grouping is also useful for keeping programs with specific functionalities together to make it easier for programmers to read and debug. Looking at lines 16 to 55, we have implemented our programs each with the <group> tag first followed by the <node> tag. Within a group, the <node> tag allows us to place any nodes under that specific namespace. For the node tag, we only need a few attributes such as the node name, the package name, and the type of file. The node name is simply to display the name on the rqt_graph. The package name is the package that contains the node to be included in the launch file. The type of file is either a C++ or Python executable. In our ROS package, we have a bin folder that holds the executable files for our C++ programs. The names of these executable files are what need to be entered for the type of file attribute. For Python programs, the executable file is where the file was first created since it is a script language. The type of file attribute in the node argument requires the .py extension be used as shown in line 26 where we use our GUI program named “script.py”. Two other common node attributes are the launch-prefix and the output, in which the first opens up a terminal and the latter determines where the programmer would like to see the output from that specific node. The type of terminal used to debug was the gnome-terminal, which is the standard terminal for the Ubuntu system. A second feature to the launch-prefix is the ability to open the terminal within a certain directory by using the command “--working-directory= . . .” after we stated which type of terminal we wanted to use. The output was chosen to be the screen of that terminal because to debug anything, we mainly used the printing functions part of the C++ and Python libraries. The launch-prefix and output attributes are not required for behind the scene computations but are very helpful during the debugging process. At this point, we have a completed launch file that includes launching both our Kinects at the same time, opening a live stream window for each Kinect, making both the Kinects' data accessible by our nodes, and launching our own computational nodes.

With our launch file set, we are able to make executing our program a lot more user friendly by using one command in one terminal. Given the user has the knowledge to work in the Ubuntu environment, they could also open up other terminals for further in depth details on what’s going on in each program and their components. Thus, from looking at Figure 42, we can see everything that will open up for the user after he or she enters the command. The windows that will all pop up for the user
to see are the two live streams of each Kinect camera, the user GUI to control the system, and the state machine node, which we use as a user instructive program.

**Communication Protocol:**

Micro Air Vehicle Communication Protocol (MAVLink)

For communicating between the three UNO32 microcontrollers (Kinect, Rail, and Arm), we chose to use Micro Air Vehicle Communication Protocol (MAVLink) by QGroundControl, which is a community groupworking (at qgroundcontrol.org/mavlink/start) to build an air vehicle ground control station. MAVLink was designed to work for wireless systems between a designated ground station and air vehicle by sending packets of data with high efficiency through data channels serially. Although QGroundControl and MAVLink were designed for wireless systems between a ground station and air vehicle, we are still able to utilize their packet protocol to communicate between components of our entire system by transporting data packets to and from a main station, which in this case is our ROS computer.

To name a few general benefits of using the MAVLink communication protocol as opposed to creating our own is its data packet framework, code generator, C/C++ or Python capability, and various data types. Similar to other wireless data framework, MAVLink utilizes a data packet framework to send and receive data between two components that begins with a specific start byte and ends with a checksum byte to ensure components receive the correct data. A visual representation of the packet framework is shown in Figure 44. The packet will differ from others in that MAVLink sets standard MAVLink data bytes (1 to 5) and contains a two byte checksum instead of one.

![MAVLink Frame – 8-263 bytes](image)

Figure 44. MAVLink packet framework

To simplify creating this packet for ease of sending and receiving between components, MAVLink provides a code generator based on an XML format that will generate functions in MAVLink header files for user designed message types. The key purpose of the code generator is to limit making packets from scratch by utilizing the provided MAVLink functions to create and decode packets. The MAVLink code generator was designed to generate header files in either C/C++ or Python since those are the more popular coding languages used for these types of systems. The ability to use it in C/C++ was a reason in choosing it for our system because it meant we could integrate it in ROS, which is in C++, and on our UNO32 controllers, which is in C. Along with these features, MAVLink has been implemented to support a wide variety of data types including characters, unsigned int8 to unsigned int32, signed int8 to signed int32, floats, and doubles. Having all of these data types available in the communication protocol is
useful for us since we have the ability to send any type of variable that we’d work with in C++/C on ROS and the UNO32.

In order to begin understanding how we integrated MAVLink onto the UNO32 microcontrollers and the ROS system, we have to first know how to generate the MAVLink header files. The MAVLink generator requires an XML file with the proper format in order to generate the header files. An example of the XML file we used for our MAVLink headers is shown in Figure 45, which is our XML file for rotating the Kinect with servo motors. MAVLink currently has two versions, 0.9 and 1.0, but we chose to use 1.0 because it is has the more recent updates. The tags (mavlink, message, field, etc.) are used for the generator to know what and how we want to format the generated header files, and essentially packets, for our system. In this example, we name the message Rotate with an ID number of three, which is byte 5 in the MAVLink packet framework. We then have a description tag that describes what this message is meant to send before we enter the type of data we want and its data variable name. For rotating the Kinect, we only needed unsigned integers for a KinectNumber, Pan, Tilt, and Error. The KinectNumber allows the UNO32 to know which Kinect to move, while the Pan and Tilt are used to either move the Kinect horizontally and vertically, respectively. The Error data variable was created in the event that we needed to check errors from the UNO32 moving the Kinects. For the rail and boogey, we set two unsigned integers, one for the rail number and the other for a position number. For the arm, we declared five unsigned integer variables: armX, armY, armZ, armNumber, armError. The X,Y, and Z variables are used to send the arm Cartesian coordinates relative to the center of the box. The armNumber variable is the KinectNumber so that the arm knows which Kinect we are looking from and the armError variable is used to communicate any acknowledgements or issues. Note that for our system, we decided to generate header files for each component in order to limit message errors and to simplify the ROS data flow by taking advantage of the ROS node features.

```xml
<?xml version='1.0'?>
<xml version='1.0'?>
  <mavlink>
    <messages>
      <message id='3' name='Rotate'>
        <description>kinect #, horizontal(pan), vertical(tilt)</description>
        <field type='uint8_t' name='KinectNumber'></field>
        <field type='uint16_t' name='Pan'>horizontal</field>
        <field type='uint16_t' name='Tilt'>vertical</field>
        <field type='uint8_t' name='Error'>acknowledge/error</field>
      </message>
    </messages>
  </mavlink>
</xml>
```

Figure 45. Example XML for MAVLink code generator

Once an XML file is ready, we open up the MAVLink generator to see a graphical user interface as shown in Figure 46. The first box is used to locate the XML file that we want to generate MAVLink for code in header files. The second box is used to state a directory for the output of all the files. The third setting is used to dictate whether the output code is generated in C or Python. The last setting is used to state the version of MAVLink we used. For our purposes, we used C and version 1.0. Once generate is
pressed, the black terminal behind it will process the XML file to header files and output them in the
directory stated.

Figure 46. MAVLink code generator

The output of the files will be similar to the ones shown in Figure 47 and 48. The directory
labeled “rotatemavlink” is named based on what we used as the name of the XML file. Note that Figure
48 inside the directory labeled “rotatemavlink”. Although all of these headers are important and useful
to utilize MAVLink, the main and only one we need to include in any of our code is mavlink.h header file.
This is due to the fact that the developers generated mavlink.h to include all the other header files
shown in these two figures.

Figure 47. MAVLink code generator
With these header files generated, we are now able to utilize MAVLink on our UNO32 microcontrollers and ROS system by including the mavlink.h header for any of our generated MAVLink code. To begin to utilize MAVLink and its functionalities, we need to understand some of the key variable declarations and functions required in order to create and decode a packet in C for serial transmission. The first thing to naturally include would be the mavlink.h header file generated as mentioned earlier. Within some of the MAVLink header files, they define all the types that MAVLink uses. Two important structs that MAVLink requires are a struct instance of the initial message data(bytes 1 to 5) and a struct instance of the message status; essentially declaring a message struct and a status struct for sending and receiving, respectively. We declared them as “static mavlink_message_t msg” and “static mavlink_status_t status”. These structs we will not need to personally manipulate the data in these structs but we will them as function parameters. With these structs defined, we also need to define specific component numbers so that components know which messages they are meant to receive. There are two numbers that need to be defined for components, the MAVLink number and the Computer ID. The MAVLink number is used to match the correct component while the Computer ID is used to match the correct computer being used in the event that more than one computer system is used. In our case we do not need to worry too much about the Computer ID since we have one system controlling everything but we still keep them different for practical purposes. For our programs, we listed the MAVLink numbers as 10, 20, 30, and we listed the Computer ID 90, 80, 70 for the rail, arm, and Kinect rotating microcontrollers, respectively.

Now that the main variables for MAVLink have been defined, we are now able to explain the important MAVLink functions that we will be using in our system. We will first begin with the functions used to create and send packets out serially with MAVLink. After generating the header files, most of the functions will be very system specific by using the name of the message listed in the XML file. For example, to create a packet, we use the function with the format: `mavlink_msg_(NAME)_pack(MAV_NUMBER, COMP_ID, &msg_struct, data . . .)`. For example, using the XML example of our Kinect rotating header files shown in Figure 45, the function we called for was `mavlink_msg_rotate_pack(MAV_NUMBER, COMP_ID, &msg, KinectNumber, Pan, Tilt, Error)`, in which the parameters after &msg are all data values that the user will adjust based on the program. Within this function to create packets, MAVLink set up the code to utilize a variety of functions from the other header files generated such as the checksum.h. By calling the `mavlink_msg_(NAME)_pack` function, the packet formation will go through the process of verifying the data type sent in compared to the message data declare in the XML file, copy the initial five data bytes that are standard for MAVLink, and then create the two byte checksum using an extra CRC value to detect mismatches.
The next important function for sending will be a function to put the packet data in a transmitting buffer and determine the overall length of the packet that is going to be sent. Since we are sending data serially, we need to know how many elements of an array buffer is our data to ensure it gets sent to the expected component. Without the length, we might over or under send data, therefore sending incomplete packets to the receiving end. To fill an arbitrary array with the entire packet and the length of that packet, we use the function `mavlink_msg_to_send_buffer(array, &msg)`. This is the same for all generated header files since the function's sole purpose is to transfer a specific packet to a buffer and then return the length of that packet. The message struct holds enough information to access the packet and transfer the data over to the buffer and it already has a data variable that stores the length of the packet. With `mavlink_msg_(NAME)_pack` and `mavlink_msg_to_send_buffer`, we are now able to send the array data serially through universal asynchronous receive/transmitter(UART) between a USB on the ROS system and mini USB on the UNO32. The transmission of data on any component is further discussed in the portion regarding the UART on UNO32 and the cereal_port package on the ROS system.

Since we now understand the functions used to send packets, we have to understand the functions to decode any packets we receive serially. To do so, we will have to continuously check the UART RX buffer and store one character at a time in a while loop. Throughout this while loop, whenever the UART RX is not empty, it will begin to parse the characters and update the MAVLink status (or state) to know which part of the packet it is expecting. To parse and update their status, MAVLink has a function called `mavlink_parse_char(MAVLINK_COMM, character, &msg, &status)` in which the first parameter is the internal MAVLink channel, in case multiple channels are running in one program, and the second parameter is the character received from the UART that should be parsed. The `mavlink_parse_char` function will return TRUE or FALSE to signify whether a packet is found or not. The way MAVLink designed the parsing function to work is it will accept the first six bytes(0 to 5), starting with the start of packet value 254, regardless since those are standard MAVLink data variables. It will then receive the payload at one point. After receiving the payload it will call a function to create the first byte checksum(or the lower checksum) and compare it to the first checksum received in the packet. If this lower checksum is good it will move on and do the same for the second byte checksum(or upper checksum). Once the second checksum has been confirmed, it will return a TRUE for the function, which signals that level of implementation to use a function to decode the data. In the event that a checksum fails, it will return FALSE and the parsing state will return to the waiting for the beginning of a new packet since no good packet was received.

The next function to understand for decoding is used to only decode the payload data into the variables stated in the XML and make them accessible for the user to use them in any way they want. To decode, we need to first define another struct called “mavlink_(NAME)_t data”, in which this struct contains all the data variables defined in the XML. Then we can decode the data by calling the function `mavlink_msg_(NAME)_decode(&msg, &data)`. This function will put data from the msgstruct to the data struct in their respective variables. Once the function is completed, to access any of the variables declared in the XML code we made, we use data.(variable name). Continuing from our Kinect rotating header files example, we would use data.KinectNumber, data.Pan, data.Tilt, and data.Error to access all.
our data. With this we can save the data into global variables if needed or simply use it right away from any purpose.

With an understanding in some of the basic MAVLink types and functions that we used, we can discuss how MAVLink was implemented onto ROS and the UNO32. The implementations onto the two systems are very different because of how we had to receive the data serially on the ROS system as opposed to the UART capabilities of the UNO32. We will first discuss the design of MAVLink on the ROS system. On ROS, the MAVLink will only be used to send data to another component when an array is received by another node, that is, the sending will only occur within the callback functions. The reason we do this is we are waiting for a reason to send the packet and wait for a response. If we did not have to wait for a callback, we would be continuously sending data to the UNO32. So in our callback, we will both send the data to any of the UNO32 controllers, depending on which part of the overall system we’re working worth, and then search for any packets sent from the controllers to ROS. The sending and waiting for a response will be in an infinite while loop until a response is found, in which a flag would break the while loop and continue the program. To keep the callback simple for future design uses, we create a function called send_(NAME)_wait(data ...) to deal with the sending and waiting. Creating a separate function to send and receive makes it convenient to use multiple callbacks to send and receive depending on how our program flows. Since callbacks are entered based on what is read from other nodes, we may have two different callbacks in which one is an initialize callback for the arm or rail and the other is the regular communications callback entered to send and receive messages.

Our function send_(NAME)_wait(data ...) will have a flow similar to the flow chart shown in Figure 49 for all three of the nodes that will send to the rail, arm, and Kinect rotating microcontrollers. At the beginning of this flow chart, we should receive the expected values to be sent from the ROS system to one of the microcontrollers and begin creating the MAVLink packets. Once a packet is made, we will write to the device serially, that is, from the USB port to the UNO32 connection. After we send the data, we set a delay because we want to give the controllers some time to do their computations and send a response back so that the ROS system knows of any errors, acknowledgements, or data for the rest of the system. Once the packet is sent and the delay is over, we use a try-catch statement to check the port of the device for any data on the buffer. If there is data, we should be able to check what was received; if there is no data, the system times out and we resend it to the controller in case it did not receive the first packet. It then continues the typical process of checking for the entire packet data, the first checksum, the second checksum, and then decodes it if the right packet is received. If the checksums fail, then most likely the packets were loss in the process when being sent from the controller to ROS. The last stage of the flow chart is to take the decoded data and save it into global variables so they can be accessed throughout our programs. The latter half of the flow chart shown is done by using the parsing and decoding functions MAVLink generated. Once this is complete, we will return from the function and continue on with our program and state machine.
Figure 49. MAVLink on ROS flow chart
With MAVLink on ROS complete, we now go over how we implemented MAVLink on the UNO32 for communication. The implementation of MAVLink for the UNO32 follows the flow chart shown in Figure 50 for the rail, arm, and rotating Kinect microcontrollers. At start up, all of the microcontrollers will initialize their own settings and then enter an idle loop to wait for a packet from the ROS system. If any of the microcontrollers see the RX buffer is not empty, it will immediately start to parse data and check the checksum with the mavlink_parse_char function. This will continue until a good packet is found, in which it will continue to decode the packer into the declared XML variable names. At this point, the data can be utilized by their microcontrollers to do their computations and move the necessary mechanical components. The ROS system will be in an idle state waiting for a response from the microcontroller. Once the rail, arm, or rotating Kinect microcontroller completes the computation, the functions to create a packet and determine the length of it will be used with data of where they’re currently at or with messages of error or acknowledgements. The microcontroller will then take the buffer and use a basic UART_putChar implementation to serially send the data to the ROS system.

Figure 50. MAVLink on UNO32 flow chart
At this point, we’ve concluded our discussion on how we used the MAVLink protocol as our communication between systems and how we implemented it on those systems. Although MAVLink was meant for wireless ground stations to air units, we were able to utilize the packet framework for our overall system’s purposes. We did use the same functions to set up and decode the MAVLink packets, but for the ROS system and the UNO32 microcontroller, we had to use different methods to serially send the data. On ROS, we had to utilize a cereal_port package to access a USB port to transmit data while the UNO32 was able to use utilize a UART to serially transmit data. Overall, MAVLink was able to simplify the amount of work we had to do to get communication up between components because of its simple packet framework and ease of use in our case.

**USB to UART (ROS System to UNO32)**

In order to communicate between the rail, arm, and rotating Kinect UNO32 microcontrollers, we had to determine the best method to transmit data. The first that came to mind was to transmit data from a computer to another device was serially through the USB to the mini port on an UNO32 since MAVLink is formatted into packets that we can send character by character. Serial data is simple enough to work with on the ROS system and on the UNO32 because of how we can send and receive on each of the system. On the ROS system, the ROS community has developed packages that are meant to detect certain devices connected to any computer’s USB port and access that port for data transmission. On the UNO32 microcontroller, the manufacturers have designed the product to have two UARTs, UART1 and UART2, in which UART1 is used when the mini port is connected to a computer. Given ROS has packages setup to transmit serial data and the features of the UNO32, transmitting data serially to communicate between the ROS system and the microcontrollers seemed the best.

To be able to send the data serially on the ROS system, we had to understand how the ROS package cereal_port worked and how to use it in our programs to send data to any of the microcontrollers. Some information of the cereal_port package in ROS can be found online at http://www.ros.org/wiki/cereal_port with a simple example. Although the name of this ROS package is cereal_port, the developers of this package designed the program to look specifically for devices attached to ports rather than just the USB ports, which is why we are able to use a USB splitter hub for the three UNO32s. Along with the splitter hub, the cereal_port package would not work without the use of a transistor-transistor logic (TTL) chip at its end in convert the USB transmission into a UART transmission. This is normally done with a TTL cable wire, but one of the beneficial features of using UNO32s is that it already includes a TTL chip on it. The TTL chip is solely connected to the mini port on the UNO32 such that any data sent from a computer to the microcontroller will go through the first UART.

To simplify our discussion of how we used cereal_port in our code, we will use the ROS example from the source site(redisplay in Figure 51). We did not do anything significantly different to work with cereal_port when comparing the example code to our code. The biggest difference would be that we sent an array of characters as opposed to the example only sending one character serially through the USB. To begin using the cereal_port package, we first had to download the package and make it a part of our ROS system in order to access it. Once we did that, we had to add cereal_port as a dependency in
any of the packages that wants the ability to connect with a USB device. At this point, we were able to utilize the cereal_port header files and example code to set up our system to communicate with each of the microcontrollers. Before we continue to explain our code, please note that because of how ROS is designed to handle data flow, we chose to make three separate programs that use cereal_port, in which each one talks to their respective UNO32 microcontroller for the rail, arm and rotating Kinect.

In our ROS code, the first requirement was to include the cereal_port header files and understand some of the functions and data types we had to work with. If we go into the cereal_port header files, we found many basic functions that serial transmission would have under the CerealPort namespace including Open(), Close(), Write(), and Read(). Next we have to define how large of a response, or a reply size (line 4), we would expect to receive from the UNO32. Since we only used the serial ports to transmit MAVLink packets, the reply size had to be at least the length of a MAVLink packet. But since we would most likely receive a lot of missing packets, and therefore useless data, we

```c
#include <ros/ros.h>
#include <cereal_port/CerealPort.h>

#define REPLY_SIZE 8
#define TIMEOUT 1000

// This example opens the serial port and sends a request 'R' at 1Hz and waits for a reply.

int main(int argc, char** argv)
{
  ros::init(argc, argv, "example_node");
  ros::NodeHandle n;
  cereal::CerealPort device;
  char reply[REPLY_SIZE];

  // Change the next line according to your port name and baud rate
  try{ device.open("/dev/ttyUSB0", 9600); }
  catch(cereal::Exception& e)
  {
    ROS_FATAL("Failed to open the serial port!!!");
    ROS_BREAK();
  }
  ROS_INFO("The serial port is opened.");

  ros::Rate r(1);
  while (ros::ok())
  {
    // Send 'R' over the serial port
    device.write("R");
    try{ device.read(reply, REPLY_SIZE, TIMEOUT); }
    catch(cereal::TimeoutExceptions e)
    {
      ROS_ERROR("Timeout!");
    }
    ROS_INFO("Got this reply: %s", reply);
    r.sleep();
  }
}
```

Figure 51. Simple example of cereal_port
chose to make a large, standard reply size of 100 across all the programs that used cereal_port. We also have to define a TIMEOUT (line 5) to control how long we want to keep checking the USB port for a response before resending the message again.

Once those were set up, we were able to begin opening a USB port for a device attached to it. In our main function we have to declare a variable under the CerealPort namespace so we declare it as device since each program will only access one device(line 13). We also need to create an array (line 14) to hold all the data received each time we read from the USB port. The next step is to check whether a USB device attach to a specific USB port and use a try-catch statement(lines 17-23) to make it accessible. The try statement will use the Open() function for the declared device and attempt to open a specific port at a certain baud rate. The port( the attached UNO32) and baud rates must be predetermined by the user. In our case, the UNO32s are being set to work with baud rates of 115200, so our baud rate on the ROS system ends are set to 115200 too. To find the microcontroller that we want to access, we have to use the Ubuntu command terminal to find the correct devices. When the terminal is open and a microcontroller is attached, type in “ls /dev” to see a list of all the devices attached and accessible. The list will contain many devices with the tty prefix, but we had to specifically look for the ones that ended with USB#. The USB# will tell us that a device, or UNO32, is connected via USB and can be accessed. Once this number is founded, it can be entered as the first parameter in the Open() function as “/dev/ttyUSB#”. The example shows the typical value for when one device is attached, but in our case we require three UNO32s through the USB ports. But in our case, after we attach three UNO32s through a splitter hub, the command “ls /dev” shows USB with # being 0, 1, or 2. To figure out which number belongs to which, the trick was in the order we plugged in the devices or we can determine which plug is always initialized as USB0, USB1, and USB2 and consistently plug the UNO32s into the same one. The catch statement will trigger an exception to occur, which will close the ROS program down since it could not find a device at the state point. At this point, assuming an UNO32 can be found, we were able to fully access it to transmit data.

Now, depending on whether we wanted to send or receive, different statements could be called. In the example shown, the program will send a letter first and then check the port buffer to read any data that comes in. In our case, we have a similar set up in that we will receive values from another ROS computation node and then check the port buffer, however, our program is set up to leave the while loop and send data out to other ROS nodes. The write statement is called using the C++ style of (variable name).write and is a function that takes the character to be sent as its parameter. In our case, we put this write function in a for loop the size of our data (MAVLink packet) and use the array and for loop index as the parameter of the write function. Once done, we put a delay to give the UNO32 enough time to run its computations and move the necessary mechanical components before checking the port buffer. We do this because if we check the port buffer too quick, there should not be any data, which leads to TIMEOUTs and being required to resend the data to the UNO32. Giving the microcontrollers enough time to finish their computations and sending a response packet back before attempting to check the port buffer will cause fewer errors to occur by over spamming the microcontrollers with packets. The read function, called similar to the write function, takes in an array, size of array, and TIMEOUT as its parameters. The array and size of array are the reply array and reply size variable.
declared earlier in our main function. As opposed to the example program shown in the figure, our program will continuously clear the reply array each time we read the buffer of data and fill it in with NULL values so that we never have an issue of rereading data if it does not get overwritten. The function returns a length of the reply array and makes the try statement valid. At this point, the cereal_port package is fully integrated into our ROS system and is ready to serially communicate with any microcontroller we access.

As mentioned earlier, the UNO32s are manufactured with a TTL chip, which makes it very convenient to receive data from the ROS system’s serially through the UARTs. The first UART, UART1, on the UNO32 has data from the mini port as its priority as opposed to data received from input and output pins. It remains accessible if no mini port is applied to it, but in our case, we require that all three microcontrollers connect via USB so that we can control the necessary components. The first thing we must do is set up the main function to be able to access the UARTs on the UNO32s. The header files for the UART and a few other important ones were provided for us and we simply had to include the files in C as normally. The important header files to send data serially are Uart.h, Board.h, Ports.h, and Serial.h. These all give us the files to program on the UNO32 board. With Uart.h included, basic UART functions such as putChar, getChar, and initializing specific UARTs were declared. At the beginning of our main, we only had to run BOARD_INIT() and the UARTs would be functional because BOARD_INIT() will also initialize SERIAL_INIT() on the UNO32, which is initializing UART1 because any serial data is sent through the mini port. Since all the UNO32s are in an idle state waiting for a packet from the ROS system, the UNO32s are set to receive data from the UART initially. To receive data, the UNO32s check the UART RX buffer to see if it is not empty. Once it is not empty, the character received will be processed through MAVLink functions. This will continue until the entire packets are received and the UNO32s leave the idle state to do their computations. If the UART RX buffer is empty, the idle state will recheck again during the next iteration. Once the UNO32s are finished with all their computations, they will want to send back data through the UART TX. To do so they make the MAVlink packets and use a loop and a basic UART putChar function to send the characters to the correct UART. We have predefined functions with two parameters for UART putChar, one to state the UART we want to send it to and another to state the character we want to send. For all the microcontrollers, they will send to UART1 because it is the only one connected to the TTL chip, which then converts the UART data to data suitable to be read on end with the USB port.

With this completes how we the ROS system and UNO32 microcontrollers are communicating to transmit MAVLink packets serially. Without the cereal_port on ROS and both the TTL chip and UARTs on the UNO32, we would not be able to simply send data across a serial channel. They each give us basic functions and features that we were accustomed to working with and made working in MAVLink a lot simpler since MAVLink’s communication protocol are in packets. As long as we keep in mind how to access each microcontroller physically on a USB hub and on the ROS system by checking the devices’ tty number, we have no problems transmitting messages continuously between the components.
**Conclusions and future work:**

This project provided a proof of concept for introducing the ceiling as plane for robotic locomotion as well as a method for assistive care. From this prototype we were able to find unexpected errors as well as better design choices. With the computer vision technology from the Kinect cameras we were able to precisely obtain coordinate points and translate them into a plane which the bogeys and arms can used to reach their desired point. The bogey is able to navigate with their tracking system do the desired point with very little error. The arm is able to extend out without the offset of the gripper to the desired point while avoiding the table height. By eliminating the obstacle collision that a traditional robot needs to account for, we are able to put more effort into constructing an assistive robot and direct our attention to the primary focus of the objective. Helping people. This robotic infrastructure is not limited only to an assistive robot, it can also be integrated with industry uses, where the environment is not safe for human interactions.

If this project progresses, better designs will be implemented to create a more robust foundation to break the physical limits that we experienced.

**Acknowledgements:**

This project was funded by Citrus, College Nine, Stevenson College, and Cowell College. Special Thanks to Pat Mantey, John Vesecky, Mircea Teodorescu, Gabe Elkaim, Super Dave, Dave Thayer, Michael Murry, Paul Naud, Life Guard team, and Nasa team for their guidance and help.
Appendix:

Miscellaneous work
Appendix

Input Capture

OpenTimer3(T3_ON | T3_PS_1_256, T3_TICK);
OpenCapture3(IC_ON | IC_EDGE_RISE | IC_TIMER3_SRC | IC_INT_ICAPTURE | IC_EVERY_EDGE);
ConfigIntCapture3(IC_INT_ON | IC_INT_PRIOR_7);

void __ISR(_INPUT_CAPTURE_3_VECTOR, ip17) IC3Interrupt(void)
{
  mIC3ClearIntFlag();
  tempBuff = mIC3ReadCapture();
  int y6;
  int porty;
  // porty = PORTS_ReadPort(PORTY);
  y6 = PORTY06_BIT;

  if(y6 == 1) {
    if(edgeCount == 1) {
      edgeCount++;
      time1 = mIC3ReadCapture();
    }
    else if(edgeCount == 3) {
      edgeCount = 1
      time3 = mIC3ReadCapture();
      pwmLength = ((time2 - time1) / (time3 - time1)) * 360;
      time1 = time3;
    }
  }
  if(y6 == 0) {
    edgeCount++;
    time2 = mIC3ReadCapture();
  }
  return;
}
Forward Kinematics Matlab

```matlab
1 - clear
2 - clc
3 - theta1 = sym('theta1');
4 - theta2 = sym('theta2');
5 - theta3 = sym('theta3');
6 - L1 = 60;
7 - L2 = 60;
8 - a1 = cos(theta1);
9 - a2 = cos(theta2);
10 - a3 = cos(theta3);
11 - a1 = sin(theta1);
12 - a2 = sin(theta2);
13 - a3 = sin(theta3);
14 - T01 = [a1, -a1, 0, 0; a1, 0, 0, 0; 0, 0, 0, 0; 0, 0, 0, 1];
15 - T12 = [a2, -a2, L1, a2, 0; a2, 0, 0, 0, 0; 0, 0, 0, 0, 1];
16 - T23 = [a3, -a3, L1+a3, a3, 0; a3, 0, 0, 0, 0, 0; 0, 0, 0, 0, 1];
17 - T03 = T01*T12*T23
18 - simplify(T03)

ans =
[ cos(theta1 + theta2 + theta3), -sin(theta1 + theta2 + theta3), 0, 560*cos(theta1 + theta2) + 560*cos(theta1)]
[ sin(theta1 + theta2 + theta3), cos(theta1 + theta2 + theta3), 0, 560*sin(theta1 + theta2) + 560*sin(theta1)]
[ 0, 0, 0, 1];
[ 0, 0, 0, 1]`
```
Inverse Kinematics Matlab

```
L2 = 860;
L4 = 660;
yc = 1000;
xc = 1000;
%yc = 560;
%xc = -560;

r = sqrt(yc^2 + xc^2);

X2 = sqrt(2*L2^2 - L4^2)/(2*L2*L4);
Y2 = [sqrt(1-X2^2), -sqrt(1-X2^2)];

Theta2 = atan2(Y2,X2);
Theta2 = Theta2*(-180/pi);
Beta = atan2(xc,yc);
CApha = (r^2+L2^2 - L4^2)/(2*r*L2);

%method1
Theta1 = [Beta - atan2(sqrt(1-CApha^2),CApha),Beta + atan2(sqrt(1-CApha^2),CApha)];

%method2
%Theta1 = [Beta - atan2(L4*sqrt(1-X2^2), L2+L4*X2), Beta + atan2(L4*sqrt(1-X2^2), L2+L4*X2)];

%solutions
Theta1 = Theta1*(-180/pi)

Theta2
ThetaGripper = -(Theta1 + Theta2)

Theta1 =
-26.2971  63.7029

Theta2 =
-43.4006  43.4006

ThetaGripper =
69.6977  20.3023

For positions
(1000mm,1000mm)
```
Appendix A:
Gantt Charts
### Robotic Arm Gantt Chart

<table>
<thead>
<tr>
<th>Task Name</th>
<th>Duration</th>
<th>Start</th>
<th>Finish</th>
<th>Team Member</th>
</tr>
</thead>
<tbody>
<tr>
<td>Research</td>
<td>30 days</td>
<td>Tue 11/27/12</td>
<td>Sun 1/8/13</td>
<td></td>
</tr>
<tr>
<td>Building robotic arm</td>
<td>34 days</td>
<td>Mon 1/7/13</td>
<td>Thu 3/21/13</td>
<td></td>
</tr>
<tr>
<td>Solid work</td>
<td>14 days</td>
<td>Mon 1/7/13</td>
<td>Thu 1/24/13</td>
<td></td>
</tr>
<tr>
<td>Gathering parts</td>
<td>7 days</td>
<td>Mon 1/21/13</td>
<td>Tue 1/25/13</td>
<td></td>
</tr>
<tr>
<td>Fabrication</td>
<td>7 days</td>
<td>Wed 1/30/13</td>
<td>Thu 2/7/13</td>
<td></td>
</tr>
<tr>
<td>Construction</td>
<td>30 days</td>
<td>Fri 2/8/13</td>
<td>Thu 3/21/13</td>
<td></td>
</tr>
<tr>
<td>Software</td>
<td>51 days</td>
<td>Mon 1/7/13</td>
<td>Mon 3/18/13</td>
<td></td>
</tr>
<tr>
<td>Slatormachine</td>
<td>51 days</td>
<td>Mon 1/7/13</td>
<td>Mon 3/18/13</td>
<td></td>
</tr>
<tr>
<td>Low level</td>
<td>21 days</td>
<td>Mon 1/7/13</td>
<td>Mon 2/14/13</td>
<td></td>
</tr>
<tr>
<td>High level</td>
<td>30 days</td>
<td>Tue 2/6/13</td>
<td>Mon 3/18/13</td>
<td></td>
</tr>
<tr>
<td>Feed back testing</td>
<td>14 days</td>
<td>Tue 1/8/13</td>
<td>Fri 1/25/13</td>
<td></td>
</tr>
<tr>
<td>Testing arm functionality</td>
<td>45 days</td>
<td>Tue 3/19/13</td>
<td>Mon 5/20/13</td>
<td></td>
</tr>
</tbody>
</table>

### Rail and Bogey Gantt Chart

<table>
<thead>
<tr>
<th>Task Name</th>
<th>Duration</th>
<th>Start</th>
<th>Finish</th>
<th>Team Member</th>
</tr>
</thead>
<tbody>
<tr>
<td>Motor Controls</td>
<td>10 days</td>
<td>Thu 1/17/13</td>
<td>Sat 1/26/13</td>
<td>Joel</td>
</tr>
<tr>
<td>Bogey Design</td>
<td>12 days</td>
<td>Thu 1/17/13</td>
<td>Mon 1/28/13</td>
<td>Kyle</td>
</tr>
<tr>
<td>Power Supply</td>
<td>12 days</td>
<td>Thu 1/17/13</td>
<td>Mon 1/28/13</td>
<td>TJ</td>
</tr>
<tr>
<td>Rail System Design</td>
<td>1 day</td>
<td>Fri 1/18/13</td>
<td>Fri 1/18/13</td>
<td>TJ</td>
</tr>
<tr>
<td>Electrify Rail System</td>
<td>42 days</td>
<td>Sat 2/9/13</td>
<td>Fri 3/22/13</td>
<td>15</td>
</tr>
<tr>
<td>Turning Mechanics</td>
<td>1 day</td>
<td>Fri 1/18/13</td>
<td>Fri 1/18/13</td>
<td>Joel</td>
</tr>
<tr>
<td>Tracking System Design</td>
<td>36 days</td>
<td>Fri 1/18/13</td>
<td>Fri 2/22/13</td>
<td>Kyle</td>
</tr>
</tbody>
</table>

### End Effector Gantt Chart

<table>
<thead>
<tr>
<th>Task Name</th>
<th>Duration</th>
<th>Start</th>
<th>Finish</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project Design</td>
<td>16 days</td>
<td>Tue 11/20/12</td>
<td>Thu 12/13/12</td>
</tr>
<tr>
<td>Build Vacuum Gripper</td>
<td>27 days</td>
<td>Thu 12/13/12</td>
<td>Fri 1/18/13</td>
</tr>
<tr>
<td>Programming ROS</td>
<td>52 days</td>
<td>Thu 1/10/13</td>
<td>Fri 3/22/13</td>
</tr>
<tr>
<td>Review Basic Functionality</td>
<td>3 days</td>
<td>Thu 1/10/13</td>
<td>Mon 2/4/13</td>
</tr>
<tr>
<td>Kinect</td>
<td>50 days</td>
<td>Mon 1/14/13</td>
<td>Fri 3/29/13</td>
</tr>
<tr>
<td>Microcontroller</td>
<td>21 days</td>
<td>Mon 1/14/13</td>
<td>Mon 2/1/13</td>
</tr>
<tr>
<td>Create Stationary/Mobile</td>
<td>11 days</td>
<td>Mon 3/18/13</td>
<td>Mon 4/1/13</td>
</tr>
<tr>
<td>Integrate End-Effect Parts</td>
<td>51 days</td>
<td>Fri 2/22/13</td>
<td>Fri 5/31/13</td>
</tr>
</tbody>
</table>
Appendix B: Budget
<table>
<thead>
<tr>
<th>Items Description</th>
<th>Subtotal</th>
<th>Tax</th>
<th>Shipping</th>
<th>Total</th>
<th>Date of Purchase</th>
<th>Vendor</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Rail and Bogey</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Screws/Nuts, Tape, Cable Ties, MDF, 2x4</td>
<td>11.65</td>
<td>0.96</td>
<td>-</td>
<td>12.67</td>
<td>2/9/2013</td>
<td>Home Depot</td>
</tr>
<tr>
<td>H-Bridge Cytron DC Motor Controller(4)</td>
<td>57.32</td>
<td>-</td>
<td>10.76</td>
<td>68.08</td>
<td>2/13/2013</td>
<td>Robot Shop</td>
</tr>
<tr>
<td>Gearmotor w/ Encoder, Mounting Hub</td>
<td>159.80</td>
<td>-</td>
<td>25.46</td>
<td>194.65</td>
<td>2/14/2013</td>
<td>Pololu Corporation</td>
</tr>
<tr>
<td>PDP Bearings(2)</td>
<td>20.00</td>
<td>1.75</td>
<td>21.75</td>
<td>21.75</td>
<td>2/23/2013</td>
<td>Pacific Wave Surf Shop</td>
</tr>
<tr>
<td>ThreadLocker, Nuts, Bolts, Eraser Board, Cutting Charge</td>
<td>50.20</td>
<td>4.39</td>
<td>-</td>
<td>54.59</td>
<td>2/23/2013</td>
<td>Probuild</td>
</tr>
<tr>
<td>Hex Bolts</td>
<td>9.91</td>
<td>0.87</td>
<td>-</td>
<td>10.78</td>
<td>3/2/2013</td>
<td>Probuild</td>
</tr>
<tr>
<td>MDF, Cutting Charge, Duct Tape</td>
<td>11.39</td>
<td>1.00</td>
<td>-</td>
<td>12.39</td>
<td>3/8/2013</td>
<td>Probuild</td>
</tr>
<tr>
<td>Tri-Pac</td>
<td>5.78</td>
<td>0.51</td>
<td>-</td>
<td>6.29</td>
<td>3/11/2013</td>
<td>Probuild</td>
</tr>
<tr>
<td>PDP Bearings</td>
<td>10.00</td>
<td>1.75</td>
<td>21.75</td>
<td>21.75</td>
<td>3/11/2013</td>
<td>Pacific Wave Surf Shop</td>
</tr>
<tr>
<td>Epoxy, Nuts, Bolts</td>
<td>13.72</td>
<td>1.47</td>
<td>-</td>
<td>15.19</td>
<td>3/26/2013</td>
<td>Probuild</td>
</tr>
<tr>
<td>Eraserboard</td>
<td>28.52</td>
<td>2.50</td>
<td>-</td>
<td>31.02</td>
<td>3/26/2013</td>
<td>ProBuild</td>
</tr>
<tr>
<td>Doubled sided tape</td>
<td>5.39</td>
<td>0.49</td>
<td>-</td>
<td>5.88</td>
<td>4/1/2013</td>
<td>Beverly's</td>
</tr>
<tr>
<td>Aluminum L Channels, Cutting charge</td>
<td>275.10</td>
<td>-</td>
<td>48.08</td>
<td>347.25</td>
<td>4/15/2013</td>
<td>OnlineMetals.com</td>
</tr>
<tr>
<td>Analog Grayscale Sensor(7)</td>
<td>30.10</td>
<td>-</td>
<td>12.00</td>
<td>42.10</td>
<td>4/18/2013</td>
<td>DFRobot</td>
</tr>
<tr>
<td>Tie Plate, wood</td>
<td>13.80</td>
<td>1.14</td>
<td>-</td>
<td>14.94</td>
<td>4/19/2013</td>
<td>Home Depot</td>
</tr>
<tr>
<td>Copper Spacer, Bolts, Nuts, Acrylic</td>
<td>58.62</td>
<td>5.13</td>
<td>-</td>
<td>63.75</td>
<td>4/22/2013</td>
<td>Probuild</td>
</tr>
<tr>
<td>Aluminum Plate, Waterjet Machining, Waterjet Layout</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>RCR Fabrication and Design Inc</td>
</tr>
<tr>
<td>14-Guage Thin Wire, Bolts</td>
<td>49.00</td>
<td>5.07</td>
<td>-</td>
<td>54.07</td>
<td>4/27/2013</td>
<td>Home Depot</td>
</tr>
<tr>
<td>Bolts</td>
<td>0.84</td>
<td>0.08</td>
<td>-</td>
<td>0.92</td>
<td>4/27/2013</td>
<td>Home Depot</td>
</tr>
<tr>
<td>4x4 Wood, Lumber fee</td>
<td>35.32</td>
<td>2.89</td>
<td>-</td>
<td>38.21</td>
<td>5/4/2013</td>
<td>Home Depot</td>
</tr>
<tr>
<td>Post Cap</td>
<td>37.98</td>
<td>3.13</td>
<td>-</td>
<td>41.11</td>
<td>5/11/2013</td>
<td>Home Depot</td>
</tr>
<tr>
<td>Welding</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>5/13/2013</td>
<td>Dave Thayer's Shop</td>
</tr>
<tr>
<td>Lamination</td>
<td>12.00</td>
<td>1.05</td>
<td>-</td>
<td>13.05</td>
<td>5/25/2013</td>
<td>FedEx Office Supplies</td>
</tr>
</tbody>
</table>

**Robotic Arm**
<table>
<thead>
<tr>
<th>Item Description</th>
<th>Unit Price (incl. tax)</th>
<th>Discounted Price (incl. tax)</th>
<th>Total Price (incl. tax)</th>
<th>Retailer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pic32(2)</td>
<td>60.00</td>
<td>60.00</td>
<td>Amazon</td>
<td></td>
</tr>
<tr>
<td>Cytron Motor Controller, alu mount hub</td>
<td>36.61</td>
<td>10.76</td>
<td>47.37</td>
<td>RobotShop</td>
</tr>
<tr>
<td>Gloss, yellow paint</td>
<td>6.84</td>
<td>0.56</td>
<td>7.40</td>
<td>Home Depot</td>
</tr>
<tr>
<td>MDF(2)</td>
<td>11.00</td>
<td>11.00</td>
<td>Probuild</td>
<td></td>
</tr>
<tr>
<td>Foamcore/Tape Measure(4)</td>
<td>7.96</td>
<td>0.70</td>
<td>8.66</td>
<td>-</td>
</tr>
<tr>
<td>Spring(11)</td>
<td>4.36</td>
<td>0.38</td>
<td>4.74</td>
<td>Probuild</td>
</tr>
<tr>
<td>Dust Pan/Broom</td>
<td>9.95</td>
<td>0.82</td>
<td>10.77</td>
<td>Home Depot</td>
</tr>
<tr>
<td>4x4, 2x4, Rigid Tie/Angle, Screws, Gear tie</td>
<td>29.01</td>
<td>2.40</td>
<td>31.41</td>
<td>Home Depot</td>
</tr>
<tr>
<td>Pic32</td>
<td>30.00</td>
<td></td>
<td>30.00</td>
<td>Amazon</td>
</tr>
<tr>
<td>H-Bridge</td>
<td>20.00</td>
<td></td>
<td>40.00</td>
<td>Amazon</td>
</tr>
<tr>
<td>Acrylic Rods</td>
<td>14.62</td>
<td>1.26</td>
<td>15.88</td>
<td>Tap Plastic</td>
</tr>
<tr>
<td>Washers, Zinc Anchor, screws</td>
<td>7.32</td>
<td>0.63</td>
<td>7.95</td>
<td>Home Depot</td>
</tr>
<tr>
<td>Gearmotor, Bracketr Pair</td>
<td>143.70</td>
<td>17.95</td>
<td>161.65</td>
<td>Pololu</td>
</tr>
<tr>
<td>hacksaw blade, gorilla glue, screws, bearings, acrylic</td>
<td>36.13</td>
<td>3.16</td>
<td>39.29</td>
<td>Probuild</td>
</tr>
<tr>
<td>Screws,hose cap, bearing, nuts, machine screws, garden hose</td>
<td>24.44</td>
<td>2.14</td>
<td>26.58</td>
<td>Probuild</td>
</tr>
<tr>
<td>Bearings, screws,</td>
<td>9.76</td>
<td>0.85</td>
<td>10.61</td>
<td>Probuild</td>
</tr>
<tr>
<td>Screws, 4x4 lumber</td>
<td>12.55</td>
<td>1.18</td>
<td>13.73</td>
<td>Probuild</td>
</tr>
<tr>
<td>Acrylic</td>
<td>6.56</td>
<td>0.57</td>
<td>7.13</td>
<td>Tap Plastic</td>
</tr>
<tr>
<td>MDF</td>
<td>7.00</td>
<td></td>
<td>14.00</td>
<td>-</td>
</tr>
<tr>
<td>Pic32</td>
<td>30.00</td>
<td></td>
<td>30.00</td>
<td>Amazon</td>
</tr>
<tr>
<td>U channels</td>
<td>19.87</td>
<td>3.48</td>
<td>43.22</td>
<td>Amazon</td>
</tr>
<tr>
<td>pipe straw, screws, v508</td>
<td>12.63</td>
<td>1.11</td>
<td>13.74</td>
<td>Probuild</td>
</tr>
<tr>
<td>Nuts</td>
<td>2.89</td>
<td>0.51</td>
<td>6.29</td>
<td>Probuild</td>
</tr>
<tr>
<td>Motor Controller</td>
<td>14.33</td>
<td></td>
<td>39.37</td>
<td>RobotShop</td>
</tr>
<tr>
<td>Steel Gears</td>
<td>-</td>
<td></td>
<td>355.00</td>
<td>-</td>
</tr>
<tr>
<td>Wires</td>
<td>-</td>
<td></td>
<td>13.00</td>
<td>-</td>
</tr>
<tr>
<td>Pressure Sensors</td>
<td>-</td>
<td></td>
<td>21.00</td>
<td>-</td>
</tr>
<tr>
<td>Item</td>
<td>Price 1</td>
<td>Price 2</td>
<td>Price 3</td>
<td>Price 4</td>
</tr>
<tr>
<td>-------------------------------------------</td>
<td>---------</td>
<td>---------</td>
<td>---------</td>
<td>---------</td>
</tr>
<tr>
<td>Retractable Extention Coord</td>
<td>43.15</td>
<td>3.78</td>
<td>-</td>
<td>46.93</td>
</tr>
<tr>
<td><strong>End Effector</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Plastic Tube/Funnel</td>
<td>4.87</td>
<td>0.41</td>
<td>-</td>
<td>5.28</td>
</tr>
<tr>
<td>Ballons</td>
<td>2.16</td>
<td>0.17</td>
<td>-</td>
<td>2.16</td>
</tr>
<tr>
<td>Coffee grinds(3)</td>
<td>3.39</td>
<td>-</td>
<td>-</td>
<td>10.17</td>
</tr>
<tr>
<td>Balloon (16&quot;)</td>
<td>3.00</td>
<td>-</td>
<td>-</td>
<td>3.00</td>
</tr>
<tr>
<td>Xbox 360 Kinect</td>
<td>59.99</td>
<td>4.95</td>
<td>-</td>
<td>64.94</td>
</tr>
<tr>
<td>Funnel(2)</td>
<td>0.99</td>
<td>0.15</td>
<td>-</td>
<td>2.13</td>
</tr>
<tr>
<td>Funnel</td>
<td>0.99</td>
<td>0.09</td>
<td>-</td>
<td>1.08</td>
</tr>
<tr>
<td>Liquid Silicon adhesive</td>
<td>13.48</td>
<td>1.18</td>
<td>-</td>
<td>14.66</td>
</tr>
<tr>
<td>Helium Balloons</td>
<td>9.74</td>
<td>0.80</td>
<td>-</td>
<td>10.54</td>
</tr>
<tr>
<td>PVC Bussing F, Adapter, PVC female adapater</td>
<td>1.03</td>
<td>0.08</td>
<td>-</td>
<td>1.11</td>
</tr>
<tr>
<td>Party Balloon(2)</td>
<td>7.99</td>
<td>1.32</td>
<td>-</td>
<td>17.30</td>
</tr>
<tr>
<td>Bushing + Adapter</td>
<td>3.08</td>
<td>0.27</td>
<td>-</td>
<td>3.35</td>
</tr>
<tr>
<td>23/4x6x1/16 Micro</td>
<td>3.49</td>
<td>0.31</td>
<td>-</td>
<td>3.80</td>
</tr>
<tr>
<td>Fastenal</td>
<td>0.85</td>
<td>0.07</td>
<td>-</td>
<td>0.92</td>
</tr>
<tr>
<td>Screw Driver + insert</td>
<td>4.58</td>
<td>0.40</td>
<td>-</td>
<td>4.98</td>
</tr>
<tr>
<td>USB Extension Cable</td>
<td>12.35</td>
<td>0</td>
<td>0</td>
<td>12.35</td>
</tr>
<tr>
<td>Adapter Barb</td>
<td>4.28</td>
<td>0.37</td>
<td>-</td>
<td>4.65</td>
</tr>
<tr>
<td>Barb Tee, Wht C, 3/80D V, 3/16IDX100'</td>
<td>6.03</td>
<td>0.53</td>
<td>-</td>
<td>6.56</td>
</tr>
<tr>
<td>Barb Elbow</td>
<td>9.96</td>
<td>0.87</td>
<td>-</td>
<td>10.83</td>
</tr>
<tr>
<td>PC Board(2)</td>
<td>4.98</td>
<td>0.44</td>
<td>-</td>
<td>5.42</td>
</tr>
<tr>
<td>USB Extension Cable(2)</td>
<td>20.3</td>
<td>0</td>
<td>0</td>
<td>20.3</td>
</tr>
<tr>
<td>Lynxmotion Pan and Tilt</td>
<td>59.86</td>
<td>-</td>
<td>10.71</td>
<td>70.57</td>
</tr>
<tr>
<td>Servo, Stepper motor, h bridge</td>
<td>69.69</td>
<td>-</td>
<td>10.76</td>
<td>80.45</td>
</tr>
<tr>
<td>Kinect</td>
<td>70.00</td>
<td>-</td>
<td>-</td>
<td>70.00</td>
</tr>
<tr>
<td>Group</td>
<td>Subtotal</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---------------------</td>
<td>------------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rail and Bogey</td>
<td>1,940.97</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Robotic Arm</td>
<td>1,116.72</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>End Effector</td>
<td>426.55</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>3,414.24</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Appendix C: Team Charters
Ceiling Helper Robot:

Arm Team Charter

Nelson Chu

BehnamZohoor

Henry Lau
# Table of Contents

<table>
<thead>
<tr>
<th>Sections</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Team members</td>
<td>3</td>
</tr>
<tr>
<td>Project overview</td>
<td>4</td>
</tr>
<tr>
<td>Mission Statement</td>
<td>4</td>
</tr>
<tr>
<td>Project description</td>
<td>4</td>
</tr>
<tr>
<td>Minimum Specifications</td>
<td>4</td>
</tr>
<tr>
<td>Division of Labor</td>
<td>5</td>
</tr>
<tr>
<td>Guidelines</td>
<td>6</td>
</tr>
<tr>
<td>Teams</td>
<td>6</td>
</tr>
<tr>
<td>Meetings</td>
<td>6</td>
</tr>
<tr>
<td>Time Commitment</td>
<td>6</td>
</tr>
<tr>
<td>Guidelines (cont’d)</td>
<td>7</td>
</tr>
<tr>
<td>Team Interactions</td>
<td>7</td>
</tr>
<tr>
<td>Documentation</td>
<td>7</td>
</tr>
<tr>
<td>Disputes</td>
<td>7</td>
</tr>
<tr>
<td>Terminations</td>
<td>7</td>
</tr>
<tr>
<td>Agreement</td>
<td>9</td>
</tr>
</tbody>
</table>
Team Members

Nelson Chu

Computer Engineering

nchu1@ucsc.edu

909-974-8192

BehnamZohoor

Electrical Engineering

bzohoor@ucsc.edu

408-497-8413

Henry Lau

Electrical Engineering

hlau3@ucsc.edu
Project Overview

Mission Statement

To develop an arm mechanism for a robot designed to transverse around the ceiling as well as through different rooms. The main purpose of this robot is to help the elderly with day to day activities as well as chores around the house.

Project Description
The ceiling helper robot will be split into 3 projects, the rail and bogey, the arm, and the end-effector. The arm portion of the robot will be able to move around freely within a given space as well as hold its current position and withstand 5lbs of force at its maximum length.

Minimum Specifications

The minimum specifications of the robotic arm is to have full motion within its maximum range of motion, hold position, extend and retract, as well as hold 5 lbs at its maximum reach.

Additional Specifications

If time permits, this arm will be integrated with the other two parts that makes up the ceiling helper robot. The arm will receive information from the end-effectors which gets its data from either user input or the surrounding environment.
Division of Labor

Nelson Chu:

Design the robotic arm’s locomotion. Develop the high and low level state machines to control the arm's behavior.

Behnam Zohoor:

Create a computerized mockup of the robotic arm as well as the physical fabrication and construction of it.

Henry Lau:

Develop code to handle sensor feedback as well as the implementation of electronic circuits for the arm.
Guidelines

Teams

This project has been divided into three sub-groups; rail and bogey, arm, and the end-effector. Each team will comprise of three people that will select a team lead amongst themselves. Each team is responsible for their respective parts such that if one team fails, the other two do not suffer. If one team begins to falter, the other two teams can provide support. Complications and arguments within the sub modules will be handled within the sub module, whereas problems with a single group will be discussed and worked out by the entire team.

Meetings

Once a week, there will be a meeting amongst the entire team to discuss their group’s success, failures, and progress on their respective section of the total project. For any reason a member cannot make it to a team meeting, a 24 notice via email or sms will be required to inform the group so they have time to either reschedule or take notes for the missing member. In addition to the team meeting, each sub team will organized their own meetings for the development of their own portion of the overall project. The team leader for each group is responsible for providing an agenda via email or sms prior to each meeting. Another member of the group will serve as a timekeeper. This sub group meetings will take place at least once every two weeks.

Time Commitment

Each member of the group is expected to put at least 30 hours a week on their respective portion of the project. Each member should finish their assigned task within the time allotted by the Gantt chart they made. If any member of the group has finished their task ahead of schedule, the member must get some sleep if they have been lacking in it, assist in other parts of the project where needed, or focus on other classes. If a member is falling behind, they should ask for assistance from fellow team members.
Guidelines (cont’d)

Team Interactions

Each member of the group will be expected to give their full attention and participate during the team meetings. Members are responsible for leaving notes of their progress and fails as well as any changes that were done to the project.

Documentation

Each member is responsible for their own engineering notebook in the lab. All information including the block diagrams, schematics, graphs and idea related to the project are put in a team binder and be readily available to other members of the group. Datasheets and app-notes are also included in the team binder in an orderly fashion. The Gantt chart must be updated accordingly and any changes will be decided among the group.

Disputes

Any conflicts between members will be resolved through a meeting with the team leader. In the case that there is a problem with the team leader, the leads of other groups will be involved in the jurisdiction of that person.
Termination

If a member does not show any effort or commitment toward the project for more than a week, that person will be given a warning by team leaders. If that person does not show improvement in their activity towards the project in a course of a week, then the mentor of the project will be notified to make a settlement regarding the member’s inactivity towards the project. If no conclusion can be met, then termination from the group will be the last resort. There will be a team meeting excluding the accused member to discuss the termination of the member from the project. A vote will be proposed and the termination will be decided through majority vote and the final approval from the mentor.

Agreement

By signing below you will agree all the rules and policies regarding to ceiling helper robot, robotic arm, university of California Santa Cruz, Jack Baskin Engineering senior design Fall 2012.
Nelson Chu  Date

Behnam Zohoor  Date

Henry Lau  Date
Ceiling Robot: End-Effector

TEAM CHARTER
Contact Information

Phillip Wong
(415) 516-0112
pwong6@ucsc.edu

Thomas Gilbert
(916) 208-9350
thgilber@ucsc.edu

Tony He
(415) 816-7883
tche@ucsc.edu
Overview

Mission Statement

To build an end effector for a ceiling mounted robot that can take visual and audio commands. The end effector must also output protocol specific commands which the Rail and Bogey team and Robotic Arm can process.

Project Description

At the end of the robotic arm, there will be the end effector module which includes a gripper capable of picking up small objects or up to five pounds. This gripper will be controlled by a Kinect sensor capable of streaming visual and audio commands to our program in the Robotic Operating System (ROS). The program will make calculations with the streamed data to calculate the gripper’s location. Then the program will determine if the gripper is within the proximity of an object and control gripper’s pump.

The program will send instructions to the Robotic Arm team and the Rail and Bogey team which follows the communication protocol document.

Minimum Specifications

Establish ROS communication with microcontroller via UART. Identify human gestures as input commands. Create a ROS program to translate Kinect Point Cloud information to controlling gripper. Mount end effector onto a platform and demonstrate functionality while in motion.

In the event that the other teams aren’t able to complete their minimum specifications, we will demonstrate our project by showing our own minimum specifications without integration. The end effector will be mounted on a manual arm to simulate an autonomous robot arm and a trolley on wheels to simulate the bogey moving along the track on the ceiling.

Additional Specifications

Implement a vacuum chamber to reduce pump noise. Receive voice commands alongside with gestures. Integrate end effector with arm and bogey. Incorporate tablet as an additional user interface.
Division of Labor

Gripper Design/Assembly

The gripper will use a latex balloon full of a granular material (ie coffee grounds) to function as a universal jamming gripper. A desired object can be picked up by placing the balloon on the object and using a vacuum to draw the air out from the system and make the coffee grounds become closely packed and rigid. The balloon will be connected to the pump with PVC connectors and a silicone seal.

Kinect Development

Through ROS, we will use the Kinect to map the robot’s surroundings. The Kinect will send data to ROS which evaluates the video, infrared, and microphone array data and make control decisions based on these signals.

Microcontroller Software

Our ROS program will send standard protocol messages to microcontroller through a UART. ROS will control the gripper’s vacuum and pressure release valve.

Team Lead

The team lead will be in charge of organizing and directing weekly meetings. He will be in charge of ensuring that deadlines are met.

Deputy Lead

The deputy lead will be in charge of the budget and doing reviews of final pieces of the project to test for robustness of parts.

Minimum Time Dedication

Each member of the group is expected to work a minimum of twenty (20) hours per week including holidays and weekends.
Project Reimbursements

The Financial Officer is in charge of all team funds, collecting receipts, and providing reimbursements. He will dispense funds as requested per the guidelines: (1) Each member must provide a receipt to receive reimbursement for orders. (2) Approval by the Financial Officer for any expenditure below $50.00 is needed for reimbursement. (3) Approval by the entire team for any expenditure exceeding $50.00 is needed for reimbursement.

In the event that our funds do not cover all of our expenses for the project, the whole group of nine will split the remaining cost.

If a team or member of the team is at fault for damaged components or parts, and requires replacement of said component or part, the persons at fault are responsible for the cost of those purchases unless reimbursement is agreed upon by the other members and teams.
Code of Conduct

Meetings

Meetings will be held at least once a week in the designated lab for progress updates. Each member will speak to success and to issues halting development. Each member will be given a chance to offer ideas for how to solve the problems.

Meeting Conduct

In the meetings, the team lead will begin the meeting. After the overview, members will speak one at a time and wait until a member is done speaking to begin speaking themselves.

Engineering Notebooks

Each member must log their work in their engineering notebooks. The pages will be labeled by topic with page numbers. Notebooks must be brought to meetings so that work can be shown to group members and issues solved more quickly.

Decision Making and Dispute Resolution

Decisions will be made by a unanimous vote. As a group of only three, the person not at the center of a dispute will act as the mediator.

Termination Policy

A team member that consistently misses deadlines, is late or misses meeting, act disrespectfully to the other group members, or acts against the goals of the group, he can be put on a two week probation. After the probation period, if the issue is not resolved, he can be terminated from the group by unanimous decision.

Addendums and Charter Changes

As a group, we can decide to make changes or additions to any part of this charter by a unanimous vote.
Agreement

By signing below you agree to abide by the terms and conditions of the End Effector group of the Ceiling Robot project, Jack Basking School of Engineering Senior Project Design.

___________________________  _____________
Phillip Wong  Date

___________________________  _____________
Tony He  Date

___________________________  _____________
Thomas Gilbert  Date
Ceiling Helper Robot: Rail and Bogey Team

Team Charter
# Table of Contents

Contact Information 3

Project Overview 4

- Mission Statement 4
- Project Description 4
- Minimum Specifications 4
- Additional Specifications 4
- Division of Labor 5

Project Reimbursement 6

Code of Conduct 7

- Teams 7
- Meetings 7
- Time Commitment 7
- Team Interactions 8
- Documentation 8
- Decisions 8
- Disputes 8
- Terminations 9

Addendums and Changes to the Charter 9

Agreement 10
Contact Information

Kyle Huey  
Rail and Bogey Team Lead
Computer Engineer  
kyahuey@ucsc.edu  
(510) 798-6286

Joel Thompson  
Computer Engineer  
jjthomps@ucsc.edu  
(916) 337-1795
Project Overview

Mission Statement

To develop a cable driven robotic arm that will navigate a rail infrastructure on the ceiling of a room as an alternative to floor-based household robots.

Project Description

This rail and bogey team will focus on developing a rail infrastructure that will power the entire system and develop a bogey that can track its position and map out its environment.

Minimum Specifications

The project will focus on developing a rail capable of providing a sufficient amount of power for the entire ceiling helper robot, and developing a bogey that can navigate the rail and track its position with precision of few centimeters.

Additional Specifications

If time permits, the team will look to improve the accuracy of the bogey’s position tracking to a few millimeters and developing a static map of the environment.

If time permits, and provided that each sub-group has developed their project to meet their minimum specifications, the team will work to integrate the modules (rail/bogey, arm, end effector) into a consolidated system.
Project Overview (cont’d)

Division of Labor

Power System: Kyle Huey

- Responsible for developing a power system that provides a sufficient amount of power to the rail to power the entire system. Also will oversee the mechanical construction of the rail and bogey.

Software: Joel Thompson

- Code the software back-end of the project by write the libraries that send and receive information from the user interface.

Sensors: Kyle Huey

- Handle the connections between the sensors and the microcontroller, which includes working with analog to digital converting, interpreting Kinect output.
Project Reimbursement

The Financial Officer is in charge of all team funds, collecting receipts, and providing reimbursements. They will dispense funds as requested per the guidelines provided below.

- Each member must provide a receipt to receive reimbursement for orders.
- Approval by the Financial Officer for any expenditure below $50.00 is needed for reimbursement.
- Approval by the entire team for any expenditure exceeding $50.00 is needed for reimbursement.

Damaged Parts Reimbursement

If a team or member of the team is at fault for damaged components or parts, and requires replacement of said component or part, the persons at fault are responsible for the cost of those purchases unless reimbursement is agreed upon by the other members and teams.
Code of Conduct

Teams

There are three sub-teams involved in the overall ceiling robot project. There is one team assigned to the rail and bogey, one team assigned to the robotic cable-driven arm, and one team assigned to the end effector. Each team has a team lead assigned by the other members of the team. Although each team is responsible for their own tasks, members of other teams can provide support when possible. Unless otherwise noted, enforcement of the Code of Conduct is the responsibility of the smallest team affected. Therefore, disputes concerning only members of one sub-team must be resolved by the members of that sub-team, while disputes involving multiple sub-teams will be resolved jointly.

Meetings

The rail and bogey team will meet once a week in lab collectively for a status updates and planning. The weekly meeting time will be determined at the conclusion of the previous week’s meeting. A 24 hour advance notice is required if a team member cannot make the scheduled meeting and the meeting will be rescheduled.

The team lead will provide an agenda via e-mail prior to each meeting. If a meeting has a guest attendee, the team is required to arrive at least 15 minutes prior to the start of the meeting.

One member present at the meeting will serve as the timekeeper, to help enforce the agenda of the meeting.

The three sub-teams involved in the ceiling robot project will meet once every two weeks for status updates and to ensure compatibility between the three sub-projects.

Time Commitment

During the course of the project, each member is expected to contribute at least 25 hours per week to work on the project with at least 20 hours in lab and 5 hours inside or outside of lab at the discretion of the team member. A team member may not work more than 60 hours in a week without the approval of the project lead. If a team member has finished their assign task for the week, remaining hours can be spent maintaining the lab, assisting other team members, or starting on the next week’s assignment.
Code of Conduct (cont’d)

Team Interactions

Each team member is expected to attempt to get 6 hours of sleep each night and act in a professional manner while in lab.

Documentation

Each team member is responsible for maintaining their lab notebook. All schematics, block diagrams, and key design concepts must be made available in the team lab notebook. Also any programming code must be commented. The team Gantt Chart must be kept up-to-date by the project lead. Any changes to the Gantt Chart must be approved by all the members of the team affected by the change. All datasheets must be printed and filed in a binder that does not leave the lab.

Decisions

If a decision to be made does not have a unanimous agreement, an impromptu meeting will be called when all the members are available to meet. At the impromptu meeting, conflicting parties will have 10 minutes to define their arguments before a simple majority vote is called. If no majority vote is reached, the team lead will make the decision.

Disputes

Any personal disputes between two or more individuals will be resolved as soon as possible by calling an impromptu meeting between the individuals involved and the team lead. If the team lead is involved, a team lead from one of the other sub-teams must be present. In the event a dispute persists and affects the performance of the team, a course instructor will be called upon to enforce the resolution. If no resolution can be reached, uninvolved individuals must meet separately the course instructor to discuss possible termination of one or more of the involved individuals.
Code of Conduct (cont’d)

Terminations

A termination can be considered if an individual does not perform to the standards defined by this charter. A face-to-face meeting will be held between the individual and the rest of the entire Ceiling Helper Robot team to outline the issues and possible solutions. After one week if the individual fails to follow the team’s proposed solutions, a course instructor will be called to facilitate a resolution. If no resolution can be reached, a meeting with the entire team excluding the member under review will be held. A member will be terminated through a 75% majority vote pending the final approval of the course instructor.

Addendums and Changes to the Charter

Proposed modifications to the team charter can be made with a 75% majority vote.
Agreement

By providing your signature below, you agree to abide by the policies set forth in this charter written for the Ceiling Helper Robot: Rail and Bogey Project, Jack Baskin School of Engineering Senior Design Fall 2012 through Spring 2012.

__________________________________________
Joel Thompson

Date

__________________________________________
Kyle Huey

Date