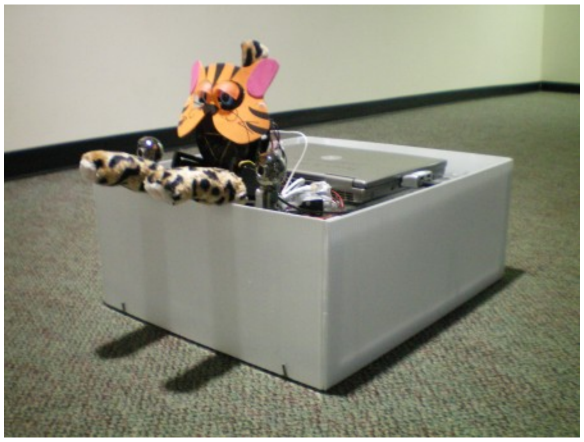
Schrödinger's Cat Robot
Mechanical and Software Improvements

Final Report

Intelligent Robotics ECE 578

Portland State University, Fall 2010

Christopher Forsstrom



Introduction

The Schrödinger's Cat robot is an ongoing project spanning several terms at Portland State University. Incremental changes have been made to the robot each term towards the eventual goal of creating an autonomous robotic agent capable of navigating its indoor environment and interacting with people there as well as other robots in the Robot Theater.

This report describes changes, improvements, and repairs made to the robot during the Fall, 2010, term to improve its operating battery life, allow it to better detect collisions and avoid trouble, improve its movement control for better navigation, and make it easier for future developers to program and experiment with.

Hybrid Differential Drive System

According to Heisenberg's Uncertainty Principle, it is possible to measure an object's position or its momentum, but not both. The Schrödinger's Cat robot was no exception, mostly due to its inefficient and inaccurate drive system.

The four wheel differential drive system from the 2010 winter term was a significant improvement over the two wheel plus casters drive system from the 2009 fall term. However, it was still a less than optimal solution since it required the wheels to be dragged sideways over the floor surface whenever the robot turned. This sideways slippage caused a significant unnecessary drain on the robot's power, and also introduced inaccuracies in the robot's motion. Not all floor surfaces offer the same friction resistance to the wheels sideways motion, and so the same target motion would result in significantly different turning rates and turning angles depending on the floor surface. A 90° turn on the carpet in the in the hall became a 100° turn on the smooth Robotics Lab floor and a 105° turn on the smooth concrete floor of the Fourth Avenue Building lobby. Imbalances in the robot's weight distribution also meant that different wheels have different rates of sideways slippage, causing the actual center of rotation to differ from the robot's center, resulting in some sideways drifting as the robot turns. This makes accurate estimations of the robot's motion and position based on dead reckoning almost impossible

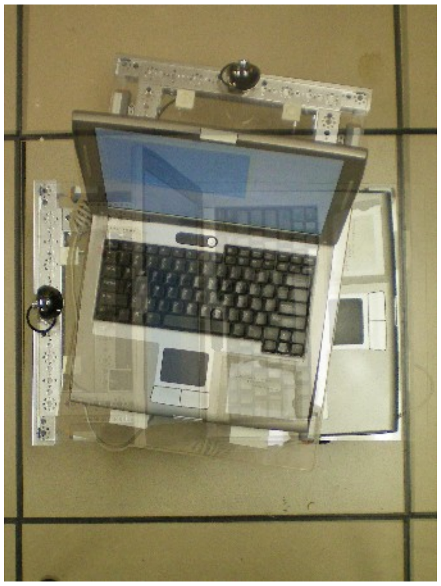


Figure 1: A supposedly 90° turn in the Robotics Lab using the old drive system. Note the actual turn of 100° and the considerable drift toward the top of the image.

The obvious solution to these problems was replacing the four conventional 100mm TETRIX wheels (Pittsco Part #W39055) at the corners with OmniWheels. Rollers along the circumference of an OmniWheel allow them to slide freely sideways, but still provide traction their rolling direction. This means OmniWheels avoid the energy and accuracy losses caused by dragging conventional wheels sideways as the robot turns.

In addition to the four new OmniWheels, two of the original conventional wheels were moved to the center axis of the robot, resulting in a six wheel conventional and OmniWheel hybrid differential drive. The two conventional wheels prevent the entire robot from slipping sideways on the OmniWheel rollers, and also allow the robot to roll over obstacles that the drive chain would otherwise hang on. Since these wheels are positioned on the center turning axis of the robot, there is no sideways component to their motion and so they are not a detriment to the robot's turning accuracy or power consumption. However, including these center wheels also required 16 tooth idler sprockets (Pittsco Part #W39165) to be installed between the front and middle wheels. The idler sprockets alter the drive chain path so that no straight length of the drive chains connect more than two sprockets. Without the idler sprockets, a loose drive chain could sag underneath the center wheel sprocket, causing the center wheel to lose power.

OmniWheel Difficulties

Unfortunately, replacing the conventional wheels in the chain driven differential drive with OmniWheels was not a simple swap. The 80mm diameter OmniWheels (Pittsco Part# W31132) listed on the Pittsco website are much smaller than the 100mm conventional TETRIX Wheels used on the Schrödinger's Cat robot. A significant amount of time was spent designing and building a rather complex suspension and multi-segment chain drive system with multiple idler sprockets to allow the smaller OmniWheels to work with the larger 100mm conventional wheels (the current idler sprockets are the last remnants of that abandoned effort). Replacing the center two 100mm conventional wheels with the smaller 75mm TETRIX wheels (Pittsco Part #W39025) was considered but rejected because they are also not the same size as the 80mm OmniWheels, even though the Pittsco website describes both as having a three inch diameter. Using 75mm wheels would also require replacing all the wheel sprockets for smaller sprockets at great expense.

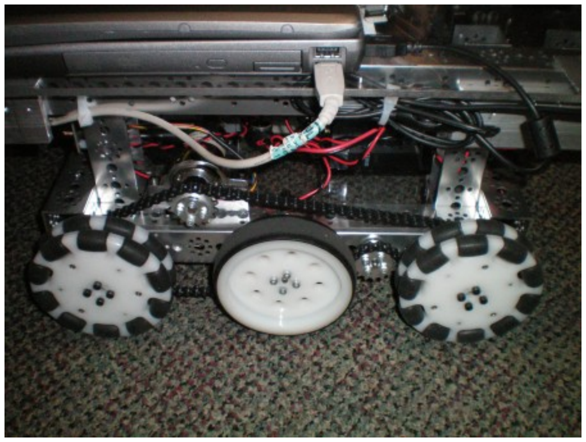


Figure 2:The Hybrid OmniWheel Drive

It was fortunate the new components were ordered by telephone and not through the Pittsco website because the sales representative mentioned that Pittsco also sells 100mm TETRIX OmniWheels (Pittsco Part #W36466) which were not listed on their website (and still aren't at the time of writing this report). Unlike the nominally 75mm OmniWheels, which are actually 80mm in diameter, the 100mm TETRIX OmniWheels actually are 100mm in diameter, which is the same diameter as the 100mm conventional TETRIX Wheels. The 100mm OmniWheels are apparently a special order item at Pittsco and were back-ordered until the latter half of November or later. Even though it meant abandoning and wasting all the effort building the mounting system for differently sized wheels, it was decided to risk the possibility the 100mm OmniWheels might not arrive before the end of the term because using wheels of the same size significantly simplifies the drive design and allowed the re-use of more of the existing components such as the 32 tooth sprockets (Pittsco Part# W39171) already in use.

Those sprockets themselves led to the second problem with replacing the conventional wheels with OmniWheels in the chain drive system.

TETRIX OmniWheels are intended to be used in drive systems where they are placed at the corners of a robot at 45°. Any robot using such a configuration can move omni-directionally along three axis of motion, right-left, forward-backward, and clockwise-widdershins. That freedom of movement comes at a price. The standard OmniWheel configuration is overly complex as it requires four independently controlled motors, one for each OmniWheel. It is also dreadfully inefficient, as each wheel loses half its movement to rolling sideways when the robot travels in a straight line.

The hybrid system used in the Schrödinger's Cat robot keeps most of the freedom of the standard OmniWheel design, reducing the axes of motion by one, but uses a much simpler and more efficient two motor differential drive. When traveling distances, all the power sent to the OmniWheels is directed along their rotation, losing none to the rollers. However, TETRIX Omniwheels were only created to be directly driven with one motor each as in their standard omni-directional configuration. They apparently were never intended to be connected to a gear train or sprocket as in the hybrid design used on the Schrödinger's Cat robot, and so components to make that connection are not for sale and don't exist.

There is no OmniWheel equivalent to the TETRIX gear hub spacers (Pittsco Part #W39090) used to connect a conventional TETRIX wheel to a gear or sprocket. Since the conventional gear hub spacer is designed to account for the thin disk of a conventional wheel, it is far too thick to be used with an OmniWheel either between the OmniWheel and the sprocket or between the two halves that make up a single OmniWheel. Trying to make that connection would position the OmniWheel too far from the sprockets and robot frame, would spread the halves of the OmniWheel too far apart, and would require unnecessarily long #6-32 mounting screws which are unavailable at any local hardware store.

In order to use the TETRIX OmniWheels with TETRIX sprockets, eight custom wheel spacers had to be manufactured. Four were made to separate the OmniWheels from the sprockets so the chain has room to move, and four were made to separate the two halves of each OmniWheel so the OmniWheel rollers have room to move. Please see the appendices for detailed instructions of how these custom spacers were manufactured.

Turning Accuracy Tests

Once installed, the hybrid OmniWheel drive system functioned much better than expected. The new design required recalibration of the motion control software as the robot is now significantly more maneuverable. What was a 90° turn with the old four wheel configuration became a 135° turn with the new hybrid OmniWheel design. That 135° turn is the same 135° turn regardless of the floor surface, whether the floor of the Robotics Lab or the hall carpet or the concrete lobby or the sidewalk out in front of the Fourth Avenue Building. After recalibration of the motion control software to account for the new design, there was no detectable difference in turning angle from one floor surface to another. The rotational center drift appears to have been eliminated as well. A full 180° clockwise rotation followed by a full 180° widdershins rotation resulted in a translation of only 7mm on the carpet in the hall, and that may have been caused by inaccuracies in the motor encoders.

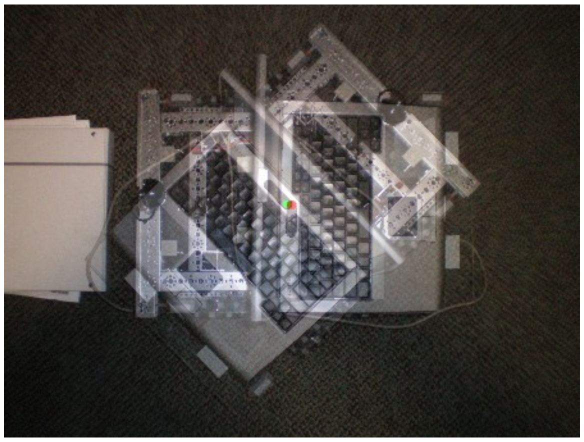


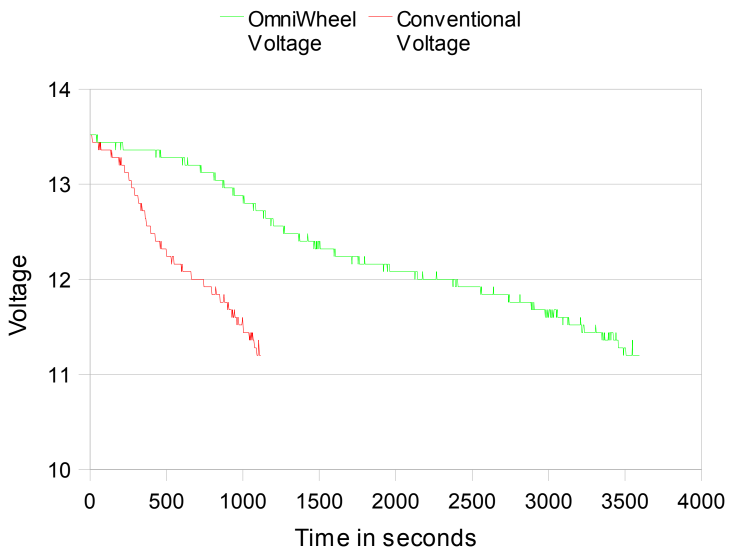
Figure 3: An uncalibrated supposedly 90° turn on the new hybrid OmniWheel drive system in the hall. Note the actual turn is 135° but there is almost no translation. The laptop power button colored red and green is almost over the exact center of rotation.

Battery Torture Tests

The new hybrid OmniWheel drive system resulted in an even greater increase in turning efficiency. While the old four wheel differential drive was still installed on the robot, a battery life torture test was conducted wherein the robot was fully charged to 13.68 volts and then repeatedly and continuously turned a programmed 90°

clockwise followed by 90° widdershins while monitoring the main battery voltage until the battery discharged sufficiently to cause the robot to stop moving. The test was conducted on the carpet in the open area in front of the elevators by the Robotics Lab to maximize the battery drain. The robot completed 274 cycles for a total of approximately 49320° of actual rotation in 1117 seconds (18 minutes 37 seconds) before the main battery could no longer power the robot.

After the new hybrid OmniWheel differential drive system was installed the test was repeated using the same software and location, though with a slightly lower initial battery charge of 13.60 volts. During the new test it became readily apparent that the new drive system turned significantly faster and farther than the old configuration, with full 135° turns instead of the previous 90°. As time passed, and continued to pass, it became apparent that the robot's longevity had increased significantly. This time the robot completed 1304 cycles for a total of 352080° of actual rotation in 3595 seconds (59 minutes 55 seconds). That's more than 300% the previous battery life time and an almost unbelievable 700% increase in rotations for a slightly smaller initial battery charge.



Upon reflection, the 700% improvement in rotations may not be all that surprising. The four corner wheels are positioned almost in a square, which means their sideways slippage is almost the same as their rolling distance during turning. So therefore half their motion during turning is wasted. Since it takes more energy to drag a conventional wheel sideways than to roll it, the majority of energy consumed while turning is used to drag the wheels instead of turning the robot. And that energy loss needs to be multiplied by four; once for each conventional wheel.

No comparison was made between the efficiencies of the old and new configurations when traveling in a straight line. Given the almost identical nature of the two configurations for straight line travel, no significant change in performance during these motions is expected. It is possible that the slightly increased weight from adding the OmniWheels might reduce the longevity of the robot when rolling in a straight line, but the clear benefits to the robot while turning outweigh any such performance reduction. At this point of robot development, under normal maneuvers in the hallways of the Fourth Avenue Building, the battery in the laptop is now the limiting factor to the Schrödinger's Cat robot's operating longevity, not the main battery.

Robot Whiskers

It is hoped that eventually the Schrödinger's Cat robot will be able to navigate its environment solely on the input from its cameras. Until that time, it needs an additional method of sensing its surroundings and avoiding potentially catastrophic navigation errors.

Touch Sensors

During the 2010 winter term two Lego Mindstorms Touch Sensors (Lego Part #9843) were attached to the front of the upper frame of the robot and connected to swinging TETRIX members with cable ties. While these sensors allowed for some impact detection when the robot collided with maze wall, they also had some significant limitations. Their high mounting points meant they were not capable of detecting impacts with objects shorter than the robot's upper platform. Their connection to the TETRIX frame members is flimsy at best and continually required repair and adjustments as impacts knocked the touch sensors loose. They are limited to no more than one contact point per touch sensor, and no more than one touch sensor per port on the NXT Intelligent Brick without using an expensive and undocumented sensor multiplexer that may not be compatible with the libnxt++ software on the laptop. The Schrödinger's Cat robot needs to monitor impacts and floor contact at at least eight locations: the four corners of the upper platform and at each of the four corner OmniWheels. Given the limited number of input ports on the NXT, and the fact that the robot already uses one input port to communicate with the HiTechnic motor controller, it limited the robot to no more than three touch sensors, which is five less than the number needed. At a price of $17.99 each (at the time of writing), not including the necessary connection cables, from shop.lego.com, they are also far more expensive than their limited utility warrants. Even if the NXT had the extra ports to connect to them, the six additional touch sensors plus cables would have cost $117.93 plus shipping and handling. Those funds were needed more for the TETRIX OmniWheels.

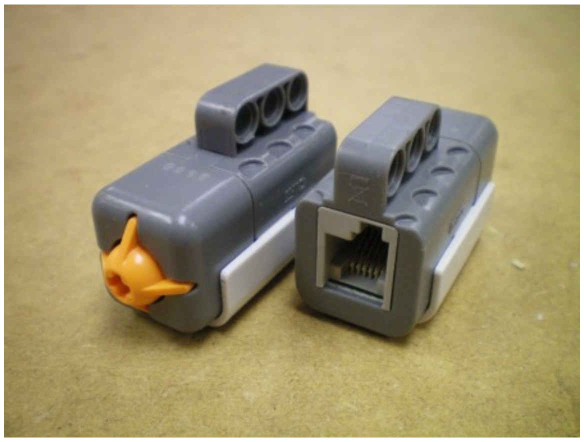


Figure 4: Lego NXT touch sensors. Note the nonstandard off center locking tab.

Sensor Analysis

It was decided to reverse engineer the Lego Mindstorms touch sensors to see if they could be recreated with a circuit using multiple switches wired in parallel so that one of the new touch sensors can detect impacts at multiple locations. An initial attempt to disassemble one of the touch sensors met with abject failure as there appears to be no means to open the case of a touch sensor short of cutting, cracking, or otherwise damaging it. There may be a simple method or tool for opening these sensors, but it is unknown what that method might be. Being unwilling to cause damage to a working touch sensor just to open its case, the focus of the reverse engineering attempt moved to treating the touch sensor as a "black box" and analyze the electrical response of the pins on the Lego communication port as the touch sensor is activated. This was accomplished by removing the retention latch from a standard six wire RJ-12 three line telephone cable, plugging it into a Lego touch sensor, and checking the resistance between each pin of the touch sensor in its activated and unactivated state.

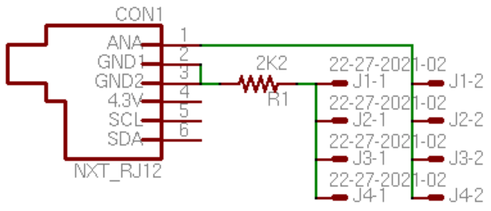


Figure 5:The multi touch switch sensor circuit.

After testing all the pins it appears the Lego touch sensor uses a very simple circuit indeed. The bumper on the front of the sensor activates a spring loaded normally-open switch than connects Pin 1 (the analog-to-digital line according to www.wikipedia.org/wiki/Lego_Mindstorms_NXT) and Pin 2 (ground), through a 2.2kΩ resistor. Pin 2 and Pin 3 appear to be shorted directly to each other, which is not surprising as they are both supposed to be ground signals. The rest of the pins, Pin 4 (+4.3 volt supply), Pin 5 (I²C clock), and Pin 6 (I²C data), appear to be electrically isolated and unused in the touch sensor. It is possible they connect to some simple integrated circuit that allows the NXT controller to identify the touch sensor via I²C when it is plugged in, which would explain the high price for the touch sensor despite being little more than a switch and a resistor. If so, such a potential feature appears to be unused by any of the example software for the NXT, all of which manually configure any port connected to a touch sensor.

Circuit Design

It was a simple matter to design a duplicate circuit to the Lego Mindstorms touch sensor that uses multiple switches connected in parallel to allow a single sensor circuit to detect impacts at multiple locations. The circuit connects four switches in parallel with four two-pin Molex KK-2021 series connectors through a single 2.2kΩ resistor to a standard six conductor RJ12 jack available from Norvac Electronics (Amphenol Part #RJE051660310).

Non-standard NXT Jacks

It needs to be mentioned at this point that the Lego NXT system does not use the standard RJ12 connectors, but rather a unique variant with the locking tab moved to the Pin 1 side of the jack. Other than that, they are identical to a standard RJ12. They are similar to the RJ12-DEC connector standard, except the RJ12-DEC jack standard has its locking tab on the pin six side of the jack. Apparently the Lego NXT system is the sole user of this non-standard RJ12 jack, and so NXT compatible jacks and plugs are not available from any common electronics component vendors. To resolve this incompatibility, a NXT connector cable pack (Lego Part #8527) was purchased from shop.lego.com. Three of the 35cm cables had one of their non-standard NXT jacks removed and replaced with a standard RJ12 jacks. These three RJ12-NXT to standard RJ12 adapter cables allow custom components with standard RJ12 jacks to function with the non-standard NXT RJ12 ports.

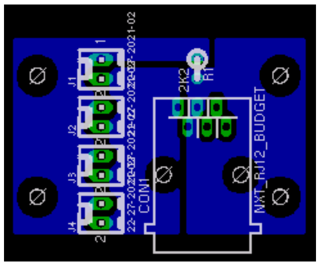


Figure 6:The multiple touch switch board.

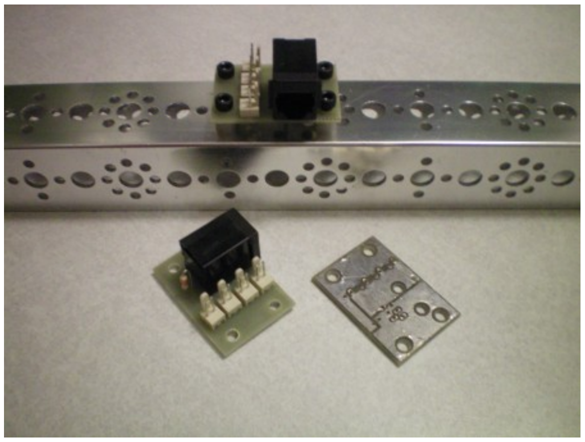


Figure 7:The new multiple touch sensor boards, etched, populated, and mounted.

Once the components were selected, the new touch sensors were designed using CadSoft's EAGLE schematic editor and printed circuit board layout software suite. A new part for the Amphenol RJ12 jack had to be created for EAGLE. At the same time a number of hole drill patterns compatible with the TETRIX building system were created to allow the creation of printed circuit boards that can be mounted directly onto TETRIX frame channels. The custom EAGLE library containing these components is included on the disc with the software for this term so that future robotics class projects and the Portland State University Robotics and Automation Society can also create circuit boards compatible with the NXT and TETRIX systems.

Four printed circuit boards for the new impact sensors were etched from single sided 1 oz. copper clad 1⁄16 inch thick FR-4 fiberglass board using the “toner transfer” method described at www.instructables.com/id/Cheap-and-Easy-Toner-Transfer-for-PCB-Making/ and other places on the internet. The toner transfer method was found to be very accurate and effective and significantly faster and cheaper than the more conventional photo-resist circuit board etching method used in previous terms. The boards were populated and three of them were mounted onto the pillars of the robot frame using #6-32 machine screws and nylon spacers in the the existing holes as they were designed to do. When connected to the NXT intelligent brick they functioned identically to the Lego touch sensor except they register a touch whenever one of four switches each are actuated.

Impact Switches

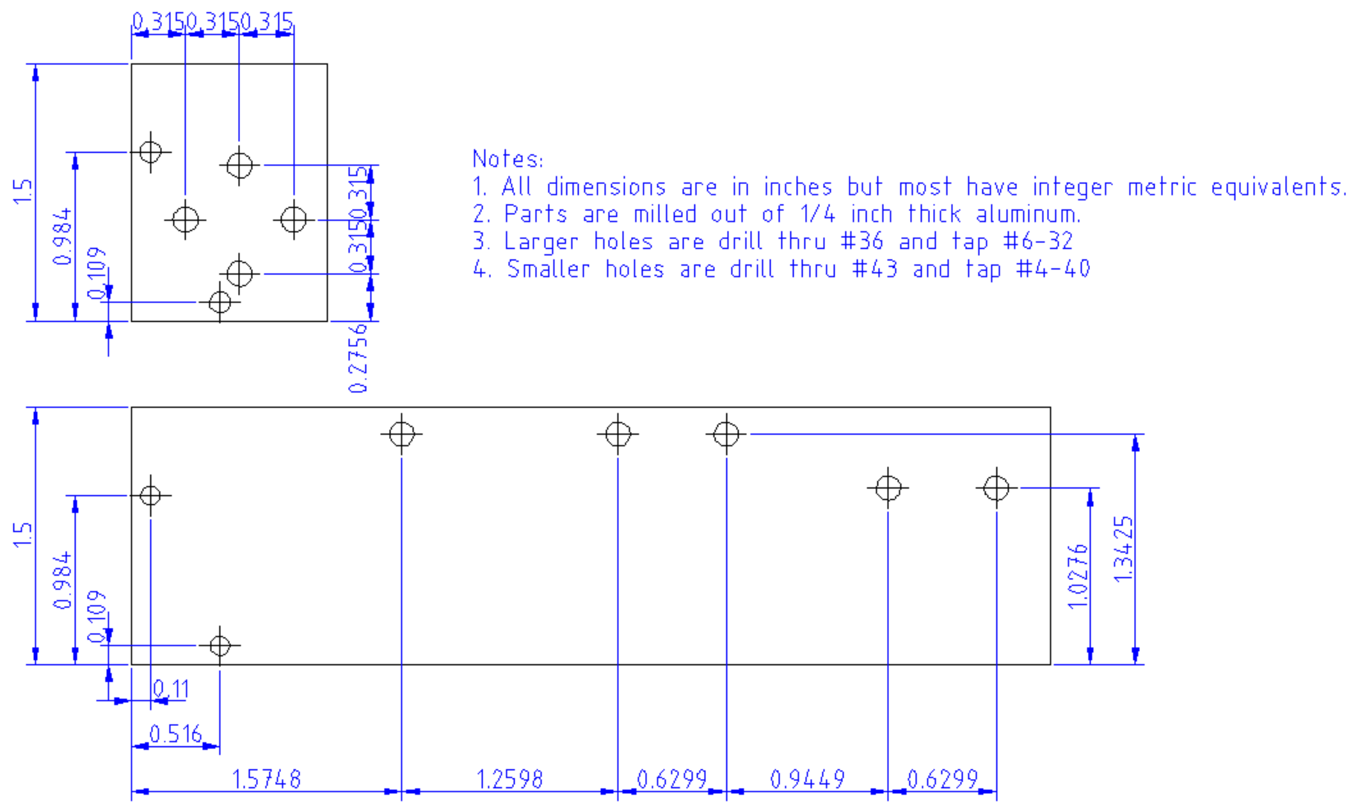
The switches chosen for the new impact sensors are long arm miniature snap action switches (Cherry Part #D42LR1XL). They were selected for their inexpensive price and common availability, but mostly for their long actuator arms which provide a great deal of adjustment flexibility and mounting options. The actuation distance, and thereby the sensitivity of these switches can be readily adjusted by simply bending their actuator arms slightly. One switch is placed at each corner of the frame's upper platform. The two switches at the front are connected using 0.187 inch spade connectors attached to their normally-open and common terminals to a single impact sensor board to detect front collisions. The two switches at the back are likewise connected to a second impact sensor board to detect collisions at the rear.

In addition to the four impact detections switches, four more switches were mounted underneath the bottom of the robot to detect whenever the robot might unintentionally and catastrophically roll over a sudden drop like the edge of a table or the top of a flight of stairs. To maintain contact between the actuator arms of these belly switches and the floor surface, four “whiskers” were constructed from ¼ inch by 1⁄16 inch brass bar stock. The ends of the whiskers were curled into approximately 30mm diameter circles to allow them to glide over the floor surface. Small brass clips were cut from thin ¼ inch by 1⁄64 inch brass stock and silver soldered to the whiskers to attach them to the belly switch actuator arms. The curls in the whiskers are positioned to contact the floor surface approximately 30mm in front of the wheels so that the robot has time to detect a drop and stop the motors before the wheels roll over it. Because the whiskers apply pressure to the belly switches while they are in contact with the floor surface, all four belly switches are connected to the third impact sensor board through their normally-closed terminal.

Switch Mounts

The miniature snap action switches have holes in their housing to mount them using #4-40 machine screws. These holes do not line up with the hole pattern in the TETRIX frame channels. Drilling more holes to match in the TETRIX frame was rejected as being unnecessarily destructive and also useless as the rear switches need to be mounted beyond the TETRIX frame to clear the polycarbonate upper platform.

To attach the switches, two different mounting brackets were designed. One is used to mount the floor sensing switches under the frame and the other to mount the impact switches at the corners of the upper platform. The brackets were cut from 1½ inch by ¼ inch aluminum bar stock. They were drilled and tapped for #6-32 machine screws to connect them to the TETRIX frame and drilled and tapped for #4-40 machine screws to attach the switches. Care had to be taken in the location of the belly switch mounting holes to ensure the switches and mounts clear all the potential obstructions at the bottom of the robot frame such as axles and motor mounting screws. The upper platform impact brackets have several extra #6-32 holes to allow the positioning to be adjusted so the same bracket design can be used to mount a switch flush with the front of the



Notes:
1. All dimensions are in inches but most have integer metric equivalents.
2. Parts are milled out of 1/4 inch thick aluminum.
3. Larger holes are drill thru #36 and tap #6-32
4. Smaller holes are drill thru #43 and tap #4-40

Figure 8:Dimensional drawing of the impact and floor sensing switch mounts.

upper platform or extended beyond the frame at the rear to place the rear impact switches flush with the rear of the polycarbonate upper platform.

Once installed, the switch mounting brackets hold the switches in place quite firmly and are a clear improvement over the flimsy linkages used to connect the NXT touch sensors. They were quite tricky to mount in the completed robot frame though, and in the future it would be preferable to mount the switches early on in the frame construction process when wheels and axles and polycarbonate platforms are not yet present to impede the allen wrench used to tighten the mounting screws.

Schrodinger's Cat's Box

Even firmly attached impact sensing switches are not useful if the impact doesn't happen on their actuators. While the new impact sensor board makes it theoretically possible to literally surround the robot frame with impact switches, such a solution rapidly becomes impractical and expensive. A method of transmitting an impact anywhere on the robot to one of the four impact detection switches was needed. With this in mind, it was decided to surround the robot frame with a loosely suspended box so that any impact anywhere against it would cause it to move relative to the robot and activate one of the impact switches.

An extensive search was made of local shipping and receiving shops for a durable cardboard box suitably sized for surrounding the robot. No such box was found so a custom box was constructed from CoroPlast corrugated polypropylene sheets purchased from Home Depot and assembled using cyanoacrylate adhesive formulated for difficult to join plastics (Loctite Part # 681925). The final box is a rectangular ring 19 inches wide by 22 inches long and 9 inches tall, with reinforcing strips glued to the inside. Unfortunately, the width of the laptop computer and the USB cables plugged into its sides require the box to be much wider than the robot frame. If a smaller computer is ever installed on the robot, a new, much narrower box could be used which would be much easier to navigate through narrow passages like doorways.

The box is freely suspended from the four corners of the upper platform with #6-32 threaded rods bent into hooks 9 inches long that engage into small notches cut into its lower rim. Threaded rod was used so that the height of the box can be adjusted by turning the nuts at the top of the rods. The box can hang low to conceal the robot frame while on smooth floors during Robot Theater, but be adjusted higher for exploring in rough terrain. After initial installation of the box, it became apparent that it was suspended a little too loosely and could be knocked off of the hooks with a sufficiently abrupt impact. Taping the bottom of the hooks to the inner walls of the box solved that problem, but a better attachment method could also be devised.

In operation the box has proven very effective at transmitting even the slightest impact anywhere on its surface to one or more of the impact detection sensors. However, this is not the only purpose for it. There is a significant aesthetic appeal to placing a box around the robot as well since the Schrödinger's Cat thought experiment is all about placing a cat, along with a few other less pleasant items, inside just such a box. With a sufficiently cat like replacement for the current ESRA-2, the Schrödinger's Cat robot could eventually end up appearing very much like a real cat in a box.

Self recharging

In addition to increasing the Schrödinger's Cat robot's battery longevity, one of the goals this term was to modify the robot so that it can be self recharging. It was hoped that a system could be made to recharge both the main motor battery as well as the laptop battery. Unfortunately it was determined that the Dell laptop used to control the robot is very finicky about where it gets its energy from. The laptop computer's recharging power supply delivers 19.5 volts DC, but when an adjustable power supply was set to deliver 19.5 volts DC and plugged into the laptop, the laptop failed to recognize it and continued to run off its still discharging internal battery. It is unknown why the laptop refuses to accept power from anything but its official supply, but it is suspected this was intended by the manufacturer to ensure that any replacement power supplies have to be purchased from them. This recharging limitation could be overcome by mounting the laptop's power supply into the robot, but that would mean the robot would need to be supplied with 120 volt AC wall current. Leaving exposed recharging plates connected to 120 volts AC around the robotics lab (or anywhere for that matter) would be extremely dangerous and irresponsible, so it was decided to limit the current recharging design to only recharging the motor battery. Eventually it is hoped that a different computer will be installed that is capable of being powered from a wider variety of sources so that it can be connected to the recharging system developed for the motor battery.

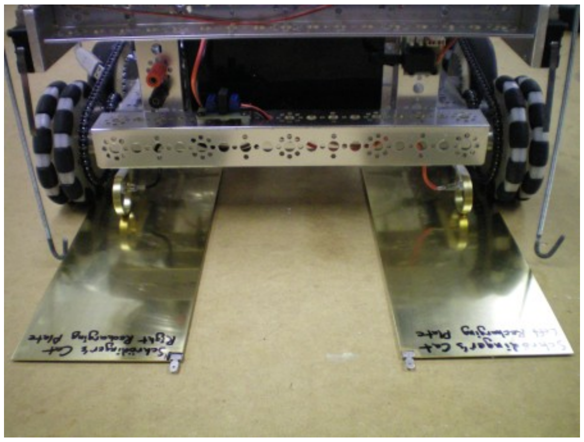


Figure 9:The charging whiskers on the charging plates. Note the power rectifier board mounted to the frame.

Charging Whiskers

To allow the Schrödinger's Cat robot to recharge its motor battery two of the floor sensing whiskers had 0.187 inch spade terminals silver soldered to them. The whiskers are then connected to a bridge rectifier to ensure that the recharging system receives the correct voltage polarity regardless of how the robot is oriented over the recharging plates. The rectifier is part of a small printed circuit board that was made to mount to the TETRIX frame channels just like the impact sensor boards. The power is then regulated down to a maximum of 14 volts and delivered directly to the battery. Since the motor battery is the simple sealed lead acid type similar to those installed in automobiles, a more complicated recharging system like those used for more sophisticated lithium-ion or nickel-metal-hydride batteries is unnecessary. The charging voltage just needs to be kept below 14.5 volts to avoid electrolysis in the battery cells and avoid damaging the HiTechnic motor controller it is directly connected to.

Charging Plates

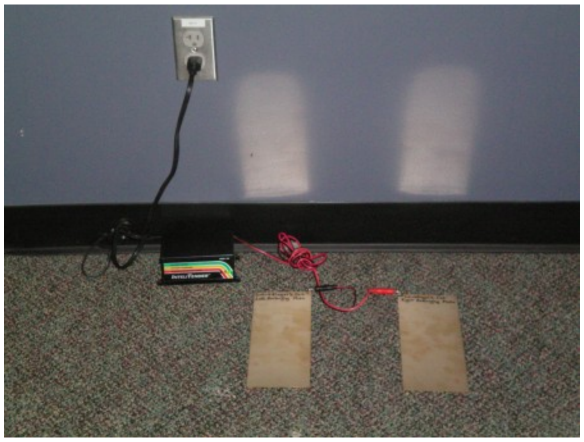


Figure 10:The charging station.

The other half of the self charging system are recharging stations made of two brass plates, 4 inches by 10 inches, with spade terminals soldered onto them to connect them to any DC or AC power supply capable of supplying approximately 15 peak volts at 1 ampere. The charging plates have velcro hook fasteners on their undersides to firmly place them on carpeted floors while still allowing easy removal for cleaning etc. It was decided to keep the recharging stations as simple and inexpensive to manufacture as possible so that many of them can be built and placed at various locations around a building the robot is set to explore. In this way the robot can recharge itself from the nearest of several recharging stations instead of continually having to return to a single station. The Schrödinger's Cat robot monitors the motor battery charge by checking the battery voltage through the HiTechnic motor controller. When the battery charge drops below 12 volts, it can recharge it by driving over the charging plates so that one of its front floor sensing whiskers comes into contact with each plate. The robot knows when it is correctly positioned over the plates when it starts detecting the charging voltage of 14 volts at the battery. After sufficient time has passed the robot drives off the charging plates and rechecks the motor battery voltage to see if additional charging is needed.

The new self charging systems works well enough at keeping the motor battery recharged. However, the new efficient hybrid OmniWheel drive system means the motor battery is no longer the limiting factor in the robot's operational time. The laptop battery now determines when the robot needs human attention to recharge, and so a computer that is also capable of being recharged from the charging station is necessary if the robot is ever going to be freed from depending on human intervention to keep itself running. A more sophisticated charging system that can monitor the current entering or leaving the battery would also be an improvement as it would give a more accurate estimation of the remaining battery life and also allow the robot to monitor the battery charge while recharging without having to roll off the recharging plates.

ESRA-2 Repairs

The Expressive System for Robotic Animation, version 2, or ESRA was in really bad shape at the start of the term. A number of broken wires in its power supply made it mostly immobile, and many of the mechanical linkages were damaged, some irreparably. After connecting the power back to the ESRA, its behavior was found to be erratic and inconsistent.

It was decided that a complete re-build was necessary to get the ESRA working adequately again for the speech generation part of the project. Originally the ESRA was mounted to a wood doll chair with an armature that allowed it to turn and tilt, at least in theory. This armature was poorly designed and assembled, and supported itself solely on the weak plastic servo splines, one of which which had nearly been polished smooth by the ERSA body being yanked about by movement and its own weight. The chair itself was falling apart, and also lifted the ESRA so that it looked much more like a robot cat sitting above a box instead of a robotic cat peeking out of a box. Since face and mouth movement is much more important to speech simulation than tilting and turning it was decided to abandon the chair and armature and re-use its components to repair the face and body of the ESRA.

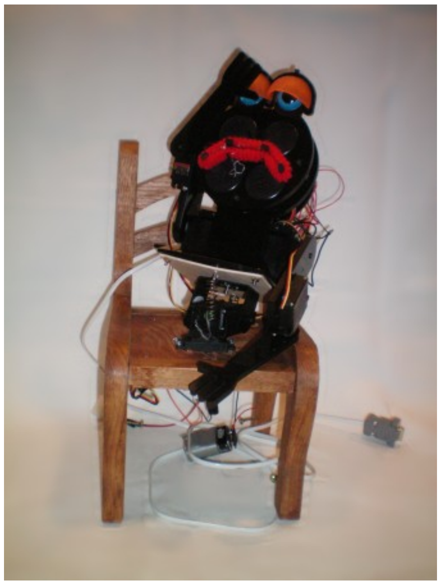


Figure 11:The ESRA-2 in bad need of repair.

As it was disassembled, an number of damaged and mis-installed components were discovered in the ESRA body. The miniSSC2 board controlling the mouth, arm, and eyelid servos had been screwed directly to the back plate of the ESRA without ant standoff spacers. Unfortunately, this was done right over one of the ESRA's machine screws, creating a short between signal lines on the bottom of that miniSSC2 board. This is believed to have caused the intermittent communication failures between the laptop computer and that miniSSC2. After detaching the miniSSC2 and cleaning up its bottom traces, it continued to exhibit communication failures, though not quite so many as when the board was still shorted, so it appears some permanent damage is present on that miniSSC2 board. Fortunately the twist and tilt servos on the abandoned chair had their own miniSSC2 controller which was undamaged, so that controller was connected to control the ESRA face servos, this time with adequate stand-off spacers.

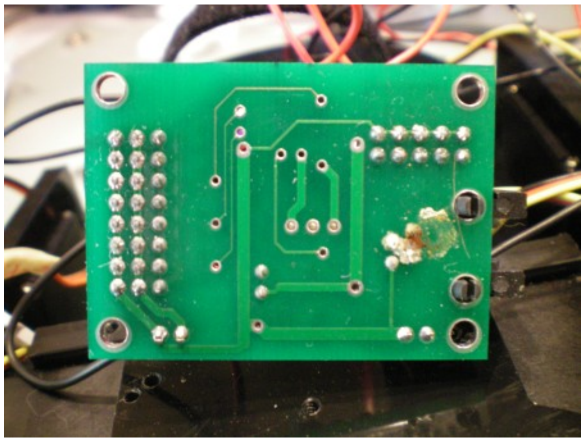


Figure 12:The damaged miniSSC2. This is why standoffs need to be used.

Once the face servos were moving consistently, it became obvious that the lower mouth servo was responding much more slowly and inaccurately than the rest of the servos. Closer inspection revealed that whoever first assembled the ESRA failed to install enough spacing washers between the front and back plates of the ESRA body. This caused the two plates to put too much pressure on the mouth servo armatures, forcing them to have to turn against unnecessary resistance. This extra load apparently took its toll on the lower mouth servo, and some damage appears to have been done to its internal motor or position sensor or both. The upper mouth servo was spared this damage, but only because the excess pressure between the plates at some point broke one of the upper #4-40 nylon screws holding the plates together, relieving pressure on that servo. The ESRA body was completely disassembled, and the lower mouth servo was replaced with the servo from the chair armature that had formerly been used to tilt the ESRA body. Then the body was completely re-assembled using new, adequately sized spacers and new #4-40 machine screws. The mouth armatures were also lightly lubricated with a dry Teflon powder to allow them to move even more freely.

When the ESRA body was reassembled, the battery pack was not replaced inside the back plate. Instead the space for the battery pack was used to hold the bundled servo cables, thereby cleaning up the back of the ESRA and hopefully helping to avoid further damage to the servo cables from accidental tangles. A new four cell AA battery pack (Radio Shack Part #270-383) and 9 volt battery clip (Radio Shack Part #270-324) were connected to the miniSSC2 board through a longer cable and a four position keyed power connector (Radio Shack Part #274-224). Moving the batteries off the ESRA body made it much easier to power since it's no longer necessary to partially disassemble the ESRA to change its batteries. It can also now be powered directly from the power supply board on the robot or from external batteries if it needs to be removed from the robot. The ESRA's weight is also significantly reduced, which will make it easier to design a new tuning and tilting armature in the future.

Easy ESRA

The ESRA is controlled through the miniSSC2 board connected to an RS-232 serial port on a computer. Unfortunately, not everyone is capable of understanding or programming RS-232 serial communications, even within Linux. In order to allow people to concentrate on robot motion an easy to use Application Programming Interface (or API), called Easy ESRA, was written for the ESRA to take care of all the apparently difficult fiddly bits of serial communications with the miniSSC2 controller. Under the Easy ESRA API, communication with the ESRA is initialized with **`ESRA_init()`**. Servos are positioned with **`ESRA_set_upper_lip()`**, **`ESRA_set_right_shoulder()`**, and so on. Communications are terminated with **`ESRA_close()`**. The Easy ESRA code automatically reverses servo positions for inverted servos so that higher values always mean the arm or lip is at a higher position. It also takes care to set limits to the servo positions so that the servos cannot be positioned beyond their range of clear motion. If an error happens, such as attempting to communicate to the ESRA on a serial port that doesn't exist or that the user does not have privileges to access, the Easy ESRA code returns a number of error messages so that the user can diagnose the problem instead of just wondering why it isn't working. It is hoped that the Easy ESRA code continues to be used in future robotics projects as it greatly simplifies the task of controlling an ESRA, or any other device using a miniSSC2, for people with only minimal programming skills.

NXT controller

The Schrödinger's Cat robot drives its motors with a HiTechnic motor controller connected to a Lego NXT Intelligent Brick over an I^{2}C connection. The NXT also connects to the impact sensors. The NXT is itself controlled by the laptop computer over a USB connection via the libnxt++ software library. This makes the NXT crucial to the correct operation of the robot. Unfortunately, the NXT used in the Schrödinger's Cat robot has proven itself to be very unreliable.

At the beginning of this term, it was assumed that the motor control systems and software from last term were still available and functional. This assumption proved to be false. When the OmniWheels finally arrived and it was time to test the old drive system before replacing it, what was thought to be the functional motor control program from last term was used to try to test the motors, only to see the robot fail to move at all.

Inspection of the source code in the folder with the motor control program revealed that it was not, in fact, the final, functional software at all, but some early version of the software that had been heavily re-written, probably unsuccessfully, to make the robot play a game of “chicken” (collision avoidance) with another robot. Or at least that's what it appeared to be supposed to do. So a new program was written to use the almost completely undocumented motor control library to turn the robot left and then right just to confirm the robot was capable of movement. This also failed to cause the robot to move, so the libnxt++ library was completely re-installed and rebuilt on the laptop. Again, the robot failed to move. A HiTechnic motor controller was borrowed temporarily from the Niels Bohr robot and connected, only to have the robot again fail to move. Since the NXT was also not responding to queries about the state of the touch sensors, the problem was determined to lie somewhere in either the laptop software or the NXT. All this time the NXT was recognizing its USB connection to the computer and vice versa and accepting commands, but simply not responding to them. There were no error messages or warnings as to why the NXT Intelligent Brick was unresponsive, it simply sat in the robot and behaved like one of the clay bricks it was named for.

Eventually Alan Cheng graciously lent the NXT from his project for testing. It was plugged into the robot and the robot moved, responding to the laptop as it should. This proved that the problem lay in the NXT Intelligent Brick and not in the laptop software. Before purchasing a replacement NXT for $144.99, an attempt was made to copy the firmware from Alan's NXT onto the Schrödinger's Cat NXT. This was done using the fwflash utility included in the library software from libnxt.googlecode.com. After copying the firmware, the Schrödinger's Cat NXT also began to respond to commands from the laptop again as it was supposed to do. This means the problem was probably caused by corrupted firmware within the NXT Intelligent Brick. It is unknown how or why the firmware was corrupted, it was just fortunate that the part of the firmware that allows uploading new firmware was not itself corrupted. A local copy of Alan's firmware was kept in case the NXT needs to be re-flashed again.

Once the robot was rolling again, tests on the battery life with the old four wheel differential drive could be performed before replacing that drive system. While these tests were ongoing, it soon became apparent that the NXT Intelligent Brick may not have been completely fixed or may have other problems or may simply not be a trustworthy design. Occasionally, the motor on one side of the robot would fail to turn. This behavior was entirely random, and was not exclusive to one motor, happening almost equally to both sides. This behavior was observed last term in the robot but was passed off at the time as an artifact of a low battery charge by the author of the motion control software. As the robot started out each battery test with a fully charged battery and the movement failures seemed to happen independently of the battery charge, this faulty behavior cannot actually be the result of a weak battery charge.

The robot chassis was partially disassembled and the connections between the NXT, HiTechnic motor controller, motors, and encoders were inspected and all found to be sound. Looking around on the internet it appears other projects have also experienced similar problems with their motors not always moving (www.chiefdelphi.com/forums/showthread.php?t=75516).

This untrustworthy behavior of the NXT and HiTechnic motor controller is unacceptable in a fully independent mobile robot. If one side of the robot fails to roll during a turn, it causes the robot's actual turn to be only half of what was intended. The robot will also drift away from its starting position because this half turn will be centered on the non-rolling side, not the robot center. Worse yet, if one side of the robot fails to roll during straight line travel, the robot will also simply turn about the non-rolling side instead of moving in the intended direction. All such positioning errors are cumulative, and make any navigation based on dead reckoning impossible requiring constant correction of the robots perceived position with its sensors. In a worst-case scenario, if one of the motors fails to turn after the robot senses a drop off through its whiskers and attempts to back away, the result could be a catastrophic fall.

Motor Control Software Rewrite

Compensating for these failures required an almost complete rewrite of the motor control software. The old software would simply send a motion instruction to the NXT controller with a **`Motor::setSpeed()`** function and assume that instruction was carried out correctly. Such assumptions have been proven false. The new **`setSpeed()`** subroutine also sends motion instructions to the NXT controller, but before doing so it stores the current encoder positions. After telling the motors to roll, it waits a delay calculated based on the intended speed of the motors to travel 5 mm, and again checks the encoder positions. If the change in encoder position for a motor is less than estimated based on the intended motor speed and the time since the instruction to move was given, the instruction to move is repeated. This is attempted five times in total before the routine gives up and returns with a warning.

A modified version of the battery torture test was repeated using the new motor control software. After 500 cycles of turning 90° left followed by 90° right taking more than 20 minutes, the motors did not show one failure to turn, and the robot had drifted only 75 mm from its original position. This increased positioning reliability comes with a heavy price, however. All the additional communication needed to monitor and correct the motion of the motors requires significantly more bandwidth from the USB connection. This means the robot's motion control can be adversely affected by other high bandwidth devices on the same USB host controller, especially if they transmit large numbers of isochronous data packets. The USB cameras used by the Schrödinger's Cat robot, unfortunately, are just such a device. When the new software controls an NXT connected to the same host controller as one of the cameras, the response time is severely delayed, sometimes taking as long as twelve seconds to respond to instructions from the laptop. Delays of this magnitude are unacceptable, especially when the laptop is monitoring the impact and floor sensors to stop the robot before a collision or fall. The laptop currently used in the Schrödinger's Cat robot only has one internal USB host controller. A second PCMCIA USB host controller was purchased in the 2010 winter term to allow both cameras to operate simultaneously, bringing the total number of host controllers to two. The new, high bandwidth NXT control software means that the robot is once again short by one USB host controller. It can currently receive stereo image data from both cameras or control the motors, but not all three at the same time, meaning it is still incapable of exploring and mapping its environment using stereo imaging.

Future Recommendations

The Schrödinger's Cat robot still has much room for improvement. The hybrid Omniwheel differential drive system has proven to work extremely well, but it could still use a suspension system to allow the robot to travel over more uneven terrain than the flat floors of the Fourth Avenue Building.

The shaft encoders on the TETRIX motors are also quite fragile and unreliable and prone to be knocked loose. A loose shaft encoder makes for inaccurate positioning. A better solution might be to build wheel encoders, or in this case sprocket encoders, out of the already installed sprockets. A disk with alternating light and dark stripes could be printed or milled into the inner surface of the 32 tooth sprockets coupled to the wheels. If reflective optical interrupters (such as the OSRAM SFH 9240) are mounted on the TETRIX frame channel facing the stripes on the disk, they will be able to detect the movement of the stripes, and therefore the movement of the sprocket and the wheel. While such an encoder would not have quite the resolution of the TETRIX shaft encoders, its accuracy would be greater than a damaged or loose shaft encoder and it would take an impact strong enough to destroy the robot itself to damage such an encoder. Another possibility might be to use a sensor like those in laser mice to detect the motion of the sprocket, or simply use such a sensor to detect the robot's movement over the floor itself like a giant laser mouse.

The laptop is severely underpowered, overly large, too finicky about its power supply, and does not have enough USB host controllers. It could be replaced with a newer laptop, but a better solution would be replacing it with a 17cm by 17cm miniITX or 10cm by 7.2cm picoITX motherboard computer. MiniITX motherboards (such as the Asus AT3IONT-I) are available with multi core Intel Atom processors, on board nVidia ION chipset CUDA video processors, 802.11 WiFi networking, and can be mounted in cases (such as the Mini-Box VoomPC-2) designed for installation in automobiles that are powered from 12 volt batteries. That is significant because it would allow the entire robot and all its components to be powered from a single rechargeable lead acid battery that the robot is already capable of keeping recharged by itself. Most miniITX motherboards include an PCI or PCIe expansion slot to allow the addition of more USB host controllers. It is unknown if there are multihost USB controller cards that will fit in an automobile installable case, but a lack of sufficient USB host controllers is a problem that is even more difficult to overcome in a laptop computer.

Figure 13:A VoomPC-2 miniITX computer.

The NXT / HiTechnic motor controller is adequate for simple robotics projects, but has proven insufficient and overly complex for larger, more complex designs controlled by more powerful computers. During a current movement cycle the laptop first calculates the robot movement based on images from its camera. Then it instructs the NXT to instruct the HiTechnic motor controller to enact the movement. Then it requests the NXT to request the encoder position data from the HiTechnic motor controller. If the encoder data is other than expected it has to again instruct the NXT to instruct the HiTechnic to turn the motors. And then again request the encoder position and so on. Once the motors are turning the computer has to repeatedly request the NXT poll the impact and floor sensors. If an impact or drop is detected it has to instruct the NXT to instruct the HiTechnic to stop the motors, then check to confirm the motors have stopped. Every one of these exchanges takes time, and on a heavily loaded USB connection they can take a lot of time. This limits the upper speed of the robot to very slow rates as faster speeds could allow the robot to crash or drop catastrophically before the message to stop reaches the motors.

A better solution for the Schrödinger's Cat robot and other projects would be replacing both the NXT and HiTechnic with a single custom motor controller more suited to interfacing with a computer. An Arduino could be used in such a project, but would require developing around the Arduino's design and firmware. A simple 8 bit microcontroller like an Atmel AVR or Microchip PIC could easily control two motors. The Atmel AVR Attiny2313 is a likely candidate. It has four symmetric Pulse Width Modulation (PWM) signal generators to control the speed of up to four DC motors through H-bridge motor drivers. Ten of its digital input pins can be configured as external interrupts allowing the encoder inputs to be monitored with a simple interrupt service routine (ISR). When the encoder output changes it triggers the appropriate ISR to increment or decrement the encoder's position variable in memory based on the state of the second encoder input. The ISR can also use one of the internal timer / counters to calculate the encoder's current rotational speed. The main program has only to access the position and speed variables for an encoder to automatically know its speed and position and can use that data to drive the motors by changing the duty cycle of the PWM outputs through a PID or Fuzzy Logic or other such control scheme. Another of the external interrupts can be used to monitor the floor and impact sensors, stopping the motors automatically when an impact is detected and then report the impact to the controlling computer. Such a motor controller would not need to wait for the controlling computer to poll a separate device to detect trouble, and so could drive the robot safely at much greater speeds than the current

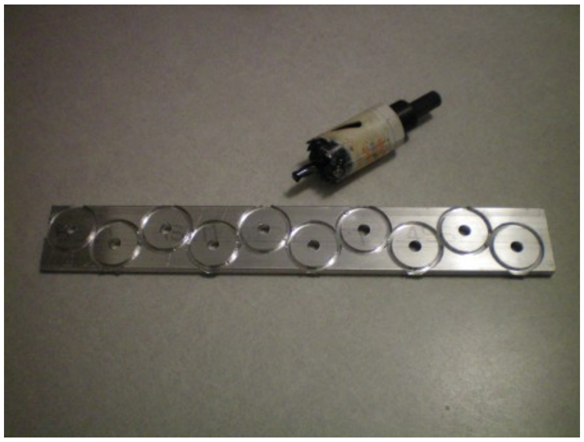
Computer / NXT / HiTechnic system can. The ATtiny2312 can communicate with a controlling computer over RS-232 or USB using emulation in its firmware using one of the many serial converter chips. Such a motor controller would not be very large and could be mounted in the space underneath the robot frame where the HiTechnic motor controller is currently located. This would allow the NXT to be removed from the center of the robot frame, making room for a much larger and higher capacity battery.

Finally, the ESRA *still* needs a decent cat costume.

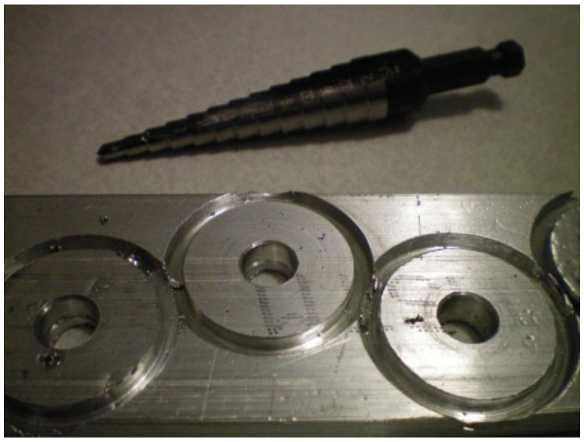
Appendix A: How to manufacture TETRIX compatible hub spacers

…without expensive machine tools.

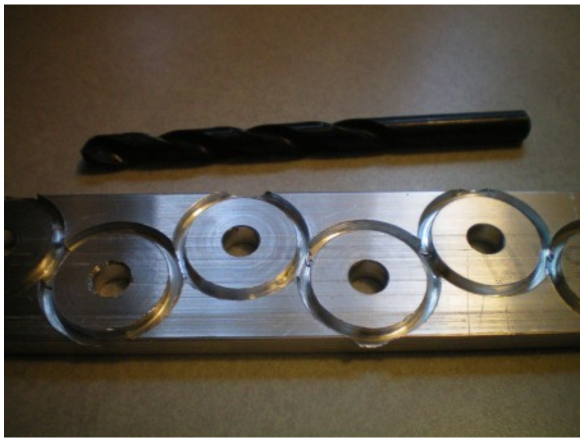
Step 1: Cut partway through a piece of ¼ inch aluminum bar stock with a 1 ⅛ inch hole saw.



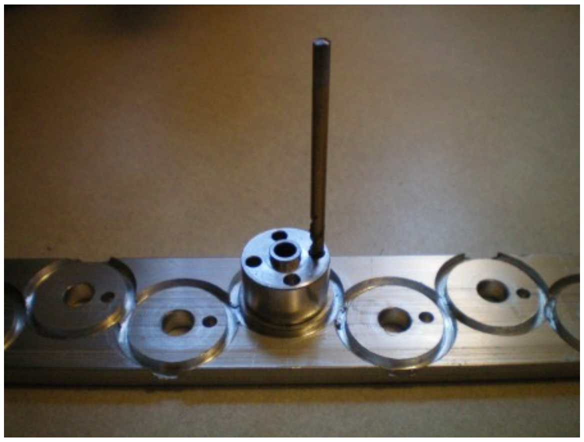
Step 2: Partially bore out the pilot hole to $^5/_{16}$ inch diameter with a step drill.



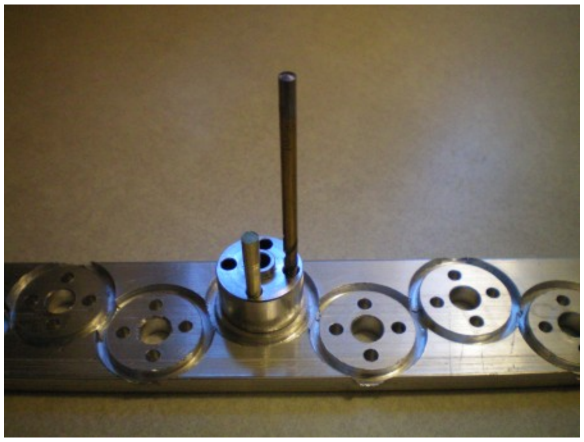
Step 3: Drill through the center hole with an 8 mm drill.



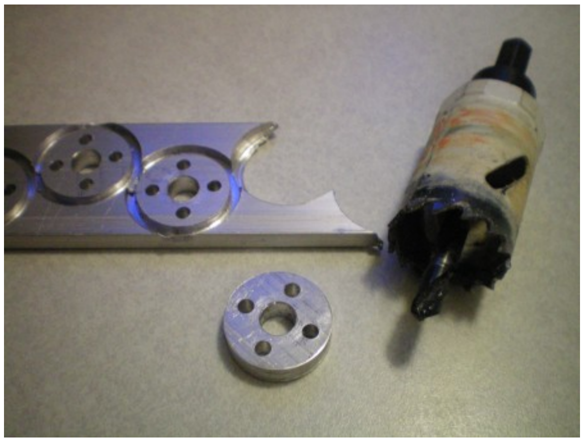
Step 4: Insert a TETRIX wheel hub spacer into the 8mm hole and use it as a drill guide to drill thru one 9/64 inch hole near each 8 mm hole.



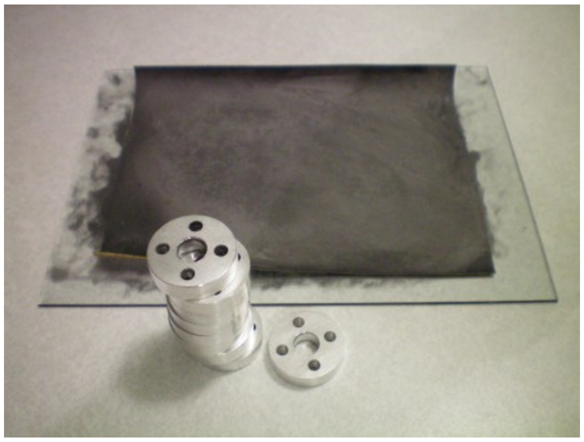
Step 5. Insert a 1/8 inch rod through one of the holes in the TETRIX wheel hub spacer and into the 9/64 inch hole just drilled into the aluminum to prevent the wheel hub spacer from turning. Drill thru three more 9/64 inch holes around the 8 mm center hole using the fixed wheel hub spacer as a drill guide.



Step 6: Cut the hubs free with the 1 1/8 inch hole saw.



Step 7: Clean up the hubs with 1000 grit wet sandpaper.



Appendix B: Source Code

Easy ESRA.c

```
/*
 * ESRA.c
 *
 * This is the Easy ESRA interface API.  In theory it should allow
 * even novice programmers to control the ESRA Human Interaction
 * Device without needing to know all the intricacies of serial port
 * programming under Linux.
 */

#include "ESRA.h"
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <termios.h>
#include <sys/stat.h>
#include <errno.h>

/* Globals */

esra_error esra_errno;
int esra_serial_fd = -1;

int esra_servo_numbers[ESRA_PART_COUNT];
int esra_servo_max[ESRA_PART_COUNT];
int esra_servo_min[ESRA_PART_COUNT];
int esra_servo_inverted[ESRA_PART_COUNT];

/*
 * The ESRA_init() function takes an integer input and treats that as
 * the serial port number the ESRA is supposedly connected to.  It
 * opens the serial port and configures it for communication by later
 * calls to this library.
 *
 * Eventually this initialization routine will also load configuration
 * data from a file (probably under /etc) that contains upper and
 * lower position limits for each of the ESRA components, but for now
 * those values will be hardcoded (or just ignored).
 */

int ESRA_init(const char *port) {

  char *default_port = "S0"; /* This may eventually be get loaded from a
                         config file */
  char *tty_dev_base = "/dev/tty"; /* The ttys all start with this */
  char tty_dev[256]; /* A buffer for the full TTY filepath */
  struct stat tty_status;
  struct termios tty_options;

  /* First lets assign some servo numbers and movement ranges.  For
     now these are hard coded, but eventually they will come from the
     config file. */
```

```
esra_servo_numbers[EYELID] = 0;
esra_servo_numbers[UPPER_LIP] = 1;
esra_servo_numbers[LOWER_LIP] = 2;
esra_servo_numbers[RIGHT_SHOULDER] = 3;
esra_servo_numbers[LEFT_SHOULDER] = 4;
esra_servo_numbers[RIGHT_ELBOW] = 5;
esra_servo_numbers[LEFT_ELBOW] = 6;
esra_servo_numbers[TWIST] = -1;
esra_servo_numbers[TILT] = -1;

esra_servo_max[EYELID] = 255;
esra_servo_max[UPPER_LIP] = 255;
esra_servo_max[LOWER_LIP] = 255;
esra_servo_max[RIGHT_SHOULDER] = 255;
esra_servo_max[LEFT_SHOULDER] = 255;
esra_servo_max[RIGHT_ELBOW] = 255;
esra_servo_max[LEFT_ELBOW] = 255;
esra_servo_max[TWIST] = 127;
esra_servo_max[TILT] = 127;

esra_servo_min[EYELID] = 0;
esra_servo_min[UPPER_LIP] = 0;
esra_servo_min[LOWER_LIP] = 0;
esra_servo_min[RIGHT_SHOULDER] = 0;
esra_servo_min[LEFT_SHOULDER] = 0;
esra_servo_min[RIGHT_ELBOW] = 0;
esra_servo_min[LEFT_ELBOW] = 0;
esra_servo_min[TWIST] = 127;
esra_servo_min[TILT] = 127;

esra_servo_inverted[EYELID] = 1;/* Bigger values should mean more
                          open, not more closed. */
esra_servo_inverted[UPPER_LIP] = 0;
esra_servo_inverted[LOWER_LIP] = 0;
esra_servo_inverted[RIGHT_SHOULDER] = 1; /* Larger positions should
                                mean the arm goes up. */
esra_servo_inverted[LEFT_SHOULDER] = 0;
esra_servo_inverted[RIGHT_ELBOW] = 1; /* Larger positions should
                             mean the arm waves to the
                             right. */
esra_servo_inverted[LEFT_ELBOW] = 1; /* And it's the same for both
                            elbows. */
esra_servo_inverted[TWIST] = 0;
esra_servo_inverted[TILT] = 0;

/* Now lets set up the serial port */

if (!port || *port == '\0')
  snprintf(tty_dev, 255, "%s%s", tty_dev_base, default_port);
else
  snprintf(tty_dev, 255, "%s%s", tty_dev_base, port);

if (stat(tty_dev, &tty_status) < 0) {
  if (errno == ENOENT)
    esra_errno = SERIAL_NO_SUCH_PORT;
  else
    esra_errno = SERIAL_INIT_FAILURE;
  return 0;
  }

  esra_serial_fd = open(tty_dev, O_RDWR | O_NOCTTY | O_NDELAY);

  if (esra_serial_fd < 0) {
    perror(tty_dev);
    esra_errno = SERIAL_INIT_FAILURE;
    return 0;
  }

  fcntl(esra_serial_fd, F_SETFL, 0);

  tcgetattr(esra_serial_fd, &tty_options);   // Get current attribute
  cfsetispeed(&tty_options, ESRA_BAUD); // Set input baud rate
  cfsetospeed(&tty_options, ESRA_BAUD); // Set ouput buad rate

  tty_options.c_cflag &= ~PARENB; // 8N1 configuration
  tty_options.c_cflag &= ~CSTOPB;
  tty_options.c_cflag &= ~CSIZE;
  tty_options.c_cflag |= CS8;

  tcsetattr(esra_serial_fd, TCSANOW, &tty_options); //Set new attribute

  return 1;
}

/*
 * The ESRA_close() function closes the serial port connection and
 * does associated cleanup.  This should only be called after the
 * linking software has finished using the ESRA and is about to exit.
 */

int ESRA_close() {

  /* There will eventually be a lot more to do here. */

  /* Should probably check for errors, but we just want to see the
     thing move right now. */
  close(esra_serial_fd);

  return 1;
}

/*
 * The ESRA_set_pos() function takes an ESRA body part and position as
 * inputs, checks to make sure the position is within range for that
 * body part, and then attempts to send that part to that position.
 * If the position is out of range, it sends the part to the maximum
 * or minimum position, sets an error code, and returns a 0.
 */

int ESRA_set_pos(esra_part part, int position) {

  static unsigned char sync = 255;
  static unsigned char servo = 0;
  static unsigned char pos = 0;

  esra_errno = NO_ERROR;
  if (esra_serial_fd < 0) {
    /* The ESRA is not yet initialized. */
    esra_errno = NOT_INITIALIZED;
    return 0;
  }

  servo = (unsigned char) esra_servo_numbers[part];
  if (servo < 0) {
    esra_errno = SERVO_NOT_ASSIGNED;
    return 0;
  }

  /* Check for inverted servos. */
  if (esra_servo_inverted[part]) position = 255 - position;
  if (position > esra_servo_max[part]) {
    position = esra_servo_max[part];
    esra_errno = POSITION_OUT_OF_RANGE;
  }
  if (position < esra_servo_min[part]) {
    position = esra_servo_min[part];
    esra_errno = POSITION_OUT_OF_RANGE;
  }

  pos = (unsigned char) position;

  /* These three writes should probably be combined into one. */
  write(esra_serial_fd, &sync, 1);
  write(esra_serial_fd, &servo, 1);
  write(esra_serial_fd, &pos, 1);

  if (esra_errno != NO_ERROR) return 0;
  return 1;
}

/*
 * And now the functions for specific body parts to make things simple
 * for novices to code.  Note that these functions only call the
 * ESRA_set_pos() function.  That's it. In c++ these would have benn
 * done with overloading.
 */

inline int ESRA_set_eyelid(int position) {
  return ESRA_set_pos(EYELID, position);
}

inline int ESRA_set_upper_lip(int position) {
   return ESRA_set_pos(UPPER_LIP, position);
}

inline int ESRA_set_lower_lip(int position) {
  return ESRA_set_pos(LOWER_LIP, position);
}


inline int ESRA_set_right_shoulder(int position) {
    return ESRA_set_pos(RIGHT_SHOULDER, position);
}
```

```c
inline int ESRA_set_left_shoulder(int position) {
    return ESRA_set_pos(LEFT_SHOULDER, position);
}

inline int ESRA_set_right_elbow(int position) {
  return ESRA_set_pos(RIGHT_ELBOW, position);
}


inline int ESRA_set_left_elbow(int position) {
    return ESRA_set_pos(LEFT_ELBOW, position);
}


inline int ESRA_set_twist(int position) {
  return ESRA_set_pos(TWIST, position);
}

inline int ESRA_set_tilt(int position) {
  return ESRA_set_pos(TILT, position);
}


#ifdef  DEBUG

/*
 * This is an example main() function to test out the API.  It inits
 * the ESRA, sends it through a few moves, and then exits.
 */

int main() {

  if (!ESRA_init("S0")) {
    printf("ESRA init error.\n");
    if (esra_errno == SERIAL_NO_SUCH_PORT) {
      printf("No such serial port.");
    } else {
      printf("Unable to init serial port for some reason.\n");
    }
    return 0;
  }

  ESRA_set_eyelid(255);
  //ESRA_set_right_shoulder(255);
  //ESRA_set_left_shoulder(255);
  ESRA_set_upper_lip(255);
  ESRA_set_lower_lip(0);
  sleep(1);
  ESRA_set_eyelid(0);
  //ESRA_set_right_shoulder(0);
  //ESRA_set_left_shoulder(0);
  ESRA_set_upper_lip(0);
  ESRA_set_lower_lip(255);
  sleep(1);
  ESRA_set_eyelid(127);
  //ESRA_set_right_shoulder(127);
  //ESRA_set_left_shoulder(127);
  ESRA_set_upper_lip(255);
  //ESRA_set_lower_lip(127);
  sleep(1);
  ESRA_set_upper_lip(0);
  ESRA_set_lower_lip(0);
  //ESRA_set_right_elbow(255);
  //ESRA_set_left_elbow(255);
  sleep(1);
  //ESRA_set_right_elbow(0);
  //ESRA_set_left_elbow(0);
  sleep(1);
  //ESRA_set_right_elbow(127);
  //ESRA_set_left_elbow(127);

  ESRA_close();
}

#endif // DEBUG
```

ESRA.h

```
/*
 * Header file for Easy ESRA interface routines.  In theory these
 * routines should allow even novice programmers to control the ESRA
 * Human Interaction Device without needing to know all the
 * intricacies of serial port programming under Linux.
 */

#ifndef ___ESRA_H___
#define ___ESRA_H___

#define ESRA_VERSION "0.01a"

/*
 * Global defines.
 * These may eventually be pit into a config file that gets read on init.
 */

#define ESRA_BAUD 9600

/* I don't expect the ESRA to ever have more than 16 moving parts. */
#define ESRA_PART_COUNT 16

/*
 * Now for some types and enums etc.
 */

typedef enum _esra_part {
  NO_PART,
  EYELID,
  UPPER_LIP,
  LOWER_LIP,
  RIGHT_SHOULDER,
  LEFT_SHOULDER,
  RIGHT_ELBOW,
  LEFT_ELBOW,
  TWIST,
  TILT
} esra_part;

typedef enum _esra_error {
  NO_ERROR,
  NOT_INITIALIZED,
  SERIAL_INIT_FAILURE,
  SERIAL_COMM_FAILURE,
  SERIAL_NO_SUCH_PORT,
  SERVO_NOT_ASSIGNED,
  POSITION_OUT_OF_RANGE
} esra_error;

/*
 * Global Variables
 */

extern esra_error esra_errno;

/*
 * And now for some public function prototypes.
 */

int ESRA_init(const char *);
int ESRA_close();
int ESRA_set_pos(esra_part, int);
inline int ESRA_set_eyelid(int);
inline int ESRA_set_upper_lip(int);
inline int ESRA_set_lower_lip(int);
inline int ESRA_set_right_shoulder(int);
inline int ESRA_set_left_shoulder(int);
inline int ESRA_set_right_elbow(int);
inline int ESRA_set_left_elbow(int);
inline int ESRA_set_twist(int);
inline int ESRA_set_tilt(int);


#endif //___ESRA_H___
```

schrodie_motor.cpp

```
/*
schrodie_motor.cpp

Loosely based on the original motor.cpp developed for the Winter 2010
term Intelligent Robotics 2.  These routines have been severely
rewritten to take into account communications problems between the NXT
brick and the HiTechnic motor controller.

The original version of this code also randomly switched from handling
the robot position as ticks of the encoders and millimeters of
distance traveled.  This was confusing and required the user to
understand how a given function will be handling the robot position
before calling that function.  I have streamlined that code so that
robot motor positions will only be handled as ticks and the
TICKS_PER_MM function will be used to convert encoder ticks to real
distances.
```

```
*/

#include "schrodie_motor.h"

#include <stdio.h>

#define SETINPUTMODE   0x05
#define LOWSPEED9V     0x0B
#define RAWMODE  0x00
#define LSWRITE  0x0F
#define LSREAD   0x10

#define BATT_V   0x54

#define M1_MODE  0x44
#define M1_PWR   0x45
#define M1_ENC   0x4C
#define M1_TRG   0x40
#define M2_MODE  0x47
#define M2_PWR   0x46
#define M2_ENC   0x50
#define M2_TRG   0x48

#define MODE_MSK 0b11111100 // mask for the mode select bits
#define MODE_PWM 0b00000000
#define MODE_SPD 0b00000001
#define MODE_SRV 0b00000010
#define MODE_REV 0b00001000
#define MODE_RST 0b00000011
#define MODE_BSY 0b10000000
#define MODE_NTO 0b00000000 // no disable timeout
//0b00010000     // disable timeout

using namespace std;

int HTPort;

/* The waitMS() function is probably not necessary since we already
   have the usleep system command as part of the standard c library.
   I'm leaving it for legacy reasons. */

void Motor::waitMS (int ms) {
      clock_t endwait;
      endwait = clock() + ms * CLOCKS_PER_SEC/1000 ;
//cout << "\t" << (double)clock()/CLOCKS_PER_SEC << endl;
      while (clock() < endwait) {}
//cout << "\t" << (double)clock()/CLOCKS_PER_SEC << endl;
}


/* The setup() function tels the NXT brick to treat the device
   connected to the given port as a motor controller.  It also sets up
   the motor controller to take commands and execute them, and tells
   Motor 1 that it needs to rotate backwards because it is on the
   right so that positive rotations roll the robot forward. */

void Motor::setup(int port) {
  HTPort = port;

  // set motor port to I2C
  ViUInt8 directCommandBuffer[] = { SETINPUTMODE,
                                 HTPort,
                                 LOWSPEED9V,
                                 RAWMODE };
  Comm::SendDirectCommand( false,
                    reinterpret_cast< ViByte* >(directCommandBuffer ),
                    sizeof( directCommandBuffer ), NULL, 0);

  // set power to 0, constant speed, reset encoder, reverse motor 1
  i2cWrite(M1_MODE, MODE_NTO | MODE_REV);
  i2cWrite(M2_MODE, MODE_NTO);
  resetPosition();
  setModeSpeed();
  setSpeed(0,0);
}

/* The resetPosition() function zeros out the encoder positions.  This
   function is very important because the encoder counts are limited
   to 16bits, which is not enough with the sprocket ratios on the
   robot to count a full 360 degree turn.  This function should be
   called before attempting to servo the robot.  If it is not, the
   robot will only turn the difference between te given angle and what
   it believs its current angle is.  Note that setting the mode in any
   way for motor 1 resets MODE_REV for some unknown reason, so it
   needs to be reminded it is running in reverse. */

void Motor::resetPosition() {

  i2cWrite(M1_MODE,
         (i2cRead(M1_MODE) & MODE_MSK)
         | MODE_RST
         | MODE_NTO
         | MODE_REV); // MODE_REV reminder
  i2cWrite(M2_MODE,
         (i2cRead(M2_MODE) & MODE_MSK)
         | MODE_RST
         | MODE_NTO);
}

/* The getPosition() function returns the current position of the
   encoders in a position structure.  This function used to be less
   important, but will be more heavily used in this revision of the
   code as it will be checked after telling the motors to move to
   ensure that the command wasn't lost and the motors are actually
   moving.  The NXT and HiTechnic motor controller cannot be trusted
   to always turn a command to an actuation, so they will need to be
   monitored by the software to ensure they respond correctly. */

position Motor::getPosition() {
  position ticks;
  ticks.left = Motor::i2cRead(M1_ENC,4);
  ticks.right = Motor::i2cRead(M2_ENC,4);
  return ticks;
}
```

```
/* The setModeSpeed() function puts the motor controller into constant
   speed mode.  Under constant speed mode, the encoders are used
   solely to control the wheel speeds, and not used for positioning at
   all.  This mode will be used for travelling long distances over
   long time frames where the laptop can montor the robot's position
   using the sensors like the cameras and also possibly by monitoring
   the encoders and resetin them as necessary.  It can allow the robot
   to travel great distances inaccurately. */

void Motor::setModeSpeed() {
  i2cWrite(M1_MODE,
         (i2cRead(M1_MODE) & MODE_MSK)
         | MODE_SPD
         | MODE_NTO
         | MODE_REV);  // Remind motor 1 it runs in reverse.
  i2cWrite(M2_MODE,
         (i2cRead(M2_MODE) & MODE_MSK)
         | MODE_SPD
         | MODE_NTO);
}


/* The setModeServo() function puts the motor controller into servoing
   mode where the encoders are used for positioning instead of speed
   control.  It is much more accurate than constant speed mode but due
   to the limits on the encoder counts cannot be used to travel great
   distances or turn at large angles. */

void Motor::setModeServo() {
  i2cWrite(M1_MODE,
         (i2cRead(M1_MODE) & MODE_MSK)
         | MODE_SRV
         | MODE_NTO
         | MODE_REV);  // Motor 1 is still reversed.
  i2cWrite(M2_MODE,
         (i2cRead(M2_MODE) & MODE_MSK)
         | MODE_SRV
         | MODE_NTO);
}

/* The setSpeed() function if the big important one.  It sets the
   speed of both motors, subject to an upper speed limit set by
   TOP_SPEED which is defined in schrodie_motor.h and sends them on
   their way.  At least in theory.

   If one of the speeds given is greater than the limit set by
   TOP_SPEED, the greater of the two speeds is reduced to TOP_SPEED by
   the speed cap factor capFactor, and the other speed is reduced so
   that the ratios between the two speeds remains the same.

   Before actually starting the motors, the function gets the values
   for the encoder positions.  Then, after setting the speeds of both
   motors, the subroutine waits for a specified time delay and then
   re-checks the encoder values.  If the encoders have not moved
   significantly for one motor, the function resends the new motor
   speed, waits the delay, and then checks the encoder position
   again. */
```

```
void Motor::setSpeed(double spdLeft, double spdRight) {
  /* Speeds are in mm/sec (at least theoretically). */

  /* Positions for checking to make sure the motors are actually turning */
  position start, current;
  int attempt;  /* What attempt is this to get things moving?
               Hopefully we will neer need more than on retry. */

  double waitDelay; /* How long in microseconds we wait before
                  checking the encoders. */

  double slowestNonZeroSpeed;

  /* Calculate the speed cap factor if needed. */
  double capFactor = min(1.0,
                   TOP_SPEED / max(
                              abs(spdLeft),
                              abs(spdRight)
                              )
                   );

  /* Adjust the speeds if needed. */
  spdLeft*= capFactor;
  spdRight*= capFactor;

  slowestNonZeroSpeed = min(abs(spdLeft), abs(spdRight));
  if (slowestNonZeroSpeed == 0)
    slowestNonZeroSpeed = max(abs(spdLeft), abs(spdRight));

  /* Just return if both speeds are zero.  Why would this be called
     with two zero speeds? */

  if (slowestNonZeroSpeed == 0) return;

  start = getPosition();

  /* Converts the speed to a power and get things moving (hopefully) */
  i2cWrite(M1_PWR, speed2pwr(spdLeft));
  i2cWrite(M2_PWR, speed2pwr(spdRight));

  /* And now we check to make sure things actually do start moving. */
  for (attempt = 0; attempt <  MOTOR_COMMAND_RETRIES; attempt++) {
    /* Sleep for a given number of micro seconds. */
    waitDelay = 1000000 * MOTOR_CHECK_DISTANCE_MM / slowestNonZeroSpeed;

    /* If the delay is more than a second (we're really slow), do a
       regular sleep for the integer seconds. */
    while (waitDelay > 1000000) {
      waitDelay -= 1000000;
      sleep(1);
    }
    usleep(waitDelay); // And sleep out the remainder.

    current = getPosition();

    if (spdLeft != 0
     && ( abs(current.left - start.left) < MOTOR_ENCODER_CHECK_THRESHOLD))
```

```
      i2cWrite(M1_PWR, speed2pwr(spdLeft)); // Try again left.

    if (spdRight !=0
      && ( abs(current.right - start.right) < MOTOR_ENCODER_CHECK_THRESHOLD))
      i2cWrite(M2_PWR, speed2pwr(spdRight)); // Try again right.
  }
  /* We've made multiple attempts to get the motors moving.  Hopefully
     they actually are. */
}

/* The stop() function stops the motors by setting their speed to
   zero.  This can probably be declared as a pre-processor macro. */

void Motor::stop() {
  setTargetTicks(0,0);
  resetPosition();
  setSpeed(0,0);
}

/* The getTarget() function returns the current target in mm.  It does
   this by calling the getTargetTicks() function and then converting
   from ticks to mm.  This function will most likely only be called
   during debugging. */

position Motor::getTarget() {
  position ticks;
  ticks.left = Motor::i2cRead(M1_TRG,4)/TIC_PER_MM;
  ticks.right = Motor::i2cRead(M2_TRG,4)/TIC_PER_MM;
  return ticks;
}

/* The setTargetTicks() function sets the target position in ticks
   relative to the current position of both motors.  It then gets the
   target position back from the motors to make sure the target
   position was actually set correctly.  Be warned that the encoder
   range limitations mean that this function could cause an encoder
   overflow if the encoder positions are already near the limits.
   Therefore, unless it is being used for small corrections in
   position, it should probably only be called after zeroing the
   positions of both encoders. */

void Motor::setTargetTicks(int trgLeft, int trgRight) {
  position here = getPosition();
  int ticLeft, ticRight;
  int returnedTicLeft, returnedTicRight;
  int attempt;

  ticLeft = trgLeft + here.left;
  ticRight = trgRight + here.right;

  for (attempt = 0; attempt < MOTOR_COMMAND_RETRIES; attempt++) {
    for (int i=0; i<4; i++) { /* Target is a 32-bit value stored in 4
                    sequential bytes. */
      i2cWrite(M1_TRG+i, ticLeft>>(8*(3-i)) & 0xFF );
      i2cWrite(M2_TRG+i, ticRight>>(8*(3-i)) & 0xFF );
    }

    returnedTicLeft = Motor::i2cRead(M1_TRG,4);
    returnedTicRight = Motor::i2cRead(M2_TRG,4);

    if (returnedTicLeft == ticLeft && returnedTicRight == ticRight) break;
  }
}

/* The setTarget() function sets the target position in mm of both
   motors relative to the current position.  It does this by calling
   the setTargetTicks() function with values generated with dist2tic()
   function. */

void Motor::setTarget(double trgLeft, double trgRight) {
  setTargetTicks(dist2tic(trgLeft),dist2tic(trgRight));
}


/* The getVoltage() function returns the current battery voltage in
   volts. This function will let the laptop know when the robot is
   getting hungry.  */

double Motor::getVoltage() {
  return (double) Motor::i2cRead(BATT_V) * .08;
}


/* The drive() function drives the robot forward at the given speed in
   mm/s */
void Motor::drive(double speed) {
  setModeSpeed();
  setSpeed(speed, speed);
}

/* The driveTo() function drives forward to the specified position
   given in mm.  Please be aware that the limits on the encoder range
   places extreme limits on the range this function is able to target,
   especially if the encoder positions are already close to their
   maximum before the function is called.  for maximum range and
   accuracy out of this function, it's probably best to zero out the
   encoder positions before calling it. */

void Motor::driveTo(double dist, double speed) {
  setModeServo();
  setTarget(dist, dist);
  setSpeed(speed, speed);
}

/* The turnAngle() function turns the robot clockwise to the specified
   angle.  In other words, a positive angle will turn the robot to the
   right and a negative angle will turn the robot to the left.  This
   function used to be significantly more complex before the new
   OmniWheel drive system was installed. Now it simply describes a
   radius based on the width of the center wheelbase.  */

void Motor::turnAngle(double angle, double revolutionsPerSecond) {
      angle = -angle; // + = right, - = left

      /* The circumference of the circle traveled by the center two
         conventional wheels. */
      double circumference = 2 * PI * MOTOR_W;

      /* The slipRatio is no longer needed for the power
         calculations because the corner wheels, being OmniWheels,
         can now slide sideways freely.  This code has been left in,
         but commented out, for historical purposes and also in case
         the old wheel architecture is ever re-installed. */

      /* Ratio of power applied to the wheels on the [left,right]
         side to power doing useful work, i.e. not slipping
         sideways. */

      /*
      double slipRatio = fabs( sqrt( pow(MOTOR_W,2) + pow(MOTOR_L,2) )
                        / (MOTOR_W) );
      circumference *= slipRatio;
      */

      /* The speed is determined by the circumference of the circle
         transcribed by the center two wheels and the revolutions
         per second. */

      double speed = circumference * revolutionsPerSecond;

      /* The target, likewise, is determined by the transcribed
         circumference and the intended angle. */
      double target = circumference * angle/360;

      setModeServo();
      setTarget(-target, target);

      /* And GO. Note that there may be some drifting because the
         motors don't always move when told to.  The setSpeed()
         function attempts to correct for this, but if the motors
         don't both fire up on the first go there will still be some
         offset in the movement as one motor gets a head start on
         the other.  there is no way to fix that without fixing the
         phantom communication errors between the NXT and the
         hiTechnic motor controller. */

      setSpeed(speed, speed);

      //cout << "circ" << circumference << endl;
      //cout << "speed" << speed << endl;
}

/* The turn() function turns the robot counterclockwise the specified
   speed.  It does set a target angle or put the motors controller
   into servoing mode, it just turns. */

void Motor::turn(double direction, double revolutionsPerSecond) {

  /* We don't need an angle, just a direction, but just in case
     something other than -1 or 1 is goven, fix it.  */

  direction = direction / abs(direction); // set to +-1

  /* The circumference is just like in the turnAngle() function. */
  double circumference = 2 * PI * MOTOR_W;

  /* And just like the turnAngle() function, we don't need the
     slipRatio calculations any more, but we're still leaving them in,
     but commented out. */


  /* Ratio of power applied to the wheels on the [left,right] side to
     power doing useful work, i.e. not slipping sideways. */

  /*
  double slipRatio = fabs( sqrt( pow(MOTOR_W,2) + pow(MOTOR_L,2) )
                    / (MOTOR_W) );
  circumference *= slipRatio;
  */

  /* Just like in turnAngle(). */
  double speed = circumference * revolutionsPerSecond;
  setModeSpeed();


  /* Again, there may be some drifting as one motor fails to pay
     attention and start spinning for some reason. */
  setSpeed(-direction*speed, direction*speed);
}

/* The busy() function USUALLY returns true if the motors are en-route
   to the target position.  BUT NOT ALWAYS.  For example, the first
   time it's called, it almost always returns a false negative.  DO
   NOT TRUST THIS FUNCTION.  Generally, if the function returns a
   true, it is, in fact, busy.  However, sometimes when it says it
   isn't busy, it actually is.  It's probably a good idea to confirm
   with the encoder positions whenever it this function claims it
   isn't busy. */

int Motor::busy() {
  return (i2cRead(M1_MODE) & MODE_BSY) || (i2cRead(M2_MODE) & MODE_BSY);
}

/* The i2cWrite() function tells te NXT to writes a byte to the
   hiTechnic motor controller via the i2c connection.  This code
   apears to be functional and is directly copied form the original
   motor.cpp code. */

void Motor::i2cWrite(int address, int data) {
  // prevent buffer overruns
  waitMS(10);

  // write 3 bytes, expect 0 back, 0x02 is padding
  ViUInt8 directCommandBuffer[] = {LSWRITE,
                          HTPort,
                          0x03,
                          0x00,
                          0x02,
                          address,
                          data};
  ViUInt8 responseBuffer[] = { 1,1};
  Comm::SendDirectCommand( true,
                    reinterpret_cast< ViByte* >( directCommandBuffer ),
                    sizeof( directCommandBuffer ),
                    reinterpret_cast< ViByte* >( responseBuffer ),
                    sizeof( responseBuffer ));
}

/* The i2cRead() function reads four or less bytes from the given
   address over the NXT's i2c connection to the hiTechnic motor
   controller.  This code has also been kept from the original
   code. */

int Motor::i2cRead(int address, int bytes) {
  // prevent buffer overruns
  waitMS(10);
  // ask for data, wait till it arrives
  int bytesRead = 0;
  do {
    // write three bytes, expect some back, 0x02 is padding
    ViUInt8 directCommandBuffer[] = {LSWRITE,
                              HTPort,
                              0x02,
                              bytes,
                              0x02,
                              address};
    ViUInt8 responseBuffer[] = {1, 1};
    Comm::SendDirectCommand( true,
                      reinterpret_cast< ViByte* >( directCommandBuffer ),
                      sizeof( directCommandBuffer ),
                      reinterpret_cast< ViByte* >( responseBuffer ),
                      sizeof( responseBuffer));
    bytesRead = NXT::Sensor::LSGetStatus(HTPort);
  } while (bytesRead < bytes);

  // read data
  ViUInt8 directCommandBuffer2[] = {LSREAD, HTPort};
  ViUInt8 responseBuffer2[] = {1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1};
  Comm::SendDirectCommand( true,
                    reinterpret_cast< ViByte* >( directCommandBuffer2 ),
                    sizeof( directCommandBuffer2 ),
                    reinterpret_cast< ViByte* >( responseBuffer2 ),
                    sizeof( responseBuffer2 ));

  long signed int result=0;
  for (int i = 3; i<3+bytes; i++) {
    result <<= 8;
    result |= responseBuffer2[i];
  }
  return result;
}

/* The speed2pwr() function takes a speed in mm/s and converts it to a
   power level sent to the hiTechnic motor controllers using the
   TIC_PER_SEC_PER_PWR constant defined in schrodie_motor.h that was
   determined through trial and error experimentation.  This function
   could probably be replaced with a simple pre-processor macro. */

int Motor::speed2pwr(double speed) {
  return (speed*TIC_PER_MM)/TIC_PER_SEC_PER_PWR;
  // mm/s * tick/mm * (s/tick/pwr)
}

/* The dist2tic() function converts a distance in mm to encoder ticks
   using the constant TIC_PER_MM defined in schrodie_motor.h.  This
   constant was also deterined experimentally.  This function could
   also be easily replaced with a pre-processor macro. */

int Motor::dist2tic(double mm) {
  return mm*TIC_PER_MM;      // mm * tick/mm = tick
}
```

schrodie_motor.h

```
#include "NXT++.h"
#include <iostream>
#include <ctime>
#include <cmath>

#ifndef ___SCHRODIE_MOTOR_H___
#define ___SCHRODIE_MOTOR_H___

// TODO: find the right numbers
#define MOTOR_W 141    // 1/2 the width of the wheelbase, in mm
#define MOTOR_L 127    // 1/2 the length of the wheelbase, in mm
#define TIC_PER_MM 11.5//8.7 // mm of travel/encoder tick
#define TIC_PER_SEC_PER_PWR 40    // ticks/second/pwr increment
                 // 1000 degrees/second at pwr=100 / 100 * 1 revolution/ 360 degrees *
1440 ticks/revolution
#define TOP_SPEED 100*TIC_PER_SEC_PER_PWR/TIC_PER_MM     // highest representable
speed in mm/s
#define PI 3.1459

/* Here are the defines for the motor speed setting retries. */

// Try to get it moving 5 times before giving up..
#define MOTOR_COMMAND_RETRIES 5
// How far in millimeters should we wait for a motor to move before
// checking up on it
#define MOTOR_CHECK_DISTANCE_MM 5
#define MOTOR_ENCODER_CHECK_THRESHOLD (TIC_PER_MM * MOTOR_CHECK_DISTANCE_MM / 2)

// motor 2 is on the left
// speeds are in mm/s, distances are in mm
// angles are in degrees

struct position {
  double left;
  double right;
};

namespace Motor {

// initalise the motor
  void setup(int port=IN_1);

// reset the encoders
  void resetPosition();
```

```
// return the distance travelled by each wheel
  position getPosition();

// sets the motors to constant speed mode
  void setModeSpeed();

// stops the motors.
  void stop();

// sets the motors to servo mode
  void setModeServo();

// sets the speed of both motors. takes care of unit conversion, and overspeed
correction.
// TODO: slippage correction to emulate 2-wheeled differential drive
  void setSpeed(double spdLeft, double spdRight);

// sets the target position of bot motors in ticks and confirms the target was
actually stored.  target is relative to current position.
  void setTargetTicks(int trgLeft, int trgRight);

// sets the target position of both motors. takes care of unit conversion. Target is
relative to current position.
// TODO: slippage correction to emulate 2-wheeled differential drive
  void setTarget(double trgLeft, double trgRight);

// return the current encoder target
// for debugging
  position getTarget();

// returns the battery voltage in volts
  double getVoltage();

// drive forward at the specified speed
  void drive(double speed = 100);

// drive forward at the specified speed for the specified distance
  void driveTo(double dist, double speed = 100);

// turn counterclockwise in a circle of the specified radius driving at the specified
speed.
// TODO: test, explain how the math works
  void curve(double radius, double speed = 100);

// turn in the specified direction at the specified speed
  void turn(double direction, double revolutionsPerSecond = .1);

// turn counterclockwise the specified angle at the specified speed.
// use busy to check for done
  void turnAngle(double angle, double revolutionsPerSecond = 0.1);

// returns true if the motors are en route to the target postion
// for use in servo mode
  int busy();

// internal use
  void waitMS(int ms);
  void i2cWrite(int address, int data);
  int i2cRead(int address, int bytes = 1);
  int speed2pwr(double speed);
  int dist2tic(double mm);

}

#endif  // ___SCHRODIE_MOTOR_H___
```

motor_battery_torture_test.cpp

```
/* This is a simple test program that turns the robot alternately
   left, then right, counts the turns, and monitors the battery
   voltage.  It prints out a .csv file on stdout for every turn count
   with the current time, number of turns, and current battery
   voltage. This will give me a power curve for the battery and tell
   me approximately what the power use is for turning.  I expect to
   run this program once with the four wheel drive configuration and
   then run it again after the OmniWheels are installed to see if
   there is any battery life gain to be had from using the OmniWheels
   instead of the dragging four wheel drive system. I can also use
   this code to determine if there's any left or right bias in the
   turning that will build up over repetitions. */

#include "NXT++.h"
#include "schrodie_motor.h"
#include <iostream>
#include <cmath>
#include <sys/stat.h>
#include <time.h>

using namespace std;

int main() {

  double battery;
  int turns = 0;
  double turn_angle = 90;
  time_t start_time, elapsed_time;

  if (NXT::Open()){ // Initialize the NXT
    Motor::setup(IN_1);


    battery = Motor::getVoltage();
    start_time = time(0);

    cout << "Starting motor torture test." << endl;
    cout << "Starting battery voltage: " << battery
       << " Volts" <<endl;
    cout << "Cycle, Seconds Elapsed, Battery Voltage" << endl;

    while (battery > 10.0 && turns < 1000) { // Repeat turning left
                                  // then right until the
                                  // battery voltage gets
                                  // below 10 volts or we
                                  // do 1000 cycles (which
                                  // isn't likely)
```

```
      Motor::resetPosition();
      Motor::turnAngle(turn_angle, 1); // Turn 90° at one rev per second.

      Motor::busy(); // Throw out the first busy result because it's
                // almost always wrong
      usleep(10000);
      while (Motor::busy()) usleep(1000); // Wait for the motors to
                              // stop being busy

      Motor::resetPosition();
      Motor::turnAngle(-turn_angle, 1); // Turn left

      Motor::busy(); // Throw out the first busy result because it's
                // almost always wrong
      usleep(10000);
      while (Motor::busy()) usleep(1000); // Wait for the motors to
                              // stop being busy

      battery =  Motor::getVoltage();
      elapsed_time = time(0) - start_time;

      cout << turns << ", " << elapsed_time << ", " << battery << endl;

      turns++;
    }
  } else {

    cout << "Unable to initialize NXT." << endl;
  }

}
```

schrodie_command_line.cpp

```
/*
  This is a quick and dirty program to drive schrodie by the command
  line.  Depending on what name it is run as it performs different
  tasks.  If the program is called as "right", it turns Schrodie to
  the right etc.  If it is given a command line parameter, that is
  interpreted as either a distance or an angle.  Distances are given
  in centimeters.  Angles are given as degrees.  If no angle is given,
  it assumes a 90 degree turn.  If no distance is given, it assumes
  three meters.  If it is called as "stop", schrodie stops.

  The program also continually polls the bump sensors.  If a bump is
  detected it exits with a -1 return value and outputs which bump
  sensor was tripped to stdout.  Otherwise it simply exits with a 0
  "success" return and gives the current battery voltage.

  This program exists because some people have difficulties
  understanding shared memory, sockets, and other elements of Linux
  Inter Process Communication, but can understand how to run a program
  from the command line.

  Note: Due to the perversity of libUSB under linux, this program MUST
  be executed suid root.  I am not happy about this, but since it
  doesn't use anything but numerical inputs, it might be OK.  And
  we're not super concerned about security on a robot anyways.
*/

#include "NXT++.h"
#include "schrodie_motor.h"
#include <iostream>
#include <cmath>
#include <sys/stat.h>
#include <time.h>
#include <string.h>

using namespace std;

/* Default angle and distance. */

#define ANG_DEFAULT 90
#define DIST_DEFAULT 300
#define SPEED_DEFAULT 100
#define RATE_DEFAULT 0.1

/* Delays to give the NXT time to communicate. */

/* Initial delay is 50ms */
#define BUSY_DELAY_COUNT 5

/* Loop delay is 1ms */
#define WAIT_DELAY 1000

/* Global variables and defines for bump sensors. */

#define BUMP_FRONT_PORT IN_4
#define BUMP_BACK_PORT IN_2
#define BUMP_BELLY_PORT IN_3

int bumpFront = 0;
int bumpBack = 0;
int bumpBelly = 0;

int bump() {

  int bumped = 0;

  if (NXT::Sensor::GetValue(BUMP_FRONT_PORT)) {
    bumpFront = 1;
    bumped = 1;
  }

  if (NXT::Sensor::GetValue(BUMP_BELLY_PORT)) {
    bumpBelly = 1;
    bumped = 1;
  }

  if (NXT::Sensor::GetValue(BUMP_BACK_PORT)) {
    bumpBack = 1;
    bumped = 1;
  }

  return bumped;
```

```
}

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

  double battery;
  char *commandName;
  int angDist = 0;  // Angle in degrees or distance in centimeters.
  double spdRate = 0;
  int delayCount = 0;
  int bumped;

  /* Find out what name we were called as.  Possibilities are:

     "f", "F", "forward"
     "b", "B", "backward"
     "r", "R", "right"
     "l", "L", "left"
     "s", "S", "stop"

     We only really care about the first character, and that as a
     lowercase.
  */

  commandName = strrchr(argv[0], '/');

  if (commandName) commandName++;
  else commandName = argv[0];

  cout << commandName << endl;

  /* If there's a value parameter, get that too.  Note that if it's
     not an actual integer number, the value of angDist remains 0 so
     the defaults will be used. */

  if (argc > 1) angDist = atoi(argv[1]);

  /* If there's a second value parameter, use it to set the robot
     speed. */

  if (argc > 2) spdRate = atof(argv[2]);

  if (NXT::Open()){ // Initialize the NXT

    cout << "Initializing the NXT" << endl;

    /* Set up the HiTechnic motor controller. */
    Motor::setup(IN_1);

    /* Configure the bump sensors. */
    NXT::Sensor::SetTouch(BUMP_FRONT_PORT);
    NXT::Sensor::SetTouch(BUMP_BACK_PORT);
    NXT::Sensor::SetTouch(BUMP_BELLY_PORT);

    /* Get the current battery voltage. */

    cout << "Getting the voltage" << endl;

    battery = Motor::getVoltage();
```

```
cout << "Executing the command" << endl;

switch (tolower(*commandName)) {


case 'b': // back
  if (angDist == 0) angDist = -DIST_DEFAULT;
  else angDist = -angDist;

case 'f': // forward
  if (angDist == 0) angDist = DIST_DEFAULT;
  if (spdRate == 0) spdRate = SPEED_DEFAULT;

  cout << "Drive" << endl;
  Motor::resetPosition();
  Motor::driveTo(angDist * 10, spdRate); /* Convert to mm.  Drive
                              at 1 meter per second
                              (or slower if the top
                              speed cap is less) */

  Motor::busy(); // Throw out first busy message.

  while (!(bumped = bump()) && Motor::busy()) {
  //    usleep(WAIT_DELAY);
  }

  /* If we were bumped, then STOP. */
  if (bumped) Motor::stop();
  break;

case 'l': //left
  if (angDist == 0) angDist = -ANG_DEFAULT;
  else angDist = -angDist;
case 'r': //right
  if (angDist == 0) angDist = ANG_DEFAULT;

  if (spdRate == 0) spdRate = RATE_DEFAULT;

  cout << "Turning" << endl;
  Motor::resetPosition();
  Motor::turnAngle(angDist, spdRate); // Turn at one rev per second.
  Motor::busy(); // Throw out first busy message.

  while (!(bumped = bump()) && Motor::busy()) {
  // usleep(WAIT_DELAY);
  }

  /* If we were bumped, then STOP. */
  if (bumped) Motor::stop();
  break;


case 's': // stop
  Motor::stop();
  break;
default:
  break;
```

```
    } // switch


    cout << battery;

    if (bumpFront) {
      cout << " FRONT";
    }

    if (bumpBack) {
      cout << " BACK";
    }

    if (bumpBelly) {
      cout << " BELLY";
    }

    cout << endl;

  } else {

    cout << "Unable to initialize NXT." << endl;
  }

  if (bumpFront || bumpBack || bumpBelly) return 1;

  return 0;

}
```