April 2008

Roof Robot Phase 2

Xu Lin
Worcester Polytechnic Institute

Zachary James Strowe
Worcester Polytechnic Institute

Project Number: ECE-KZS-0701

Roof Robot Phase 2

A Major Qualifying Project Report:

submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the

Degree of Bachelor of Science

In Electrical and Computer Engineering

by

Xu Lin
Zachary Strowe

____

____

Date: April 24th, 2008

Approved:

____

Prof. Kenneth Stafford, Advisor

____

Prof. William Michalson

Acknowledgements

We would like to thank the following for their collective involvement in this project:

Kenneth Stafford

Brad Miller

William Michalson

Michael Ciaraldi

Turin Pollard

Matthew Goon

Abstract

In 2006 and 2007, a team of students completed a rooftop robotic platform for a sponsor. While that project performed admirably, there were numerous electrical, mechanical, and control issues with the robot. In order to address these issues, the team performed numerous enhancements on the previous year's design in order to grant the robot superior traction as well as a more centralized and detailed user interface. Work on the platform is expected to continue in future MQPs.

Executive Summary

In the previous year, a team of students developed a robotic platform for performing roof inspections. The platform met many of the target goals for the project, however there were still several points which were left to be improved upon. Beyond additional functionality, the previous team left several problems with the electrical system which would need to be worked on. Our task was to perform these required updates to the electrical systems of the robot, as well as adding to the overall functionality of the system.

Project Goals

At the outset of the project, we were told that, while the robot could maintain a stationary position on a 12/12 pitch roof, its maneuverability on that surface was severely limited. As such, we decided that one of our key goals with this project would be the development of a better traction control system. Being Electrical Engineers, we chose to focus our efforts on developing improvements in the form of updated control code for the robot. In tandem with this development, we chose to investigate additional sensors which might allow us to more easily develop further improvements to the robot control system.

Another major issue we were informed of was the camera system installed on the robot. The camera installed at the beginning of the project was a wireless security camera with its own transmitter and receiver system. This camera faced several problems in field testing, including significant interference while the transmitter was not pointed at the receiver and the receiver picking up video feeds from other cameras in the testing area. These issues prompted us to investigate new camera options that could either run through a controller or use a different communication method than the previous design. The desire to improve the camera also impacted our search for a new controller for the robot, as the Vex controller installed in the original design is a relatively weak controller in terms of memory and computational power.

After testing the robot in the beginning of the year, we also decided that some additional maintenance would need to be performed on the robot. The power system for the robot was overly complex, and we therefore decided to attempt to reduce the number of power sources required for the robot. In reducing the number of batteries, we would

also have the opportunity to re-wire several portions of the robot where connectors were coming apart from improper connection in the previous year. We also decided to improve the charging mechanism in the robot, as detaching the batteries in the previous configuration was challenging enough to be an annoyance.

Results

Overall, the majority of our tasks were completed sufficiently. Our major regret is our inability to replace the Vex controller with a more powerful system. However, new control code was developed for the Vex, allowing it to improve its performance. This improvement also was aided by the addition of new inclinometers on the robot, allowing it to sense the tilt and roll of each half of the robot independently. This new control code also used a new communication protocol and a newly designed user interface.

The camera system of the previous year was replaced with a new Wi-Fi camera system. This new system allows the robot to take pictures and video of its surroundings, as well as combating the interference problems displayed in the previous year. The new camera is significantly bulkier than the previous model however, and additional research may result in a more compact solution in the future.

The power system wiring of the robot has been completely redone. In addition to repairing some of the weak connectors, the ring terminals have been replaced with spades to ease maintenance work in the limited space available in the robot. In addition, the previous circuitry had no circuit protection, and so self-resetting circuit breakers were added between the batteries and the motors. Additional fuses were added to the battery lines in order to prevent batteries from being connected backwards.

Recommendations

The majority of the major electrical work has been done at this point, however there are still major improvements that can be performed on the robot. The most important of these would be to replace the current Vex controller with a more powerful system. This component is currently the weakest link in the overall control structure of the robot, and replacing it would allow significant improvements to be made, as well as possibly reducing the number of additional components used for communication. In addition, a new controller could allow the camera system to be integrated more fluidly into the controller. This would allow for a smaller camera to be used in place of the current model.

Additional work could also be performed in research for new sensors to be added to the robot. New sensors would allow more data to be gathered about the robot's operation as well as the roof to be inspected. Any additional sensors would also suggest an improvement to the current user interface. Some desired functionality in the user interface is lacking from the concept design, leading further work to be necessary in that area as well. The final electrical issue would be to continue to modify the current wiring scheme to maximize power efficiency and minimize stress on the battery packs, which is another issue which could potentially be resolved by using a new controller.

Although not electrical problems, there are also some points which require further mechanical work. The wheels could probably be improved from their current performance through further research into traction and wheel structure. Additional changes might also be in order for the chassis to contain additional sensors and electronics. Finally, an ascender system should be designed to allow the robot to access a third story roof from ground level. With the current design, this ascender would also have to accommodate a wireless access point with the robot.

Table of Contents

Table of Figures

Introduction

In 2006 and 2007, a team of MQP students designed a robotic platform for performing roof inspections. This robot was a 4 wheeled vehicle with a unique center joint designed to allow separate articulation while traversing a rooftop environment, as shown in Fig. 1. The wheels of this vehicle are covered in EPDM foam rubber to improve traction between the wheels and the roof surface. The camera mounted on the top of the robot is used for both navigation and visual inspection of roof areas. There were some minor mechanical issues with the robot at the outset of our project, but the robot itself is an impressive mechanical design.

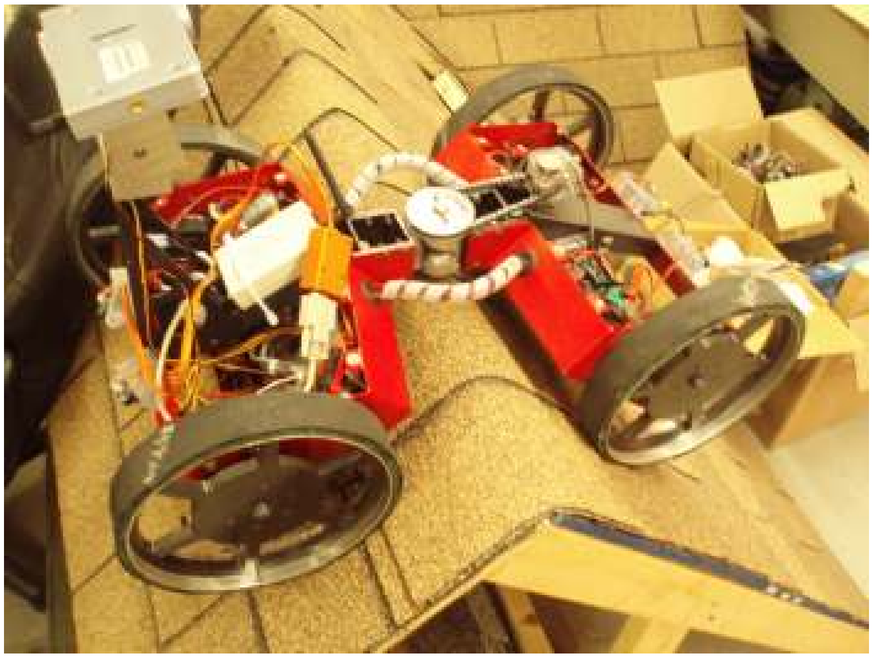


Fig. 1. Roof Robot

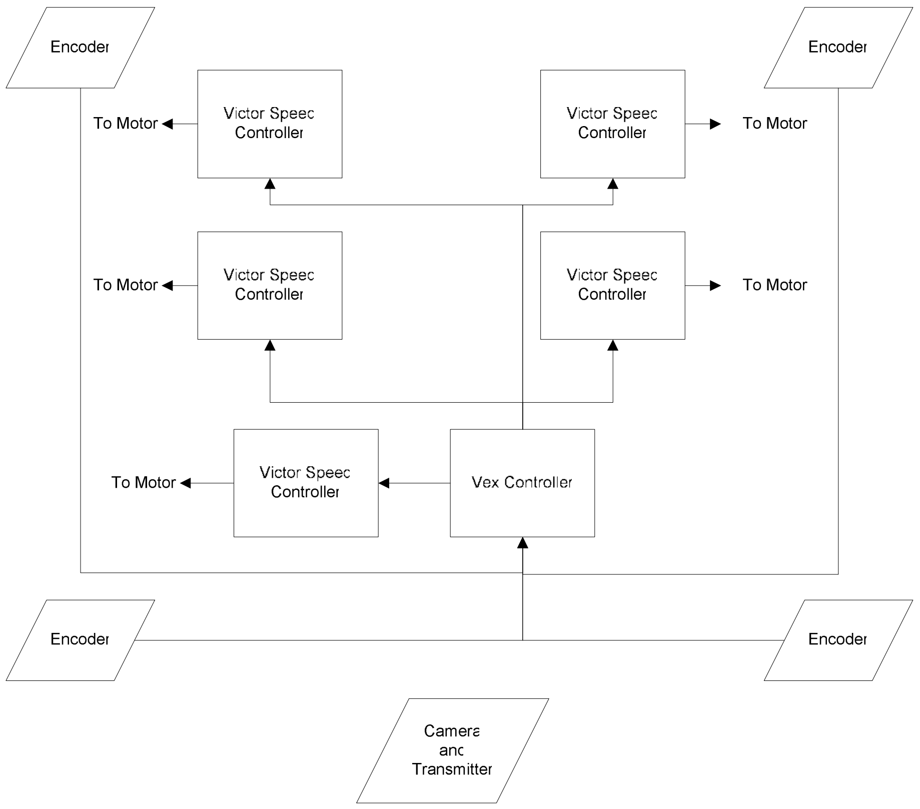


Fig. 2. Robot Electronics Layout

However, there were several areas electrically which required improvement. The general layout of the electronic systems can be seen in Fig. 2, although power is not shown in that diagram. The controller for the robot is a relatively low end controller, called a Vex. This controller has two major drawbacks in this setting; a need for a different voltage power source from the rest of the robot and a normally single way communication scheme. Additional problems were encountered with the camera in the previous year, as the camera suffered from interference as well as having a very directional antenna. The robot also had no circuit protection installed, which could result in damage to the system in cases of stalled motors or improper battery connection.

Although less technical in development, there were also several wiring improvements that were deemed necessary. The system as a whole previously ran off of multiple battery sources, requiring multiple charging methods. Aside from the challenges of tracking multiple chargers, removing the batteries for charging was inconvenient and time consuming. Rewiring some portions of the system was also desirable, as the motor speed controllers were not wired in a logical pattern. Turning the robot on was also more involved than necessary, as there was effectively a start-up sequence that the user would need to perform to have the robot activate properly.

Our initial goal was to tackle all of the aforementioned electrical issues with the system. After exploratory research on several different controller products, we decided to remove the update to the robot controller from our list of challenges to face. We replaced that goal with a new concept idea to improve the robot's usability through a new user interface. Our final objectives were to improve the robot's control code, improve its overall usability, and to perform general maintenance and improvements on the electrical systems as a whole.

1 Background

In order to better understand what upgrades and modifications were required to improve the robot, research was performed in several areas. To address the control issues, we studied other robots which had to perform well on non-level terrain. Improving the camera was also a priority, which resulted in additional research into video compression. Product research was also performed in regards to several areas of improvement, and will be discussed more thoroughly in the Methodology section of this report.

1.1 Control Algorithms

One of the major concerns with the robot from the previous year was that its desired goal of navigating a 12/12 pitch roof was only partially met. While the robot could maintain its current position on such a roof, its mobility was severely limited. In order to address this issue, several new control algorithms were researched to improve traction through control code. Understanding traction control algorithms required a basic knowledge of how traction normally functions.

$$F = \mu * F_N \qquad (1)$$

The most basic aspect of traction is the application of frictional force. Calculating the force of friction on a system can be performed using (1). The symbol μ represents the coefficient of friction, while F_N represents the normal force on the system. While the coefficient of friction generally considers either a static system or a dynamic system, for purposes of our project we use the coefficient of static friction. The opposing force to friction in our case is the traction force being generated by the wheels. The traction force is the total driving torque of the wheel divided by the radius of the wheel, as shown in (2).

$$F_T = T * r \qquad (2)$$

Another important aspect of traction used in control is the concept of slip. The slip ratio (λ) refers to the differential velocity of the system as a whole against the velocity of the wheels for that system, as described in (3) below. Allowing for some slip to occur in the system is unavoidable, however too much slip can bring the system out of control. This allowable slip is also determined by the coefficient of friction used in the system, as shown in Fig. 3. The code generated by the previous team used a modified slip control system that would monitor the velocity of the wheels in relation to one another. While this control system allowed the robot to self correct to a limited extent, the lack of sensors allowed only limited feedback to the system and the overall control style was therefore forced to use a proportional control scheme. Two of the more promising control possibilities are described below.

$$\lambda = \frac{V_W - V}{V_W} \tag{3}$$

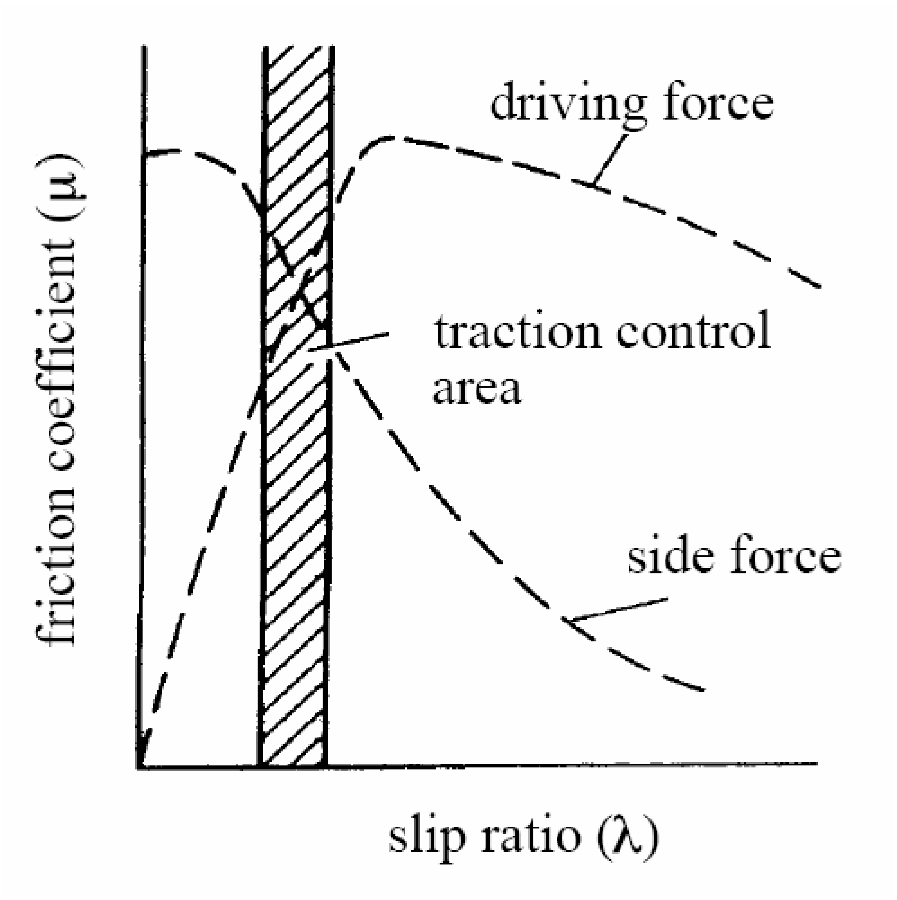


Fig. 3. Slip Ratio versus Coefficient of Friction [1]

1.1.1 Model Following Control

The more comprehensive of the two control styles is the Model Following Control (MFC) style. In this type of control, a model of the system is made and the results expected for the inertia of the system are simulated. Those simulated results are then used in the system and compared against the actual values being read by the onboard sensors. The equations to perform these checks are shown in (4) and (5) below. They use the moment of inertia of the system (J), the moment of the drive shaft (J_W), the ideal moment (J_{MODEL}), the mass of the vehicle (M), the radius of the wheels, and the slip ratio. Comparison between the real and expected moments of the system can then be used to adjust the drive train on the vehicle with high precision. While this system is very accurate, it also requires a complex system of sensors to be truly effective. Also, the computational power to run these constant simulations and differentials is much larger than the second type of control style explored.

$$J = J_W + M * r^2 * (1 - \lambda) \tag{4}$$

$$J_{MODEL} = J_W + M * r^2 \tag{5}$$

1.1.2 PI with Slope Control

The PI with Slope Control is similar to the original control code designed in the previous year. This control scheme uses a simple PI control system while also including a slope measurement to adjust its controls accordingly. While this system is less precise than the MFC control system, it is substantially less computationally intensive on the controller. Another benefit of this control scheme is the limited number of additional feedback required to implement the system, as it only requires some method of detecting the slope of the system. While both control styles were at least partially tested, a modified form of the PI with Slope system was the final implementation and is described in detail in the Methodology section of this report.

1.2 Video compression

One of the largest concerns when sending high resolution video information between two systems is limiting bandwidth consumption. Picture and video data can quickly consume large quantities of bandwidth; a 640x480 camera transmitting at 30 frames per second would use over 27 megabytes per second in uncompressed transmission. Unless we develop and maintain very high bandwidth networks, we have to compress the data. There are two types of video compression, lossless and lossy compression. Lossless compression is data being compressed in such a way that it can be reconstituted without loss of detail or information. These are referred to as bit preserving or reversible compression systems. A lossy compression is any method of data compression that reconstructs the original data approximately, rather than exactly. The losses are often imperceptible to the human eye, unless extreme compression is performed on the data.

There are many different compression standards used in the world today. The most commonly encountered ones are MPEG-1, MPEG-2, JPEG, DivX, and FLV. MPEG-1 and MPEG-2 are similar in general application, with the major differences between the two being that MPEG-1 is generally optimized to operate at 352x240 at 30 frames per second while MPEG-2 is a higher resolution image at 720x480. This means that aside from quality of image, the primary difference is in total bandwidth consumed for those two compression schemes. JPEG has traditionally been used as a static image storage method, but can also be used as a video feed when the MJPEG standard is used. DivX is a company brand name, but is also a type of video codec. DivX is actually a form of MPEG-4 encoding, which takes many of the portions of MPEG-1, MPEG-2, and other video formats. This compression format allows a wide range of quality and is only roughly standardized. The FLV, or Flash Video format, is a type of compression that is commonly found on web sites with video. Flash video is generally lower in resolution that other video compression formats, but this stems from its extensive use in online situations.

All of these compression standards are based on one or more of the following compression algorithms; the Discrete Cosine Transform (DCT), Vector Quantization (VQ), Fractal Compression, or the Discrete Wavelet Transform (DWT). The Discrete Cosine Transform algorithm is the basis of the JPEG and MPEG standards. This algorithm transforms an image into the frequency domain, samples at regular intervals, and then works on smaller pieces of the desired data. DCT then discards those data which do not affect the image in any human perceptible level. Vector Quantization is a lossy compression algorithm that looks at an array of data, instead of individual values. It can then generalize what it sees, compressing redundant data, while at the same time retaining the desired object or data stream's original intent. Fractal Compression is a form of vector quantization technique. The compression is performed by locating self-similar sections of an image and then using a fractal algorithm to generate those sections. The Discrete Wavelet Transform converts an entire image into the frequency domain, and produces a hierarchical representation of an image in layers. Each layer represents a different frequency band located within the image.

2 Methodology

In the beginning of the project, we held meetings with our advisors and our sponsor to determine what changes and improvements needed to be made to the robot. While our sponsors desired some basic additional functionality, our advisors informed us of several design issues which had previously been forgiven for the sake of completing a prototype on schedule. The issues we chose to explore were as follows:

- Improving the overall traction control of the robot
- Improving the camera on the robot
- Unifying the power for the robot's systems
- Easing further maintenance on the robot electrically
- Modifying the controller to improve overall performance
- Creating a new user interface

2.1 Traction Control

One of the primary concerns our sponsor had was the robot's limited ability to react on a 12 pitch roof. In order to determine the physical reason for the robot's tendency to slip and fall on higher inclines, we tested the performance of the robot on that slope to determine the mode of failure. Through our observations, we determined that the robot was not slipping equally on all wheels when a slip was occurring. From a stopped position, the robot would begin to spin all of its wheels with equal speed as the control program dictated. However, with the lighter weight load on the wheels upslope, the front wheels of the robot would actually begin to physically spin before the wheels at the back could move. This behavior would cause the front wheels to lose traction, and begin to slide which would then destabilize the entire robot from its position. In order to counteract this problem, we performed research on control algorithms which could be used to increase the performance of the robot through code. The overall structure of the control code can be seen in Fig. 4. This research also helped us determine what types of additional sensors might be useful for improving traction performance.

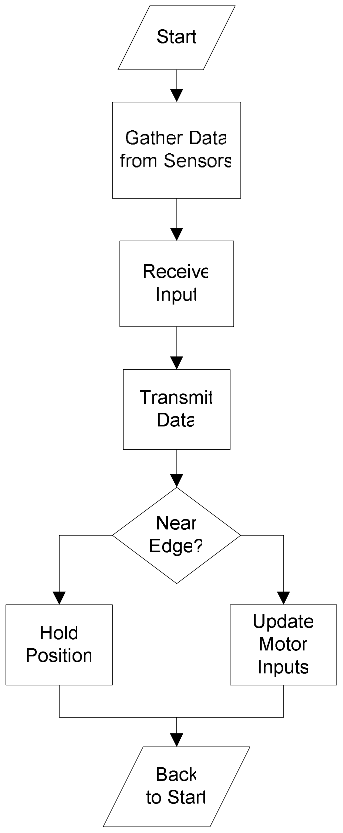


Fig. 4. Robot Control Flow

2.1.1 Original Code Design

Before developing our own algorithms, we chose to analyze the code that had been written in the previous year. The code had been written by a team of mechanical engineers, and the structure of the code was less than optimal. As such, one of our first tasks was to revise the code completely in order to make it more readable in the future. Originally, the control code was written using primarily global variables, which in turn resulted in the majority of the functions taking no parameters and returning void. After rewriting the original version in more standard C, we discovered that the Vex controller was so limited on space that the use of many global variables and small parameter lists was necessary to force the code into memory. Despite this issue, we did manage to perform some minor revisions which made portions of the code more readable.

The general structure of the code is shown in Fig. 4, but a detailed description of its operation follows. The code began by checking the starting position of the center joint to prevent the robot from jittering at startup and initialized a sequence of timers and encoders for the wheels, seen in lines 81-95 in Appendix B. This led to the main loop of the control code, which is an infinite "while" loop of all the key functions to control the robot. The code would then check the values generated by the remote control and store them internally through the getInputs() function. It then took the input values and used them to update the target values of the motors and servos on the robot, calling a sequence of functions in lines 178-332. A breakdown of the actual code design can be seen in Fig. 5.

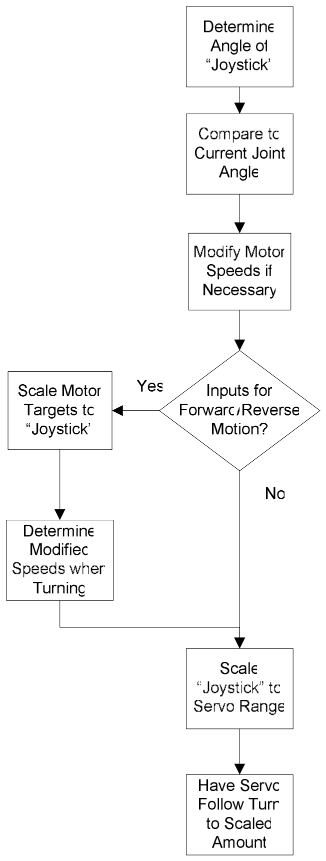

Determine Angle of "Joystick"
Compare to Current Joint Angle
Modify Motor Speeds if Necessary
Inputs for Forward/Reverse Motion?
Yes
Scale Motor Targets to "Joystick"
Determine Modified Speeds when Turning
No
Scale "Joystick" to Servo Range
Have Servo Follow Turn to Scaled Amount

Fig. 5. Determining Motor/Servo Targets

Once the target values had been determined, the robot checked the center joint and the wheels for feedback, and the wheel speeds were calculated based on the inputs and feedback from the system in lines 419-520. An "if" check connected to a short loop then checked the range finders to make sure that the robot is not near the edge of a roof, and stopped the motors if it was. The final functions would send the finalized control signals to the motors and camera servos before looping back to repeat the process again. Our final code was very similar to the original, and reused the majority of the code. New functions were added to check the inclination of the robot and to output useful data to the user interface we developed, as well as extracting the original driveMotors() and drivePanTilt() functionality and placing it in the main loop to reduce the space used on the controller.

2.1.2 Sensor Selection

After reviewing several papers, [1]-[3], on traction control, the most useful type of sensor to add to the robot appeared to be an accelerometer. By adding this type of sensor, we would be able to generate additional data for the robot. Our theory was that by mounting accelerometers on both halves of the robot, we would be able to more accurately determine the current speed and of the robot as well as the current inclination of the system as a whole. We initially hoped to use the new sensors in conjunction with the wheel encoders in order to accurately determine the position of the robot. This could then have been used to measure the actual distance covered by the robot, as well as potentially creating a function to automatically return the robot's starting position. This idea was not implemented, as we discovered that the actual accuracy of both sets of sensors would not be suitable to the precision needed for those functions to work.

After determining what type of sensor we wished to explore, we began researching the specifications we would require of the sensor. We determined that the sensor we chose should be able to read angles of at least 50 degrees to insure full operation on the robot, which would reach 45 degrees. We also determined that we would require either a 3-axis or multiple 2-axis accelerometers per side in order to determine the robot's pitch and roll. The sensors would also be required to sense at least normal gravity to determine pitch and roll. The sensor that most closely followed these specifications was the Analog Devices ADXL330, which was a low power 3-axis accelerometer. Before purchasing this sensor, we were recommended to speak to Professor Furlong, from the Mechanical Engineering department, who had some knowledge of the types of sensors we were investigating.

In speaking to Professor Furlong, we learned that our initial specifications were flawed. Aside from being incorrect about the level of accuracy provided by these sensors over time, we were also incorrect about the forces that would be acting upon the sensors. He informed us that our results would be greatly influenced by any unevenness while driving the robot, and that these sudden changes of acceleration would affect our reference to earth gravity as well. He did provide a suggestion toward using a type of sensor called an inclinometer. This type of sensor does not need to be held smoothly to output its data, although we did lose the ability to quickly detect slippage through unexpected accelerations. The other benefit to using an inclinometer was that it greatly simplified the code required to translate the raw data from the sensor to a meaningful value we could use. The final sensor we chose to use in our project was the VTI Technologies SCA100T-D02 inclinometer, which is a 2 axis inclinometer that can read to a 90 degree incline. Fig. 6 shows the PCB we designed to mount the sensor in the robot. Although we chose to use an analog output signal from the sensors, this PCB contains both an analog and serial output circuit for future controllers. We separated the two sensor versions by cleaving the board and mounting only the analog half.

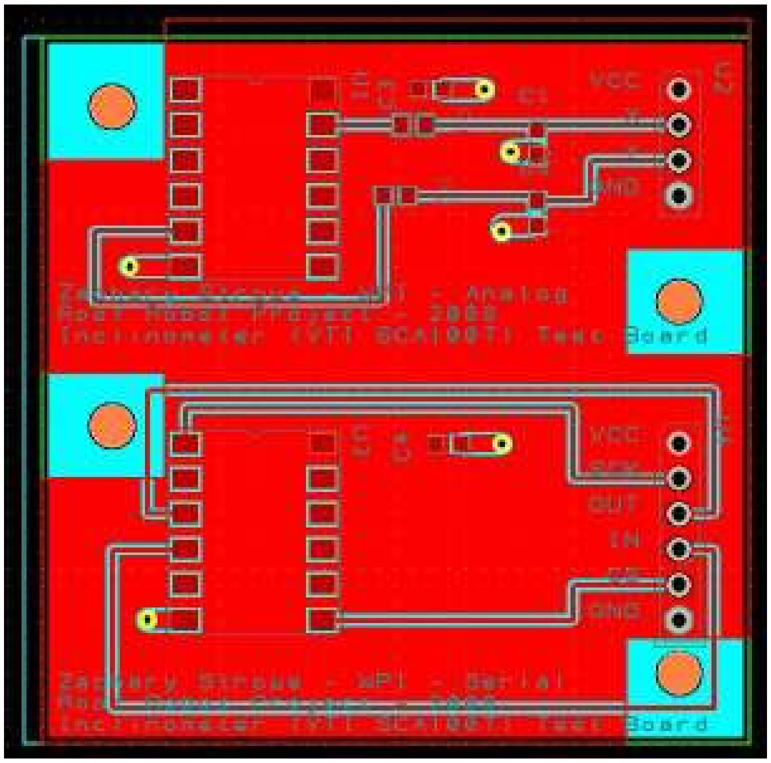
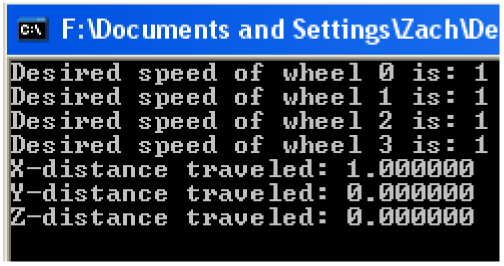


Fig. 6. Inclinometer PCB Design

While the inclinometer choice made coding substantially simpler, there was a cost involved. Accelerometers could potentially have been used in our system as an alternative to the single chip inclinometer option we chose. The 3 axis accelerometers we investigated were priced at less than13 dollars per chip, while the inclinometers were over 65 dollars per chip. While this price difference was large, the benefits of the inclinometer went beyond code reductions. In order to have the design function as an incline sensor, we would need to develop a substantially more complex mounting board for the circuitry, as well as purchasing additional components. We managed to defray some of the costs involved with the inclinometers through samples donated to our project by VTI Technologies. In addition, further production costs of the inclinometers, as well as the PCBs which hold them, would of course be lower than our prototyping costs.

2.1.3 Truth Model

While designing the new control code for the robot, Professor Michalson suggested that we develop a truth model in order to better understand the system we were working with. Our truth model was a simulation, developed in C code, which allowed us to manipulate a virtual copy of our robot in order to determine the expected sensor outputs and position of the robot given set inputs. This proved very useful in development, as it not only allowed us to test algorithms in a simulated environment, but also identified several bugs with algorithm ideas without the need to load and test the robot with buggy code. The first iteration of the code was designed to accept an input speed on each wheel and apply it to the system for 1 second while determining the robots theoretical position in xyz coordinates. Later revisions included turning radius, as well as demonstrating the difficulty in determining the forces on the robot while turning. A screen shot of the model is shown below in Fig. 7.



```
Desired speed of wheel 0 is: 1
Desired speed of wheel 1 is: 1
Desired speed of wheel 2 is: 1
Desired speed of wheel 3 is: 1
X-distance traveled: 1.000000
Y-distance traveled: 0.000000
Z-distance traveled: 0.000000
```

Fig. 7. Example Run of Truth Model

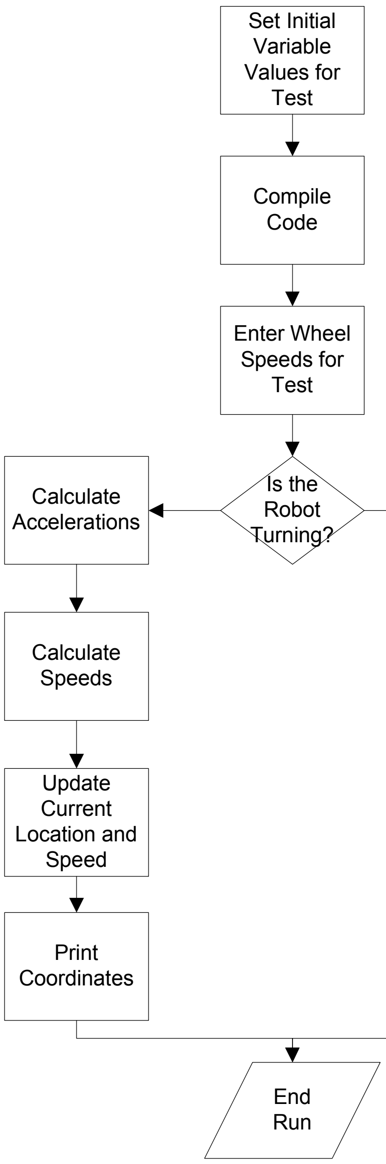

Set Initial Variable Values for Test
Compile Code
Enter Wheel Speeds for Test
Is the Robot Turning?
Calculate Accelerations
Calculate Turning Radius
Calculate Speeds
Print Turning Radius
Update Current Location and Speed
Calculate Forces on Wheel
Print Coordinates
Print Force to Counter Turning Forces
End Run

Fig. 8. Truth Model Flow Diagram

Operation of the truth model was relatively simple, and the flow of the program logic can be found in Fig. 8. Before compiling the code, we would set the variables controlling the overall pitch, yaw, and roll of the robot in the simulation. Although it would have been possible to modify these values as additional inputs, we chose to hard code them initially to minimize repetition when running repeated tests. Once the choice of orientation was made, the code was compiled and the input sequence shown in Fig. 7 was used as the feed into the virtual robot. From this point the code would check to see if the robot was traveling in a straight line or turning. If it was traveling in a straight line, then the new expected coordinates of the robot would appear as the output. If the robot was determined to be turning, then the program would call a function to determine the turning radius of the robot. With the value of the turning radius given, the code would then attempt to determine the forces on each of the wheels. The first version of the code would run for a set time in seconds, while later revisions accepted variations in time and subdivided the amount of runtime into tenths of a second. This caused some problems with accelerations and velocity calculations, as values would jump from at rest to full speed in one tenth of a second, skewing results. Although the version of the code given in Appendix D works with accelerations, future versions of the code more closely resembled the algorithm code given in Appendix C.

The truth model was designed to be as accurate to reality as possible, but some exceptions were made. As we were unable to accurately describe the action of the center joint, we treated the robot in our model as a solid chassis platform. In addition, as we are unable to generate truly continuous data flow into the pitch, roll, and yaw of the robot, we were forced to use small discrete time slices and slowly modify our inputs when looking for detailed results. This time jumping also caused some problems in initial runs of the model, as some pieces of the model would be calculated as instant jumps in motion which in turn caused the acceleration and force values to be unreliable. Despite these issues, the truth model was also modified later in the project to generate the weight distribution on the robot and test the algorithms for determining that distribution.

2.1.4 Tread Testing

Although not electrical in nature, additional testing was performed on the physical traction material of the wheels themselves. The team in 2007 had performed some research and performed tests on different materials for the tread surface as well as the substrate to be used below the tread. It was suggested that in addition to examining control code improvements that we investigate traction in further detail. While we performed additional research on materials, we did not come to any decision about a material that might be better for our purposes. Another theory we decided to test was to generate several variations on the original tread pattern to potentially increase the coefficient of friction on the tires. We did achieve some results with that line of testing, but our testing setup was not very accurate and the results lacked true repeatability.

2.1.5 Algorithm Implementation

We developed a new code modification in the form of a novel weight distribution algorithm to upgrade the overall performance of the traction control system. The general form of the algorithm is shown in Fig. 9, and a complete version of the code can be found in Appendix C. The system takes in the inputs from the inclinometers and performs an arcsin function on that information, as per the equations in the data sheet. With the angle of incline now calculated in radians, the controller takes the front and back tilt values and uses a tangent function with the length of the robot to distribute weight between the front and rear. With the weight now distributed between front and back, similar functions are performed to determine the roll on the front and back sections independently. Once the full weight of the vehicle has been distributed, the load on each wheel is compared to a threshold value and the control signal to each wheel's motor is modified according to the current heading of the robot and weight on that wheel.

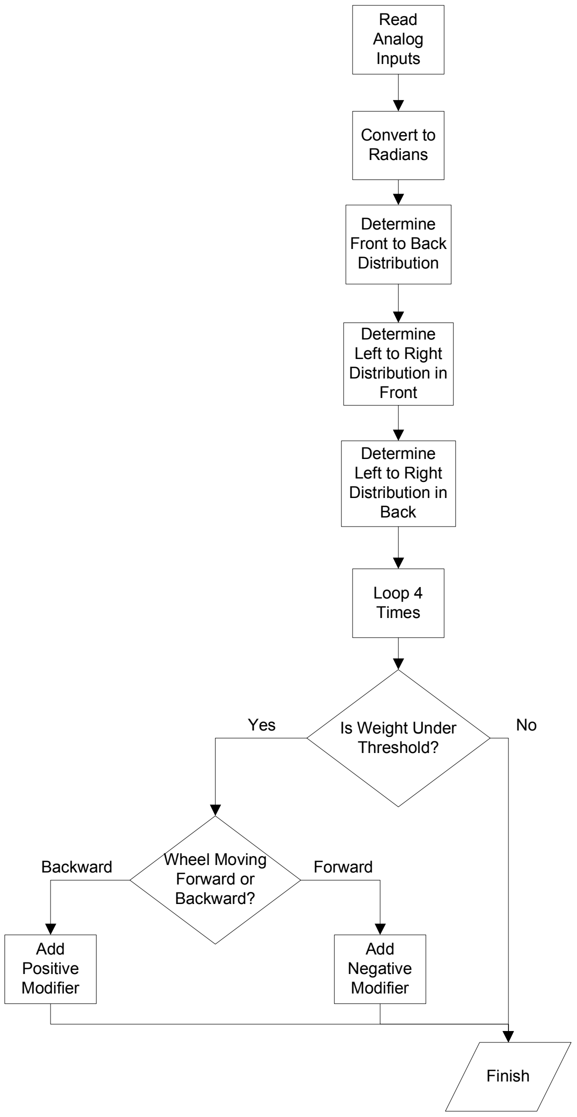

Read Analog Inputs
Convert to Radians
Determine Front to Back Distribution
Determine Left to Right Distribution in Front
Determine Left to Right Distribution in Back
Loop 4 Times
Is Weight Under Threshold?
Yes
No
Wheel Moving Forward or Backward?
Backward
Forward
Add Positive Modifier
Add Negative Modifier
Finish

Fig. 9. Algorithm Flow Chart

2.2 Upgrading the Camera

The original camera mount on the roof robot was an X10 wireless home security camera, as seen in Fig. 12. The X10 wireless camera runs off a different power source than the robot, which results in multiple power sources being required to provide power to the systems on the robot. Another drawback of the X10 camera was its ability to transmit information to the user; the camera used a standard frequency for video signals as well as a directional antenna. The directionality of the antenna transmission resulted in a loss of data when the robot was not in line of site. Analysis of the camera's antenna output is shown in Fig. 10 and 11, where a 23 decibel difference in signal strength is visible. This also created a situation where the receiver for the camera would then pick up other wireless cameras in the area, displaying the video feed from the newly acquired signal. The robot now carries a Hawking Technology network camera for navigation and a visual inspection, as seen in Fig. 13. In the current prototype, we upgraded to this new camera in order to address both the directionality and interference from the previous camera.

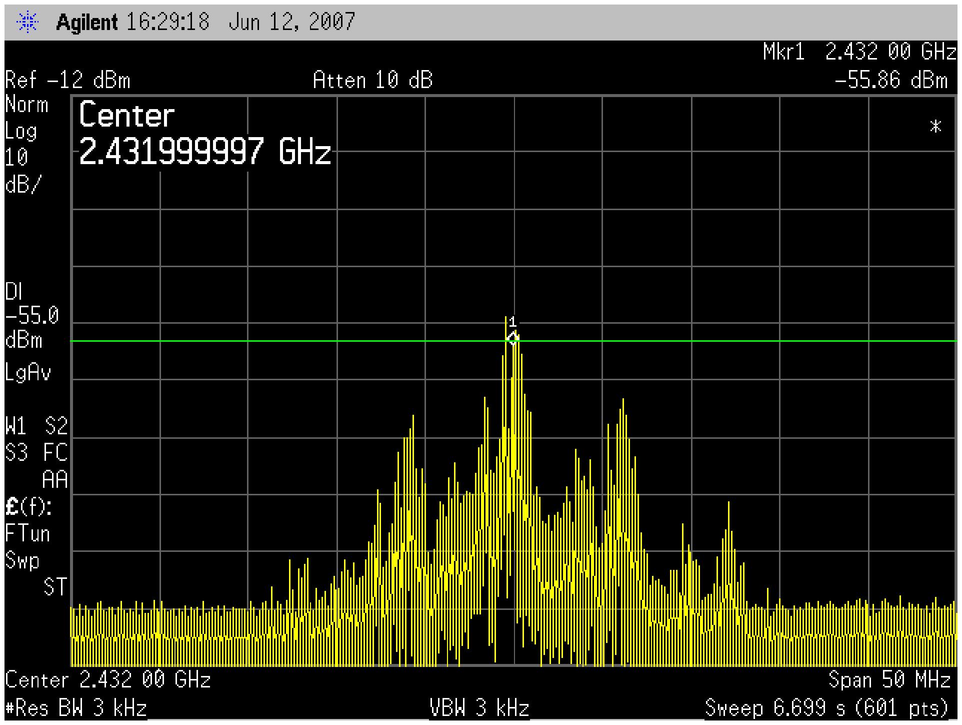


Fig. 10. Antenna Facing Receiver

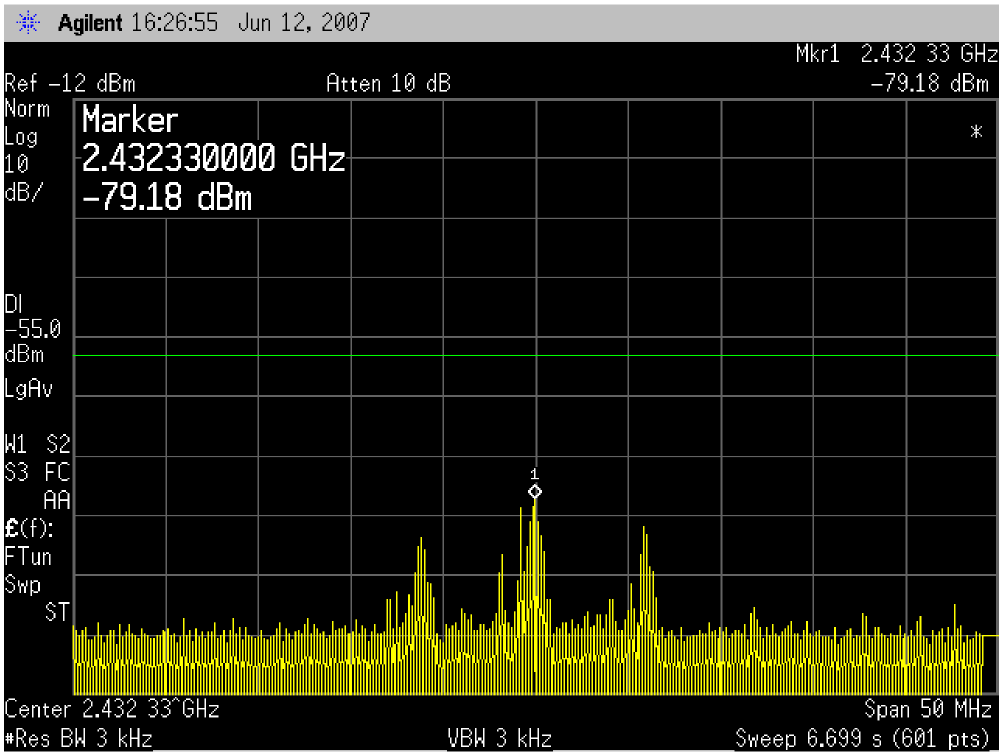


Fig. 11. Antenna Facing Away from Receiver

At the beginning of the project, we investigated several different camera options for the robot. Webcams have a very attractive price point, as they can be found as inexpensively as 20 dollars. Our initial choice was to use a webcam, both from a pricing standpoint and because we were investigating the Qwerk controller which had built in support for webcam feeds. Unfortunately, with the Qwerk microcontroller being unsuitable to our application, we began to investigate other alternatives. As we decided to continue using the Vex microcontroller, a wireless camera with its own transmission system was needed to record roof inspection data. To this end, we began investigation IP cameras, which offered similar functionality to the webcam option, but with a substantially higher price. The Hawking Technology camera which we installed was donated by Professor Michalson, which removed the immediate need for further cost analysis on that component. The router we used for the system was an inexpensive Linksys router, chosen for being the least costly of the wireless routers readily available. The router allows the camera and computer controller to communicate through a wireless link effectively, but a more powerful router may be appropriate in future work to increase range and reduce the chance of signal loss.

The camera is located at the highest point on the robot, mounted on a raised arm in the rear of the robot. In this position, the user can use the camera for navigation as well as inspection through the pan and tilt servos mounted on the bracket. The level of servo articulation allows the user to maneuver the camera through a wide field of vision, as well as being able to look almost directly down at the main body of the robot. The camera's resolution is high enough to provide an accurate sense of the robot's surroundings, as well as following the motion of the robot as turn commands are given. An additional feature in this version of the control scheme is the ability to center the camera with a single command, allowing a quick way to return the camera to a known forward facing position. As the camera is a Wi-Fi based, it no longer suffers from the interference displayed in the X10. The camera control software included with the Hawking camera also allowed us to generate snapshots and video feeds from the camera while in operation, saving us development time on creating our own custom version of such a system. The final benefit to this new camera model was the new power requirement of 12V. As this power level matches the power already supplied by the motor battery packs, a simple additional power feed from the central hub on the fuse block allowed us to quickly connect this camera without the need for any additional batteries or circuitry.

Fig. 12. X10 Camera

Fig. 13. Hawking Technology IP Camera

2.3 Unifying Power Sources

The robot at the outset of the project used 3 different power sources on board the robot, as well as requiring power for both a remote control and a laptop to be used from the ground. Externally, our new control system removed the need for an additional charger for a remote control battery. Replacing the old camera system removed one of the three sources from the robot. The other source on the robot was the special supply for the Vex controller. As the Vex controller is required for the operation of the robot, it was not possible to simply remove it from the design. Instead, we modified one of the battery packs that drive the robot's motors to supply the 7.2 volts required by the controller. In addition to reducing the number of sources on board the robot, we also added additional protection to the batteries in the form of fuses. The inspiration for this change came when we were connecting the batteries to the robot to test it and reversed the connection between 2 of the packs. This resulted in one of the battery packs being burned while the other was still usable.

2.4 Easing Maintenance

In addition to rewiring the batteries and simplifying the overall power scheme of the robot, we also made some adjustments to the internal wiring of the system. We chose to move the power switch from its old position in the front of the robot to a new location to allow the user to more easily reach it. This move also allows the user to more easily replace the battery packs in the front of the robot. In addition to moving the switch, it also now controls the ability of the robot to be charged. We created a single charging port on the robot in order to allow all of the batteries to be charged simultaneously via a single charger. This port is wired through the power switch in order to insure that the robot cannot be charged while in operating mode.

Additional wiring changes were made between the robot's motors and the switch. Each of the motors on the robot is now wired through a self-resetting fuse block in order to protect the motors from burnout, as seen in Fig. 14. Additionally, the motors have been rewired in such a way that the front portion of the robot controls the four wheel motors while the rear half holds the majority of the electronics and the speed controller for the center joint motor. In order to accommodate the increase in wires running between the two halves of the robot, the holes which had previously been drilled needed to be expanded for the new load. After dealing with the large hassle of disconnecting the speed controllers from their power, and two from the motors they had been previously connected to, we decided to replace the previous ring terminations with spades to ease further electrical maintenance. The last change made was to correct the color matching between the motors and their speed controllers.

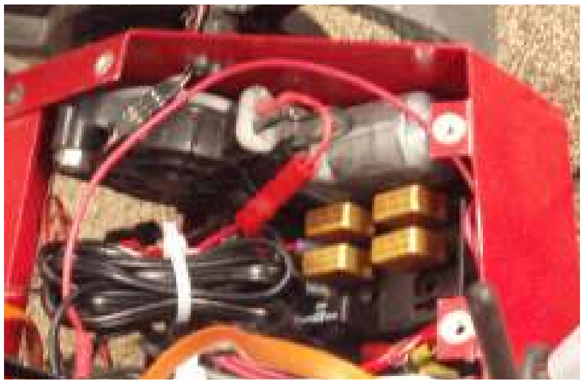


Fig. 14. Self Resetting Fuses

Although additional expenses are incurred for self-resetting fuses, the 20 amp self-reset fuses allow the robot to potentially recover from situations where one motor becomes overworked. While self-resetting fuses were the logical choice between the motors and the power source, the fuses between the batteries themselves did not require this self-resetting behavior. Without the need to use the more expensive self-resetting fuses for the battery connections, we instead chose to purchase standard in-line fuse holders and the appropriate fast acting fuses. Any potential issue where the batteries are the source of an electrical problem should occur long before the robot is actually in the difficult to reach roof position. The robot was originally supplied with three 12V battery packs, one of which was destroyed when we attached it incorrectly before a test run. The choice to buy additional batteries was therefore a necessary expense to insure that we could continue to work on the robot in the case of future problems.

2.5 Control System Modifications

At the outset of the project, we believed that the Vex platform would be insufficient to our needs in terms of both processing power and communication methods. As such, we began assessing other possible control systems to replace the Vex on the robot. We examined several other systems, including a substantially deeper look at the Qwerk system mentioned in the previous project report. We did not find any of the alternative solutions acceptable, and we therefore decided to expand the Vex controller. Our eventual solution was to attach a wi-fi to serial converter to the robot and rewrite the robot control code to accept a custom set of serial commands we designed.

2.5.1 Other Controllers

We explored several options in alternative controllers for the robot. We used several criterions to determine the viability of alternate controllers to the Vex. The first was expandability, as the Vex controller was currently using almost all of its motor and interrupt ports. Secondly, we were looking for a replacement that had, either built in or readily available, some way to communicate outside of a proprietary remote control. We were also looking for a product which would have more processing power and memory for holding progressively more intensive programs, both now and in the future. The overall size of the controller would also be an issue, as space was already fairly limited within the robot. Robustness was also a requirement, as the robot is designed to be outdoors and will potentially be traversing rough terrain. We also wished to maintain a low cost solution which further directed our choice of products.

What we determined from our research was that we were in a fairly difficult position. The problem we faced was that our requirements seemed to hit a sparse area in the overall market of controllers. While there are many robotics controllers available, most of them have relatively limited numbers of inputs and outputs for control. These controllers did have substantially more processing power and memory room than the Vex though. We were also guided to talk to Professor Ciaraldi about other potential options for controllers. Two potential options were given through discussion with the professor; however both were unusable in our design. A system called Gumstix was one option given, which was a highly modular system of very small components. While the size and power of the system met our needs, the need for the system to be a linked series of modules limited the robustness of the system to vibration. The other option we were given was single board computers. These single board computers would also easily provide the processing power and memory we required, as well as adding the potential for extended data storage on board. However, these boards suffered from a lack of protection similar to the Gumstix modules, as well as being too large to fit well in the robot.

2.5.2 Qwerk

The Qwerk platform deserves special mention as the most promising of all the options we explored. Not only did the controller meet and exceed our needs in terms of communications, processing power and storage, but it also was designed extremely ruggedly and could be run off the same 12 volts as the rest of the system. This system had been looked at by the previous project group, and dismissed because of communication issues they had with the system as well as the limited support of the system at the time. While we also had difficulty communicating with the system initially, we did eventually gain functionality. Despite these findings, we were still forced to reject this system.

The Qwerk had great potential to be the perfect solution to our controller problem, but the lack of certain key features killed its prospects. One major issue we had with the controller was that some of the features shown in the hardware were not supported yet in software. Specifically, the quadrature encoder ports on the hardware had no actual connections that could be software accessed. Another problem with the system was its method of storing code. While the actual GUIs supplied in the source code were fairly simplistic in design, actually re-writing the control code was proven to be relatively difficult. Another major code problem was the total lack of any locally running code on the system. While the Vex controller runs all code itself, the Qwerk was essentially just a board that would talk to a computer that had connected to it. This rendered it useless to us, as a loss of communication between the PC and the Qwerk could potentially result in the robot traveling off the roof.

2.5.3 The Serial Converter

After finally determining that for our project we would stick with the Vex controller, we needed a different way to communicate with it in order to eliminate the need for the remote control. A limitation of the Vex controller was that the only way to input data, other than the remote control, was via serial communications. In order to get a serial signal to the Vex, we needed to find a method of transmitting serial data. The solution that we implemented was a Wi-Fi to serial converter unit. This unit acts as a remote serial port that can be controlled by a computer via a wireless network. The other benefit of this system is that the serial port is controlled in the same way as a standard serial port, which meant that no additional code needed to be written in order to transmit or receive data on the PC. The Wi-Fi to serial converter we chose was the Grid Connect model Wi232. This converter module was one of the lowest priced models we could find, while maintaining a small form factor and the essential functionality that we needed. Most of the products available were 240 dollars or more, leaving the Grid Connect model the clear choice at 200.

The serial converter was used to replace the remote control through modifications made to the original control code of the robot. We developed a custom set of serial commands that can be recognized by the robot to replace the normal signals from the remote control. This change in structure also required reworking how some of the code functioned, as normally the code would be receiving constant updates via the remote. We circumvented this problem by creating a system which would essentially increment internal speeds according to the serial commands sent to the robot. While the robot loses some of its ability to rapidly speed up and maneuver, it does still function as well as no longer being as susceptible to a user rapidly changing the input on the remote control. Despite the loss of rapid control from the user, this new control scheme gives us the ability to both send data to and receive data from the robot, which was used in the newly designed interface.

2.5.4 Algorithm Changes

Space constraints on the Vex controller limited our ability to fully implement the traction control algorithm as described earlier in this section of this report. The reference code for performing the algorithm described there is included in Appendix C. Without the ability to implement the full weight distribution algorithm, the end code result is a substantially less computationally intensive heuristic function. This new function, while less accurate than the full algorithm, uses the basic principles developed in the complete code without actually computing the weights on individual wheels. Through additional testing this function can be made to perform better with additional tuning, but the lack of true weight distribution removes the level of fine control that may be desired.

2.6 New User Interface

The previous model of the robot used a standard Vex remote control as an input device, shown here in Fig. 15. The left joystick was used to drive the robot motors, while the right joystick was specific to the camera's pan and tilt functions. The major drawback of the Vex remote was the lack of two way communication between the robot and the user, which meant that the user was forced to use a laptop independently of the controller to receive the video feed. As an additional feature to add for the end user, we developed a new control interface concept which would unify the control of the robot and camera, while removing the Vex remote control as an input device. Our design concept, as shown in Fig. 16, was a unified dashboard program that could take in data from the robot and provide visual feedback to assist the driver through features that track the general status of the robot.

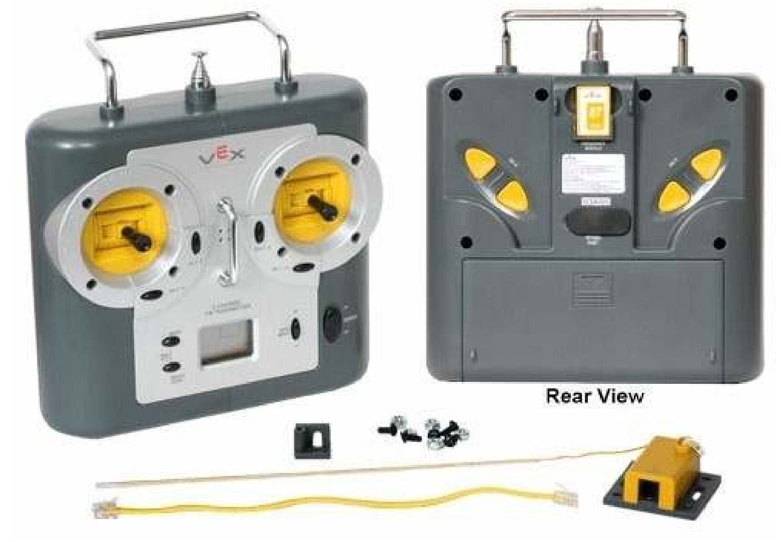


Fig. 15. Standard Vex Remote Controller

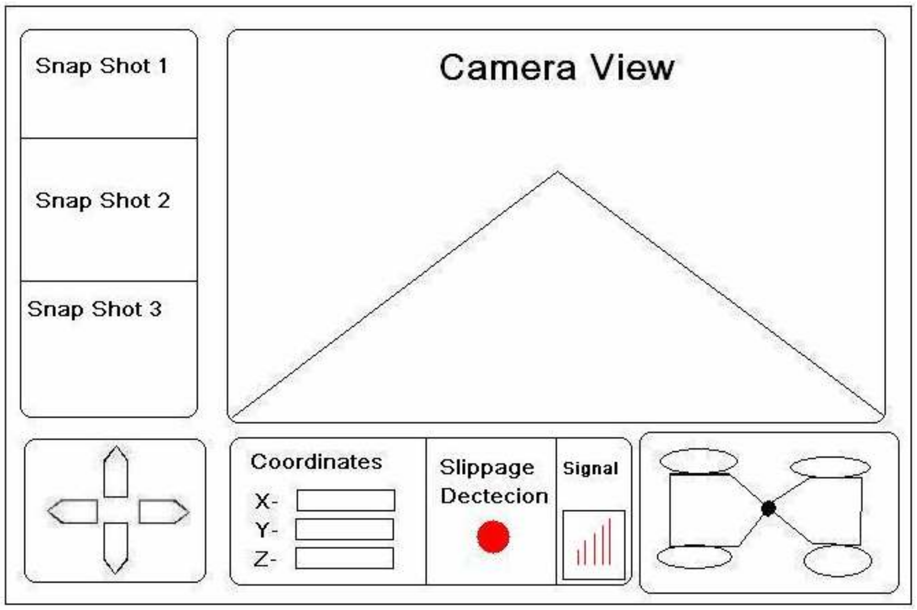


Fig. 16. User Interface Concept Sketch

2.6.1 User Interface Implementation

The new user interface is written in Visual Basic (VB), and there are three key functions defined in this program. A flowchart of the program behavior can be seen in Fig. 17. The first function is used to enable the VB program to communicate through serial port on the computer, which is the MSComm function built into VB. Under the Form_Load event, lines 3-27 of Appendix E, we declare the necessary settings for the VB serial controller, including the baud rate and the particular serial port we wish the program to communicate through. We also use this event to set how the VB program responds to receiving serial data on that communication port.

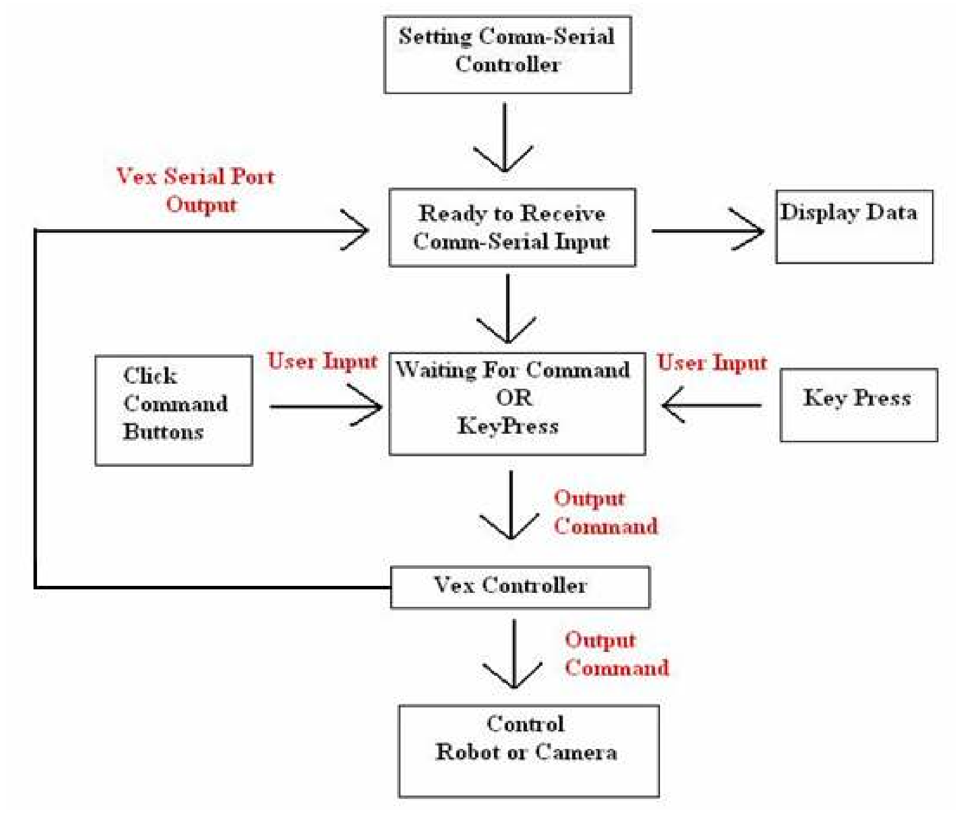


Fig. 17. VB Program Flow Chart

The second function is used to enable VB to receive input from the robot. The On_Comm event, lines 29-55 of Appendix E, is triggered once the serial input buffer contains the number of characters specified in the Form_Load event. This event is designed to take in the serial data from the robot and display it in a text box. The final set of functions is designed to accept user input on the user's computer and convert them into signals that are recognizable by the Vex controller. There are two groups of command buttons available for user input, one to navigate the robot and the other to control the camera. Users can either click the on screen buttons or press specific keys on the keyboard corresponding to each command. The VB program converts these button presses to a set of custom serial commands that the control code is designed to handle.

The motor control group has five command buttons, forward, back, left, right and stop. These commands are also mapped to the W, S, A, D, and Esc keys respectively. The same control concept is also applied to the camera, as there are five command buttons for camera control; left and right command buttons for pan, up and down command buttons for tilt, and a center for returning the camera to its default position. The up, down, left, right, and center commands, similarly to their motor control counterparts, are mapped to keyboard keys; I, K, J, L and C respectively. There are also two text boxes in the interface. The lower text box displays the command that was most recently executed by the user, while the upper text box displays the data feed coming from the robot.

2.7 Safety Concerns

Any safety issues on the robot can be divided into electrical and mechanical concerns. The primary electrical concern was the potential for a short circuit in the system, which can result in destruction of internal circuitry as well as fire. Another fire hazard arose from the current draw being placed on the battery packs, which can potentially overheat them. Both of these concerns have been addressed through the addition of the numerous fuses in the system. In addition, the general rewiring of the robot corrected the poor wiring connections of the previous year, further reducing the risk of sparks or fires. One electrical issue which was not addressed in this project was that of waterproofing the circuitry. The current design of the robot does not have a watertight seal between the halves of the robot, which could allow water to seep into the internal areas. Essentially all of the sensors on the robot are currently exposed to water contact, with the exception being the inclinometers.

Mechanically, there are several issues which we did not address, but should be considered in the future. The wheels and chain on the center joint currently allow full access while the robot is running, which could result in injury should the user place their hands near any of those areas. Although we can partially avoid this situation through the better placement of the control switch and the full stop command in the user interface, there are still some risks involved when handling the robot while it is active. Another potential safety issue arises in the event that the robot falls from the roof or ascender system. The control code runs on the Vex controller, which should effectively handle the robot driving off of the roof in the event of a signal loss, but a situation where the robot slips could still result in it falling off of the roof. As there is no way to insure that the robot will not fall in all circumstances, our best option is to simply insure that there are no people within some radius of the operational area of the robot while it is active in a raised environment.

3 Results

Overall, the final results of our project have been satisfactory. The robot has maintained all of the functionality it previously had, as well as gaining from the addition of the new camera, control modifications, and physical upgrades. All of the code has been modified several times now, and the actual platform itself is significantly more user friendly in terms of both maintenance and general usability.

3.1 Camera

The Hawking Technology network camera is able to produce high quality resolution video feedback to the user. The video camera software package that came with the camera allows user to produce a record of visual inspection through snapshots and full video recordings. While the camera offers these advantages to the previous installation, there are several drawbacks to its use. The network camera is larger in both size and weight compared to the X10 wireless camera, which produces some additional strain on the servo motor, shown in Fig. 18. Also the network camera requires a wireless access point, a Linksys router in our case, to send video information to the user, while the X10 wireless security camera comes with its own antenna communication system. However, the broad coverage of the Hawking camera proves to be superior in this type of system.

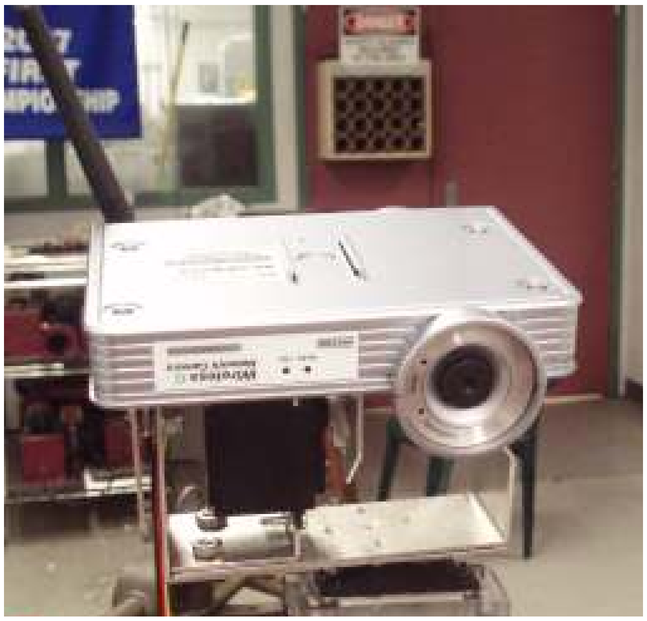


Fig. 18. Camera Mounting Bracket

3.2 *Coding*

The single largest contribution to the code is the overall readability improvement, due to both accurate comments and rewriting the code to make it more compatible with coding standards. The original code was written using almost no local variables as well as having only two functions that took parameters and returned values. Initially, the rewritten code contained almost no global variables other than those required for interrupts, but that modification had to be changed due to the restrictions in the Vex controller. Those restrictions also limited the number of parameters which could be sent between functions, rendering the majority of the changes to pointer style code unusable. The final result of the space constraints placed on us by the Vex controller is the code in its current form, which could possibly be simplified or made more efficient given adequate storage space for the end program.

Aside from the general code rewrite, the new portions of the code are behaving as expected. The new control algorithm appears to have improved performance over the previous control scheme. This assessment is purely subjective, as there is no hard data to show a performance increase. The current code specifications allow the robot to maintain its position and heading on the roof more easily than at the outset of the project, although additional tuning of the algorithm can still be performed to potentially further increase performance. There were several interesting behaviors observed in testing the new algorithm. The largest piece of note was the increased performance of the system when we lowered the total amount of modification being performed per wheel. The most dramatic changes were viewed when changing the modifier on line 34 of Appendix B. Using a modifier of 4 caused the system to lock the front wheels when climbing, while leaving the back wheels to spin. Reducing that modifier to 2 proved to work substantially better, and allowed the wheels to move as expected. We also found that increasing the threshold before the algorithm begins also improved performance, as our initial cutoff values of 10 and negative 10 in lines 569, 574, 580, and 585 also caused the wheels to lock when on an incline. The robot can safely maintain its position on a 12/12 roof, although a lack of traction is still apparent from test driving the system. Although further improvements to the control algorithm are still recommended future work, increasing the coefficient of friction between the wheels and the roof surface should be a higher priority as the failure method of the system now is indicative of the wheels being unable to grip versus the previous problem with unequal slippage.

The serial control method has been proven to be functional in conjunction with the new interface. Currently, the output style is sending string data which contains the wheel speeds of the robot. Output from the Vex controller can be quickly modified to send any additional data as required through a single line of code, as well as the potential to send raw byte data if needed. Another unforeseen benefit of our new control system is that the robot is effectively given time to accelerate. The original control code allowed the user to enter a full throttle command immediately, which would cause the robot to slip very quickly. The new control scheme forces the user to accelerate the robot more slowly, which lessens the initial slip when the system begins moving.

3.3 Usability

The modifications made to the chassis and internal wiring has resulted in a robot with substantially improved overall usability to the end user. By adding a number of fuses throughout the system, the overall safety of both the system and the users has been increased. Changing the connectors between the batteries and the Victor Speed Controllers has also improved the ease of maintainability by reducing the number of poor connections from the previous year, as well as replacing the difficult to remove ring terminals with spades which do not require a complete removal of the holding screw to detach. Although difficult to see in Fig. 19, the front of the robot shows the new wiring scheme, as well as demonstrating how the speed controllers are now better balanced in terms of their logical position in relation to the motors they control. In addition, the main power connections are now completed through quick disconnect terminals, further easing the process of changing wires or connections within the power system. The modifications made to the power system also created a single charging port which is easily accessed from the outside of the robot. Tying the charging port to all of the batteries has eliminated the need to remove the batteries from the robot, unless a pack is determined to be dead. This also results in the end user only requiring one battery charger, which further simplifies the number of accoutrements required to operate the robot.

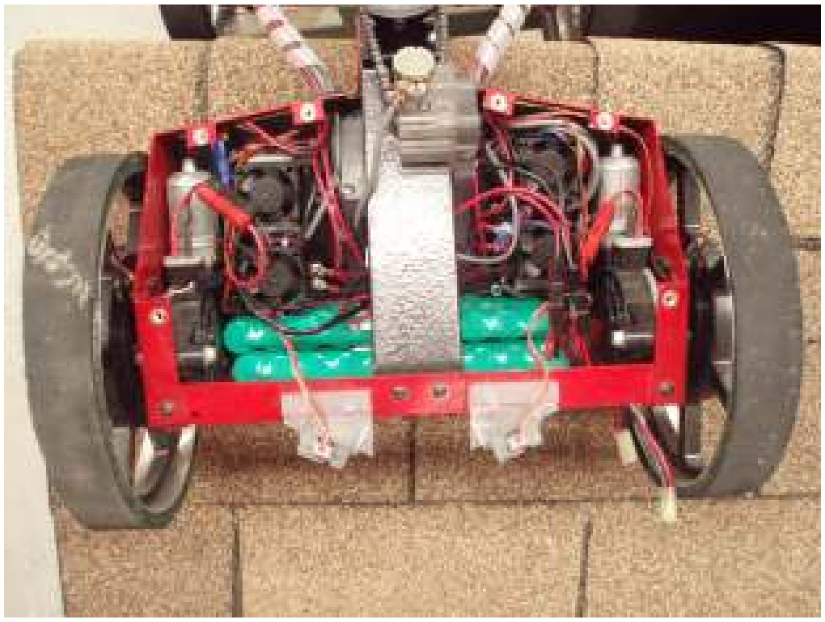


Fig. 19. Robot Front Region

The final piece of beneficial wiring is in the movement of the switch to a more easily accessed area on the robot chassis, shown in Fig. 20. Not only does this make the robot easier to power up through a single switch throw, but also insures that the robot cannot be charged while in operation. That feature saves the charger from potential damage by being overdrawn from the motors directly. The single switch feature was excellent in theory, but a bizarre wiring condition seems to exist between the Vex controller and the rest of the robot. When the switch on the controller was left in the "on" position, as would be required for the normal operation of a single switch power-up, the circuit on the robot was completed despite the lack of an external completion. This suggests that the Vex controller somehow allows a completed circuit through itself, which eliminated the ability to use a single switch system.

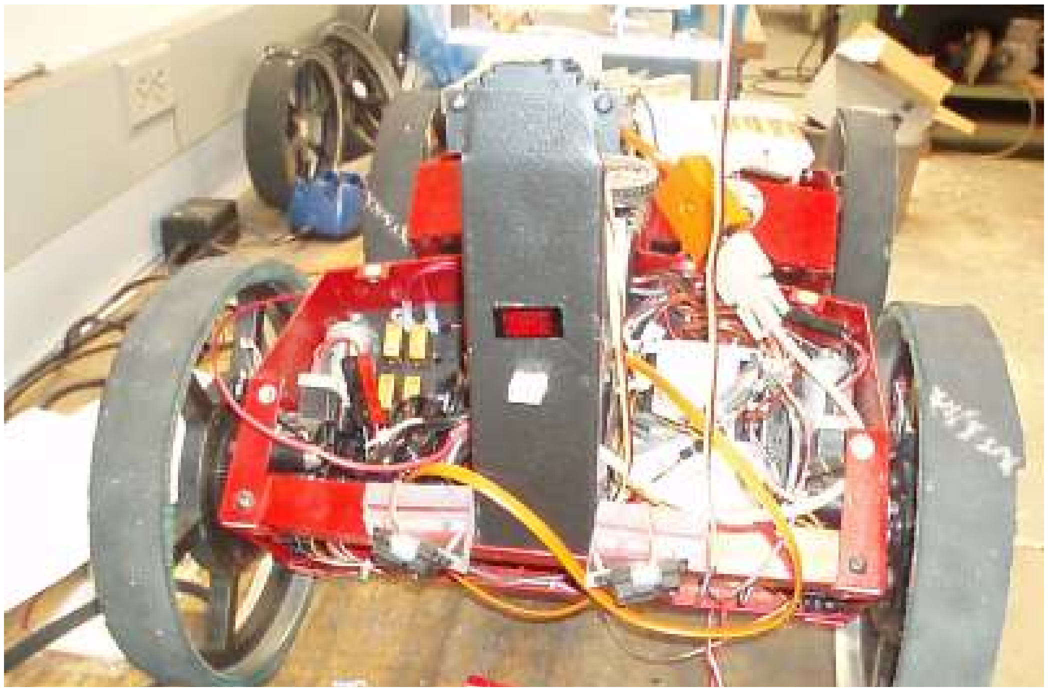


Fig. 20. Switch and Charging Point Placement

3.4 User Interface

The user interface implementation is a success; it has most of the essential functionality from the concept design, such as navigation for the robot as well as control for the camera. The video viewer from the Hawking Technology camera displays high quality video feedback and is able to take pictures, which can be used as a record of the roof inspection, as shown in Fig. 21. At present, these images do not appear on screen in the manner depicted in the concept sketch, but the images are saved to a location on the host computer. The user interface is also able to display the orientation and the individual motor speeds of the robot. Certain features do remain to be fully developed however. The 3D model of the robot while in operation is incomplete, and the data being transmitted to the interface is currently improper for truly performing useful conversions on the base PC controlling the robot. In addition, the coordinate system and slippage detection features are not currently available, as they are not supported by the robot's control code at this time.

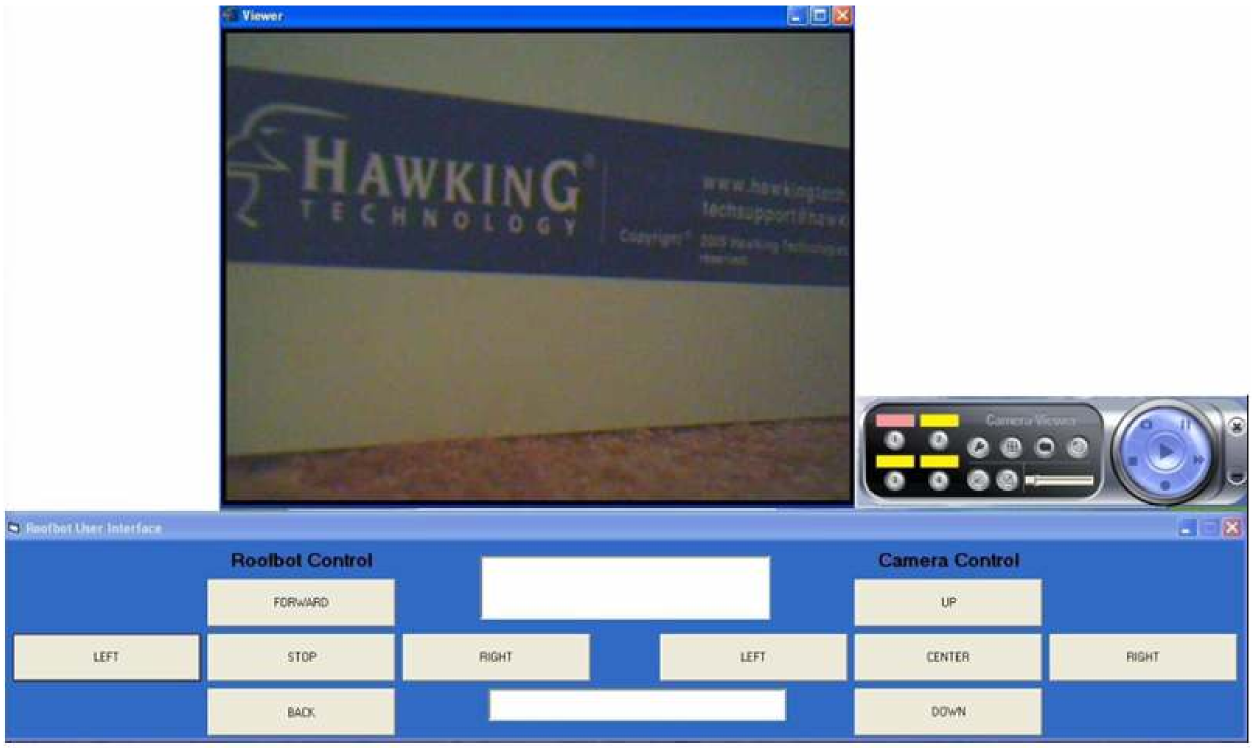


Fig. 21. Final User Interface

4 Future Work

Despite our progress in this project, there is still substantial work to be done on the system. While the number of electrical concerns has decreased, there is still room for improvement both in terms of the internal wiring of the system and the control system. Mechanically there are also several potential areas for improvement, with further work on traction materials and possible restructuring of the chassis being in the forefront.

4.1 New Controller

One absolutely essential upgrade which must be performed on this system in the near future is to find a new controller. The Vex controller is being pushed to its limits in terms of storage and power on the current system, and any further sensor upgrades will push it beyond. Should the Qwerk platform become viable, it should certainly be investigated again. A more recent suggestion was a platform called nanoITX, which are full computer systems which fit on a very small form factor board. As long as future expansion is kept as a primary goal in the platform, alternatives must be able to deal with additional heretofore unexpected input and output ports. The other major benefit to a future controller upgrade is the potential to reduce the cost and weight of systems on the robot. The controller is the primary weakness in the system as of the completion of this project.

4.2 Further Traction Control Tuning

While the current traction control system is a marked improvement from the previous year, there is still progress to be made with the traction control code. Additional tuning of the current version may result in increased performance. Another possible addition to the current algorithm would be to modify the allowed slip on a sliding scale, as low velocity may require less overall adjustment than when the system is moving at full speed. Exchanging the current heuristic with the original algorithm would also likely assist in the overall performance of the traction controls. In addition to the current choices made in control code, a more powerful system could also potentially take advantage of more sophisticated controls, such as the Model Following Control scheme described in the Background. Additional research could also provide more options for control schemes beyond those mentioned in this report, as robotic development continues to be a major area of study. Further sensor and mechanical upgrades may also result in the need to completely change the current control scheme through either changing the current drive control sensors or through novel concepts that allow new sensors to be more fully integrated with the overall robot control scheme.

4.2 Additional Sensors

The current sensors on the robot are sufficient for general navigation, but are limited in utility beyond that. Research into additional sensor packages could result in additional control systems being developed, as well as increasing the potential utility of the robotic platform. New sensors could include additional cameras for closer visual inspection, as well as other sensors targeted to specific types of damage. An additional feature which was discussed was creating an automatic return function, which would allow the robot to return to its starting position in the ascender through a single button press. Several safety issues might also be addressed through additional sensors, such as preventing users from being caught in the wheel spokes or chain. Additional sensors would be required for performing this task. Space restrictions will become a factor in future sensor upgrades, and a new controller will be necessary for any additional inputs to be processed.

4.3 Additional Camera Work

The new camera on the robot meets the rough expectations of the camera system, but could still use further improvement. Upgrading the controller could allow a substantially smaller camera to be used and integrated with the control system itself. A smaller camera could also potentially be found that would reduce the overall weight of the system. Any changes to a smaller camera would not only result in a reduction in camera weight, but the added possibility of reducing the servos currently used to maneuver the camera. Additional future improvements for the camera system would be to find a camera that has some level of zoom and a higher resolution in order to make the inspection process easier. It has also been suggested that the camera possibly be mounted in an even higher location on the robot, but feasibility of this request may have to be investigated further.

4.4 Continued User Interface Improvements

While the new user interface is a dramatic improvement from the previous year, the total functionality of the system is still fairly limited. The current interface, as shown in Fig. 21, displays a portion of the desired functionality, but the issues mentioned in the Results section could most likely be addressed. This interface is currently written in Visual Basic, which was adequate to our early development, but a more efficient language might be better suited to the task. Another issue to address is the use of 2 separate programs to run the interface. Developing an interface which could combine the two current programs into one overall executable would be ideal. A combined program would also allow future developers to more closely match the interface with the current concept sketch. It is our recommendation that at least one Computer Science student be recruited to continue software development on the interface.

4.5 Additional Wiring

The majority of the system has been rewired in this project for easier maintenance and to be more standardized internally. However, there are still some unsolved issues with the system electrically. The current method of powering the controller required custom soldering to be performed on one of the battery packs, which means that when that pack fails the work will have to be redone. This also causes the customized battery pack to discharge unevenly, which will shorten its life. While changing the controller may eliminate the need for a different voltage for the controller, developing a DC/DC converter would also potentially be a good solution to the problem. It is also highly likely that any new controller would require new wires to be run between the sensors and the controller, unless the same style of connecting point is used.

4.6 Continued Wheel Improvements

The current wheels on the robot are barely adequate for operation on high pitch roofs, with the wheels being unable to allow full maneuverability on steeper inclines.

Further research in traction materials, tread patterns, and possibly additional or modified wheel encoders would be recommended. Our testing suggested that even relatively simple tread patterns could potentially increase the functionality of the wheels with little work. Other tread materials and different wheel substrates may also improve the general functionality of the wheels. The final suggestion would be to remove the current open optical encoders and replace them with a closed quadrature encoder. The open nature of the current encoders causes them to be inaccurate at times, as well as making some parts of the operating code more complex than necessary.

4.7 Chassis Upgrades

It was mentioned in the previous report that the center joint on the chassis could benefit from additional work. We also determined this to be the case, as the joint had loosened considerably since the previous year when we began our project. Aside from the joint, the overall structure of the robot could possibly be reworked to create more room for electronics. Space is already extremely limited in the current incarnation of the robot, and any attempts to add further functionality will almost assuredly find that space for additional parts will be difficult to find.

4.8 Ascender System

One of the major mechanical components of the system which was not covered this year was the need for a new ascender system for the robot. The original goal of the system was to be able to place the robot on a roof up to 3 stories off of the ground. The current system is only capable of approximately 1 story, and is exceptionally heavy for the total load it needs to carry. Development of a system would require not only thought about moving the robot itself, but also a wireless access point with some power source, unless a new system is developed for communication between the robot and the ground.

5 Conclusions

The primary goal of this MQP was to perform improvements to the project of the previous year. Despite difficulties in discovering a new controller module, our modifications have been mostly successful. Overall wiring, camera performance, and drive performance have been improved. The rewiring work performed on the robot also improved safety for the end user, by reducing the risk of electrical short circuits causing fires or electrical damage within the system. Future work on this project can continue, and should be substantially easier with the new wiring used in the current design. Further development of both the control code and the physical robot itself will continue to be important goals in future projects.

Appendix A – References

[1] Y. Hori, Y. Toyoda, and Y. Tsuruoka, (1998, Sep/Oct). Traction Control of Electric Vehicle: Basic Experimental Results Using the Test EV "UOT Electric March". *IEEE Transactions on Industry Applications*. [Online]. 34(5), 1131-1138. Available: http://ieeexplore.ieee.org/iel3/5036/14041/00645576.pdf?tp=&isnumber=&arnumber=645576

[2] D. Caltabiano and G. Muscato. (2002, Oct). A Comparison Between Different Traction Control Methods for a Field Robot. *Intl. Conference on Intelligent Robots and Systems*. [Online]. 1, 702-707. Available: http://ieeexplore.ieee.org/iel5/8071/22324/01041473.pdf?tp=&isnumber=&arnumber=1041473

[3] D. Caltabiano, D. Ciancitto and G. Muscato. (2004, April). Experimental Results on a Traction Control Algorithm for Mobile Robots in Volcano Environment. *Intl. Conf. on Robotics and Automation*. [Online]. 5, 4375-4380. Available: http://ieeexplore.ieee.org/iel5/9126/28923/01302406.pdf?tp=&arnumber=1302406&isnumber=28923

[4] Video Compression Tutorial. [Online]. Available: http://www.wave-report.com/tutorials/VC.htm

[5] J. Wijering. (2005, Nov). FLV Video Compression. [Online]. Available: http://www.jeroenwijering.com/?item=FLV_Video_Compression

[6] J. Dunn. (2002, Nov). Digital Video Compression Explained. [Online]. Available: http://www.microsoft.com/windowsxp/using/moviemaker/expert/digitalvideo.mspx

[7] P. Tudor. (1995, Dec). MPEG-2 Video Compression. [Online]. Available: http://www.bbc.co.uk/rd/pubs/papers/paper_14/paper_14.shtml#TOP

[8] C. Manning. Digital Video Compression. [Online]. Available: http://www.newmediarepublic.com/dvideo/compression.html

[9] MPEG Video Compression Technique. [Online]. Available: http://vsr.informatik.tu-chemnitz.de/~jan/MPEG/HTML/mpeg_tech.html

[10] D. Marshall. (2001, Oct). Video and Audio Compression. [Online]. Available: http://www.cs.cf.ac.uk/Dave/Multimedia/node200.html

[11] W. Buchanan. (1999, June 23). PC Interfacing, Communications and Windows Programming (1st Ed.) [Online]. Available: http://www.dcs.napier.ac.uk/~bill/pdf/Io_ch26.PDF

[12] S. Holzner, *Visual Basic 7 Black Book: The Only Book You'll Need on Visual Basic*, Scottsdale, AZ: The Coriolis Group, 1998.

[13] R. Stephens, *Visual Basic Graphics Programming*, John Wiley & Sons, 1999

[14] Free Visual Basic Source Code. [online]. Available: http://www.freevbcode.com/

[15] Visual Basic Keycode [online]. Available: http://www.gh-gold.co.uk/keycodes.php

[16] Visual Basic Town. [online]. Available: http://cuinl.tripod.com/

[17] Extreme Visual Basic Talk. [online]. Available: http://www.xtremevbtalk.com/forumdisplay.php?s=5ac1d973949612b23c25b649925094a2&f=17

[18] MSComm Control [online]. Available: http://www.yes-tele.com/mscomm.html

Appendix B – Robot Control Code

```
/*
Zachary Strowe

Roof Robot Control Code - Revision 2.5
This is a followup to the original control code written by
the team of Nicholas McMahon, Samuel Feller and Nathan Malatesta.

For the original source code used in their project, reference Appendix B of
project MQP-KZS-0602.

NOTE: The original code used the VEX remote control as the primary input, while
this revision uses serial commands.  Some pieces of code are legacy and may not
be necessary - additional pruning is suggested.
*/

// runRobot.c : implementation file
#include "API.h" //this is the API to access all the Vex’s sensors/motors/etc.
#include "math.h"

#define servoNeutral 127
#define potNeutral 447
#define halfLength 9
#define halfWidth 9
#define rateOfTurn .05
#define P 50
#define P2 30
// INCREMENT is the amount of change per instruction through the serial commands
#define INCREMENT 5
// These thresholds define the speed correction cutoffs for the inclinometers
#define INCHTHRESH 662
#define INCLTHRESH 362
#define INCMOD 2

void runRobot(void);
void getJointFeedback(void);
void getWheelFeedback(void);
//void getInputs(unsigned char*);
void getSerialInputs(void);
void updateTargets(void);
void setTargetsForDriving(void);
void setTargetsForTurning(float);
void setTargetsForPanTilt(void);
void calcSpeed(int);
void adjustForError(int);
//void driveMotors(void);
//void drivePanTilt(void);
void lockOut(void);
void transmitData(void);
void inclination(void);

// indicator for connection stability, currently disabled until software support is
// available on the transmitting side - would then require additional timing logic
//unsigned char connected = 0;
// inputs are 0 = front/back, 1 = left/right, 2 = tilt, 3 = pan, 4 = override
// This convention is a legacy from the remote control programming
unsigned char inputs[5] = {127, 127, 127, 127, 0};
// motor variables are 0 = FL, 1 = FR, 2 = BL, 3 = BR
int mTargets[4], output[4];
// cam targets 0 = pan, 1 = tilt
unsigned char camTargets[2] = {127, 127};
// pan/tilt delays 0 = count, 1 = delay
int panTiltDelays[2];
// joint 0 = reading, 1 = angle
float joint[2];
// the count and click time variables are for wheel feedback
int newCount[4], newClickTime[4], oldCount1[4], oldCount2[4], stuckWaiting[4],
delay[4];
float expectedClicksPerSec[4], clicksPerSec[4];
unsigned long oldClickTime1[4], oldClickTime2[4];
// helps control wheels based on angle of the center joint
float rightToLeftRatio;
float targetAngPWM, targetAngle, D;
char I;
int incline[4];

void runRobot(void)
{
   int i;
   unsigned long sender = 0;

   getJointFeedback();
   targetAngle = joint[1];
   D = joint[0];
   panTiltDelays[0] = 0;

   // start encoders and timers
   for( i = 0 ; i <= 3 ; i++)
   {
      PresetEncoder(i+1, 0);
```

```
            StartEncoder(i+1);
            PresetTimer(i+1, 0);
            StartTimer(i+1);
        }

        while(1)
        {
            getSerialInputs();
            updateTargets();
            getJointFeedback();
            getWheelFeedback();
            for (i = 0; i < 4; i++)
                calcSpeed(i);

            for( i = 2; i <=5 ; i++)
            {
              if (GetAnalogInput(i) < 85 && inputs[4] != 255)
              {
              lockOut();
              Wait(50);
              }
            }

            // transmits data back to interface at 1 second intervals *approximately*
            if (GetTimer(1) > sender)
            {
              sender = sender + 1000;
              transmitData();
            }

            inclination();

            //driveMotors();
            SetPWM(1, (unsigned char)(output[0] + servoNeutral));
            SetPWM(2, (unsigned char)(output[1] + servoNeutral));
            SetPWM(3, (unsigned char)(output[2] + servoNeutral));
            SetPWM(4, (unsigned char)(output[3] + servoNeutral));
            SetPWM(5, targetAngPWM);

            //drivePanTilt();
            SetPWM(6, camTargets[1]);
            SetPWM(7, camTargets[0]);
        }

    }
    //*****************************************************
```

```
void getJointFeedback(void)
{
    joint[0] = GetAnalogInput(1);
    joint[1] = (joint[0] - potNeutral) / 508;
}
//*****************************************************
void getWheelFeedback(void)
{
   int i;
   int countDiff1, countDiff2;
   //long timeDiff1, timeDiff2;

   for (i = 0 ; i <= 3 ; i++)
   {
      newCount[i] = GetEncoder(i+1); //this section of code prevents the encoder
      newClickTime[i] = GetTimer(i+1); //counters from overflowing

      if (newCount[i] > 32000)
      {
         countDiff1 = newCount[i] - oldCount1[i];
         countDiff2 = oldCount2[i] - oldCount1[i];
         oldCount2[i] = 0;
         oldCount1[i] = countDiff2;
         newCount[i] = countDiff1 + countDiff2;
         PresetEncoder(i+1,newCount[i]);
      }
      /* This correction would only happen after 11 days of continuous battery operation
      if (newClickTime[i] > 1000000000) //this prevents the timers from overflowing
      {
         timeDiff1 = newClickTime[i] - oldClickTime1[i];
         timeDiff2 = oldClickTime1[i] - oldClickTime2[i];
         oldClickTime2[i] = 0;
         oldClickTime1[i] = timeDiff2;
         newClickTime[i] = timeDiff1 + timeDiff2;
         stuckWaiting[i] = newClickTime[i];
         PresetTimer(i+1,newClickTime[i]);
      }*/
   }
}
//*****************************************************
void updateTargets(void)
{
   float joystickInRadians = (float)((servoNeutral - inputs[1]))/ 175;
   float halfTangent = tan(joint[1]/2);

   rightToLeftRatio = (halfWidth + halfWidth * halfTangent)/(halfWidth - halfWidth * halfTangent);
   /* the above is a confusing magic conversion, so that a full throttle
   joystick position should correspond to about .785 radians, or a full,
   45 degree turn*/
   // NOTE: It may be possible to use a linear function in place of the
   // above "magic conversion".
   setTargetsForTurning(joystickInRadians);

   if ((inputs[0] - servoNeutral) < -15 || (inputs[0] - servoNeutral)> 15)
      setTargetsForDriving();
   setTargetsForPanTilt();
   if (panTiltDelays[0] < panTiltDelays[1]) // I did this to slow the pan tilt down
   {
      if ((inputs[3] - servoNeutral) < -15 || (inputs[3] - servoNeutral)> 15) //set deadbands
        if ((inputs[2] - servoNeutral) < -15 || (inputs[2] - servoNeutral)> 15)
           panTiltDelays[0]++;
   }
   else
      panTiltDelays[0] = 0;
}

//****************************************************
// set DRIVING TARGETS
void setTargetsForDriving(void)
{
   //float rightToLeftRatio;
   float maxPower = 50.;
   float powerScaleToJoystick;
   float slowSide, fastSide;

   if (rightToLeftRatio > 1 || rightToLeftRatio < -1)
      powerScaleToJoystick = maxPower / rightToLeftRatio; //dynamically scales the throttle range
   else
      powerScaleToJoystick = maxPower * rightToLeftRatio; //dynamically scales the throttle range

   slowSide = ((float)(inputs[0] - servoNeutral) / 128) * powerScaleToJoystick;
   // the code below should keep it out of the deadband
   // it assumes that the maxPower setting will keep it from maxing out
   if (slowSide > -15 && slowSide < 0)
      slowSide = -15;
   if (slowSide >= 0 && slowSide < 15)
      slowSide = 15;
   if (rightToLeftRatio > 1 || rightToLeftRatio < -1)
   {
      fastSide = slowSide * rightToLeftRatio;
      mTargets[1] = (int)(slowSide);
      mTargets[3] = (int)(slowSide);
      mTargets[0] = (int)(fastSide);
      mTargets[2] = (int)(fastSide);
   }
   else
   {
      fastSide = slowSide / rightToLeftRatio;
      mTargets[1] = (int)(fastSide);
      mTargets[3] = (int)(fastSide);
      mTargets[0] = (int)(slowSide);
      mTargets[2] = (int)(slowSide);
   }
}
//*****************************************************
// set TURNING targets
void setTargetsForTurning(float joystickInRadians)
{
   float slowSide, fastSide;

   targetAngle = joystickInRadians ;
   if((joint[1] - targetAngle) > -.037 && (joint[1] - targetAngle) < .037)
   {
      I = 0;
      slowSide = 0;
   }
   else
   {
      if ((D - (int)joint[0]) > -5 && (D - (int)joint[0]) < 5 && (-20 < I < 20))
      {
         if (joint[1] < targetAngle)
         {
            slowSide = -12;
            if (I > -12)
               I = -12;
            else
                I--;
         }
         if (joint[1] > targetAngle)
         {
            slowSide = 12;
            if (I < 12)
               I = 12;
            else
```

```
                I++;
            }
        }
    }

    D = joint[0];
    targetAngPWM = (unsigned char)(servoNeutral + I);

    if (rightToLeftRatio > 1 || rightToLeftRatio < -1)
    {
        fastSide = slowSide * rightToLeftRatio;
        mTargets[1] = (int)(slowSide);
        mTargets[3] = (int)(-slowSide);
        mTargets[0] = (int)(-fastSide);
        mTargets[2] = (int)(fastSide);
    }
    else
    {
        fastSide = slowSide / rightToLeftRatio;
        mTargets[1] = (int)(slowSide);
        mTargets[3] = (int)(-slowSide);
        mTargets[0] = (int)(-fastSide);
        mTargets[2] = (int)(fastSide);
    }
}
//*******************************************************
// set the PAN/TILT
void setTargetsForPanTilt(void)
{
    int turnOffset;
    int scaledPan;
    float scale;


    if (panTiltDelays[0] < panTiltDelays[1])
    {
        if ((inputs[2] - servoNeutral < -15) && camTargets[1] > 1)
            camTargets[1]--;
        if ((inputs[2] - servoNeutral > 15) && camTargets[1] < 255)
            camTargets[1]++;

        panTiltDelays[0]++;
    }
    else
        panTiltDelays[0] = 0;

    // setup for camera tracking the motion of the robot
    turnOffset = (servoNeutral - inputs[1]) / 1.5;

    // scaling the turned camera with further controls
    if ((inputs[3] - servoNeutral) > 0)
      scale = (float)(127 - turnOffset) / 127.;
    if ((inputs[3] - servoNeutral) <=0)
      scale = (float)(turnOffset - (-127))/127;

    scaledPan = (inputs[3] - servoNeutral) * scale;
    camTargets[0] = (unsigned char)(scaledPan + turnOffset + servoNeutral);
}
//*******************************************************
void getSerialInputs(void)
{
     char signal, i;

     // read serial port, ignore 0 bytes
     if ((signal = ReadSerialPortOne()) != '0')
     {
       if (signal == '$')
       {
         //connected = 1;

         switch(ReadSerialPortOne())
          {
            // motor commands
            case('f'):
               inputs[0] = inputs[0] + INCREMENT;
               break;
            case('b'):
               inputs[0] = inputs[0] - INCREMENT;
               break;
            case('l'):
               inputs[1] = inputs[1] + INCREMENT;
               break;
            case('r'):
               inputs[1] = inputs[1] - INCREMENT;
               break;
            // camera commands
            case('u'):
               inputs[2] = inputs[2] + INCREMENT;
               break;
            case('d'):
               inputs[2] = inputs[2] - INCREMENT;
               break;
            case('z'):
               inputs[3] = inputs[3] - INCREMENT;
               break;
            case('x'):
               inputs[3] = inputs[3] + INCREMENT;
               break;
            // override command
            case('o'):
               inputs[4] = 255;
               break;
            // stop command
            case('s'):
               inputs[0] = 127;
               inputs[1] = 127;
               inputs[4] = 0;
               lockOut();
               break;
            // center camera
            case('c'):
               inputs[2] = 127;
               inputs[3] = 127;
               break;
         }

      // This code is a basic proto-form of the connected signal
      //if (signal == 'h')
      //   connected = 1;*/

      // insure that the inputs never roll over
      for (i = 0; i < 4; i++)
      {
         if (inputs[i] > 245)
           inputs[i] = 250;
         else if (inputs[i] < 10)
           inputs[i] = 5;
      }
    }
   }
}
//*****************************************************
/* This function is for using the VEX remote, getSerialInputs is the current
// method of getting input data to the controller
void getInputs(unsigned char* in)
{
   int i;

```

```
     for (i = 0; i < 5; i++)
     {
        in[i] = GetRxInput(0, i+1);
     }
}*/
//*****************************************************
// calculate current Speed
void calcSpeed(int i)
{
   int timeInterval, numClicks;
   float expectedTimeBetweenClicks;

   expectedClicksPerSec[i] = (float)(mTargets[i])*.85;

   if (mTargets[i] < 0)
     expectedClicksPerSec[i] = -1 * expectedClicksPerSec[i];

   expectedTimeBetweenClicks = (1200 / expectedClicksPerSec[i]);

   if (expectedTimeBetweenClicks > 120)
     expectedTimeBetweenClicks = 120;
   // the above should give you the time between clicks...
   // there is a little extra leeway to account for rounding error
   // and stuff like that
   if(newCount[i] > oldCount1[i])
   {
      timeInterval = newClickTime[i] - oldClickTime2[i];
      numClicks = newCount[i] - oldCount2[i];
      oldClickTime2[i] = oldClickTime1[i];
      oldClickTime1[i] = newClickTime[i];
      stuckWaiting[i] = newClickTime[i]; //a click has occured,
      oldCount2[i] = oldCount1[i]; //so reset everything
      oldCount1[i] = newCount[i];
      clicksPerSec[i] = 1000 / (float)(timeInterval / numClicks);
      adjustForError(i);
   }
   if ((newClickTime[i] - oldClickTime1[i]) > expectedTimeBetweenClicks)
   {
      clicksPerSec[i] = 0;
   }
   if ((newClickTime[i] - stuckWaiting[i]) > expectedTimeBetweenClicks)
   {
      stuckWaiting[i] = newClickTime[i];
      delay[i] = 2;
      adjustForError(i);
   }
```

```
}
//*******************************************************
// adjust output for error
void adjustForError(int i)
{
   int Pwheel;
   int speedBehind, correction;
   float percentError, error;
   char comp;

   delay[i]++;

   // run this function every on every other call
   if(delay[i] < 2)
      return;
   else
      delay[i] = 0;

   //figures out which wheel is on the same side (left and right sides)
   if( i == 0)
      comp = 2;
   if( i == 1)
      comp = 3;
   if( i == 2)
      comp = 0;
   if( i == 3)
      comp = 1;

   error = clicksPerSec[i] - expectedClicksPerSec[i];
   percentError = error / expectedClicksPerSec[i];
   speedBehind = (int)(clicksPerSec[i] - clicksPerSec[comp]);

   if (percentError > .1)
     Pwheel = (-percentError * 2) - 1;
   else if (percentError < -.1)
       Pwheel = (-percentError * 2) + 1;
   else
      Pwheel = 0;

   if (percentError < -.1 && speedBehind < -12 )
     Pwheel = Pwheel + 10;

   if (mTargets[i] > 0)
     correction = Pwheel ;
   else if (mTargets[i] < 0)
       correction = -Pwheel;
```

```

    output[i] = output[i] + correction;
    //***********************
    //dead zone
    if (mTargets[i] == 0 )
      output[i] = 0;
    else if (mTargets[i] > 0 && output[i] < 12)
      output[i] = 12;
    else if (mTargets[i] < 0 && output[i] > -12)
      output[i] = -12;
    //***********************
    //dont max out
    if (output[i] > 127)
      output[i] = 127;
    if (output[i] < -127)
      output[i] = -127;
}
//*****************************************************
// The following two functions are only called once, and have been moved to the
// main program to conserve space on the controller
/*void driveMotors(void)
{
    SetPWM(1, (unsigned char)(output[0] + servoNeutral));
    SetPWM(2, (unsigned char)(output[1] + servoNeutral));
    SetPWM(3, (unsigned char)(output[2] + servoNeutral));
    SetPWM(4, (unsigned char)(output[3] + servoNeutral));
    SetPWM(5, targetAngPWM);
}*/
//*****************************************************
/*void drivePanTilt(void)
{
    SetPWM(6, camTargets[1]);
    SetPWM(7, camTargets[0]);
}*/
//*****************************************************
void lockOut(void)
{
    output[0] = 0;
    output[1] = 0;
    output[2] = 0;
    output[3] = 0;
    targetAngPWM = 127;
}
//*****************************************************
void transmitData(void)
{
    // The data transmission is led with a 'z' character to signal the interface
    // that data is incoming
   PrintToScreen("z%d, %d, %d, %d", output[0], output[1], output[2], output[3]);
}
//****************************************************
void inclination(void)
{
    int i;

    // The following is a substantially less elegant version of the traction
    // control algorithm discussed in the paper.
    incline[0] = GetAnalogInput(6);
    incline[1] = GetAnalogInput(7);
    incline[2] = GetAnalogInput(8);
    incline[3] = GetAnalogInput(9);

    // handle inclination from front to back
    if (((incline[2] + (1024 - incline[0])) / 2) > INCHTHRESH)
      if (output[0] > 20 && output[1] > 20)
      {
        output[0] -= INCMOD;
        output[1] -= INCMOD;
      }
      else if (output[0] < -20 && output[1] < -20)
      {
        output[0] += INCMOD;
        output[1] += INCMOD;
      }
    else if (((incline[2] + (1024 - incline[0])) / 2) < INCLTHRESH)
      if (output[2] > 20 && output[3] > 20)
      {
        output[2] -= INCMOD;
        output[3] -= INCMOD;
      }
      else if (output[2] < -20 && output[3] < -20)
      {
        output[2] += INCMOD;
        output[3] += INCMOD;
      }

    // handles roll of the front end
    if (incline[1] > INCHTHRESH && output[1] > 10)
      output[1] -= INCMOD;
    else if (incline[1] > INCHTHRESH && output[1] < 10)
      output[1] += INCMOD;
    else if (incline[1] < INCLTHRESH && output[0] > 10)
```

```
          output[0] -= INCMOD;
       else if (incline[1] < INCLTHRESH && output[0] < 10)
          output[0] += INCMOD;

       // handles roll of back end
       if (incline[3] > INCHTHRESH && output[3] > 10)
          output[3] -= INCMOD;
       else if (incline[3] > INCHTHRESH && output[3] < 10)
          output[3] += INCMOD;
       else if (incline[3] < INCLTHRESH && output[2] > 10)
          output[2] -= INCMOD;
       else if (incline[3] < INCLTHRESH && output[2] < 10)
          output[2] += INCMOD;

       // keep the output from rolling when the drive conversion is run
       for (i = 0; i < 4; i++)
           if (output[i] > 126)
              output[i] = 126;
           else if (output[i] < -127)
                output[i] = -127;

   }
```

Appendix C – Algorithm Code

```
void checkInclination(void)
{
    float tilt, roll, frontBack, leftRight; // temp
    float frontWeight, backWeight;
    int i;

    frontIncline[0] = GetAnalogInput(6);
    frontIncline[1] = GetAnalogInput(7);
    backIncline[0] = GetAnalogInput(8);
    backIncline[1] = GetAnalogInput(9);

    tilt = (arcsin((((float)(frontIncline[0] + (1024 - backIncline[0])) / 2) - offset) /
sensitivity));
    //temp = tan(tilt) * height;
    frontBack = ((.5 * length) - (tan(tilt) * height));
    frontWeight = ((frontBack / length) * Weight);
    backWeight = (Weight - frontWeight);

    #define froll (arcsin(((float) frontIncline[1] - offset) / sensitivity))
    //temp = tan(roll) * height;
    #define fleftRight ((.5 * width) - (tan(froll) * height))
    weight[0] = (fleftRight / width) * frontWeight;
    weight[1] = frontWeight - weight[0];

    #define broll (arcsin(((1024 - (float) backIncline[1]) - offset) / sensitivity))
    //temp = tan(roll) * height;
    #define bleftRight ((.5 * width) - (tan(broll) * height))
    weight[2] = (bleftRight / width) * backWeight;
    weight[3] = backWeight - weight[0];

    for (i = 0; i < 4; i++)
    {
       if (weight[i] < threshold)
       {
         if (output[i] > 10)
            output[i] = output[i] - mod;
         else if (output[i] < -10)
              output[i] = output[i] + mod;
       }
    }
}
```

Appendix D – Truth Model Code

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <math.h>
/* add -lm on gcc to get this to work right

Additional notes:

Wheel positions are as follows:
    0       1



    2       3

This version is treating the vehicle as having 2 wheels
Turning should work in 2 dimensions, needs checking for 3
Distance traveled while turning is calculated in short, straight line estimates
Retrieving speed data from wheels needs to be developed
Interpreting and retrieving information from accelerometers needs to be developed
*/

const double radius = .1143;
const double wheelbase = .32;
const double mass = 11.748;
const double gravity = 9.81;
double dSpeed[4];
double cSpeed[4];
double oldSpeed[4] = [0, 0, 0, 0];
double oldLoc[3];
double curLoc[3] = [0, 0, 0];
double distance[3];
double acc[3];
double pitch = 0, roll = 0, yaw = 0;
double delta = 0;

void getSpeed(int);
void calcAcc(double);
void calcDist(double);
double calcForce(int);
double calcForceAdjust(double);
void calcSpeed(int, double);
double calcSlip();
double calcTurnRadius();
```

```
void updateLoc();
void updateSpeed();

int main(int argc, char **argv)
{
   int time, i, j;
   char holder[128];
   double rad;

   time = 1;

   for (i = 0; i < 2; i++)
   {
      getSpeed(i);
      oldSpeed[i] = dSpeed[i] / (radius * 2 * 3.1415);
      calcSpeed(i, time);
   }

   if (cSpeed[0] == cSpeed[1])
      for (i = 1; i <= time; i++)
      {
         for (j = 0; i <= 10; i++)
         {
           calcAcc(.1);
           calcDist(.1);
           updateLoc();
           updateSpeed();
         }

         printf("X-distance traveled: %f\n", curLoc[0]);
         printf("Y-distance traveled: %f\n", curLoc[1]);
         printf("Z-distance traveled: %f\n", curLoc[2]);
      }
   else
   {
      rad = calcTurnRadius();
      printf("Radius is: %f\n", rad);
      printf("Counter-force is: %f", calcForceAdjust(rad));
   }
   // More to be added later

   fgets(holder, 128, stdin);
}

void getSpeed (int wheel)
{
    char input[128];

    printf("Desired speed of wheel %i is: ", wheel);
    fgets(input, 128, stdin);
    dSpeed[wheel] = atof(input);
}

void calcAcc (double time)
{
    int i;
    double diff = 0;

    for (i = 0; i < 3; i++)
    {
        diff = ((cSpeed[0] - oldSpeed[0]) + (cSpeed[1] - oldSpeed[1])) / 2;
        // Though the above seems correct, it doesn't seem to work as intended
        // Most likely the problems are time-constant related
        acc[i] = diff / time;
        printf("Acc[%d] = %f\n", i, acc[i]);
    }
}

void calcDist (double time)
{
    distance[0] = cos(yaw + delta) * ((cSpeed[0] * radius * 2 * 3.1415 * time) + (.5 *
acc[0] * 2 * time));
    distance[1] = sin(yaw + delta) * ((cSpeed[1] * radius * 2 * 3.1415 * time) + (.5 *
acc[1] * 2 * time));
    distance[2] = sin(pitch + delta) * ((cSpeed[2] * radius * 2 * 3.1415 * time) + (.5 *
acc[2] * 2 * time));
}

void calcSpeed (int wheel, double interval)
{
    cSpeed[wheel] = dSpeed[wheel] / (radius * 2 * 3.1415 * interval); // replace with
wheel readings later
}

void updateLoc ()
{
    oldLoc[0] = curLoc[0];
    oldLoc[1] = curLoc[1];
    oldLoc[2] = curLoc[2];

    curLoc[0] = oldLoc[0] + distance[0];
    curLoc[1] = oldLoc[1] + distance[1];
    curLoc[2] = oldLoc[2] + distance[2];
}

double calcTurnRadius ()
{
      double rad;
      double slope;

      if (cSpeed[0] > cSpeed[1])
      {
          slope = (cSpeed[0] - cSpeed[1]) / wheelbase;
          rad = (cSpeed[0] / slope) - (wheelbase / 2);
      }
      else
      {
          slope = (cSpeed[1] - cSpeed[0]) / wheelbase;
          rad = (cSpeed[1] / slope) - (wheelbase / 2);
      }

      return rad;
}

double calcForce(int w)
{
      double force;
      double front, back, left, right;

      if (yaw > 45 || roll > 45)
      {
            printf("Unstable position.\n");
            return 0;
      }

      back = 1 - cos(pitch);
      front = 1 - back;

      force = .25 * mass * gravity;

      return force;
}

double calcForceAdjust (double rad)
{
      double force;

      force = (((oldSpeed[0] + oldSpeed[1]) / 2) * ((cSpeed[0] + cSpeed[1]) / 2)) / rad;
```

```

        return force * mass;
    }

    void updateSpeed()
    {
        oldSpeed[0] = cSpeed[0];
        oldSpeed[1] = cSpeed[1];
        oldSpeed[2] = cSpeed[2];
    }
```

Appendix E – User Interface Code

```
Private Sub Form_Load()

    ' Fire CommReceive Event Every X Bytes(we specify)
    MSComm1.RThreshold = 1

    ' When Inputting Data, Input X at a time(we specify)
    MSComm1.InputLen = 1

    ' 115200 Baud, No Parity, 8 Data Bits, 1 Stop Bit
    MSComm1.Settings = "115200,n,8,1"

    ' Disable DTR
    MSComm1.DTREnable = True
    MSComm1.NullDiscard = False
    MSComm1.Handshaking = comNone

    'Open COMM Port #5
    MSComm1.CommPort = 5
    MSComm1.InBufferSize = 2048
    MSComm1.PortOpen = True     'open the comm port

    text1.Text = ""             'Clear the output text box
    Text2.Text = ""

End Sub

Private Sub MSComm1_OnComm()

    'testing program 1


    Dim sData As String    ' Holds our incoming data

    ev = MSComm1.CommEvent


    If ev = comEvReceive Then


    sData = MSComm1.Input ' Get data (2 bytes)

    If sData = "z" Then   ' Clear text box for new input data

    text1.Text = " "

    End If

    text1.Text = text1.Text + sData 'Append newest character


End If

End Sub
Private Sub Command1_Click()

    'Sending command to Robot
    Text2.Text = "left"
    MSComm1.output = "$l"

End Sub
Private Sub Command2_Click()

    'Sending command to Robot
    MSComm1.output = "$f"
    Text2.Text = "forward"

End Sub
Private Sub Command3_Click()

    'Sending command to Robot
    MSComm1.output = "$b"
    Text2.Text = "back"

End Sub
Private Sub Command4_Click()

    'Sending command to Robot
    MSComm1.output = "$r"
    Text2.Text = "right"

End Sub
Private Sub Command5_Click()

    'Sending command to robot
    MSComm1.output = "$s"
    Text2.Text = "stop"

End Sub

```

```
Private Sub Command6_Click()

   'Sending Command to camera
   Text2.Text = "camera left"
   MSComm1.output = "$z"

End Sub

Private Sub Command8_Click()
   'Sending Command to camera
   Text2.Text = "Camera down"
   MSComm1.output = "$d"

End Sub

Private Sub Command9_Click()
   'Sending Command to camera
   Text2.Text = "camera center"
   MSComm1.output = "$c"

End Sub
Private Sub Command10_Click()

   'Sending Command to camera
   Text2.Text = " Camera up"
   MSComm1.output = "$u"

End Sub

Private Sub Command11_Click()

   'Sending Command to camera
   Text2.Text = "camera right"
   MSComm1.output = "$x"

End Sub
Private Sub Form_KeyDown(KeyCode As Integer, Shift As Integer)
   Select Case KeyCode
      Case 83

         'assign to keyboard "S"
         Call Command3_Click
         Text2.Text = "back"

      Case 87

```

```
            'assign to keyboard "W"
            Call Command2_Click
            Text2.Text = "forward"

        Case 65

            'assign to keyboard "D"
            Call Command1_Click
            Text2.Text = "left"

        Case 68

            'assign to keyboard "A"
            Call Command4_Click
            Text2.Text = "right"

        Case 27

           'assign to keyboard "Esc"
           Call Command5_Click
           Text2.Text = "stop"

        Case 73

           'assign to keyboard "I"
           Call Command10_Click
           Text2.Text = " Camera up"

        Case 75

           'assign camera up to K
           Call Command8_Click
           Text2.Text = "Camera down"

        Case 74

           'assign camera up to J
           Call Command6_Click
           Text2.Text = "camera left"

        Case 76

           'assign camera up to L
           Call Command11_Click
           Text2.Text = "camera right"

        Case 67

           'assign camera up to C
           Call Command9_Click
           Text2.Text = "camera center"


        End Select


End Sub
```

Appendix F – Part Cost List

Bill of Materials			
Item	Quantity	Cost	Total
Hawking Technology Camera	1	109.95	$109.95
Serial to Wi-Fi Converter	1	200.00	$200.00
Inclinometer (VTI SCA100T-D02)	4	65.88	$263.52
20 AMP auto reset fuse	4	4.19	$16.76
Power Adaptor for Qwerk	1	21.00	$21.00
Misc Wires and Hardware	1	20.00	$20.00
Batteries	3	60.00	$180.00
PCB layout	2	33.00	$66.00
Linksys Router	1	40.00	$40.00
X10 Wireless Camera	-1	100.00	-$100.00
		07-08 Subtotal	$796.23
		06-07 subtotal	$2,487.00
		Total	$3,283.23