|  |
| --- |
| Ninja TurtlEEs |
| Final Report |
| John O’Brien, Casey Morris, Josh Vahala, Justin Yoder |

2014

Table of Contents

1 Introduction……………………………………………………………………………..…….…2

2 Detailed System Requirements………………………………………………….…….….……..5

3 Detailed Project Description…………………………………………………….…….….……..8

3.1 System Theory of Operation……………………………………………….…....….…8

3.2 System Block Diagram………………………………………………….….…………9

3.3 Detailed Design/Operation of Motor Controls…………………………….………….9

3.4 Detailed Design/Operation of Environment Sensing……………………..…….……18

3.5 Detailed Design/Operation of User Interface…………………………………….….24

4 System Integration Testing…………………………………………………………………….30

5 User’s Manual/Installation Manual………………………………………………………….…32

6 To-Market Design Changes……………………………………………………………………35

7 Conclusions…………………………………………………………………………………….36

8 Appendices………………………………………………………………………………….….37

**1 Introduction**

The city of South Bend, Indiana derives its name from the southern bend of the Saint Joseph’s River. Flowing into this southern bend are many different tributaries and watersheds. However, one such tributary, Bowman Creek, has historically had many problems. Most of the streams around the river are able to support a flourishing ecosystem, but Bowman Creek has long been unable to support such aquatic life. Now, the city of South Bend is working toward revitalizing the Bowman Creek area. City officials believe that by bringing life back into the Bowman Creek watershed they can stimulate the local communities and promote the welfare of the people who live there.

The city of South Bend officials also know that the reason Bowman Creek cannot support life is that the water levels in the tributary are too low. However, the officials do not know why they are too low. One theory is that the water from the stream is leaking into sewer pipes that run beneath the tributary in different points. If this is true, it would explain why there is no life. It would also mean that the sewage treatment plant is treating many gallons of water more than it should be. However, in order to investigate this theory, the officials need to be able to explore the sewage pipes underground. The city currently owns a robot that has the ability to take video of the sewer pipes so that the city engineer can remotely explore the tunnels. However, this robot is too small to image the large tunnels that run underneath Bowman Creek. This is why the city of South Bend has turned to Notre Dame to help solve this problem. The city officials, represented by Gary Gilot as a liaison, pitched the idea of a senior design team designing a robot that can investigate sewer tunnels and film what is going on down there. This is the inspiration for our team’s senior design project.

One of our first tasks in this project was to talk to city engineer Patrick Henthorn. Patrick was able to give us more details on the project. These details included potential goals as well as information on the operating environment. First, he informed us that the pipe size that we would be working with would be 66”-70” diameter. He also warned us that the environment would be very wet and so any electronics would have to be waterproofed before going down to the sewers. Along with that, the robot would have to be able to drive over any types of debris that would be down there including mud, clay, dirt, or trash. Lastly, he expressed his desire to have the video output something that could be as interactive as google maps’ street view. He said that his job would be easier if he was able to control looking through the tunnel video after the actual exploration was done.

Having heard all of the goals and expectations, the team then began to design a robot that we felt would be able to accomplish what is needed. First, we decided to tackle the problem of the limited mobility inside of the sewer pipes. The team has decided that the best method of getting around the sewer pipes would be to have a chassis with four wheels attached to it. We need the wheels to be large enough that the robot can maneuver over debris on the ground, but small enough that it can have good control and use a small amount of power. We also needed to choose motors that would be able to generate a reasonable amount of torque and gearboxes that would increase this torque output. The other main hurdle for this project is having the robot locate leakage areas. In order to most efficiently look for the leakage areas, the team has decided that it is necessary to have a camera that can rotate 360 degrees horizontally and 90 degrees vertically. This camera would allow for viewing of the entire sewer area. We want the robot to be able to locate any areas of leakage and to not be limited in its exploration of the pipes. A camera that is mounted on a track that allows for motion in two different directions will be able to focus in on any part of the pipe.

Next, we needed to overcome the hurdles of control and detecting leakage. We have decided that the robot will have two functioning modes. The main mode will be a human controlled mode in which the robot is simply driven around the pipes by a human with a remote control. This human will have a controller that will control the motors and the camera. The information that will be used to control the robot will be sent wirelessly to the robot. Likewise, the video will also be sent back to a computer wirelessly. Then, the video would be sent to a video processor that will be able to identify where leaks are in the pipe based on the phosphorescent dye. The secondary mode will be an autonomous mode in which the robot will drive itself down the pipes and scan all of the walls as it goes along. This mode will most likely be slower and require more power from the robot because it would need to do a very thorough search in order to make sure the video processing is accurate in locating leakage spots. The robot would also include a functioning infrared (distance) sensor and gyroscope. The distance sensor would be pointed in front of the robot so that the robot would be able to detect possible objects in its path. The gyroscope would be used to make sure the robot would not tip over. It will be especially useful if the robot tries to drive up the circular wall. The gyroscope would detect the change in level and force the wheels to correct the robot.

In the end, the robot functioned very well. We were able to successfully integrate all of our subsystems into a robot that could investigate sewer tunnels. We figured out how to make the robot a wirelessly controlled system and we were able to use an Xbox controller to control the motion of the wheels and camera on the robot. We also came out with very good results for the sensors of the robot. Both the infrared sensor and the gyroscope sensors worked very well. However, the video processing part of the project never translated out of the design phase. We wanted to make it so that the robot would be able to detect cracks in the walls by itself, but instead a human is still needed to explore the video that is taken in the sewers. Likewise, we never got around to coding an autonomous mode for the robot. This would have taken extensive amounts of code to be implemented and tested and we just ran out of time to implement that function. Still, the robot was an extreme success as this was a very ambitious project that yielded good results.

**2 Detailed System Requirements**

Our proposed solution is a robot that can scan the tunnels. The robot will be approximately 22”x18”x12” (HxLxW). An image of the body is shown below in Figure 1.

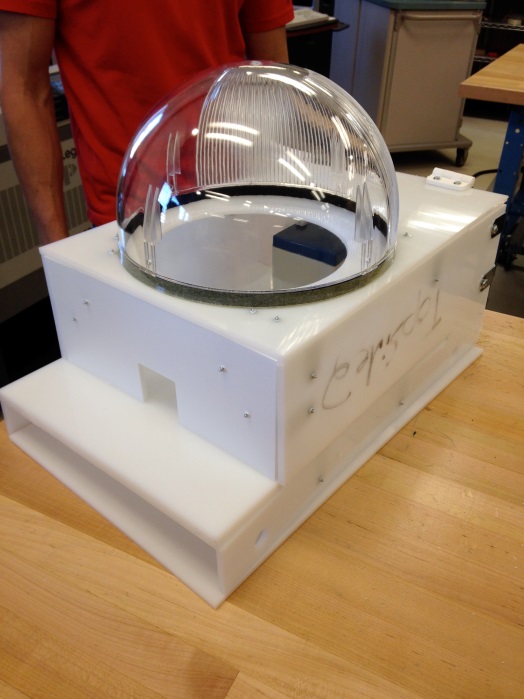


Figure Body Design

It should also be light enough that a user could easily lift and maneuver it. Besides these obvious design points, we are designing this robot so that it will reliably operate in a sewer environment and it will be user-friendly such that different operators can run it. Considering all of these general design criteria, we began thinking more specifically about the requirements of our system. Firstly, we decided to tackle some of the problems associated with the environment. The robot needs to move without slipping on the mud or water that may be present in the pipes, but it also needs to be able to climb over little obstacles such as sticks or rocks. Therefore, we implemented a system of four wheels that are each 8” in diameter so that they give the robot suitable clearance. The wheels are shown in Figure 2.



Figure Designed Wheels

Along with that, the entire casing for the robot must be waterproof so as to protect the electronic systems that are at work in the robot. Lastly, the robot must be equipped with a system of LED lights that will work to light up the tunnel in the direction that the camera is pointing. We will use LED lights because they are the most efficient and they have the ability of being pointed in a specific direction.

After the challenges of the operating environment, we considered some of the systems that we would need to actually run the robot. We have decided that the robot will have two electric motors that will control the left- and right-back wheels of the robot. These motors will be controlled by an on board microcontroller via a user interface. This user interface will also be able to control two motors that will be used to move the camera vertically ninety degrees as well as horizontally three hundred and sixty degrees. Furthermore, the robot will have two sensors that will allow it to be protected from some user error. The first sensor is an infrared sensor that acts as a distance sensor. It will stop the robot if it registers that an obstacle is too close. The second sensor is gyroscope or tilt sensor. This is to protect the robot if it begins driving up the circular wall of the pipe. If the tilt sensor returns a dangerous value then the robot will correct its movement to level out again. Both of these sensors will feed data to the microcontroller.

Additionally, we decided to make the robot a wireless robot. This decision opened up many more challenges for the group to overcome. Perhaps the most difficult was the task of receiving and sending signals to the microcontroller. To do this, the group decided to purchase a Raspberry Pi computer that was mounted on the robot. This Raspberry Pi computer has a Wi-Fi dongle that allows it to send and receive signals to a router and then a computer. The Raspberry Pi was connected to the microcontroller so that the computer could communicate with the microcontroller. Along with the decision to make the robot wireless was the decision to make it battery powered. Thus, we found a suitable battery that had 18 amp hours and could supply 12 volts. We also implemented a battery level sensor so that a user could know how much battery power was left.

Lastly, this project required an extensive user interface. This user interface needed to include sections for each of the sensors as well as a way to show the wireless feed from the camera. It also needed an Xbox controller so that the user could control the motions of the wheels and the camera. These all needed to be coded to operate through a TCP connection with the robot at certain timed intervals so that there was no lag. It also needed to be simple enough to use that any user could figure out what the different sections meant.

**3 Detailed Project Description**

3.1 System Theory of Operation

This project is a very complex system with my integrated parts. The system starts with the user interface that is loaded onto a computer. From this user interface a person can communicate with a Raspberry Pi computer through a TCP connection with Xbox controller and receive feedback from different sensors. The Raspberry Pi is, in turn, interfaced with our microcontroller board using an SPI connection. That board is then hooked up to two motors, two servos, and three sensors. The board also has a power section that is used to regulate a 12V battery to 5V and then again to 3.3V. This all works quickly and seamlessly as one functioning robot. A full frontal view of the robot can be seen in Figure 3.

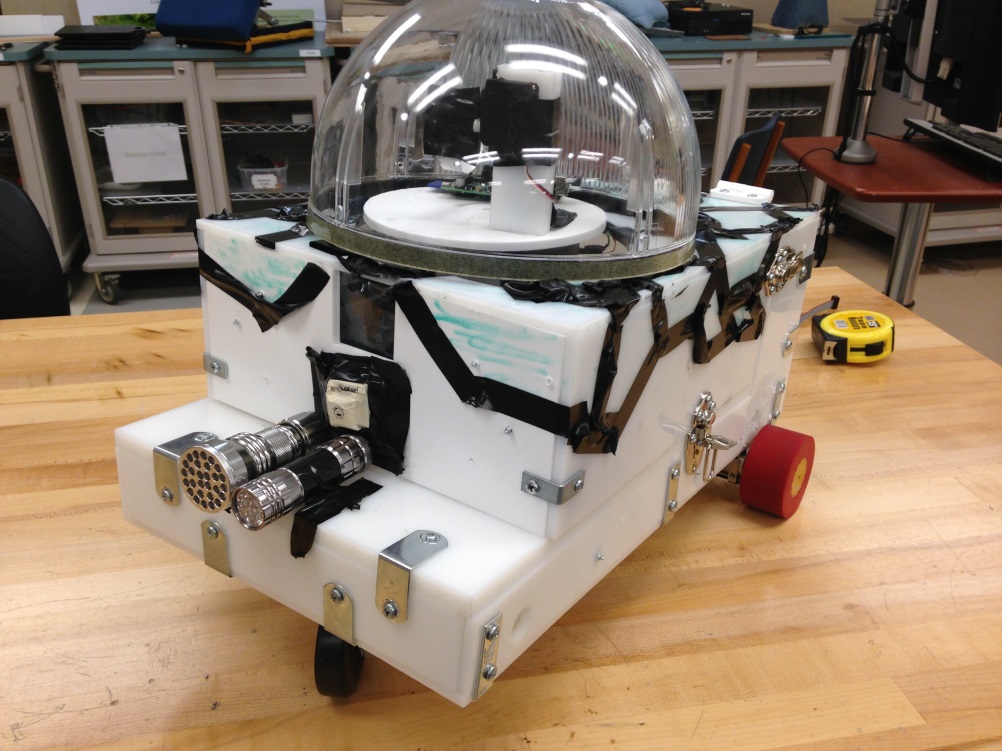
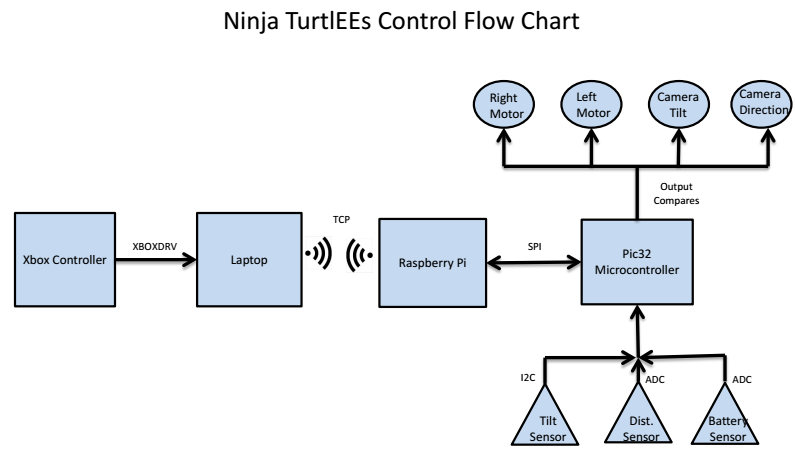


Figure Front View

3.2 System Block Diagram



3.3 Detailed Design/Operation of Motor Controls

The project consisted of four separate motors that were capable of being controlled by the user. There were two servo motors to control the angle and position of the camera. A continuous motion servo on which we mounted a plate with the camera and which spun to change the horizontal direction in which the camera was facing. A conventional servo was then used to control the tilt position of the camera, or the vertical direction it was looking. The other two motors were the brushed DC motors which we connected to our rear wheels and used to drive the robot. Both servos and the brushed DC motors share the trait that their position and speed are based on a duty cycle applied to their terminals. The duty cycle is controlled through the output compare pins on the microcontroller. Before going in further detail about that, I will first explain how the microcontroller determined the duty cycles needed for each motor.

*SPI*

As previously explained, the user is able to determine the speed and direction to move the robot as well as the position of the camera using an Xbox controller connected to his or her laptop. The laptop then processes the Xbox controller data and sends position commands to the Raspberry Pi for each motor in a known protocol, which is shown in the flow chart in user interface subsystem section below. Once the Raspberry Pi receives the command bytes from the TCP connection, it utilizes the functions downloaded as a part of the wiringPi.h C class to send these bytes via Serial Peripheral Interface (SPI) to the microcontroller. The Raspberry Pi is the master in our SPI set up and when it has new information from the laptop, it initiates a sequence of 4 bytes to be sent to the microcontroller. SPI was chosen because it could be most easily configured on the Raspberry Pi and because the project did not call for two masters to initiate data transfer. The SPI runs at 10Mhz, as that was plenty fast enough for our purposes and was capable on both the Pi and microcontroller.

On the microcontroller side, the program runs a while loop, waiting for an interrupt from the SPI. Because the bytes are sent via SPI one at a time, the microcontroller waits until it has placed all four data bytes into a buffer on the microcontroller before setting a semaphore telling the main program that new data has arrived and the motors should be updated.

Once the semaphore is set, the main program then steps through the data one byte at a time. As previously explained, the first byte received controls the speed of the drive motors, and is sent as either a speed up, or slow down command in a particular direction. The second byte controls the turning, which is done by spinning the drive motors at different speeds. The third byte sets the vertical position of the camera and the last byte spins the continuous motion servo. Below is explained how the commands effect the duty ratios of the motors and what they mean.

*DRIVE MOTORS*



Figure Motor and Gearboxes

As mentioned above, the control of the drive motors is done by the duty cycle of the signal applied to its terminals. As the duty cycle on the motor increases, more power is applied to the wheel and the robot can move faster. The part of the microcontroller that allows the use of duty cycles is the Output Compare module which has a PWM (Pulse Width Modulated) function. The setup of the output compares is relatively simple, all that is needed is to set the OCM bits to 7 and PWM mode is enabled. One other choice that needs to be made before going forward is which Timer module on the microcontroller will be used to set the PWM for a particular output compare, and that is controlled by the OCTSEL bit, 0 meaning Timer2 and 1 meaning Timer3. Once the timer has been selected, the last thing to do is to set the frequency of the timer and its initial duty cycle.

For our drive motors, we chose a signaling frequency of 1kHz, which is well below the maximum switching frequency of any transistor switches, but it is still fast enough to apply enough continuous power to our motor. To set the frequency, the Timer3 module had to be configured to allow for this. To get a 1kHz signal with the best resolution, we set the Timer prescaler (bit TCKPS to 0) which controls the speed at which the counter increases. With this setting, we set out frequency to be 8000 pulses by setting PR3 = 8000. This means that the timer will count up to 8000, which we set to take I ms, before resetting and starting over. To set the duty cycle, the OCxRS register in the output compare module is set to an integer between 0 and 8000, corresponding to the duty cycle, i.e. OCxRS = 4000 means the duty cycle would be .5. Thus changing the value in the OCxRS register would change the duty cycle. Exactly how the duty cycle effected the speed depended on the motor controller used. In our project, we actually used two different methods; however, they both had the same effect.

*H-BRIDGE*

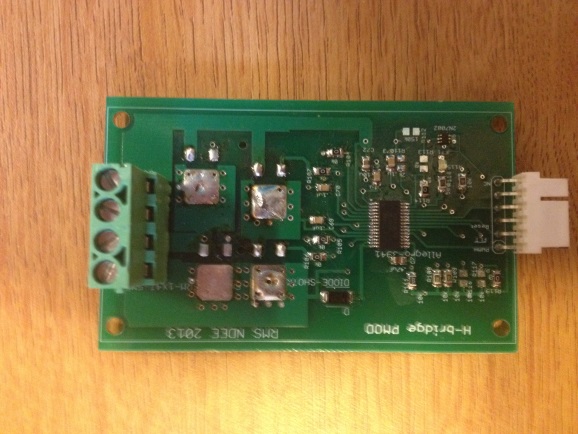


Figure Broken H-Bridge

The first method we used to control the speed of the motors was through the use of an H-Bridge. Basically in an H-Bridge, the motor is connected between two switches and the duty cycle is applied to the switches, thus applying the voltage to the motor when the switches are closed. The H-Bridges we explored were controlled through an Allegro H-Bridge driver, which took as an input the duty cycle and the phase, or the direction in which to spin the motors. This method makes the most sense as the duty cycle is directly applied to the switches so a duty cycle of 0 means the motors are off and a duty cycle of 1 meaning they are full speed, having increased the torque applied to the wheels linearly.

In this set-up, the microcontroller had to take the information from the Xbox controller and change the duty cycle and set the phase pin accordingly. In our project, top speed of the robot was not a concern, as we only needed it to move at a relatively slow, controlled pace, so we set a maximum speed, or duty cycle to apply. Thus, the microcontroller took the speed up and slow down commands and direction from the Xbox controller and set the pins accordingly.

This was a relatively easy portion to test, as the only things to calibrate were the duty cycle that related to the top speed we need. Also, we wanted to test for the percent we would want to speed up or slow down the robot as it accelerates or decelerates. The only limitation with this was that was dependent on having the robot constructed and connected before we were able to get a good idea for these values, however, it was very easy to calibrate.

Thus, the first data byte sent to the microcontroller determined the base speed to send to both motors by telling it to increase or decrease the duty cycle until the limits (0 and max speed) were reached. The second byte controlled the turning of the robot, which was done by setting the duty cycles of the left and right motors to different values so they would spin at different rates. To simply things, we implored only a turn right, go straight, and turn left command. With the speed of the robot already set with the first byte, the only thing the second byte was used for use modifying those numbers. Obviously, if the go right command was received, the motors would be applied the same duty cycle; however, if a turning command was sent, the duty cycle of one of the motors would be overridden and set to 0 so that only one motor would spin and thus turn the motor. For example, if a turn right command was given, the right motor would be set to a duty cycle of 0 and just the left one would spin at the predetermined speed. This worked fairly well, however, with more time, we would wish to implement more robust ways of spinning, as in more precisely controlling the duty cycle of each motor and be able to turn with a more precise desired turn radius. This is how the motors were controlled using the H-Bridge set up. This was not the perfect solution however, as when a motor would stall, the H-Bridge boards had a propensity of blowing up, sometimes ruining the main board, so we had to search for an additional method.

*MOTOR CONTROLLER*

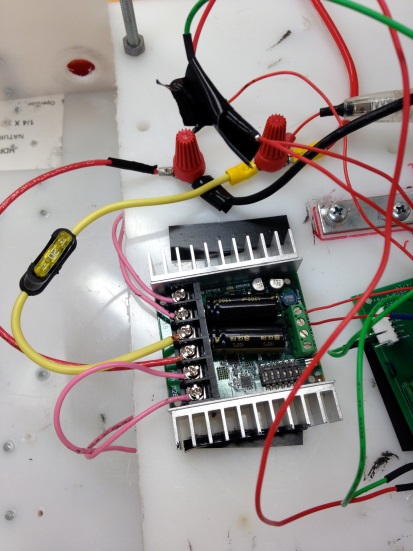


Figure Motor Controller

Instead of designing our own H-Bridge and H-Bridge driver, a motor controller, such as the Sabertooth 2x25, could be bought off of the shelf. The overall idea of using this motor controller to spin the motors was similar, as in the duty cycle was what the microcontroller needed to control, but some of the details were actually quite different. The motor controller could be set up in a variety of modes of which we chose Analog mode. Basically, the motor controller reads the voltage on the input signal, for ours between 0V and 5V and changes the speed and direction of the motor based on that. In this mode, a voltage of 2.5V at the input corresponds to not moving, and voltage above 2.5 spin the motor forward at a speed based on the voltage difference, and any voltage below 2.5V on the input drive the motors backward. To control this with a PWM signal, a low-pass RC filter is needed to filter out the switching and to give one voltage value. The setup is shown in Figure 7.

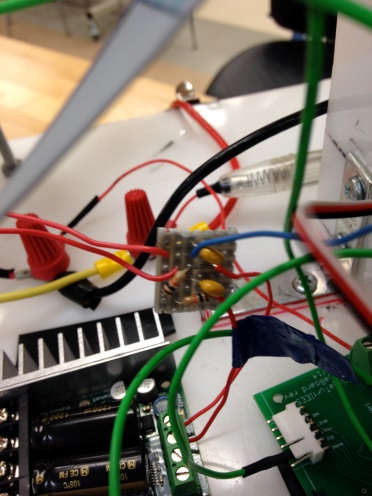


Figure Low-Pass RC filter

The main difference between this set up and H-Bridge set up was calibration of the duty cycles that corresponded with 2.5V and the maximum speed, or voltage, we wanted to signal with. This set up was quite easy having had everything enabled when running the H-Bridge method, and was actually easier as the Phase did not have to be controlled by the microcontroller explicitly because it was already accounted for in the PWM signal. This method was more robust than the H-Bridge method, however, no difference between the two methods was noticeable from the user’s prospective.

*SERVOS*

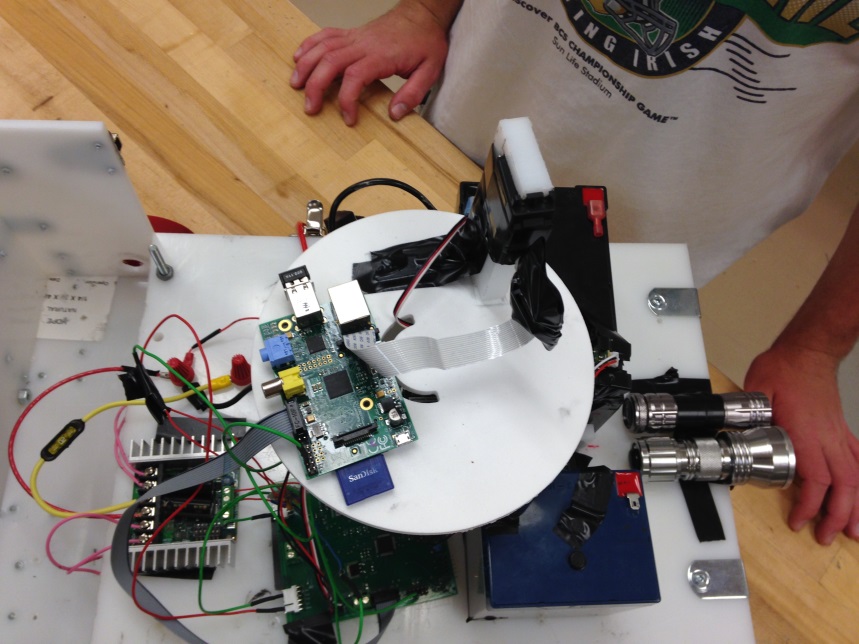


Figure Continuous Spins Base and Tilt Moves Camera

*Continuous motion*

Just as the drive motors are dependent on the duty cycle applied to its terminals, so too is the continuous motion servo used to control the horizontal position of the camera. This servo, however, reacts a little differently to the duty cycle than the drive motors. The first difference is that the period of the signal applied to the needs to be between 20 and 30 milliseconds, instead of 1 ms. It also works in that the servo holds its current position when a pulse of width 1.5 ms is applied. When the duty cycle increases, the servo begins to spin in one direction, and when it decreases below 1.5 ms, it spins in the opposite direction, and the speed of the spin is proportional to the difference in duty cycle from 1.5 ms width, with a pulse width not to exceed 2 ms.

To configure our microcontroller to provide this signal, we set up another output compare, but instead it was set to signal with Timer2, which we enabled with the 25 ms period (see code for specifics). We then tested the servo to the find the integer values needed in the OCxRS register that corresponded to the duty cycles that held the motor still and also spun in each direction at the speed that we needed.

This type of servo will spin continuously as long as the width of the pulse at its terminals is not 1.5 ms, so that presented us with two questions to answer. The first question we had to answer was how long to let the servo spin to best coordinate with the user input (i.e. we did not want the motor to spin half way around when the user only tried to move the camera a few degrees). The second question was how to prevent the camera from rotating around completely which was an issue because on the rotating disk we sat both the Raspberry Pi and camera, which had hard wired connections to the microcontroller below it. Thus, because we were limited by the cords, we had to find a way to prevent the wires from becoming tangled and being pulled out of the board.

The first question we were able to answer quite easily because of other design decisions we had made previously. For our protocol for messages between the laptop/Xbox controller and the Raspberry Pi/microcontroller, we stipulated that messages be sent every 150 ms. This was fast enough to allow the servo to react very well with the user joystick, so we only changed the duty cycle (the OCxRS register) when a turn command was given and then did not change that value until a stop command was received.

The biggest dilemma we had was deciding how to best limit the angle of rotation for the continuous servo. We debated whether that logic and control should be done on the microcontroller or on the laptop before it sent commands to the Raspberry Pi. Ultimately, we decided that it could be done best on the laptop, as it had more processing power. To enable this on the laptop, we had a variable counting the number of times a turn command had been sent in each direction, and when the limit had been met, it would send a stay put command instead of the next turn command. The only thing we had to test for this was what we wanted to set as the maximum number of turns we would allow. This was done quite easily.

*Tilt Motion*

The last motor we used was a conventional servo motor whose position is set by the duty cycle. It requires the same period as the continuous motion servo, so was set up using the same Timer2. This type of servo only has 180 degree motion available, and instead of continuously spinning based on a duty cycle, the position corresponds to a specific duty cycle. For example, a pulse width of 1.5 ms is the “neutral” position, a pulse width of 1 ms corresponding to a position 90-degree from the neutral position in one direction and 2 ms pulse corresponding to a position 90 degrees from the neutral position in the other direction. Pulses in that range thus hold their position based on that pulse. This was the easiest of the motors to test, as the only thing we really had to test was the maximum and minimum points we wanted to set for the camera, as 180 degrees of movement was not necessary. As with all of the other motors, the exact values and initialization code is in the appendix.

3.4 Detailed Design/Operation of Environment Sensing

The goal of this subsystem is to allow the robot to sense its environment so that the user does not need to be constantly aware of the surroundings at all times. The user should be able to continue to move forward while looking upward with the camera and not worry about the robot running into debris or falling over because of the rounded walls in the sewer pipe. Also, the user should know when it is time to retrieve the robot so that the battery does not die. This subsystem accomplishes all of these goals with an infrared distance sensor, an accelerometer, and a battery level sensor.

Each of these sensors must send data to the microcontroller so that it may relay the information to the user located on the surface. The infrared and battery level sensors accomplish this by feeding the signal through a unity gain op-amp and then into the PIC32 ADC module which allows for analog to digital conversion. The accelerometer does this through the PIC32 I2C module.

*OBSTACLE DETECTION*

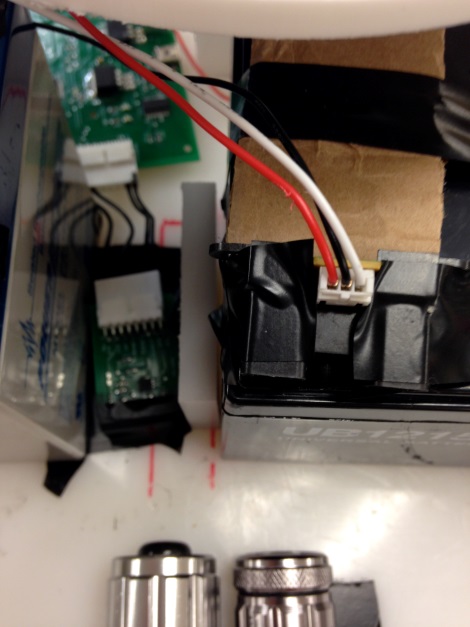


Figure IR Sensor (Right) and Tilt Sensor (Left)

The idea behind obstacle detection is that the user could be looking at the ceiling of the sewer for cracks when the robot could run into an obstacle that it cannot traverse. In order to combat this, we have decided to use an infrared sensor angled slightly toward the ground to track changes in the floor height. Possible scenarios are shown in Figure 10. The robot can either see nothing in its line of sight (i.e. the value will represent a known constant and the user will be shown a green block), see something to possibly worry about if the infrared sensor is cut at a far distance (i.e. the value will be less than some warning threshold and the user will be shown a yellow block), or see something imminent in its range that the user must act quickly to avoid (i.e. the value will be less than some “danger zone” threshold and the user will be shown a red block). 

Infrared line of sight

No obstacle

Infrared line of sight

Possible obstacle

Infrared line of sight

Obstacle

Figure 10 a) No obstacle detected. b) Possible obstacle detected. c) Obstacle detected

In order to accomplish this functionality, we chose a SHARP GP2Y0A02 Infrared distance sensor with a range of 20 cm to 150 cm and a 5V operating voltage. We chose this distance range because we wanted the user to have fair warning before running into obstacles. After testing, we found that we sent green if the distance returned something greater than 1 meter, we sent yellow if the distance returned something less than 80 cm, and we returned red if the distance returned something less than 60 cm. In choosing these distance values, we first consulted the device datasheet to learn expected voltage readings for different distances. This is shown in Figure 11. Without a simple curve to follow, we focused on ADC conversion values instead of a conversion to distance exactly. Also, we performed all testing with minimal sunlight entering the sensor because the infrared from the sun could change our values from what we expect in a sunless sewer tunnel.

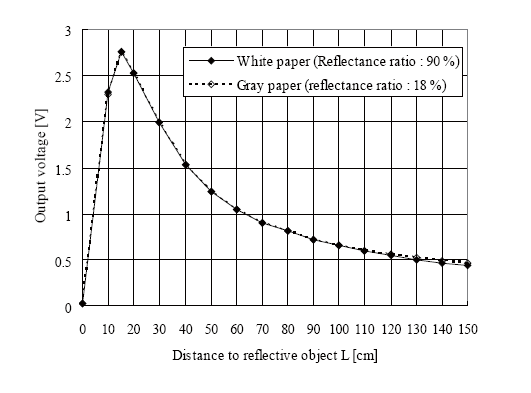


Figure 11 Infrared sensor expected output values with distance

On the robot, we angled the sensor so that these values worked with our goals.

With only three wires (power, ground, and output signal), the sensor was very simple to set up and use. In connecting it to our microcontroller, we chose to send it first through a unity gain operational-amplifier so that the IR sensor cannot load down the rest of the circuit. From there the signal travels to the AN13 pin for analog to digital conversion. We chose this pin because of its proximity to where we wanted to place the IR sensor on the board. However, we had difficulties because we were unaware that the JTAG pin defaulted at this location. After turning off JTAG, we were able to get accurate data from the device. This flow is shown in Figure 12. For further communication protocols after the signal arrives at the PIC32 microcontroller, see other subsystems. Also, related to the physical placement of the sensor, we applied a thin plastic film (from a plastic bag) to the front of the robot for the infrared sensor to look through. We had tested on thicker materials like acrylic and glass but the signal did not effectively return to the sensor. In a future update, an IR passing material should be purchased and more permanently mounted on the robot.

IR sensor

Unity gain op-amp

PIC32 microcontroller

Figure 12 IR Sensor Flow

In physically setting the ADC module up, we decided to sample at the slowest possible rate. We chose this because our robot cannot move quickly and thus more than a hundred thousand samples per second are plenty for our use.

*BATTERY LEVEL SENSING*

The goal behind this sensor is to warn the user of the state of the battery when the robot is in the sewer. The sensing is based upon a voltage divider that takes the 12V battery input and makes it into a value tolerable to a 5V I/O pin on the microcontroller. Because the battery depletes in voltage with time and use, we can check what the current voltage is and give the user an approximation that can aid them in using time wisely.

The battery level sensing works very similarly to the IR sensor because they both use the ADC module of the PIC32 and also use unity gain op-amps to isolate the sensor from the rest of the board. However, instead of checking for distance based voltages, we are looking for depletion based voltages. We do this with the voltage divider shown in Figure 13. Z1 and Z2 are 300kohm and 100kohm respectively. This circuit sends approximately one fourth of the battery voltage to the microcontroller, thereby sending a usable value less than 5V at all times.

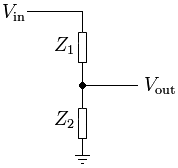


Figure 13 Battery Sensor Voltage Divider

We chose what voltage values to convert to ADC threshold numbers to warn the user of low battery by consulting the battery datasheet. This datasheet showed that the battery drained pretty linearly over the battery’s lifespan. 100% became a 13V value while 0% was declared to be 11.7V. In the end, we had two four thresholds that sent data to the user: Full battery, >50% battery, >25% battery, Replace now. We believe these to be enough values to tell the user when it is time to worry about the battery. In order to get these values for each threshold, we set a voltage source to the voltage we wanted and had the kit board send the ADC value to the UART.

*INVENSENSE MPU6050 ACCELEROMETER AND GYROSCOPE*

The goal behind this sensor is to warn the user if the robot tilts to its side too much. Because the pipes are rounded, the robot has the potential to drive up the side, and it would be nice if the user were warned of this.

The sensor is set up to communicate via I2C to the microcontroller. The data inside of the device is set up as a two byte register for each axis of each sensor (accelerometer X,Y,Z, gyroscope X,Y,Z, and temperature). By taking the device out of sleep mode, data can be read. We only cared about the y-axis accelerometer data to determine whether the robot was tilted along that y-axis. See the device data sheet for exact axes, but the one mentioned is probably the one expected by the reader’s intuition. A function was set up to burst read from two registers at a time and to concatenate them into a single 16 bit int.

Once the data was accessible, we used the kitboard to read the values of the y-axis accelerometer at various angles and we guesstimated the final angle that we would like to warn the user, and the angle that we would like to tell the user they are in the danger zone. Figure 14 shows the various options that the user can see when the sensor is tilted. The GUI also gives direction which it is tilted (clockwise or counter clockwise).

Safe zone

Warning zone

Danger Zone

Figure 14 MPU 6050 tilt sensor

We used the slower rate that the device runs at (100KHz) is choosing our timing frequency because we don’t care for a high speed sensor because the robot does not move very quickly. Also, we repeatedly took it out of sleep because it would not always get out after the first try. Therefore, we did it three times.

3.5 Detailed Design/Operation of User Interface

*XBOX CONTROLLER COMMUNICATIONS*

This subsystem needed to allow the user a way to control the robot. We decided to use an xbox controller because its use is extremely familiar to most people, it has existing programs in place that allow it to interface to a laptop, and because the dual joysticks allow us to independently control the camera motion and driving. We chose to write this program in C++ because it’s a commonly used and versatile language, and because we are very familiar with it.

This subsystem works in two pieces. The first piece is a program called xboxdrv that must be downloaded and installed from the internet. While we do not have the source code for this program, it is relatively simple to use. When running the program, values for the joysticks and buttons are printed out to the terminal window. The joystick values range from -32767 to 32767 in integer values. The buttons values are either “1” if pushed or “0” is not pushed. xboxdrv continuously prints out these values in a formatted fashion line by line. The rate that the values are printed out to the screen varies, but it averages to about 30 lines per second. A catch to the program is that a new line is only printed if at least one value has changed since the last printout. The joysticks are very sensitive, so the program typically prints out a lot of data. This output to the screen is then sent to the second piece of the subsystem via a Linux piping command.

The second piece of this subsystem is the NT\_Laptop\_5\_1\_11a.cpp program that we wrote to serve as the communication link between the xbox controller and the robot, as well as be the graphical user interface for the user. The graphical user interface will be discussed specifically below. After initialization, the program begins by setting up a separate thread that runs independently and asynchronously. The sole function of this thread is to receive all the input data from the xboxdrv program and update the string that will be used to set the current values of the joysticks and buttons. The string that this thread is updating is a global variable that the main program also has access to. Multiplexer locks are used by both the thread and the main program in order to ensure that the data is not corrupted due to attempts at simultaneous modification of the data. We set up the thread because the xboxdrv program sends controller data much faster than we want to use it in our program. Because the thread is asynchronous, it can easily handle the varying rates that xboxdrv sends data, without interrupting the flow of the main program.

The first action of the main program is to copy the data from the string that the thread has been updating. This string houses all of the data from the controller that we want. After updating the string, the main program parses the updated string to extract only the necessary information. The only data we really need is that x-axis and y-axis data from both joysticks, as well as whether or not the “A” button has been pushed. This data is converted to integers so it can be used later on. Once the main program has the necessary values, it must decide what commands to send to the robot. This will be discussed in detail in below.

*COMMANDS VIA TCP*

This subsystem transferred data and commands between the user’s laptop and the Raspberry Pi on the robot. A Raspberry Pi is a very simple and very inexpensive Linux-based computer. Raspberry Pi’s also have an available camera that can be purchased separately, making it ideal for our needs. A TCP connection is a type of data transfer protocol that runs over internet protocol (IP). It is a dedicated connection, and all data requires an acknowledgement from the other side that the data was successfully received. If this acknowledgement is not received within a specified period of time, the data is resent. This ensures that all data is transferred correctly. TCP is slower than its counterpart UDP, however given the low volume of data that we needed to transfer, and the importance of every bit of data that we were transferring, TCP was the obvious choice for the connection.

Upon startup of the Raspberry Pi, TCPserverTest2.c must be run in order to communicate with the laptop. Typically this is done through the laptop with SSH. The Raspberry Pi doesn’t have a dedicated screen. It only has a HDMI port, which makes it impractical to try to use directly out in the field. TCPserverTest2.c sets up the server side of a TCP connection and waits until another program requests to make the connection on the correct port. This is done by first setting up a socket, binding to that socket, then listening for connection attempts on that socket. This must be done before the laptop can connect to it. Once a connection attempt has been received, it will accept that connection. At this point, the connection has been set up, and the program waits to receive data. During initialization of the NT\_Laptop\_5\_1\_11a.cpp program on the laptop, the laptop connects to the Raspberry Pi. The NT\_Laptop\_5\_1\_11a.cpp utilizes a command line argument that specifies the IP address of the Raspberry Pi. This makes it simpler to connect when switching between networks. The laptop program then sends four commands at a time to the Raspberry Pi based on the data it receives from the controller. The first command tells the Pi to either “speed up” or “slow down” in either the forward or reverse direction, or “do nothing.” The second command tells the Pi to either “turn left”, “turn right”, or “do nothing.” The third command tells the Pi the camera should either “look up”, “look down”, or “do nothing.” The fourth command tells the Pi that the camera should either “turn right”, “turn left”, or “do nothing.” The program sets up a deadzone for the joystick data that prevents rapid back-and-forth changes in the commands being sent when the joystick is close to, but not exactly, zero. The laptop program incorporates data received from the sensors and will override user commands if necessary. The program includes a user override that will disable the override commands if the user desires. Once the Pi receives the commands, it immediately relays these commands to the microcontroller through the SPI interface.

*GRAPHICAL USER INTERFACE (GUI)*

The graphical user interface provides the user with the necessary information to control the robot. It provides visual cues to quickly and easily inform the user of the state of the sensors and allows the user to turn the manual override on or off. The GUI is a part of the aforementioned program running on the user’s laptop. It is written using the curses library that is a part of C++.

After sending commands to the Raspberry Pi, the main program must wait for the Pi to send return data. The return data is sent as four bytes, but only two independent commands, so the laptop must parse the data to extract the two different commands. This data contains information about the state of the sensors that needs to be relayed to the user through the GUI. The GUI is set up so that red, yellow, and green boxes indicate danger, warning, or “all-clear” respectively. The tilt sensor section indicates whether or not the robot is tipping too far to one side, and indicates which direction it is tipping. The distance sensor section indicates how far away obstacles are. Periodically, the Pi will also send updates about the state of the battery. The commands that are sent to the laptop are intentionally kept very simple. The commands only reflect which box should be displayed and what color it should be. Battery updates are also sent only as a color. The microcontroller decides what these commands should be, so the laptop does very little processing once it receives these commands. The colored boxes are sent up as independent “windows” that can be updated independently without updating the entire screen. Turning on the appropriate window is done through a switch case. The state of the manual override is indicated by highlighting either “on” or “off” as appropriate. The entire GUI screen is updated as the final step in each loop of the main program.

*VIDEO TRANSFER*

This subsystem allows the user to see what the camera on the robot is seeing in real-time. We used a program called motion that streams the video feed to a website. The user must go to this website to see the live stream. VLC is a video player, but it can also be used to access video streams from websites. We chose these two programs to handle video transfer because of their ease of use and professional-level functionality. Using VLC to view the video stream is preferable to using a web browser because VLC allows you to record the video stream at any point and automatically saves it as a file so it can be viewed later.

Both VLC and motion are programs that must be downloaded and installed. Motion only needs to be installed on the Raspberry Pi, and VLC only needs to be installed on the user’s computer. Motion is run on the Pi through SSH from the user’s laptop. The laptop and the Pi will need to be on the same network in order for this to work, but this would obviously be the case when working out in the field.

Table Sending Commands

|  |  |
| --- | --- |
| **First Command** |  |
| ‘a’ | Speed up forward until max speed is reached |
| ‘b’ | Slow down forward until no longer moving |
| ‘c’ | Speed up in reverse until max speed is reached |
| ‘d’ | Slow down in reverse until no longer moving |
| ‘e’ | Slow down until not moving |
| ‘f’ | Stop immediately |
| **Second Command** |  |
| ‘A’ | Turn left |
| ‘B’ | Do nothing |
| ‘C’ | Turn right |
| **Third Command** |  |
| ‘w’ | Look up |
| ‘s’ | Look down |
| ‘5’ | Do nothing |
| **Fourth Command** |  |
| ‘d’ | Look right |
| ‘a’ | Look left |
| ‘2’ | Do nothing |

Table . Receiving Commands

|  |  |
| --- | --- |
| **Distance Sensor** |  |
| ‘G’ | All clear |
| ‘Y’ | Caution |
| ‘R’ | Warning |
| **Tilt Sensor** |  |
| ‘X’ | Warning: Too far left |
| ‘C’ | Caution: Tilted left |
| ‘V’ | Flat |
| ‘B’ | Caution: Tilted right |
| ‘N’ | Warning: Too far right |
| **Battery Sensor** |  |
| ‘1’ | Full battery |
| ‘2’ | 50% |
| ‘3’ | 25% |
| ‘4’ | Change Immediately |

**4 System Integration Testing**

4.1 Test Description

Most of the subsystem tests were done using the original kitboard that was provided to each of the senior design groups. First, each subsystem was tested by itself in an attempt to make sure that each was working up to par. Once we had each one working a very difficult part of the project was trying to design a circuit board that incorporated all of these subsystems. After this process we learned many things about the microcontroller that we had not known before. This includes, but is not limited to, that some pins default as JTAG or analog and some pins must be set to open drain if you are using a pull-up resistor. However, eventually we were able to pull all of our subsystems together and get a working robot.

4.2 Show how Testing Demonstrates that the Overall System meets Design Requirements

In order to test the robot that we had built we needed to simulate the environment in which it would be operating. Luckily, Notre Dame has a similar environment that runs all around campus: the steam tunnels. We got access from the Utilities Department on campus to go down into the tunnels and test out robot. The test went really well and we demonstrated that each of our subsystems worked and that together they formed a complete working project.



Figure 15 Exploring Sewer Tunnels



Figure 16 Outside the tunnels



Figure 17 Successful Demonstration!

**5 Users’ Manual/Installation Manual**

5.1 How to Install

To install this project you will need to follow several steps. First, you need to run the code from a Linux operating system. You can find instructions on how to download a Linux system online. You will also need to download the xboxdrv.exe file in order to read data from the xbox controller. Thirdly, you will need some type of video play. The VLC player works best. Lastly, you will need to download the code that is included at the end of this report.

5.2 How to Setup

First, you need to put the battery into its slot and plug it in. Once you hit the switch, the Raspberry Pi should begin to boot up and there will be a light on the microcontroller board indicating power. Once the Pi has booted up you need to SSH into the Pi using its IP address. You may need to find the IR address by hooking it up to a monitor first. After you are connected to the Pi you need to run sudo motion to start the video software. Lastly, you need to run the executable currently at ~/Desktop/copies/casey/exec42414. Then, on the laptop, you need to change directories into the folder with the executable and run sudo xboxdrv | ./executableName PI\_IP\_Adress. This will bring up the GUI and allow you to drive the robot around with the Xbox controller.

5.3 How the user can tell if the product is working

The most obvious signs of malfunction are the absence of lights on the boards. If the power light on either the pi or the microcontroller board is not lit then the robot is not working correctly. You should recheck the connections in the robot and try again. Likewise, if you fail to load the GUI properly then you will have problems controlling the robot and the best procedure is to just start over and try it again. If you are getting a response from the sensors and the motors in the GUI then the robot is working fine. Lastly, and probably most easily to miss, make sure that everything shares a common ground. We had several problems when pieces did not share a common ground.

5.4 How to Troubleshoot

The GUI is set up to display all commands that the laptop is receiving from the Pi. Make sure these commands are correct. This data should be moving quickly during normal operation. To debug, you should slow down the rate that the program is running. To do this, change the value inside the “usleep()” command at the end of the while loop in the main function in the NT\_Laptop\_5\_1\_11a.cpp file. Typical run speed should have this set at 150,000 microseconds. While debugging, 1,500,000 is a good value. If you need help understanding the flow of commands refer to Figure 18.

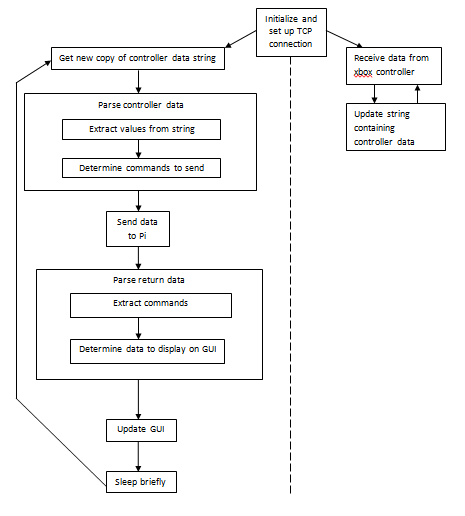


Figure 8 Command Flow

1. **To-Market Design Changes**

After completing the project, there are five main issues that would need to be dealt with before taking the product to market. The first, and most glaring, issue is the ability to turn. This is a simple matter of adding a little code to work with the new motor controller, but we just did not have time to code enough with the new motor controller. Second, different wheels will need to be mounted on the back of the robot. The wheels that are on there now will drive it around a hallway fine, but once it gets into a tunnel they will not have enough clearance to get over any debris. They also will just slip in mud and not provide enough torque. Third, the entire casing needs to be waterproofed better. This could simply mean just putting caulk into all of the cracks and crevices, but as it is the robot would not survive long in a wet, moist environment. Fourth, it would’ve been nice if there was some sort of video processing on the back end to recognize cracks in the pipes, or to create a street-view type interface with the video. This may be too much to ask of a senior design group, but I at least wanted to attempt this. Lastly, we wanted the robot to have an autonomous mode, but we never quite got it there with the coding. Our struggle with the H-bridges did not allow us to branch out and try new things with the code.

**7 Conclusions**

The project went extremely well. We were very ambitious with our goals for the project and we accomplished the majority of them. Our robot could be controlled wirelessly from outside of an underground tunnel and told to move itself, change the direction of the camera, sense obstacles in its path, sense whether it was about to fall over from too much tilt, and check its battery level all from a well-designed board and master-slave interface between the pic32 microcontroller and a raspberry pi. At the end of the project, we produced a robot that could perform the function for which it was designed and we demonstrated this in the Notre Dame steam tunnels. We had to work extremely hard all semester, but this final product was worth the effort.

**8 Appendices**

Poster:

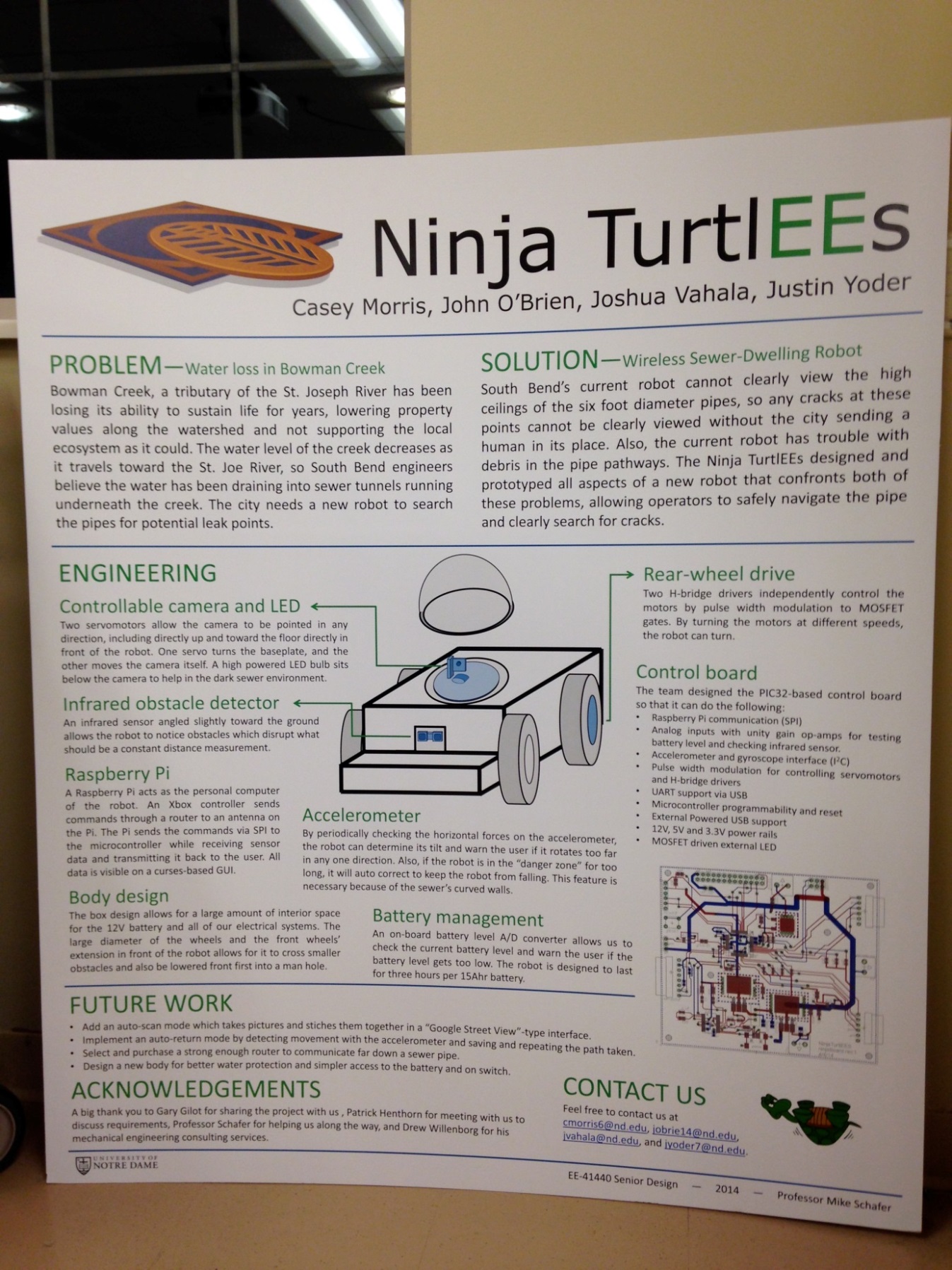


Figure 18. Ninja TurtlEEs poster

Schematic and board pictures:

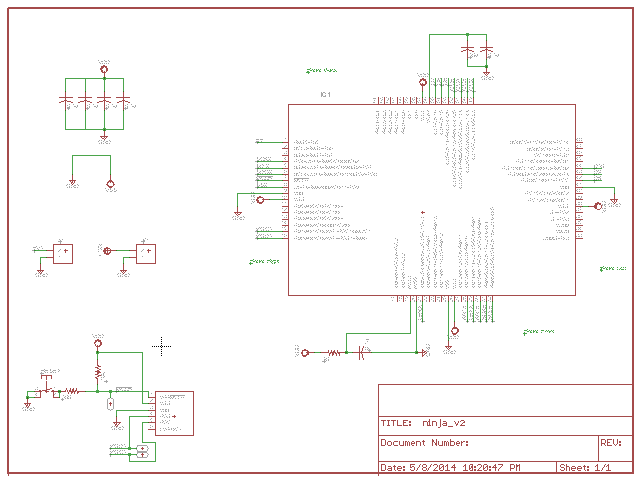
****

Figure 19. Schematic page 1

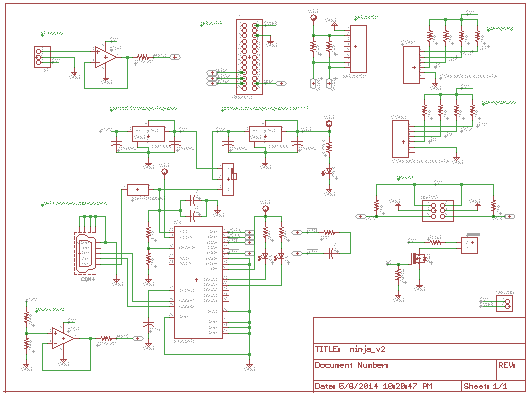
****

Figure 20. Schematic page 2

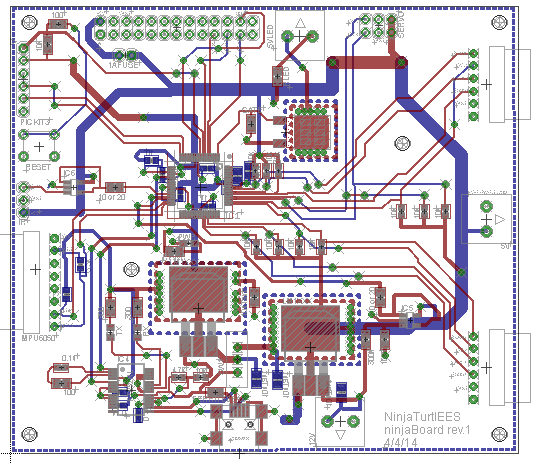
****

Figure 21. Board layout

Data Sheets:

Motor - <http://banebots.com/p/M5-RS550-12>

Gearboxes - <http://banebots.com/pc/P60K-S5/P60K-44-0004>

Wheels - <http://www.fingertechrobotics.com/proddetail.php?prod=ft-sumo-wheel&cat=11>

Servos - <http://www.parallax.com/sites/default/files/downloads/900-00008-Continuous-Rotation-Servo-Documentation-v2.2.pdf>

Battery - <http://www.batterystuff.com/batteries/ub12150-40658.html>

Pi - <http://www.mcmelectronics.com/product/RASPBERRY-PI-RASPBRRY-MODB-512M-/83-14421>

Camera - <http://www.mcmelectronics.com/product/RASPBERRY-PI-2302279-/28-17733>

Wifi Module - <https://www.adafruit.com/products/814>

IR sensor – <http://www.pololu.com/product/1137>

MPU Sensor - <http://www.invensense.com/mems/gyro/documents/PS-MPU-6000A-00v3.4.pdf>

Motor Controller - <http://www.dimensionengineering.com/products/sabertooth2x25>

H-Bridges –

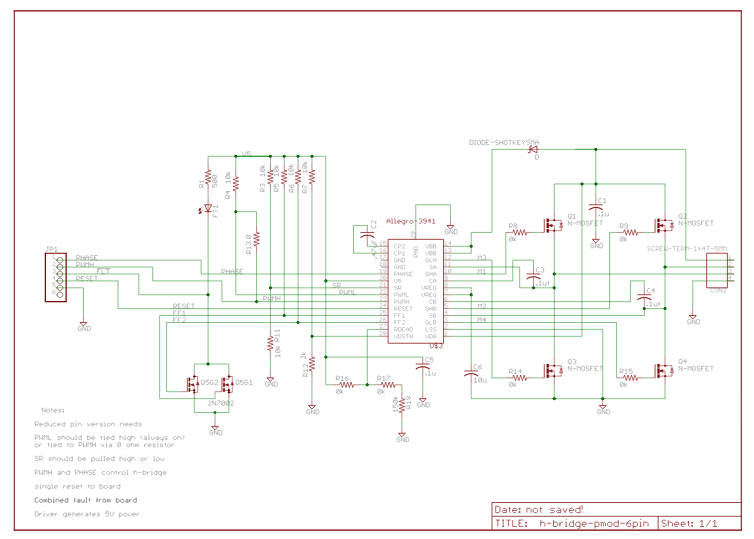


Figure 22. H-Bridge Pull Up schematic

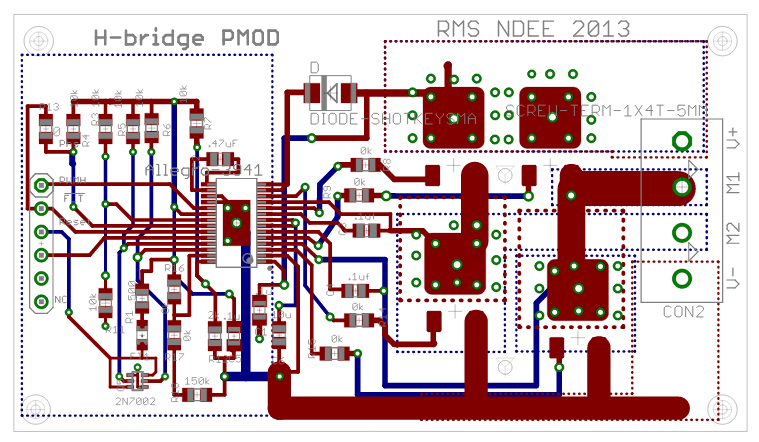


Figure 23. H-bridge board layout

Circuit Board Parts:

Microcontroller Data Sheet - <http://ww1.microchip.com/downloads/en/DeviceDoc/61156H.pdf>

Op-Amps - <http://www.digikey.com/product-detail/en/OPA344NA%2F250/OPA344NACT-ND/362263>

Voltage Regulator LM1117 - <http://www.digikey.com/product-detail/en/LM1117SX-3.3%2FNOPB/LM1117SX-3.3%2FNOPBCT-ND/2469092>

Voltage Regulator LM1084 – 5V - <http://www.digikey.com/product-detail/en/LM1084ISX-5.0%2FNOPB/LM1084ISX-5.0%2FNOPBCT-ND/3526808>

10uf Tantalum – 16V rated - <http://www.digikey.com/product-detail/en/298D106X0016R2T/718-1861-1-ND/2403774>

10uf Tantalum - <http://www.digikey.com/product-detail/en/TACR106K010RTA/478-5224-1-ND/1913294>

All of the other parts came from the common parts from Dr. Schafer.

Code:

/\*

\* File: XboxMotorMain.c

\* Author: Ninja TurtlEEs

\*

\* Created on March 27, 2014, 2:44 PM

\*/

#include <stdio.h>

#include <stdlib.h>

#include <xc.h>

#include "configbits32.h"

#include "kit32r7lib.h"

#include <plib.h>

//need a fault pin

//currently using OC1, D0 pin, for PWM

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

still need to create a separate timer for the servos and motors

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

//#define RESET\_O\_R TRISDbits.TRISD5 //Fault is D6

#define RESET\_O\_R TRISEbits.TRISE5 //Fault is D6

#define PHASE\_O\_R TRISDbits.TRISD7

#define PHASE\_O\_L TRISDbits.TRISD10 //Fault is D9

#define RESET\_O\_L TRISDbits.TRISD8

//#define RESET\_R LATDbits.LATD5

#define RESET\_R LATEbits.LATE5

#define PHASE\_R LATDbits.LATD7//0 for phase is forward, 1 for backward

#define RESET\_L LATDbits.LATD8

#define PHASE\_L LATDbits.LATD10

#define SPI\_RXIF IFS1bits.SPI2RXIF

#define timer\_2\_max 3124//represents duty cycle of 1

#define max\_speed\_drive timer\_2\_max\*.27

#define min\_speed\_drive 600//minimum duty at which weel spins

#define motor\_R OC3RS

#define motor\_L OC2RS

#define motor\_CH OC4RS //continuous motion

#define motor\_CV OC5RS//normal servo

#define grad\_turn 50

#define medium\_turn 125

#define sharp\_turn 300

#define buffLen 1000

#define cautionZone 280

#define dangerZone 370

#define ch\_stationary 233

#define rotate\_speed 3

#define batteryCheckIter 400

#define speedChange\_decel (max\_speed\_drive-min\_speed\_drive)\*.2//5 times will fully accel/decel

#define speedChange\_accel speedChange\_decel/2

#define busy (I2C5CONbits.SEN || I2C5CONbits.PEN || I2C5CONbits.RSEN || I2C5CONbits.RCEN || I2C5CONbits.ACKEN)

//i2c functions

void forceInitI2C5(void);

void init\_I2C5(int baud\_scaler);

void send\_slave\_address(int address7bit, int r\_w); //r = 1, w = 0

void write\_byte(int data\_byte); //can send up to 10 bytes at once

int readSingleByte(int address7bit); //address of slave

int burstRead(int address7bit); //Reads both high and low bits and concatenates them

//CS is the SS3 pin, on D9 I think?

//uses SPI3 Timer2 and OC1

//Need to be sure to use code for all of the motors when have that info

void enable\_MotorController();//OC1 and T2

void init\_drive\_Motors();

void change\_motor\_base(unsigned char c);

void set\_drive\_OCs(unsigned char c);

void change\_servo\_Pos(unsigned char c);

void enable\_ServoControl();

void setData\_6axis();

void setData\_IR();

void enable\_ADC();

void enableSlaveOp();

void compute\_Dist\_Avg();

int returnADCBuffData(int a);

void checkBatteryStatus();

//void serial\_init6(unsigned long rate);

//void enable\_PC\_Com();

int motorDuty\_R;

int motorDuty\_L;

int motorDuty\_CH;

int motorDuty\_CV;

int faultCount;//hopefully won't need this bc of the fix to SPI

int changeDirection;//0 for no, 1 for yes

int cont\_motion\_count;

int vert\_Servo\_Position;

int byteNum;

int numBytes;

int ADCValue; //need this as global only if using interrupts...currently not

int runningAvg = 0;

int runningSum;

int buffer [buffLen];

int batteryCheckCount;

int IS9add = 0x68;

unsigned char axisData= 'X';

unsigned char IRData = 'G';

int turning;

int dutyChange = 50;

int dutyDiff = 100;

unsigned char data [4];

int newData; //0 or 1 depending if new data read from SPI ISR

unsigned char dataToSend[4];

int main(int argc, char\*\* argv) {

//TRISDbits.TRISD5 = 1;//set to input. tied to t

//Open drain capability

ODCDbits.ODCD1 = 1;

ODCDbits.ODCD2 = 1;

ODCDbits.ODCD3 = 1;

ODCDbits.ODCD4 = 1;

ODCDbits.ODCD5 = 1;

ODCDbits.ODCD6 = 1;

ODCDbits.ODCD7 = 1;

ODCDbits.ODCD8 = 1;

ODCDbits.ODCD9 = 1;

ODCDbits.ODCD10 = 1;

RESET\_R = 1;

RESET\_L = 1;

runningAvg = 0;

byteNum = 0;

numBytes = 4;

faultCount = 0;

motorDuty\_R = 0;

dataToSend[0] = 'G';

dataToSend[1] = 'G';

dataToSend[2] = 'V';

dataToSend[3] = 'V';

newData = 0;

TRISE = 0;

LATE = 0x55;

INTConfigureSystem(INT\_SYSTEM\_CONFIG\_MULT\_VECTOR);

INTEnableInterrupts();

cont\_motion\_count = 0;

vert\_Servo\_Position = ch\_stationary;

init\_drive\_Motors();

enable\_MotorController();

enable\_ServoControl();

enable\_ADC();

enableSlaveOp();

init\_I2C5(0x02F);

batteryCheckCount = 390;

while (1){

// OC3RS = 3124;

// RESET\_R = 1;

// RESET\_L = 1;

// break;

if (newData == 1){

setData\_IR();

dataToSend[0] = IRData;

dataToSend[1] = IRData;

setData\_6axis();

dataToSend[2] = axisData;

dataToSend[3] = axisData;

change\_motor\_base\_MC(data[0]);

set\_drive\_OCs(data[1]);

change\_servo\_Pos(data[2]);

change\_servo\_Pos(data[3]);

/\*if (faultCount >=5){

motorDuty\_R = 0;

OC1RS = motorDuty\_R;

RESET\_R = 0;

}\*/

newData = 0;

/\*if (byteNum == numBytes){

byteNum = 0;

}\*/

} //else {//when dont have data to send back, check the IR sensor

//compute\_Dist\_Avg();

//}

}

return (EXIT\_SUCCESS);

}

void \_\_ISR(\_SPI\_2\_VECTOR, ipl7auto)\_\_SPI2Interrupt(void){

unsigned char c; // Read SPI data buffer

c = SPI2BUF;

//printf("\nreceived data is %c\n", c);//print out what we got

SPI2BUF = dataToSend[byteNum];

SPI\_RXIF = 0;

//how to change the unsigned int into

data[byteNum] = c;

LATE-=1;

if (byteNum == 3){

newData = 1;

byteNum = 0;

} else{

byteNum++;

}

batteryCheckCount++;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

// before exiting the service routine.

}

void enableSlaveOp(){ // from data sheet

SPI2CONbits.ON = 0;//turn SPI off while configuring everything

SPI2CONbits.MSTEN = 0;//master mode off

SPI2CONbits.MODE16 = 0;//8 bit SPI

SPI2CONbits.MODE32 = 0;//will be 1 when using 32 bit mode

SPI2CONbits.SSEN = 1;

//Enable interrupts

IEC1bits.SPI2RXIE = 1;

// IEC0bits.SPI3EIE = 1;

//IEC0bits.SPI3TXIE = 1;

IPC7bits.SPI2IP = 7;

IPC7bits.SPI2IS = 1;

//IPC6SET = 0x0000001d; //sets ipl7 i hope!

SPI2CONbits.CKE = 1;//THIS WAS CRUCIAL to get correct data

//sSPI3CONbits.SMP = 1;

SPI2CONbits.ON = 1;

//SPI3CON=0x8000;

//LATE = 0x0F;

SPI\_RXIF = 0;

//printf("about to set SPI3Buf\n");

SPI2BUF = dataToSend[0];

//printf("Currently SPI buf is set and TXIF = '%d'\n", TXIF);

return;

// from now on, the device is ready to receive and transmit data - 8 bits

}

void init\_drive\_Motors(){

RESET\_O\_R = 0; //set to an output RESET - set to global

PHASE\_O\_R = 0; //set to an output PHASE - set global name

PHASE\_R = 0;

RESET\_R = 1;

RESET\_O\_L = 0; //set to an output RESET - set to global

PHASE\_O\_L = 0; //set to an output PHASE - set global name

PHASE\_L = 0;

RESET\_L = 1;

}

void enable\_MotorController(){

int initial = 5350;

//Right Motor

OC3CON = 0x0000; // Turn off the OC1 when performing the setup

OC3R = initial; // Initialize primary Compare register

OC3RS = initial; // Initialize secondary Compare register

OC3CONbits.OCTSEL = 1;

OC3CONbits.OCM = 0b110; // Configure for PWM mode without Fault pin enabled

//Left Motor

OC2CON = 0x0000; // Turn off the OC1 when performing the setup

OC2R = initial; // Initialize primary Compare register

OC2RS = initial; // Initialize secondary Compare register

OC2CONbits.OCTSEL = 1;

OC2CONbits.OCM = 0b110; // Configure for PWM mode without Fault pin enabled

//Will we need a use a different timer with a different period than the servos? if so, will T1 work

PR3 = 8000; // Set period

IFS0bits.T3IF; // Clear the T2 interrupt flag

IEC0bits.T3IE;

//IEC0SET = 0x0004; // Enable T2 interrupt

//IPC1SET = 0x001C; // Set T2 interrupt priority to 7

IPC3bits.T3IP = 0b011;

T3CONSET = 0x8000; // Enable Timer2

T3CONbits.TCKPS = 0b00;//64 prescale

//Set open drain!

OC3CONSET = 0x8000; // Enable OC3

OC2CONSET = 0x8000;

}

/\*

void enable\_MotorController(){

//Right Motor

OC3CON = 0x0000; // Turn off the OC1 when performing the setup

OC3R = motorDuty\_R; // Initialize primary Compare register

OC3RS = motorDuty\_R; // Initialize secondary Compare register

OC3CON = 0x0006; // Configure for PWM mode without Fault pin enabled

//Left Motor

OC2CON = 0x0000; // Turn off the OC1 when performing the setup

OC2R = motorDuty\_R; // Initialize primary Compare register

OC2RS = motorDuty\_R; // Initialize secondary Compare register

OC2CON = 0x0006; // Configure for PWM mode without Fault pin enabled

//Will we need a use a different timer with a different period than the servos? if so, will T1 work

PR2 = 3124; // Set period

IFS0CLR = 0x00000100; // Clear the T2 interrupt flag

IEC0SET = 0x00000100; // Enable T2 interrupt

IPC2SET = 0x0000001C; // Set T2 interrupt priority to 7

T2CONSET = 0x8000; // Enable Timer2

T2CONbits.TCKPS = 0b110;//64 prescale

//Set open drain!

OC3CONSET = 0x8000; // Enable OC3

OC2CONSET = 0x8000;

}

void enable\_MotorController(){

//Right Motor

OC3CON = 0x0000; // Turn off the OC1 when performing the setup

OC3R = motorDuty\_R; // Initialize primary Compare register

OC3RS = motorDuty\_R; // Initialize secondary Compare register

OC3CON = 0x0006; // Configure for PWM mode without Fault pin enabled

//Left Motor

OC2CON = 0x0000; // Turn off the OC1 when performing the setup

OC2R = motorDuty\_R; // Initialize primary Compare register

OC2RS = motorDuty\_R; // Initialize secondary Compare register

OC2CON = 0x0006; // Configure for PWM mode without Fault pin enabled

//Will we need a use a different timer with a different period than the servos? if so, will T1 work

PR1 = 8000; // Set period

IFS0bits.T1IF; // Clear the T2 interrupt flag

IEC0bits.T1IE;

//IEC0SET = 0x0004; // Enable T2 interrupt

//IPC1SET = 0x001C; // Set T2 interrupt priority to 7

IPC1bits.T1IP = 0b011;

T1CONSET = 0x8000; // Enable Timer2

T1CONbits.TCKPS = 0b00;//64 prescale

//Set open drain!

OC3CONSET = 0x8000; // Enable OC3

OC2CONSET = 0x8000;

}

\*/

void \_\_ISR(\_TIMER\_1\_VECTOR, ipl7auto) T1\_IntHandler (void){//Do i need to interrupt with this?

//I dont think it is necessary

/\*if (motor\_CH != ch\_stationary){

if (cont\_motion\_count >5){

motor\_CH = ch\_stationary;

}

cont\_motion\_count++;

} else {

cont\_motion\_count = 0;

}\*/

IFS0bits.T1IF = 0; // Clearing Timer2 interrupt flag

}

void \_\_ISR(\_TIMER\_2\_VECTOR, ipl7auto) T2\_IntHandler (void){//Do i need to interrupt with this?

//I dont think it is necessary

/\*if (motor\_CH != ch\_stationary){

if (cont\_motion\_count >5){

motor\_CH = ch\_stationary;

}

cont\_motion\_count++;

} else {

cont\_motion\_count = 0;

}\*/

IFS0CLR = 0x0100; // Clearing Timer2 interrupt flag

}

void change\_motor\_base(unsigned char c){

//PHASE\_R 0 for forward, 1 for backward

//motor\_Duty\_R motor\_R

if (c =='a'){//speed up

if((PHASE\_R == 0)){

if (motorDuty\_R == 0){

motorDuty\_R = min\_speed\_drive;

}

motorDuty\_R+=speedChange\_accel;

if (motorDuty\_R > max\_speed\_drive){

motorDuty\_R = max\_speed\_drive;

}

} else if ((PHASE\_R == 1)&&(motorDuty\_R >= min\_speed\_drive)){//if change from backwards to forwards, slow down to zero then change direction

motorDuty\_R -= speedChange\_decel;

if (motorDuty\_R <= min\_speed\_drive){

motorDuty\_R = min\_speed\_drive;

RESET\_R = 0;

RESET\_L = 0;

PHASE\_R = 0;

PHASE\_L = 0;

motor\_R = motorDuty\_R;

motor\_L = motorDuty\_R;

RESET\_R = 1;

RESET\_L = 1;

}

}

} else if (c == 'b'){//slow down

if((PHASE\_R == 0)){

motorDuty\_R-=speedChange\_accel\*3;

if (motorDuty\_R <min\_speed\_drive){

motorDuty\_R = 0;

}

} else {

//what to do? is this possible?

}

} else if (c == 'c'){//reverse speed up

if((PHASE\_R == 1)){

if (motorDuty\_R == 0){

motorDuty\_R = min\_speed\_drive;

}

motorDuty\_R+=speedChange\_accel;

if (motorDuty\_R > max\_speed\_drive){

motorDuty\_R = max\_speed\_drive;

}

} else if ((PHASE\_R == 0)){

motorDuty\_R -= speedChange\_accel;

if (motorDuty\_R < min\_speed\_drive){

motorDuty\_R = min\_speed\_drive;

RESET\_R = 0;

RESET\_L = 0;

PHASE\_R = 1;

PHASE\_L = 1;

motor\_R = motorDuty\_R;

motor\_L = motorDuty\_R;

RESET\_R = 1;

RESET\_L = 1;

}

}

} else if (c == 'd'){//reverse slow down

if((PHASE\_R == 1)){

motorDuty\_R-=speedChange\_decel\*3;

if (motorDuty\_R <min\_speed\_drive){

motorDuty\_R = 0;

}

} else {

//what to do? is this possible?

}

} else if (c == 'e'){//do nothing

if (motorDuty\_R > min\_speed\_drive){

motorDuty\_R -= speedChange\_decel;

}

if (motorDuty\_R <= min\_speed\_drive){

motorDuty\_R = 0;

PHASE\_R = 0;//have it ready to go forward

PHASE\_L = 0;

}

} else if (c=='f') {

motorDuty\_R = 0;

}else {//didnt recognize data

faultCount++;

}

motorDuty\_L = motorDuty\_R;

//is this necessary? probs not, if problems, edit this.

if (changeDirection == 1){

RESET\_R = 0;

RESET\_L = 0;

PHASE\_R = !PHASE\_R;

PHASE\_L = PHASE\_R;//intentional R

motor\_R = motorDuty\_R; //actually will be mtor\_speed\_r when all logic figured out

motor\_L = motorDuty\_L;

RESET\_R = 1;

RESET\_L = 1;

changeDirection =0;

}else{

motor\_R = motorDuty\_R; //actually will be mtor\_speed\_r when all logic figured out

motor\_L = motorDuty\_L;

changeDirection = 0;

}

//motor\_R = motorDuty\_R; //actually will be mtor\_speed\_r when all logic figured out

//motor\_L = motorDuty\_L;

}

void change\_motor\_base\_MC(unsigned char c){

if(c=='c'){

if (motorDuty\_R >= 5350){

motorDuty\_R += dutyChange;

} else {

motorDuty\_R = 5350;

}

if (motorDuty\_R > 5750){

motorDuty\_R = 5750;

}

} else if (c =='d'){

if ( motorDuty\_R >= 5350){

motorDuty\_R -= dutyChange;

} else {

motorDuty\_R = 5350;

}

} else if (c =='a'){

if ( motorDuty\_R <= 5350){

motorDuty\_R -= dutyChange;

} else {

motorDuty\_R = 5350;

}

if (motorDuty\_R < 4950){

motorDuty\_R = 4950;

}

} else if (c == 'b'){

if ( motorDuty\_R <= 5350){

motorDuty\_R += dutyChange;

} else {

motorDuty\_R = 5350;

}

} else if (c == 'e'){

/\*if ( motorDuty\_R > 4000){

motorDuty\_R -= dutyChange;

} else if (motorDuty\_R < 4000){

motorDuty\_R+= 4000;

}\*/ motorDuty\_R = 5350;

} else {

motorDuty\_R = 5350;

}

motorDuty\_L = motorDuty\_R;

motor\_R = motorDuty\_R;

if (motorDuty\_R > 5350){

motor\_L = motorDuty\_L+dutyDiff;

} else if (motorDuty\_R < 5350){

motor\_L = motorDuty\_L-dutyDiff;

}

}

//Set up Timer one for either of the types of motors

void enable\_ServoControl(){

OC5CON = 0x0000; // Turn off the OC1 when performing the setup

OC5R = ch\_stationary; // Initialize primary Compare register

motor\_CV = ch\_stationary; // Initialize secondary Compare register

OC5CON = 0x0006; // Configure for PWM mode without Fault pin enabled

OC4CON = 0x0000; // Turn off the OC5 when performing the setup

OC4R = ch\_stationary; // Initialize primary Compare register

motor\_CH = ch\_stationary; // Initialize secondary Compare register

OC4CON = 0x0006; // Configure for PWM mode without Fault pin enabled

// Configure Timer2 interrupt. Note that in PWM mode, the

// corresponding source timer interrupt flag is asserted.

// OC interrupt is not generated in PWM mode.

//USE TIMER 1

PR2 = 3124; // Set period

IFS0CLR = 0x00000100; // Clear the T2 interrupt flag

IEC0SET = 0x00000100; // Enable T2 interrupt

IPC2SET = 0x0000001C; // Set T2 interrupt priority to 7

T2CONSET = 0x8000; // Enable Timer2

T2CONbits.TCKPS = 0b110;

OC5CONSET = 0x8000; // Enable OC1

OC4CONSET = 0x8000; // Enable OC5

}

void change\_servo\_Pos(unsigned char c){

switch (c){

case 's' :{//up 'w'

if (vert\_Servo\_Position <322 ){//324

vert\_Servo\_Position+=11;

motor\_CV = vert\_Servo\_Position;

}

motor\_CH = ch\_stationary;

break;

}

case 'w' :{//down 's'

if (vert\_Servo\_Position > 201){//201

vert\_Servo\_Position-= 11;

motor\_CV = vert\_Servo\_Position;

}

motor\_CH = ch\_stationary;

break;

}

case 'a':{//left 'a'

motor\_CH = ch\_stationary +rotate\_speed;

cont\_motion\_count = 0;

break;

}

case 'd':{//right 'd'

motor\_CH = ch\_stationary -rotate\_speed;

cont\_motion\_count = 0;

break;

}

case 'r':{//reset to default position

vert\_Servo\_Position = ch\_stationary;

motor\_CV = vert\_Servo\_Position;

break;

}

case '2':{

break;

}

case '5':{

motor\_CH = ch\_stationary;

break;

}

default: {

}

}

}

void set\_drive\_OCs(unsigned char c){

int motor\_speed\_r = motorDuty\_R;

int motor\_speed\_l = motorDuty\_L;

//make sure that motorDuty\_L is set in function above

//then set motor\_R and \_L based on those based on the turn signal

if (c== '@'){

motor\_speed\_r = motorDuty\_R;

motor\_speed\_l = motorDuty\_L;

turning = 0;

//motor\_R = motor\_speed\_r;

//motor\_L = motor\_speed\_l;

}else if ((motorDuty\_R == 5350) || (turning == 1)){

switch (c) {

case 'A':////Turn Right

RESET\_R = 0;

RESET\_L = 0;

PHASE\_R = 1;

PHASE\_L = 0;

motor\_speed\_r = min\_speed\_drive + 2\*speedChange\_accel;

motor\_speed\_l = min\_speed\_drive + 2\*speedChange\_accel;

RESET\_R = 1;

RESET\_L = 1;

motor\_R = 4950;

motor\_L = 5750;

turning = 1;

break;

case 'B'://Turn Left

RESET\_R = 0;

RESET\_L = 0;

PHASE\_R = 0;

PHASE\_L = 1;

motor\_speed\_r = min\_speed\_drive + 2\*speedChange\_accel;

motor\_speed\_l = min\_speed\_drive + 2\*speedChange\_accel;

RESET\_R = 1;

RESET\_L = 1;

motor\_R = 5750;

motor\_L = 4950;

turning = 1;

break;

default:

motor\_speed\_r = motorDuty\_R;

motor\_speed\_l = motorDuty\_L;

turning = 0;

break;

}

//motor\_R = motor\_speed\_r;

//motor\_L = motor\_speed\_l;

} else {

turning = 0;

if (motorDuty\_R > 5350){

motorDuty\_R -= dutyChange ;

}

if (motorDuty\_R <= 5350){

motorDuty\_R = 5350;

PHASE\_R = 0;//have it ready to go forward

PHASE\_L = 0;

}

motorDuty\_L = motorDuty\_R;

motor\_R = motorDuty\_R;

motor\_L = motorDuty\_L;

}

}

void setData\_IR(){

//This function is going to read the ADC buffer and then

//convert that to a distance and decide if it is

//G - green, nothing in path

//Y - something is kind of close

//R - Stop, must override to keep going forward

//

//compute\_Dist\_Avg();

runningAvg = returnADCBuffData(0);//ADC1BUF0;

if (runningAvg < cautionZone){//means in the okay zone

IRData = 'G';

} else if (runningAvg < dangerZone){//in the caution zone

IRData = 'Y';

} else {//DANGER Zone

IRData = 'R';

change\_motor\_base('d');

}

// printf("runningAvg is '%d'\n", runningAvg);

//dataToSend = 'Y';

}

void setData\_6axis(){

//This function reads the data of the tilt sensor

//and determines if it is too far to one side and sets the dataToSend

//V - okay, tilt is fine

//B - right side is a little higher

//N - right side is too high, must override to go that way

//C - left side to a little higher

//X - left side is too high

/\* if (count < 50 ){

dataToSend = 'N';

}else if (count <100){

dataToSend = 'V';

}else if (count < 150){

dataToSend = 'X';

} else {

count = 0;

}

\*/

signed int mpuData[3]; //accX,accY,accZ,temp,gyroX,gryoY,gyroZ

int registerAddr = 0x3B; //start at ACCEL\_XOUT\_H

int i = 0;

int j = 0;

signed int totalY = 0;

//for (j; j<3; j++){

for (i; i<3; i++){

I2C5CONbits.SEN = 1;

while(busy);

send\_slave\_address(IS9add, 0);

write\_byte(registerAddr);

mpuData[i] = burstRead(IS9add);

//printf("num: %d\n",mpuData[i]);

registerAddr += 0x02;

//delay\_ms(1);

}

totalY+= mpuData[1];

//}

signed int avgY = totalY;

if (avgY>57000 && avgY< 60000 ){// Right warning

axisData = 'B';

} else if (avgY>7300 && avgY< 10500 ){//left high warning

axisData = 'C';

} else if (avgY< 57000 & avgY>48000){//Right too high

axisData = 'N';

} else if (avgY>10500 && avgY< 16000 ){//left way too high

axisData = 'X';

} else {//you're fine

axisData = 'V';

}

//put the check battery status thing here

if (batteryCheckCount >= batteryCheckIter){

//checkBatteryStatus();

//checkBatteryStatus();

//checkBatteryStatus();

batteryCheckCount = 0;

}

}

void enable\_ADC() {

AD1PCFGbits.PCFG13 = 0; //set AN13 to input

AD1PCFGbits.PCFG15 = 0; //set AN15 to input

TRISBbits.TRISB13 = 1; //set B13 to input

TRISBbits.TRISB15 = 1;//for the battery

DDPCONbits.JTAGEN = 0;//freaking disable jtag

AD1CHSbits.CH0SA = 0b1101; // using MUXA positive input on AN13 Mux A

//for the battery, do

AD1CON1bits.FORM = 0b000; //int 16bit DOUT = 0000 ... 0000 00dd dddd dddd

AD1CON1bits.SSRC = 0b111; //auto convert after sampling

AD1CON1bits.ASAM = 1; //auto sample and convert

AD1CON2bits.VCFG = 0b000; //VR+ = AVdd, VR- = AVss

AD1CON2bits.CSCNA = 0; //scan mode disabled

AD1CON2bits.SMPI = 0b0000; //interrupt after each sample. Only ADC1BUF0 used.

AD1CON2bits.BUFM = 0; //does not split into two 8-bit buffers

AD1CON2bits.ALTS = 0; //always use MUXA

AD1CON3bits.ADRC = 0; //use PBCLK instead of ADC internal RC clock

//our sensor doesnt change its output fast enough to merit a ton of sampling

AD1CON3bits.SAMC = 0b00001; //go 1 T\_AD of sampling before converting

AD1CON3bits.ADCS = 0b11111111; //PBCLK prescaler...any value fits specs

//IFS1bits.AD1IF = 0; //clear the ADC interrupt

//IPC6bits.AD1IP = 7; //set priority

//IEC1bits.AD1IE = 1; //enable ADC interrupt

AD1CON1bits.ON = 1; //turn it on

}

void compute\_Dist\_Avg(){

/\*

int i = 0;

int ADCValue = returnADCBuffData(0);//only question mark left

runningSum = buffer[(buffLen-1)];

while (i<(buffLen-1)){

buffer[i+1]=buffer[i];

runningSum += buffer[i];

i++;

}

buffer[0]=ADCValue;

runningSum += buffer[0];

runningAvg = (double)runningSum/(double)buffLen;

\*/

//runningAvg = ADC1BUF0;//returnADCBuffData(0);

}

int returnADCBuffData(int a){

if (a ==0) {//read from AN13

AD1CHSbits.CH0SA = 0b1101; // using MUXA positive input on AN13 Mux A

} else if (a ==1){//read from AN15

AD1CHSbits.CH0SA = 0b1111; // using MUXA positive input on AN15 Mux A

} else {

return -1;

}

while(!(AD1CON1 & 0x0001));

return ADC1BUF0;

}

void checkBatteryStatus(){//currently unused

//will need

int battLevel = returnADCBuffData(1);

//1 = Fully Charged > 950

//2 = 50% >930

//3 = 25% >910

//4 = Replace Immediately

//these numbers are not accurate. also, find new letters to send back

if (battLevel <950){//<900 real bad. 925 warning 12V

if (battLevel < 930){

if (battLevel < 910){

axisData = '4';

} else {

axisData = '3';

}

} else{

axisData = '2';

}

} else {

axisData = '1';

}

//care about battery getting low

//int 615

//dangerously low. replace immediately

//int 600

}

//initializes I2C5 with designated baud scaling factor

//baud\_scaler: (0x009 for 400kHz, 0x02F for 100kHz)

void init\_I2C5(int baud\_scaler) {

I2C5CONbits.ON = 0; //keep off during inittialization

TRISFbits.TRISF4 = 0;

LATFbits.LATF4 = 0;

I2C5CONbits.A10M = 0; //7 bit slave address

I2C5CONbits.SIDL = 0; //keep module operation in idle mode

I2C5CONbits.ACKDT = 1; //send a NACK

I2C5BRG = baud\_scaler; //set to 100kHz baud rate

I2C5CONbits.ON = 1; //turn on

while(busy);

//printf("ACKSTAT (0 is ACK): %d\n", I2C5STATbits.ACKSTAT);

I2C5CONbits.SEN = 1;

while(busy);

//printf("ACKSTAT (0 is ACK): %d\n", I2C5STATbits.ACKSTAT);

send\_slave\_address(IS9add, 0);

write\_byte(0x6B);

write\_byte(0b10000000);

I2C5CONbits.PEN = 1;

while(busy);

I2C5CONbits.SEN = 1;

while(busy);

send\_slave\_address(IS9add, 0);

printf("slave address sent\n");

write\_byte(0x75);//who am i register

printf("write to who am i\n");

int who\_am\_i = readSingleByte(IS9add);

printf("who am I: %i\n", who\_am\_i);

I2C1CONbits.PEN = 1;

printf("stopped\n");

while(busy);

//get it out of sleep

send\_slave\_address(IS9add, 0);

write\_byte(0x6B);

write\_byte(0x01);

I2C5CONbits.PEN = 1;

while(busy);

I2C5CONbits.SEN = 1;

while(busy);

//get it out of sleep again

send\_slave\_address(IS9add, 0);

write\_byte(0x6B);

write\_byte(0x01);

I2C5CONbits.PEN = 1;

while(busy);

I2C5CONbits.SEN = 1;

while(busy);

//get it out of sleep again again

send\_slave\_address(IS9add, 0);

write\_byte(0x6B);

write\_byte(0x01);

I2C5CONbits.PEN = 1;

while(busy);

I2C5CONbits.SEN = 1;

while(busy);

return;

}

//send address of your slave with read(1)/write(0)

void send\_slave\_address(int address7bit, int r\_w){

int byteToSend;

byteToSend = (address7bit << 1) + r\_w;

//wait for idle bus, then send byte

while(busy);

//printf("Sending byte: %i\n", byteToSend);

I2C5TRN = byteToSend;

while(I2C5STATbits.TRSTAT);

//wait until send it complete, then return

while(busy);

//printf("R/W bit: %d\n", I2C5STATbits.R\_W);

//printf("ACKSTAT (want '0'): %d\n", I2C5STATbits.ACKSTAT);

return;

}

//writes a data\_byte

void write\_byte(int data\_byte){

//printf("sending data: %i\n", data\_byte);

while(busy);

I2C5TRN=data\_byte;

while(I2C5STATbits.TRSTAT);

while(busy);

//printf("ACKSTAT (want '0'): %d\n", I2C5STATbits.ACKSTAT);

return;

}

int burstRead(int address7bit){

int Word[2];

printf("in burst\n");

I2C5CONbits.ACKDT = 0;

while(busy);

printf("ackdt setA32 B27 \n");

I2C5CONbits.RSEN = 1; //send restart

while(busy);

printf("restart sent \n");

send\_slave\_address(address7bit,1); //read from slave

while(busy);

printf("slave address sent with read\n");

I2C5CONbits.RCEN = 1; //enable receive sequence

while(busy);

printf("receive complete \n");

I2C5CONbits.ACKEN = 1; //enable ACK sequence of 1

while(busy);

printf("NACK sent\n");

I2C5CONbits.ACKDT = 1;

Word[0] = I2C5RCV;

while(busy);

printf("word[0] set \n");

I2C5CONbits.RCEN = 1;

while(busy);

printf("RCEN sent \n");

I2C5CONbits.ACKEN = 1;

while(busy);

printf("ACK sent \n");

Word[1] = I2C5RCV;

while(busy);

printf("word[1] set \n");

I2C5CONbits.PEN = 1; //stop transmission

while(busy);

printf("stop sent\n");

//printf("H: %i L: %i\n", Word[0], Word[1]);

return ((Word[0]<<8)+Word[1]);

}

//sends restart, then reads byte and returns the byte

int readSingleByte(int address7bit){

int byteRead;

I2C5CONbits.ACKDT = 1; //send a NACK when ACKEN is set

while(busy);

//printf("ACKSTAT (want '0'): %d\n", I2C5STATbits.ACKSTAT);

I2C5CONbits.RSEN = 1; //send restart

//printf("restart enabled.\n");

while(busy);

//printf("ACKSTAT (want '0'): %d\n", I2C5STATbits.ACKSTAT);

send\_slave\_address(address7bit, 1); //read from slave address

while(busy);

//printf("ACKSTAT (want '0'): %d\n", I2C5STATbits.ACKSTAT);

I2C5CONbits.RCEN = 1; //enable recive sequence

//printf("receive sequence enabled.\n");

while(busy);

while(!I2C5STATbits.RBF); //wait for Recieve Buffer to be full (safeguard)

byteRead = I2C5RCV; //transfer data from the receive buffer

//printf("byteRead: %i\n", byteRead);

//printf("receive full?: %i\n", I2C5STATbits.RBF);

while(busy);

I2C5CONbits.ACKEN = 1; //initialize the ACK sequence

//printf("ACKEN started.\n");

while(busy);

//printf("ACKSTAT (want '0'): %d\n", I2C5STATbits.ACKSTAT);

I2C5CONbits.PEN = 1; //stop process

while(busy);

return byteRead;

}

void serial\_init6(unsigned long rate) {

U6MODEbits.ON = 1;

U6STAbits.URXEN = 1;

U6STAbits.UTXEN = 1;

U6MODEbits.BRGH = 1;

unsigned long baudRate = (10000000)/(4\*rate) - 1;

U6BRG = baudRate;

}

void change\_motor\_base\_old(unsigned char c){

//PHASE\_R 0 for forward, 1 for backward

//motor\_Duty\_R motor\_R

if (c =='a'){//speed up slowly)

if((PHASE\_R == 0)){

motorDuty\_R+=speedChange\_decel;

if (motorDuty\_R > max\_speed\_drive){

motorDuty\_R = max\_speed\_drive;

}

} else if ((PHASE\_R == 1)&&(motorDuty\_R >= min\_speed\_drive)){//if change from backwards to forwards, slow down to zero then change direction

motorDuty\_R -= speedChange\_accel;

if (motorDuty\_R < min\_speed\_drive){

motorDuty\_R = min\_speed\_drive;

changeDirection = 1;

}

}

motorDuty\_L = motorDuty\_R;

} else if (c == 'b'){//speed up quickly

if((PHASE\_R == 0)){

motorDuty\_R+=speedChange\_accel;

if (motorDuty\_R > max\_speed\_drive){

motorDuty\_R = max\_speed\_drive;

}

} else if ((PHASE\_R == 1)&&(motorDuty\_R >= min\_speed\_drive)){//if change from backwards to forwards, slow down to zero then change direction

motorDuty\_R -= speedChange\_accel;

if (motorDuty\_R < min\_speed\_drive){

motorDuty\_R = min\_speed\_drive;

changeDirection = 1;

}

}

motorDuty\_L = motorDuty\_R;

} else if (c == 'c'){//slow down slowly

if((PHASE\_R == 0)){

motorDuty\_R-=speedChange\_decel\*3;

if (motorDuty\_R <min\_speed\_drive){

motorDuty\_R = 0;//because it is supposed to be stopped

}

} else {

//what to do? is this possible?

}

motorDuty\_L = motorDuty\_R;

} else if (c == 'd'){//slow down quickly

if((PHASE\_R == 0)){

motorDuty\_R-=speedChange\_accel\*3;

if (motorDuty\_R <min\_speed\_drive){

motorDuty\_R = 0;

}

} else {

//what to do? is this possible?

}

motorDuty\_L = motorDuty\_R;

} else if (c == 'e'){//do nothing

} else if (c == 'f'){//reverse speed up

if((PHASE\_R == 1)){

motorDuty\_R+=speedChange\_accel;

if (motorDuty\_R > max\_speed\_drive){

motorDuty\_R = max\_speed\_drive;

}

} else if ((PHASE\_R == 0)&&(motorDuty\_R >= min\_speed\_drive)){

motorDuty\_R -= speedChange\_accel;

if (motorDuty\_R < min\_speed\_drive){

motorDuty\_R = min\_speed\_drive;

changeDirection = 1;

}

}

motorDuty\_L = motorDuty\_R;

} else if (c == 'g'){//reverse slow down

if((PHASE\_R == 1)){

motorDuty\_R-=speedChange\_accel\*3;

if (motorDuty\_R <min\_speed\_drive){

motorDuty\_R = 0;

}

} else {

//what to do? is this possible?

}

motorDuty\_L = motorDuty\_R;

} else {//didnt recognize data

faultCount++;

}

//is this necessary? probs not, if problems, edit this.

if (changeDirection == 1){

RESET\_R = 0;

RESET\_L = 0;

PHASE\_R = !PHASE\_R;

PHASE\_L = !PHASE\_L;

motor\_R = motorDuty\_R; //actually will be mtor\_speed\_r when all logic figured out

motor\_L = motorDuty\_L;

RESET\_R = 1;

RESET\_L = 1;

changeDirection =0;

}else{

motor\_R = motorDuty\_R; //actually will be mtor\_speed\_r when all logic figured out

motor\_L = motorDuty\_L;

}

//motor\_R = motorDuty\_R; //actually will be mtor\_speed\_r when all logic figured out

//motor\_L = motorDuty\_L;

}

/\*

void enable\_PC\_Com(){

serial\_init6(56700);

set\_output\_device(1);

U5STAbits.URXISEL = 0b00;//trigger interrupt when a character is received

IEC2bits.U5RXIE = 1;

IPC12bits.U5IP = 7;

// IPC12bits.U5IS = 7;

}

void serial\_init6(unsigned long rate) {

U5MODEbits.ON = 1;

U5STAbits.URXEN = 1;

U5STAbits.UTXEN = 1;

U5MODEbits.BRGH = 1;

unsigned long baudRate = (10000000)/(4\*rate) - 1;

U5BRG = baudRate;

}

\*

\*

\*

\* void set\_drive\_OCs\_old(unsigned char c){

int motor\_speed\_r = motorDuty\_R;

int motor\_speed\_l = motorDuty\_L;

//make sure that motorDuty\_L is set in function above

//then set motor\_R and \_L based on those based on the turn signal

switch (c) {

case '@' ://go straight

motor\_speed\_r = motorDuty\_R;

motor\_speed\_l = motorDuty\_L;

break;

case 'D':////gradual turn right

motor\_speed\_r = motorDuty\_R-grad\_turn;

motor\_speed\_l = motorDuty\_L+grad\_turn;

break;

case 'E'://medium turn right

motor\_speed\_r = motorDuty\_R-medium\_turn;

motor\_speed\_l = motorDuty\_L+medium\_turn;

break;

case 'F'://sharp turn right

motor\_speed\_r = motorDuty\_R-sharp\_turn;

motor\_speed\_l = motorDuty\_L+sharp\_turn;

break;

case 'C'://gradual turn left

motor\_speed\_r = motorDuty\_R+grad\_turn;

motor\_speed\_l = motorDuty\_L-grad\_turn;

break;

case 'B'://medium turn left

motor\_speed\_r = motorDuty\_R+medium\_turn;

motor\_speed\_l = motorDuty\_L-medium\_turn;

break;

case 'A'://sharp turn left

motor\_speed\_r = motorDuty\_R+sharp\_turn;

motor\_speed\_l = motorDuty\_L-sharp\_turn;

break;

default:

motor\_speed\_r = motorDuty\_R;

motor\_speed\_l = motorDuty\_L;

break;

}

if(motor\_speed\_l > max\_speed\_drive){

motor\_speed\_l = max\_speed\_drive;

}else if (motor\_speed\_l < min\_speed\_drive){

RESET\_L = 0;

PHASE\_L = !PHASE\_L;

(motor\_speed\_l = min\_speed\_drive);

RESET\_L = 1;

}

if(motor\_speed\_r > max\_speed\_drive){

motor\_speed\_r = max\_speed\_drive;

}else if (motor\_speed\_r < min\_speed\_drive){

RESET\_R = 0;

PHASE\_R = !PHASE\_R;

(motor\_speed\_r = min\_speed\_drive);

RESET\_R = 1;

}

motor\_R = motor\_speed\_r;

motor\_L = motor\_speed\_l;

}

\*

\* \*/