Implementation of a Deliberative Robot Control Architecture on an Inexpensive Robot Platform

by Gary R. Mayer, Bachelor of Science

A Thesis Submitted in Partial
Fulfillment of the Requirements
for the Master of Science Degree

Department of Computer Science
in the Graduate School
Southern Illinois University Edwardsville
Edwardsville, Illinois

June, 2004

TABLE OF CONTENTS

APPENDICES

LIST OF FIGURES

LIST OF TABLES

CHAPTER I

INTRODUCTION

Problem Statement

Robotic platforms are being used to enhance the education of students from elementary school to graduate college courses. A growing population of inexpensive platforms is among these.[1] Nevertheless, some people have viewed the capability of these inexpensive platforms as too limited to create anything but rudimentary robotic architectures (reactive control). The intent of this project is to show that the full potential of these platforms has yet to be explored; robotic architectures can be created from inexpensive platforms like LEGO Mindstorms and used for teaching a complex concept like path planning and navigation in robotics and artificial intelligence courses. But, before understanding the "how" of using these inexpensive platforms, it is important to understand "why" they have come to play such a large role.

Weinberg and Yu [16] state that the reason for robotics success as an educational tool is three-fold. First, robots offer a unique learning experience by providing hands-on experience with an integrated system. Second, the cost of robotic platforms has dropped considerably over the last decade, making them affordable to schools with small budgets.[2] Third, the new platforms offer a "plug-and-play feel". The latter has lessened the need for instructors and students to have a broad knowledge of how to design and integrate the

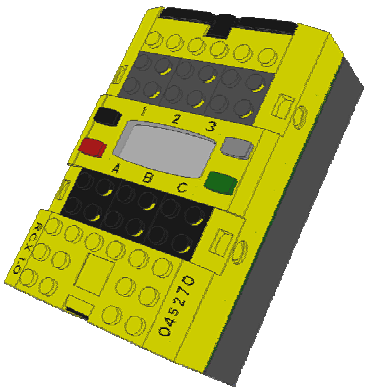
[1] A broad range of applications can be examined in [2],[5],[7],[8],[10],[13],[14].
[2] The LEGO Mindstorms Robotics Invention System – with over 700 LEGO pieces and the controller – can be bought for under $200.00 (US).

mechanical, electrical, and computational components of a robotic system. Thus, robotics is no longer restricted to large institutions, but available to others for a number of purposes.

Another reason that robotics was limited in its application was a lack of a framework for insertion into the curriculum. With the Association for Computing Machinery (ACM) / Institute of Electrical and Electronics Engineers, Inc. (IEEE) Computing Curriculum 2001 (CC2001) developing curricular guidelines for undergraduate programs in computing, this barrier has been breached [10]. Klassner and Anderson [10] believe that robotics projects can support at least seven of the 14 knowledge areas in the CC2001. They then pose the following criteria for a robotic platform to meet these criteria: it must support multiple sensors, have a modifiable chassis, support communication between robots and / or a personal computer, be programmable in several languages commonly used in computer science courses, and the robot platform requires a central processing unit (CPU) and enough random access memory (RAM) to support complex programming. Lastly, it must be affordable. The LEGO Mindstorms kit is strong in many of these areas. Its success in the hobby community has helped fill in the gaps where it is weak as an off-the-shelf product.

In 1998, collaborating with the Massachusetts Institute of Technology (MIT), the LEGO Group released LEGO Mindstorms [4]. Mindstorms was a new product line that makes use of LEGO's existing plastic bricks, motors, and gears and allows the user to automate a mechanical creation with a mobile programmable computer. Its computer, called the Robotic Command Explorer (RCX) (Figure 1, below), is based on a Hitachi H8 microcontroller running at 16 MHz and with 32 KB of static random access memory (RAM) for firmware and programs. The LEGO product makes it simple to construct a mechanical

structure (Figure 2, below) through the use of very familiar LEGO bricks. Furthermore, the sensors and motors all have a standard interface that makes them easy to connect to the RCX and a graphical user interface provided by LEGO that simplifies the programming process. The graphical programming environment is intended to be simplistic and allows programming to be as easy as attaching two virtual LEGO code bricks together. Further, the platform has been so popular that a number of hobbyists and schools have created a variety of C, Java, LISP, and Visual Basic programming environments to support the LEGO RCX and a growing interest in robotics. These environments have also added capabilities beyond the LEGO graphical programming tool like floating point variables, multidimensional arrays, structures, and pointers [3],[6].



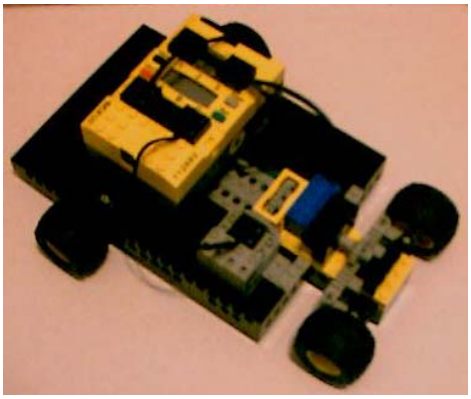


Figure 1: LEGO Mindstorms RCX Figure 2: Example of a Robotic Mechanical Structure

Since the 1990s, robots using LEGO Mindstorms have appeared as educational tools for elementary, high school, and college students around the world [4]. The approach and purpose of the projects differ, but the emphasis remains to capture students' interest in the subject matter by providing the ability to logically devise, create, and test their ideas on a platform capable of interacting with the real world. The success of these platforms and their

effect on students can be seen by their use in international robotics competitions at both the college and high school levels.[3] Furthermore, robotics in education has grown enough to warrant its own forum in professional conferences and publications of organizations like the American Association for Artificial Intelligence (AAAI) and IEEE.[4] While the popularity of these inexpensive platforms has proven their usefulness, there is still some argument that the utility is very limited to robotics education and research due to the reduced processing capability, memory capacity, and sensor limitations. In her book, Robin Murphy specifically states, "Lego Mindstorms….robots can be used for the first six chapters [on reactive paradigm robotics], but their current programming interface and sensor limitations interfere with using those robots for the more advanced material [such as deliberative paradigm robotics using path planning and map making]." [12] While it is true that the limited resources of the inexpensive platforms do not support the spectrum of current robotic architectures as well as the more expensive robotics platforms specifically intended for robotics research, a platform like LEGO Mindstorms can still be used.

The reactive paradigm discussed above is one of three robotic paradigms in current robotics theory. The other two are deliberative and hybrid. An architecture using a pure reactive paradigm tightly couples sensor input and action. The robot does not plan nor retain world knowledge; it simply reacts to whatever its sensors tell it at that moment. Reactive architectures are best applied to dynamic environments because they do not derive a specific

[3] The KISS Institute for Practical Robotics Botball and RoboCup tournaments, for instance.
[4] See the 2001 and 2004 AAAI Spring Symposium on Robotics in Education conference notes and IEEE Robotics & Automation Magazine. vol. 10, no. 2. June 2003 and vol. 10, no. 3, September 2003 for IEEE special issues on robotics in education.

control plan. Conversely, they have difficulty completing specified goals. This is the architecture for which the inexpensive platforms are known, as a strong coupling of sense and reaction does not require a lot of memory or processing power. An architecture based on a pure deliberative paradigm maintains world knowledge, gathers input from its sensors and plans its next move based upon what it knows about the world around it. The storage requirements for world knowledge and computing resources required for planning algorithms make this approach much more processor and memory intensive in most cases. However, it tends to be much more efficient at completing goals than the reactive architecture when the environment is static. It is less efficient in a dynamic environment that forces frequent replanning. A hybrid paradigm refers to an architecture using a combination of reactive and deliberative to best achieve the task at hand [1],[12].

Purpose of the Project

This project demonstrates that an inexpensive platform is capable of being used to teach a deliberative robotic control that includes navigation and planning. Navigation incorporates determining a goal location, planning a path to get there, maintaining knowledge of where the robot has been, and maintaining present location within the world representation [12]. Planning involves taking knowledge that the robot has acquired and using it to develop a sequence of actions that achieve a goal [15]. To enable a more direct compare-and-contrast of a deliberative architecture's performance versus a reactive architecture's performance, two robots were developed. The two LEGO Mindstorms robots were built using standard pieces with the exact same physical properties. One construct housed an RCX with a deliberative software architecture, the other and RCX with reactive architecture programming.

Once development and testing of the two platforms was complete, the deliberative robot design was used to teach robotics concepts of planning and navigation in an upper-division class on artificial intelligence. To maximize the applicability of the results of this project to Southern Illinois University Edwardsville and other universities, the project makes use of the LEGO Mindstorms set, a commercially available suite of LEGO rotation, light, and touch sensors, and C-like programming environments that can be obtained by educational institutions free of charge.

CHAPTER II

RELATED WORKS

In the article, “Mobile Robot Labs,” by Lloyd Greenwald and Joseph Kopena [8], the RCX and Handy Board[5] are both being used to examine how far inexpensive robotic technology can be pushed for teaching more advanced artificial intelligence and robotics concepts. Similar to my project, advanced AI techniques are being used such as map-based path planning. Further, deductive reckoning is the approach being taken for localization. Interactive C (IC) is also being used as the software development platform. Unlike this project, other advanced techniques are being explored – such as Bayesian representation, resource-bounded reasoning, and real-time control. The Handy Board and third-party sensors are the primary focus and a website reference is given for substituting the LEGO RCX for the Handy Board. However, third-party sensors are still required.

“6.836 Final Project: Evolution in the Micro-Sense: An Autonomous Learning Robot,” by Chuang-Hue Moh [11], is another project that explores ways to expand the usage of low-cost platforms for non-traditional, more advanced purposes. Specifically, it examines the use of genetic algorithms (GAs) on a RCX platform to enable the robot to learn. Besides the use of GAs, the project also differs from mine through its use of a reactive architecture for motion behavior, not deliberative path planning and navigation.

[5] The Handy Board is an open platform microcontroller system. It is capable of running four motors and has inputs for 16 sensors (analog and digital). More information may be found at http://handyboard.com.

Dr. Dean Hougen's "Hybrid Deliberative/Reactive Systems" [9] is an undergraduate student project for developing hybrid robotic architectures. Students are requested to design, build, program, and demonstrate a robot that efficiently acquires a target and returns it to the robot base station. The robot is given some *a priori* data and unknowns exist. This is very similar to the deliberative robot in my project and the implementation plans for applying the results of this project to an introductory robotics lesson. In addition, the project uses the RCX and IC for software. Unlike my project, this project makes use of the Handy Board, potentiometers, photoresistors, and other non-standard LEGO components. Again, my project focuses strictly on the LEGO RCX brick and standard LEGO building components.

Each of these projects further demonstrates the educational community's interest in exploring just how far inexpensive platforms can go. However, a number of projects turn to the Handy Board due to the LEGO RCXs limitations in memory and sensor ports. But, the RCX remains the easiest platform to use for students and instructors not desiring to delve into the technicalities of sensors and soldering based upon its plug-and-play ports. It's for this reason that this project is unique. This project seeks to push the LEGO RCX and standard LEGO components as far as they can go. The goal is to understand what the system can do and then incorporate what is learned into the curriculum so that students can use the platform to continue their education beyond the platform's apparent current means.

CHAPTER III

ARCHITECTURE

Model

To demonstrate the deliberative robot's ability to navigate and plan, we needed a goal. Animal behavior provided a template for the design of both of the robots and something for comparison of the results [1]. The robots for this project were modeled after a foraging animal. The robots' purpose became a fetch and retrieve mission. The robots' goal was to gather as many targets as efficiently as possible and return them to the starting area. Efficiency, in this case, emphasizes timeliness in completing the task successfully. It was also necessary for the robot to find its way around obstacles in its environment. From this concept of the robots, their niche – an arena containing food and obstacles – was also developed.

Current Architecture

To model the robots as foraging animals that could collect an item and return it to a specific position, the robots required mobility, the ability to detect targets and obstacles, and the ability to find their way back to the start location. Implementing the two different robot paradigms – reactive and deliberative – required two very different approaches to logic within the software. As mentioned previously, the hardware design was constructed to support both of these software architectures. A number of software revisions and different hardware architectures were developed over the course of the project and they are explained

here so that the reader may gain an understanding of the strengths and weaknesses of different models for their own usage.

Reactive paradigm software architecture

Five distinct phases emerged while designing the software for the reactive robotic paradigm – finding a goal item (a target), acquiring it, bringing the target back to a set location, releasing the target, and avoiding any obstacles encountered while executing the other behaviors. To detail the transition between these phases, a Finite State Acceptor (FSA) Diagram was developed (See Figure 3, below, and associated Table I, below). Note that the reactive paradigm robot does not have a specific end state. Once it successfully returns a target to the designated location, it goes back out and searches for another. Each of these phases was coded as a separate behavior. The interaction between these behaviors is shown as a Suppression Arbitration Network (SAN) (Figure 4, below). What the SAN illustrates is that a behavior with a higher priority overrides, or suppresses, the output of the behaviors below it. This is done without the lower priority behaviors knowing that their output is being suppressed. Mechanically, none of the behaviors directly control any of the robots sensors or effectors. An independent process acts as arbitrator. It reads the sensor inputs and makes them available to the other processes. The behavior processes set flag variables stating their desired output. The arbitrator then sets the highest priority behavior with a desired output to become the robot output – using motors, reset encoders, and switching state on internal state variables.

The reactive paradigm robot maintains no world knowledge. It uses a direct SENSE-ACT approach [1]. When the robot first starts, it records the average value of the ambient

light source and a target light source. Values are maintained as a percentage (0 – 100) with 100 being the brightest light source. Raw sensor values could not be used, as the readings were unstable with the default firmware. The robot begins with no knowledge of its location within the arena and retains no data about what it has sensed or past actions. The one exception to this is one internal state variable that stores the position of the target trap arm (up or down). The reactive software employs the use of the timer function in the RCX to generate random numbers that determine the robot's direction and amount of movement. This prevents the robot from establishing a pattern in corners and other symmetrical areas, which may lead to the robot becoming stuck in that area. If the robot has no target, it exhibits a foraging behavior and randomly moves about the arena. If the light sensor detects a target while foraging, the robot changes its behavior to acquiring the target and moves directly toward the lit target. If the light level falls back below a threshold determined to be a target, then the robot does a quick pan by turning to the left and right. If the light is seen again, the robot again moves straight forward. If the light is not seen during the pan, then the robot resumes the foraging behavior. When the robot senses that the target light level is very high, indicating a high-probability that the lit target is in the target trap, the robot lowers the trap arm. This completes the acquire behavior. The return behavior then searches for the place to return to using a Marco-Polo algorithm. The robot emits an infrared signal, "Marco." If a second RCX at the return location – referred to as "Home" – detects the Marco signal, it emits a signal back to the robot, "Polo." If the robot detects the Polo signal, it assumes that it is facing the return location and moves forward. This behavior repeats until the robot no longer receives the Polo signal or it impacts Home. If the robot does not receive the Polo

signal, it randomly turns and moves about, resending the Marco signal until it again receives Polo. This approach physically drives the robot directly into Home. However, the Home RCX is fitted with a bumper to detect the robot's impact and emits an "At Home" signal when it occurs. The robot releases the target upon receiving an At Home signal by backing up, nudging forward to ensure that the target is not against the trap arm, and lifting the trap arm. Before the release behavior completes, it then turns the robot around about 180 degrees to allow a human time to retrieve the released target and prevent the robot from trying to recapture the same target. The robot then exhibits the forage behavior once again. If the robot impacts an obstacle, causing the touch sensor to be depressed, the avoid obstacle behavior takes control. When an obstacle is impacted, the robot backs away slightly from it and then turns a random direction and random amount between approximately 30 and 90 degrees.

The internal state variable that monitors the target trap arm is used to prevent the return home and release target behaviors from assuming control when inappropriate. While a constant check of target light level might have been used, it was unreliable as the target's light level fluctuated too rapidly due to battery problems (See "Problems encountered with the current architecture", below).

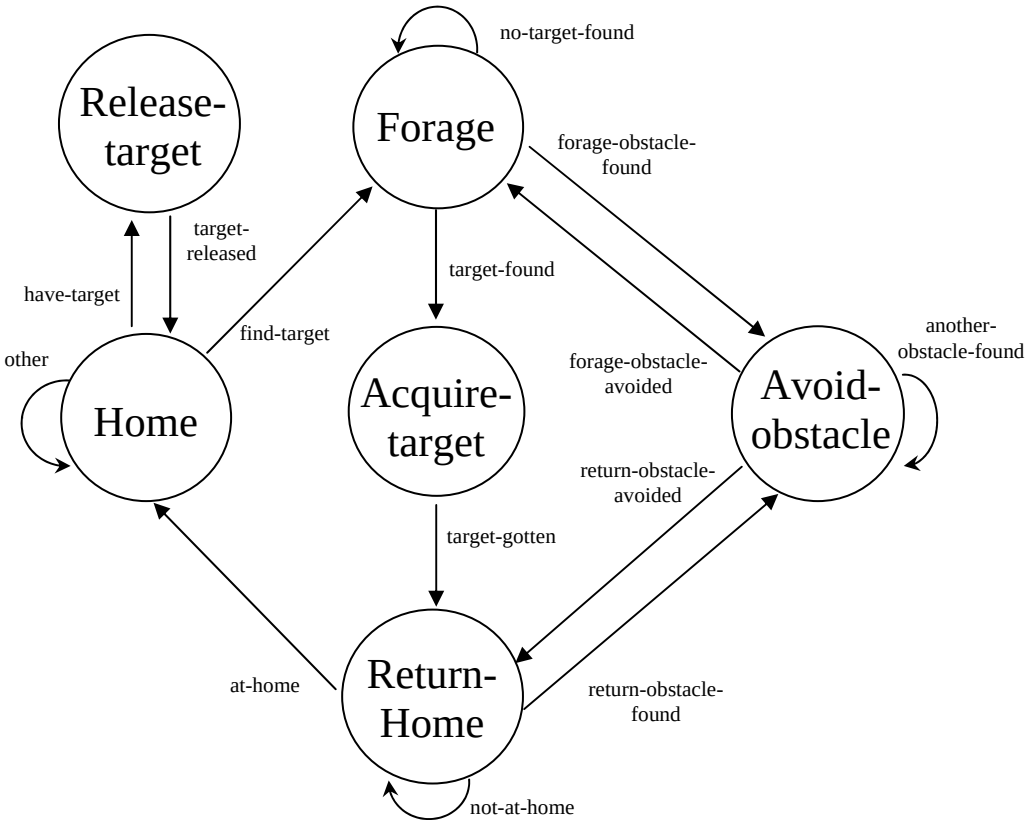


Figure 3: Finite State Acceptor Diagram

Table I: Finite State Acceptor Diagram

δ	*Q*	*input*	δ (*q*, *input*)
	Home	have-target	release-target
	Home	find-target	forage
	Home	other	home
	Forage	target-found	acquire-target
	Forage	forage-obstacle-found	avoid-obstacle
	Forage	no-target-found	forage
	acquire-target	target-gotten	return-home
	avoid-obstacle	forage-obstacle-avoided	forage
	avoid-obstacle	return-obstacle-avoided	return-home
	avoid-obstacle	another-obstacle-found	avoid-obstacle
	return-to-home	at-home	home
	return-to-home	return-obstacle-found	avoid- obstacle
	return-to-home	not-at-home	return-to-home
	avoid-return-obstacle	return-obstacle-avoided	return-to-home
	release-target	target-released	home

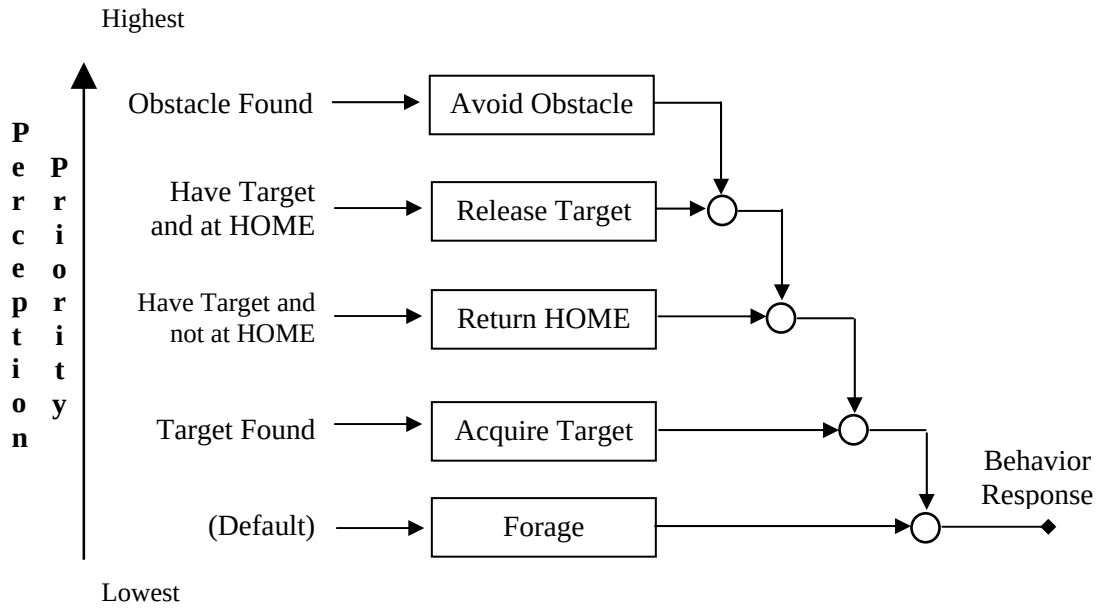


Figure 4: Suppression Arbitration Network

The reactive paradigm software is programmed using Dave Baum's Not Quite C (NQC) version 2.5 R1. It is a C-like programming language that uses the LEGO Mindstorms firmware [3]. The language was originally used based upon availability and documentation at the start of the project. Due to limited program space, the programs for both software architectures were ported to Interactive C. However, unexplained, erratic light sensor readings were encountered with the reactive architecture software using IC. Thus, the final version of the reactive robotic software was built using NQC. A copy of the reactive robotic code may be found in Appendix F.

Deliberative paradigm software architecture

The deliberative robotic paradigm software is very functionally oriented. There are two main processes that run concurrently – one that goal searches and retrieves the goal targets and the other that looks for unknown targets and obstacles that cause plan failure. The Retrieve Goals process uses functions that are sectioned by purpose into initialization, cartographer, navigator, and pilot. The first, the initialization functions, calibrate the ambient and target light levels (similar to the reactive design). For improved accuracy, the raw light values were used, giving a range between 100 – 1000 with the lower values normally indicating a brighter light source. The initialization functions also receive user input to build an *a priori*[6] map, including known targets, known obstacles, and robot starting position and heading. The cartographer functions build and maintain the robot's world map – a grid stored as a two-dimensional array. The navigator functions build the robot's plan for capturing the

[6] *A priori*: Information known independent of experience.

known targets. It uses a wavefront algorithm (see Appendix A) to generate a path. The pilot functions are the formal motor output functions that make the robot interact with the physical world. The robot is programmed to move horizontally and vertically. It changes direction by turning in place. The robot also possesses a target trap arm – which can be raised and lowered – that prevents a target from exiting the trap when the robot turns or backs up.

The deliberative paradigm robot uses a cyclic SENSE-PLAN-ACT approach [1]. The robot uses arrays to store representation of its world – the arena, the known obstacles, and the known targets. The *a priori* information is used to develop the robot's initial plan to move from its current location to the nearest goal. The plan is stored as an array of successive grid coordinates through which the robot should travel. As the robot progresses through the plan, it senses its environment through the touch sensor connected to a front bumper and the forward-looking light sensor for an indication of plan failure. If either indicates an object is present where none is expected, the robot backs up to the grid it previously occupied and the current plan is scraped; knowledge of obstacles and goals is then updated as appropriate. The robot then rebuilds the map and creates a new plan from its current position.

Object information is placed onto the grid each time a map is built. The map grid starts as an empty 2-dimensional array. The hard coded edge obstacles (arena boundaries) are placed into the grid as the other spaces are cleared. Then, the program traverses an obstacle location array and inserts the known obstacles onto the grid. The robot's heading and position are maintained in variables. If the robot is using the path-planning algorithm to determine the closest target, targets are represented as empty spaces. This prevents targets from becoming unreachable if located behind other targets in a corridor. However, when the

robot plans a path to a specific target, the other targets are represented as obstacles during the path-planning process to prevent the robot from running through their grid space and possibly knocking them out of the way.

Once the map is built, the robot uses a wavefront algorithm to find the closest known target (target priority is based upon proximity to the robot). The robot finds the closest target by propagating the wave throughout the arena and then evaluating the value of the space occupied by the target in question. It does this for each known target and the closest target (the one with the lowest value) is maintained as the current closest target. To conserve memory, the grid values resulting from the wavefront propagations are not saved. Thus, once the closest target is found, the wavefront algorithm is run again for just that target. Then, a path generator follows the wave propagation from the robot to the target, saving the grid coordinates of each step. Once a path is generated, the robot follows the path. The robot maintains its current heading as it advances. To execute each step of the plan it first determines if it needs to turn left or right; using its turn rotation sensor to execute the motion as required. It then moves straight to the next coordinate in the plan using its linear rotation sensor to determine proper distance.[7] To account for gear slop and improve localization, the robot maintains which direction it last moved. It then adjusts the threshold values for its rotation sensors accordingly. Once the robot reaches a target location, it stops and captures the target by dropping the target trap arm. The wavefront algorithm is then used again to plan a path back to Home. The robot follows the plan Home in a similar fashion. Once Home, the

[7] See Hardware architecture, below, for calculation of rotation sensor ticks required for linear and turning motion.

robot releases the target, turns around, and plans a path to the next nearest target, if any. If no known targets remain, the robot stops.

The robot is capable of seeking multiple targets at once. But, since the wavefront algorithm can only handle one goal at a time, objects on the grid have to be manipulated to support whether the algorithm is being used to find the closest target or make a path to a target. Storing object positions independent of the grid best supports this approach by allowing positions of specific objects to be easily modified or added. As mentioned above, when determining the closest target, the robot sets all other targets besides the one it is looking at to empty spaces. Given that the robot maintains no goal knowledge except location until the goal is determined to be the closest, it does not matter if the other targets are set as a space or an obstacle when searching each target, as long as a contingency exists if the robot cannot reach the target – a condition that's possible if there is a target in the only path between the robot and the current target. If the target cannot be reached, one can assume it either cannot be reached at all or there is a closer target. However, if additional target data is kept to improve performance when finding the next closest target, then the other targets should be set to spaces to allow a truer measure of target distance from the robot and to alleviate one target blocking another.

Once a target is captured, the robot plans a path Home. This is done because the robot may have had a plan failure (such as finding an obstacle in its original path) prior to acquiring the current target and following the existing path back may not represent the shortest path from the robot's current location to Home. For the purpose of planning a path to Home, all remaining target locations are marked as obstacles. This is necessary since the

robot cannot sense other targets when one is in the target trap. Thus, a plan that did not consider existing targets as obstacles may cause the robot to pass through another known target's location and knock it out of position, causing the robot's plan to fail when it returned to capture this rogue target.

A plan under execution can fail in three ways – an unknown obstacle is encountered, an unknown target is found, or what was given as a known target is not there. For the first, the robot ceases execution of the current plan when the bumper detects an obstacle. The Plan Failure process then moves the robot back to the last occupied position and updates the appropriate obstacle array. The Retrieve Goals process is then restarted, which recalculates the closest target using the new data as the new obstacle may require a much longer path to what was the closest target. The robot then automatically generates a plan from its current location to whatever grid is determined to contain the closest target. Next, the robot executes the new plan. The robot does the same for unknown targets except the robot can only detect the presence of unknown targets while it is traveling to a target. If the robot has captured a target, the captured target blocks the robot's light sensor that it uses to detect targets. If the robot finds that no target exists where it expects, it removes the target from its goal array. Then, it determines the closest remaining target from its current location and plans a path to it. If no further known targets exist, it stops.

A number of steps were taken to make the deliberative robotic paradigm program easier to use and understand. First, it is programmed using Interactive C (IC) version 4.2. IC 4.2 was chosen because the C language is commonly used in a majority of courses at SIUE and other universities. Second, it replaces the LEGO Mindstorms firmware with one that has

a much smaller memory footprint. The IC4 firmware provides a lot more memory space for storing grid, obstacle, and target information, removing the need to resort to bitwise encoding for a small arena. NQC resides on top of the existing LEGO firmware so does not offer these benefits. Third, to ease the burden of maintaining localization, the robot was limited to ninety-degree turns. No diagonal movement between grid spaces was allowed. Further, the array contents represented grid vertices. Thus, obstacles and targets were assumed to occupy the whole grid space. A copy of the deliberative robotic code may be found in Appendix G.

About LEGO hardware

The commercially available LEGO kits provide a number of motors and sensors for a variety of different purposes. The current LEGO Robotics Invention System (RIS) includes 9 Volt (V) geared DC motors. The RCX output ports, strictly digital devices, control the speed of these motors through Pulse Width Modulation (PWM). This means that the RCX always outputs the maximum voltage but does so in pulses. The RCX sends these pulses every 8 milliseconds (ms). The length of the pulse over the 8 ms determines the duty-cycle. Thus, if the pulse were 3 ms long, the duty-cycle would be 37.5% (3 ms / 8 ms = 0.375). The lowest power is 1 ms in length while the highest power setting lasts the entire 8 ms. The longer the pulse, the faster the motor shaft spins. The light sensor contains both a red Light Emitting Diode (LED) that emits a visible red light and a clear phototransistor/receiver which allows more current from the RCX to flow through as a stronger light source is detected. The light sensors are not very efficient and drastically different readings can be obtained from two light sensors under the same lighting conditions or from the same sensor under even slightly different ambient lighting conditions. Note that the distance the sensor is from the target

surface can influence the latter as well. The light sensors are best used to detect a distinct difference between a very dark object and a very light object; though a third mid-range color can be detected if ambient lighting conditions are supportive. The rotation sensors measure the amount of rotation on an axle. When the axle within the rotation sensor moves, it causes it to pass through four different states, each generating a tick and enabling the RCX to tell that the rotation sensor has been moved and in which direction. A full rotation of the axle causes the rotation sensor to pass through each of these four states four times. Thus, each full axle rotation will yield a count of 16 ticks. Gearing up from the wheel axle to the axle that the rotation sensor is mounted on will improve the sensitivity of the sensor feedback by increasing the number of revolutions of the rotation sensor axle for each rotation of the wheel axle. However, the motor tends to have an average loaded angular velocity of 250 rotations per minute (rpm) (350 rpm unloaded) and it was found that, using the standard RCX firmware, the rotation sensors begin loosing accuracy below 50 rpm and above 300 rpm. Thus, it may be best to directly mount the rotation sensor to the motor axle and gear down to the wheel axle. Other firmware may yield different accuracies depending upon sample rates. The touch sensors are simple on / off switches. When the sensor is depressed, current flows. When released, the circuit is broken [3],[6].

Hardware architecture

The project hardware has two main components, the arena in which the robot moves and the physical robot construct. The arena was designed around the robot and the goals of the project. But, the robot itself underwent many modifications. As a result, certain aspects of the arena’s design drove changes to the robot.

The arena had to provide enough space to support the project requirements but be portable enough to allow for easy teardown. The arena was in a multifunctional room used by other students and could not be left standing at all times. When erected, it had to fit on the available table space, as floor space was limited. Further, the most convenient, continuous surface to use was a 4’ x 8’ x ¼” sheet of board with a melamine coating. The melamine coating also provides good traction to most LEGO rubber wheels. However, the laminate tends to come off, especially if tape is used repeatedly on the surface. This led to irregularities on the surface, which did occasionally hinder the robot’s navigation, though not too significantly. They would likely cause a greater problem for robots using light sensors that face downward. This board size also met the need for most grid sizes approximations that the software could support.

Planning for the deliberative robot to use grid representation, initial estimates of the robot’s turning requirements determined that a 10” x 10” square would work best. The original language of choice was NQC. With the limited memory program memory available, the arena size was restricted to a 4 x 6 grid. It should be noted that the arena included a border of obstacles in the robot’s memory. Thus, internally, the robot stored a 6 x 8 grid. This made the use of the wavefront algorithm easier. When the language was transitioned to IC 4, it was unknown just how much memory would be freed. So, the original 4 x 6 physical grid remained. This grid size proved to be adequate when considering the 5 – 10 second processing time required to propagate the path-planning wave and the time required to actually move the robot from one end to the other, due to the slower speed caused by gearing the wheels down from the motor. This grid setup provided enough space to create open areas

and boxed canyons and was small enough that only a few obstacles are required to make a one grid-space-wide opening.

The obstacles, being physical in nature, were built as required to ensure that the robot's sensor was impacted and that the obstacle would interfere as little as possible with the other robot functions. Since the reactive robot did not have an internal representation of walls around the arena, a physical one was built using Polyvinyl Chloride (PVC) piping. With the robot's target trap, its bumpers had to be built up, above the height of the targets, to prevent the robot from mistaking a target for an obstacle. The obstacle wall was then built to stand a couple of inches above the arena floor.

The unique bumper design on the robot created a number of problems for a freestanding obstacle within the arena. To mimic the approach that the border obstacles took, a mushroom-shaped obstacle was required, one with a central column and a cylinder lying flat on top. However, none could be devised with the materials available that were also heavy enough to withstand a robot impact and not move or topple. To compensate, the robot's bumper was extended forward, beyond the trap arm, so that it would trigger when the robot ran into a vertical wall obstacle. These wall obstacles were constructed of cardboard boxes filled with rocks and brick to prevent them from moving when the robot impacted them. The obstacles were designed to be only 5" x 5", 10" tall, and sit in the middle of a grid space. This provided some allowance for the deliberative robot to be off center during its travels. The ramification of this design was that it allowed the robot to fit between two obstacles in adjacent grid spaces, especially diagonally. So, for runs involving the reactive robot, the obstacles were placed on their side to cover a 5" x 10" portion of the grid space. Being hand-

constructed, the obstacles had tape on the outside to hold the seams together. Further, because the boxes were made from much larger cardboard boxes, there were odd bows and creases in the cardboard sides. The tape and irregularities on the obstacle surfaces routinely caused problems for the reactive robot as it became stuck when its bumper would not properly trigger. Revisions to the bumper alleviate most of these errors. The deliberative robot, typically staying on course and only hitting the obstacles dead center, did not suffer from these same problems. An example of the arena (with robot and targets) is shown in Figure 5, below.

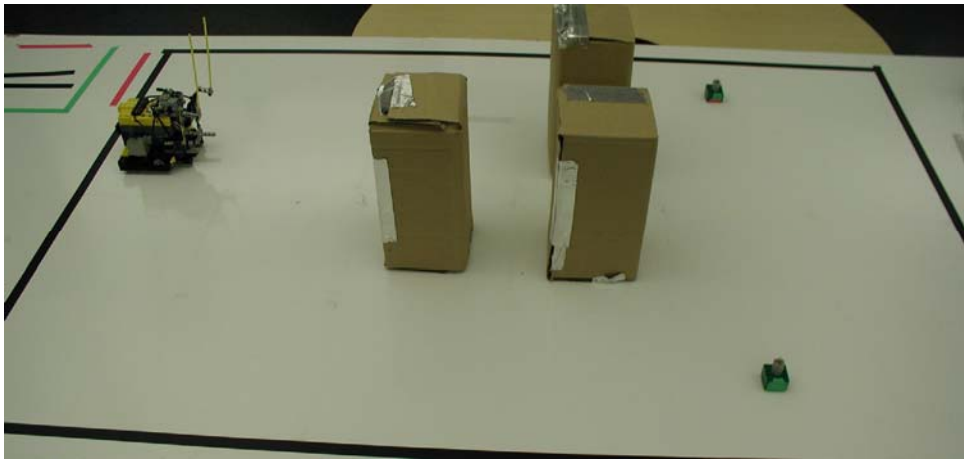


Figure 5: Arena Example

The robot's targets (see Figure 6, below) were LEGO pieces with a square base and round stalk rising up about 1 ½". A 12 V lamp was placed on the top of the stalk and two alkaline 12V batteries were placed in the square base. The 12 V lamp provided a very bright light source for the robot to detect and enabled it to be detected from 20" to 30" away with fresh batteries. The problem was finding a lamp and battery combination that would keep the

lamp well lit over long periods of a few hours. While many batteries can provide a lot of milliamps well in excess of the power requirements for the lamp, they can only do so for short periods at a time. After which, the light quickly dims and the battery is unusable for an hour or so. The best-lit target that could be developed used a 12 V lamp and two 12 V batteries in parallel. Placing the batteries in parallel doubles the current rather than the voltage. Using this configuration, the target light source remained usable for about an hour with minimal reduction in the luminescence of the lamp. However, the falloff at the time when the batteries did cease to provide enough power was dramatic. As the robot calibrated to the target light source, it was important to ensure that all targets were approximately the same luminescence. If the target used for calibration is too bright, the robot fails to properly capture a dimly lit target, thinking the source to be further out. Conversely, if a dim light source is used to calibrate, then the robot closes the target trap too soon, missing the target ahead. These problems only arise with the reactive robotic architecture. The deliberative architecture moves the robot to where it believes the center of the target grid space is located. It does not use the light to guide itself toward the target; it only checks if the light is actually present in the trap when it arrives.

Figure 6: Lit Target

As mentioned previously, the robot hardware architecture is designed to support both the deliberative and reactive software architectures. However, the deliberative architecture levies much more stringent requirements to ensure successful localization. Therefore, the majority of the hardware architecture was designed with the deliberative robotic paradigm in mind.

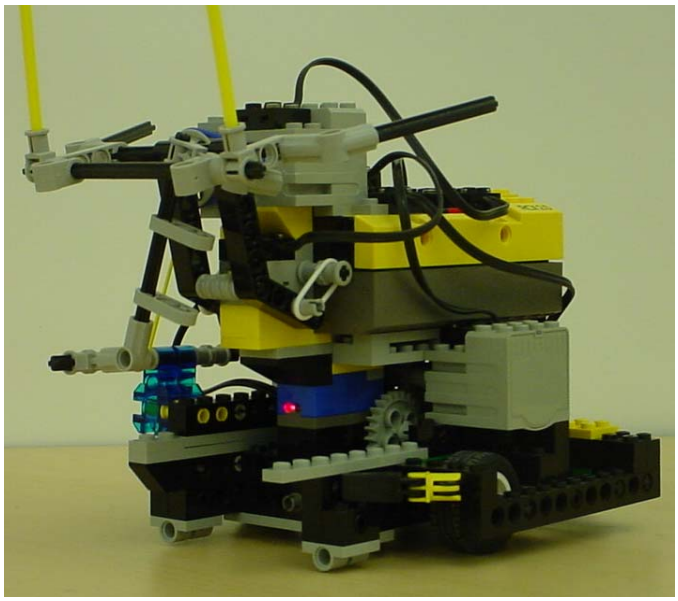


Figure 7: “Diane / Rea”, The Current Architecture

The robot’s physical configuration (see Figure 7, above) is designed around its gear train. A dual-differential system (see Appendix B) is used to ensure that the wheels travel at equal velocities. Two motors are used, each connected to one of the differential shells. One motor provides power for driving and the other for turning. By turning on one motor and locking the other, it could be asserted that the wheels were rotating at the same velocity for forward and reverse movement, or equal speed and opposite direction for turning. This

system allows a single axle to be used for both driving and turning. With a centrally mounted axle, it allows the robot to turn in place. Additionally, by turning on both motors at the same time, a turn with a turning radius could be generated, if desired.

Rotation sensors were required to provide deductive reckoning localization. The LEGO motors were always set to maximum speed, as they do not provide enough power to move a robot laden with the weight of an RCX unless a top speed, or near top speed, setting is used. To avoid loosing accuracy on the rotation sensors by exceeding their angular velocity maximum, they were geared directly to the motors with a 1:1 ratio. To gain more granularity in the rotation sensor ticks, the axle was geared down from the motor to a 1:6 ratio. With 16 ticks per rotation sensor rotation, this provided 96 ticks per wheel axle rotation. This meant that every tick equated to only 3.75° or, given a wheel circumference of about 95.5 mm, 1 mm traveled. While this provided much improved accuracy and torque to handle the loads, it also caused the robot to move much slower.

When constructing the robot, a balanced design was created with motors on each side of the body and the RCX positioned to help the robot rest on its skid plate in the back. Further, the wheels were enclosed within the frame structure to ensure that the axle was supported on both sides of the wheel. This design helped to avoid any unnecessary tilt or motion that would lead to immeasurable errors. In addition, the compact, balanced design made it easier for the robot to turn as the load was balanced both forward and aft of the turning axle.

The target trap was designed around the targets. The light sensor is placed at the optimal height to receive maximum luminescence from the target lamps. The width of the

trap is twice the width of the target stalk, allowing some room for error when trapping but forcing the lamp to remain in the light sensor's field of view. A trap arm was used to ensure that the target did not slide out when the robot backed up or turned. To maintain the balanced design, a third motor for the trap was mounted center on top of the robot. Rubber bands and pulleys were used to transfer motion from the trap motor to the arm. This allowed a timing function to be used to control the trap arm without having to resort to a sensor to determine when the arm was in place. The time was purposely set a little long. When the arm was in position, the rubber bands slipped on the pulleys. Unfortunately, the trap system obscured the RCXs IR port, which sometimes made the robot miss an IR signal from the Home RCX. It also added difficulty to the creation of an obstacle bumper system.

As stated previously, the bumper on the robot was designed to react to high obstacles only. As a result, tall whiskers were placed on the robot. They were stiff enough to allow the touch sensor to note the impacts but flexible enough that the robot's movement was not altered grievously by the impact. It should also be noted that the robot only had a bumper in the front. Most rearward movement was very short distances and mostly into space from which the robot just came. By removing the rear bumper, structural and code complexity were reduced with minimal loss to operational capability. This enabled more processing time for the other sensors.

Four sensors are required for the robot. A touch sensor is required to monitor the bumper discussed above. Two rotation sensors monitor the linear and turning motions and each required their own port. A port is also required for a light sensor to detect targets. With only three ports on the RCX, multiple sensors on the robot were ganged together on a single

port. Because of the nature of how ticks are read, it does not appear possible to gang two rotation sensors. It is even less feasible to gang a rotation sensor and light sensor. However, a light sensor and touch sensor can easily be ganged and it is not too difficult to tell which sensor input is being provided. When a touch sensor ganged with a light sensor is pressed, the RCX (reading the sensor input as a light sensor) registers an extreme value indicating a very bright source. For raw values, it is typically impossible to get the light sensor itself to read such a low value, even with a bright light source shining directly into the light sensor. However, if the light sensor is read as a percentage, the conversion from raw value to percentage enables a very bright source to register as 100% - making it difficult to distinguish between a bright light source and touch sensor impact. To prevent the target lamp from registering as 100%, the trap was modified to maintain some distance from the light sensor. Even with brand new batteries, the lamp only registers in the low-90s.

Detailed instructions on how to build the LEGO robot can be found in Appendix E.

Problems encountered with the current architecture

The most significant problems encountered while testing the robots was environmental hardware related. The issue was maintaining consistent lighting from the targets and trying to maintain the light level for any usable period of time. As mentioned above, the 12V, dual-battery system provided much better results but it too gave out, often at very inconvenient times. This issue plagued both design paradigms. The other problems are paradigm related.

The most troublesome part of the reactive robotic paradigm was the use of IR messaging. First, IR messages from Home were often misinterpreted because of the

reflection of the signal around the room or it was missed entirely. The later is expected to be a result of both the trap motor mounted on top blocking some of the signal and the positioning of the IR emitter and detectors within the IR port itself. The second problem was trying to properly time the receipt of the IR messages within a multithreaded environment. Wait delays and loop sizes all had a factor and numerous *while* statements had to include an IR message check in the Boolean expression to ensure that a message was not missed.

The one issue with the deliberative robot was localization. It was typically successful on single runs to the target and back. It was even successful going 1-way through a 16-step "S-Curve" with seven turns. However, a lot of movement creates enough errors to throw the robot off of its assumed position. The majority of these errors appear to develop during a turn. Slippage of the wheels on the arena surface is assumed to be the biggest cause of the errors. Irregularities in the surface are the other expected problems. This includes dirt, which fell out of the obstacles that were using rocks for weight. The dirt and irregularities typically caused the skid plate to jerk the robot at a small angle. While a small change in direction is negligible initially, the errors increase as the robot moves along. To maintain proper localization, after the robot dropped of each target, it was repositioned back at the center of the start grid before it was allowed to continue to the next target.

Previous Architectures

The robot's software and hardware architectures went through a number of revisions before the final design was reached. Each design increment offered improvements over the previous and yielded important lessons to be learned from the design's shortcomings.

First architecture, "Buggy"

The first hardware architecture was modeled after a car with rear-wheel drive (See Figure 8, below). While the 9 V motors are efficient, they are not always consistent with the velocity derived from the same power input. This means that if one motor is connected to each wheel, even if both wheels are set to the same speed, the robot is likely to turn, as the motor outputs are not exactly the same. To make a straighter path, a differential was used to ensure that the drive wheels had the same rotational velocity. A single motor powered the differential shell and this, in turn, powered both wheels. A rotation sensor was connected to the motor axle to measure distance traveled. To facilitate turning, a steering drive was created for the front wheels. A second motor controlled the front wheels' angle of turn. A rotation sensor was also connected to the turn motor axle to measure angle of turn. In addition to providing consistent angular velocity to both wheels during straight-line travel, the differential also allowed the rear wheels to move independently during turns, reducing drag across the wheel in the outside of the turn. While this model may offer some useful experience with front-wheel steering, it also creates complexities undesirable for this project. First, since Ackerman steering was not used – which allows the two front wheels to turn at different angles – there was drag induced during turns on the front wheels. This meant that one wheel had to slip and it was harder to ensure that the robot traveled in the desired arc for a specific turning distance. Second, the turn angles and distance to travel for a turn had to be calculated. Third, the large turning radius required a larger test area.

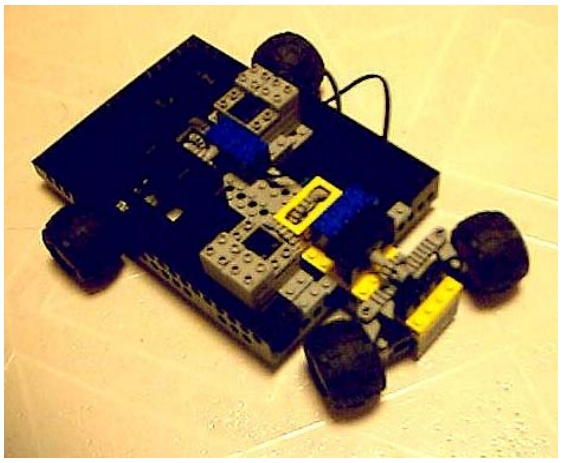
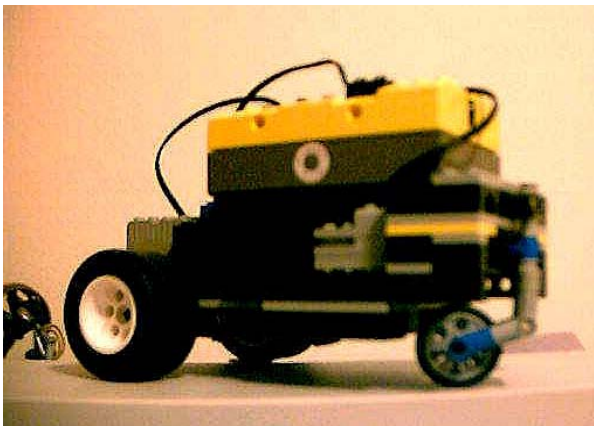


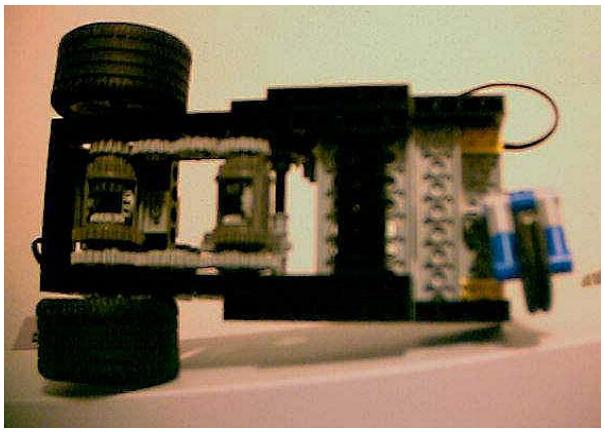
Figure 8: “Buggy”, The 1st Hardware Architecture

Second architecture, “Seeker”

To compensate for the turning problems with the first architecture, a robot that could drive and turn off of the same axle was required. Thus, the second architecture with a dual-differential drive was developed (See Figure 9, below). Since the rear axle now controlled the drive and turning motion, a caster wheel was employed at the front for balance. Initial tests proved this model to be successful. It greatly reduced the turning area required by turning about the center of its rear axle and made turning calculations simpler. A slight problem noted was that after a turn, the robot would waddle a little while the drag wheel straightened out. The next step was to add a way to capture a target so that it could be returned to the starting location.



Side View



Bottom View

Figure 9: "Seeker", The 2nd Hardware Architecture

Third architectures, "Mantis" and "Trap"

The third design employed a grabber arm to give the robot the capability to capture a target (See Figure 10, below). The second architecture's base was elongated to accommodate the motors and structure for the arm. Two motors were required – one to raise/lower the arm and one to capture/release the target. With a requirement for four motors and only three output ports on the RCX, a non-commercial multiplexer called a "backpack" was used.[8] The backpack had three states controlled by output port A. Each state allowed three different input ports and two different output ports to be active. This worked well for this design as when the two target capture motors were needed, the two drive and turn motors were not being used. Also, as a single caster wheel design proved unstable, a dual-caster system was used. With a working robot design, an arena was constructed. The lit targets were also devised. They originally looked like a 2" tall dumbbell standing on end and housed a small 3

[8] The "backpack" was developed and provided to this project courtesy of John Barnes.

V penlight and two 1.5 V N-sized batteries in the base. The design worked but suffered from the extreme weight added by the grabber arm. Further, there was a lot of difficulty getting the initial target lamp design to remain lit for any sufficient amount of time. Even when the target was properly identified, it was difficult to get the robot into the correct position to pick up the target. This was due to the long distance between the turning wheels and the grabber arms. While the rotation sensors on the wheels perceived they had moved only a small distance, the arc length of the distance the grabbers traveled was much greater – often overshooting the target. Also, if the target was dropped in the middle of pick up, it often fell over, leaving the robot very little chance of reacquiring the target given the sensors and the configuration of the lift arm grabbers. Once a target was acquired, the robot would seek an Infra-Red (IR) signal from another RCX near the starting location. The robot would send an IR signal back – something the base RCX would only see if the robot were facing toward it. Once this second signal was detected, the base robot would change its signal, indicating to the robot that it should move forward. If it lost the robot's signal, it would return to transmitting the original signal. This would indicate to the robot that it was no longer headed toward the starting location. The robot would then react by randomly moving about and resending its signal until the base RCX once again responded with the appropriate second signal. This IR version of Marco-Polo was used to guide the robot back to the starting position. However, it is not always as easy as following a straight-line course. As when the robot first went searching for the targets, it would have to avoid any obstacles on its way back. A problem that occasionally occurred is that the robot would bump into the target and react to it as if it were an obstacle. As long as the target remained upright, this posed no

problems. However, it would fall over half the time, resulting in a similar situation as when the target was dropped during pick up.

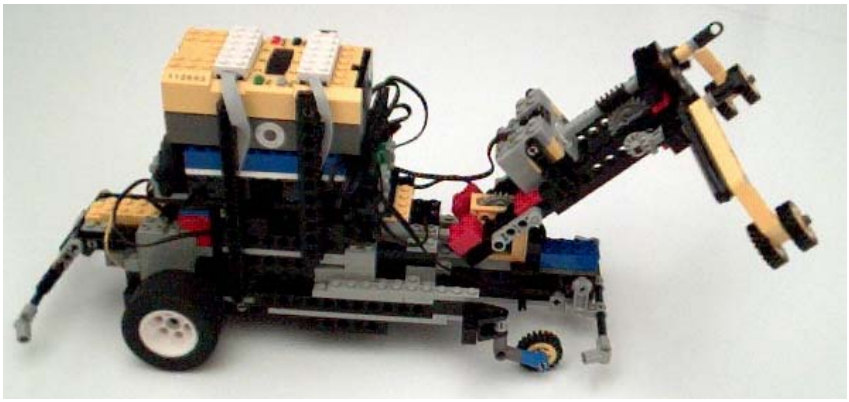


Figure 10: “Mantis”, The 3rd Hardware Architecture

The issues mentioned in the previous paragraph were managed by modifying the existing hardware architecture. To reduce the extra weight, the grabber arm was replaced by a trap mechanism (Figure 11, below). Guided by the light sensor, the robot would move toward the target with a wedge shaped trap. Once near, the target would slide into the trap as the robot moved forward. As the robot neared the lit target, the light intensity value increased. When the light sensor detected that the target was close enough, a single trap motor would lower an arm in place to ensure that the target did not slide out during turns. Since the robot no longer had to pick it up, the target’s top platform was removed and round bricks were used for the central column to enable the target to slide into the trap regardless of what direction it was encountered from. This gave the target its current shape as shown in Figure 11, below. To alleviate the possibility of the target being knocked over by the robot, the front bumper was modified to be higher than the base of the target. If the bumper

encountered the top of the target, they would go around each other. In case the target went toward the outside of the robot, angled pieces were placed so that the target would slide across the robot's side, to be picked up later. While this version of the hardware architecture had all of the essential pieces required to make it useful, a number of problems remained that still prohibited its success.

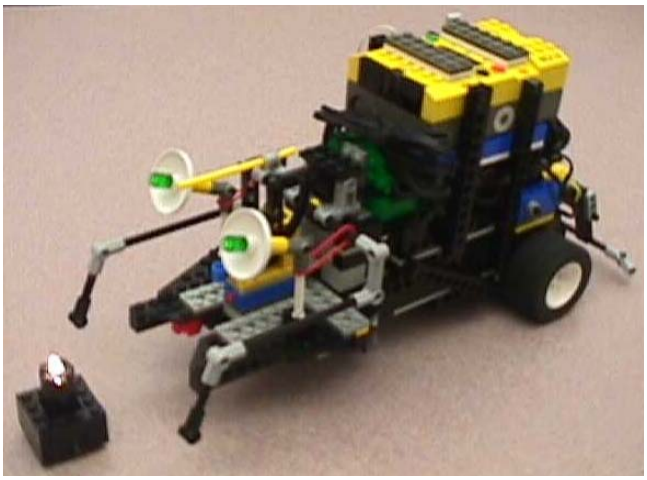
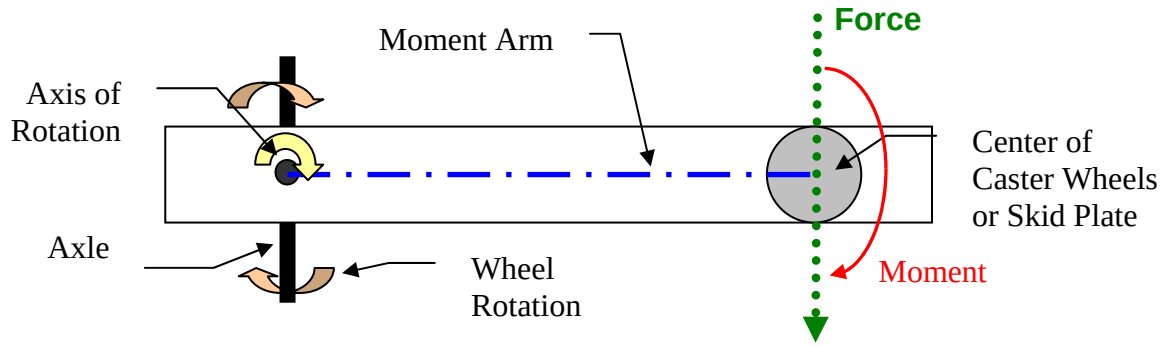


Figure 11: "Trap", Revised 3rd Architecture

These problems stemmed from the fact that the last two revisions were extensions of the second architecture rather than a complete redesign. The first problem was that the turning axle was still located at the very rear of the robot. Even when the added weight of the grabber arm was removed, the wheels at the very back had to create a large moment to move the structure (See Figure 12, below). Think of a very heavily loaded shopping cart and trying to turn it from the handle alone. This required a lot of power and often resulted in the wheels slipping – yielding an incalculable amount of error. Even when the robot did turn correctly, the dual-caster system induced more error as the caster wheels straightened out when the

robot moved forward after a turn. Second, the rotation sensor was geared 1:1 with the movement axle. This meant that every tick of the rotation sensor equated to 22.5 degrees of turn. The rear axle is 9.25 inches from the front. This means that the front could stop up to 3 inches from target position and the error would be undetectable. Next, the two main wheels were mounted to the robot structure inward of the two wheels. Outside of the wheels is free-floating. The LEGO axles tend to be flexible and any heavy weight (like the RCX loaded with 6 AA batteries) will cause them to bend (See Figure 9 “Bottom View”, above, and examine the rear wheels). This bending causes the robot to deviate from its desired path and the error is typically incalculable. Lastly, with the increased length due to adding the target capture arm, the area required for turning was once again very large. The robot’s length was about 10 inches. For a 180-degree turn, the grid spaces would have to be twice that (20 in). Thus, for only a 6 x 4 grid space, the robot would require an arena that was at least 10 ft x 6 ft 8 in. To correct for these errors, the next robot hardware design was built from scratch with elimination of these errors in mind.



A moment arm is defined as the perpendicular distance between the line of action of a force and an object's axis of rotation.

Moment (in*lb) = Force (lb) * Moment Arm (in) (Equation 1)

Force (lb) = Moment (in*lb) / Moment Arm (in) (Equation 2)

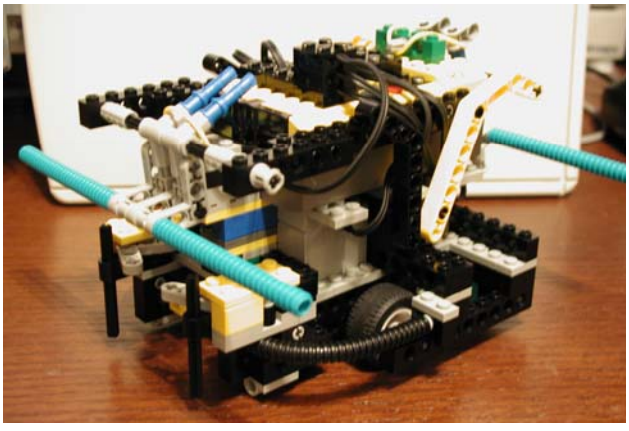
The same amount of force is required to overcome the coefficient of friction and move the robot. So, as the moment arm length increases, the moment itself must increase. In this case, the moment is the rotational movement caused by two wheels turning in opposite directions on the axle. Thus, to increase the moment, power to the two wheels turning must be increased.

Figure 12: Moments Explained

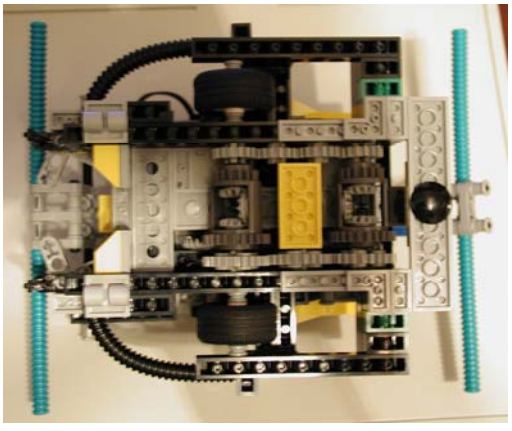
Fourth architecture, "Balanced"

The ability to turn in place is the biggest change to the fourth hardware architecture (See Figure 13, below). The single drive and turning axle was moved to the center of the robot while the dual-differential gear assembly expanded to the rear. Further, the motors are placed on top of the dual-differential assembly instead of fore and aft, making the robot shorter. Further, it enabled the creation of a gear train with a 3:1 ratio between rotation sensor rotations and drive axle rotations, allowing for 48 ticks per rotation (7.5 degree error). The RCX was placed to set the center of gravity of the robot just aft of the drive axle so that it

would rest nicely on a skid plate. While some kneeling did occur if the robot stopped abruptly, no errors such as slippage of the wheels was found due to this phenomena. The weight, bearing directly down on the center axle, also helped to reduce any slippage on the wheels even under a heavy load. Further, the wheels were mounted between two frame components. This eliminated the errors associated with the bowing of the axle. It is important to note that to make framing the wheels effective, some load must be transferred to the outside wheel frame or the horizontal pieces must be made sturdy enough such that the outside frame itself does not bend upward under load. The effect was such that the robot could turn in place, toward any direction and then move forward, avoiding positioning calculations for movement with a turning radius. Another modification to the fourth architecture is that the trap motor was removed. Additionally, the trap was redesigned to take advantage of the existing target's shape so that it would remain in the trap during turns. Finally, the backpack was removed to remain consistent with the original plan of using only commercial off-the-shelf pieces.



Side View



Bottom View

Figure 13: "Balanced", the 4th Architecture

With an earlier version of the return home behavior in the current architecture, the forward facing light sensor often interpreted any messages from Home as a bright light source if the robot was close. So, the robot would charge into Home trying to get the elusive target. While changing the height of the Home RCXs IR port helped a little, it did not alleviate the problem entirely. The problem was that, originally, Home constantly emitted an "over here" signal. When the robot entered into the return home state, it would look for this signal. This was modified to the Marco-Polo algorithm described above. Thus, Home now only emits a signal while the robot is using the return behavior.

CHAPTER IV

RESULTS

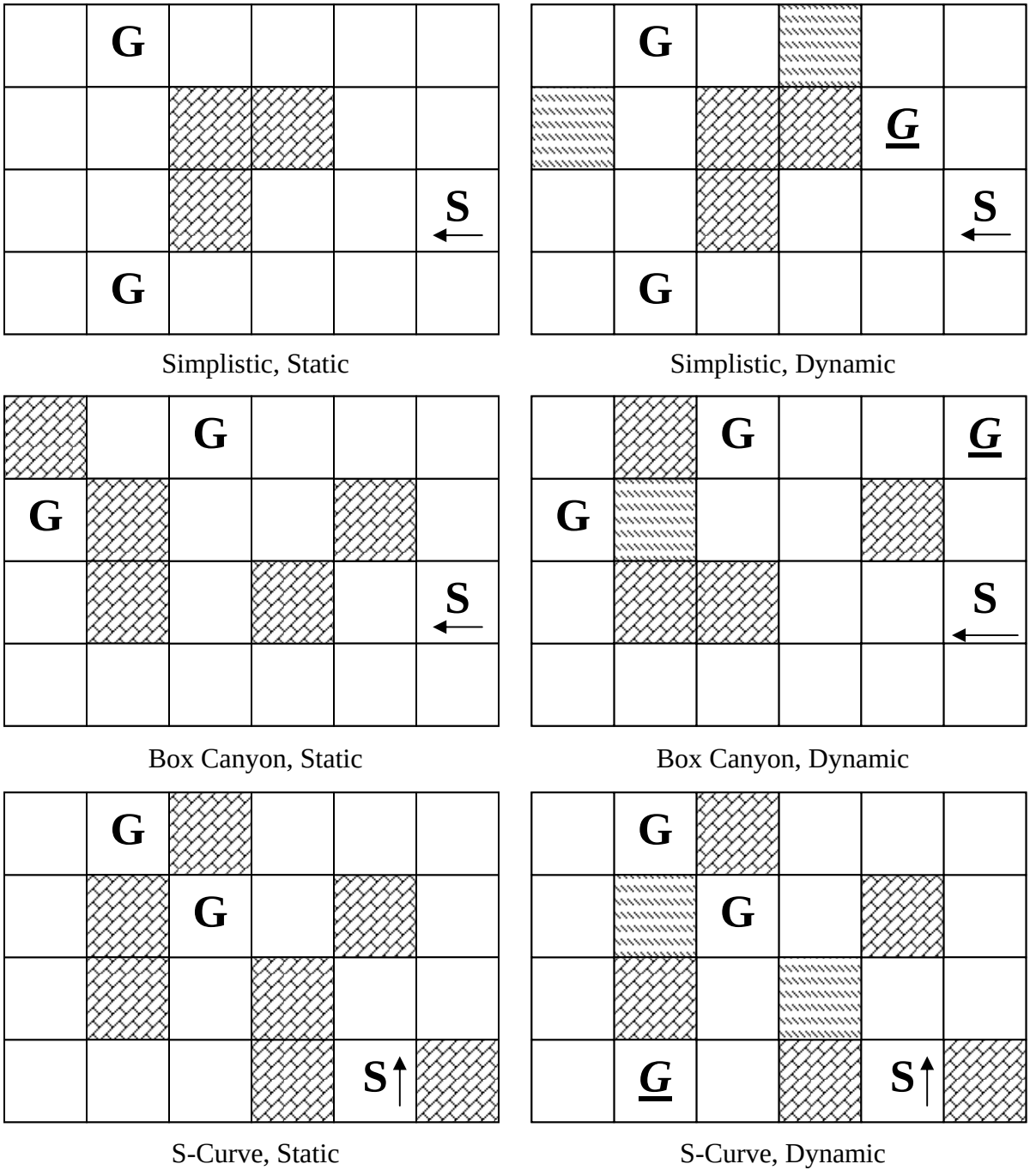
Test Run Results

To emphasize the capabilities of the deliberative robotic architecture and assess its weaknesses, a series of test runs were conducted to compare the performance of the reactive robotic architecture with the deliberative one. Six different arena layouts were developed with three general formats. The formats were a simplistic model with obstacles in the center; an S-Curve composed of a single, long corridor that meandered the whole arena; and a Box Canyon, an area with only one entry/exit with which reactive robotic architectures typically have some problems navigating. There were two obstacle approaches to each format – static and dynamic. The obstacle approach means very little to the reactive robotic architecture, which retains no world knowledge. However, for the deliberative robotic architecture, a static obstacle environment means that the *a priori* data contains the location of all obstacles throughout the entire run. In the dynamic obstacle runs, the deliberative robot is given the location of some of the obstacles or an obstacle may be removed. In short, the deliberative robot's plan is forced to fail somewhere during the run. All six of the arena layout patterns are presented below in Figure 14. A picture of each arena configuration is shown in Figure 15, below.

The reactive robotic architecture successfully completed only the Simplistic courses. The robot frequently got stuck in corners and box areas; though random timing functions did help it to work its way out. However, this consumed a lot of time. After 20 minutes, the S-

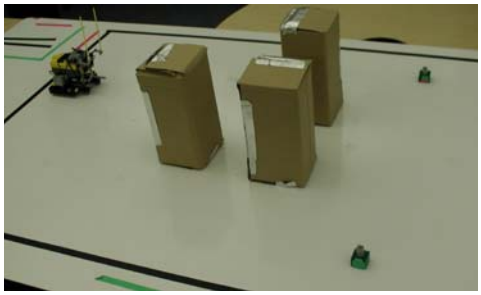
Curve and Box Canyon tests were deemed unsuccessful. It appears that the complexity of the other courses, with numerous turns, offered too difficult a challenge for a random motion robot. Had the reactive architecture used a wall-following technique, the results may have been more favorable.

The deliberative robotic architecture successfully completed all courses. The Simplistic courses were completed in much less time than it took the reactive robot, even with approximately 10 seconds per path planning event. As the robot followed the plan, localization remained somewhat accurate with severe errors only arising over long runs or runs with a lot of turns. To compensate, the deliberative robot's code was modified to stop after each target was delivered to allow the user to realign the robot. Other error corrections including nudging the target within a grid space to ensure it would be properly captured when the robot arrived. As the focus of the project was to understand the limitations of the deliberative robotic architecture using the RCX, it was felt that the robot should not be allowed to completely fail a course if the localization kept the robot at least within the same grid space. A drawback of the deliberative robotic architecture is that it is less likely to find unknown targets that are away from paths to known targets. With this in mind, after the deliberative robot completed the test runs, it was concluded that the RCX could be successfully used as a teaching tool with some modifications to the robot and exercise requirements.

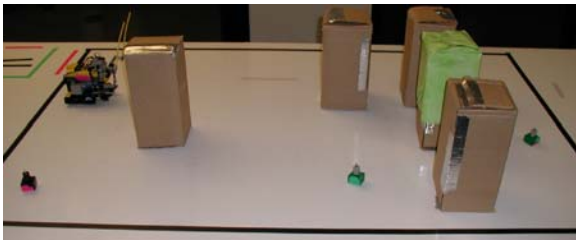


Key: **G** – Known Goal, ***G*** – Unknown Goal, **S** – Start Location (arrow indicates heading)
– Known Obstacle, – Unknown Obstacle

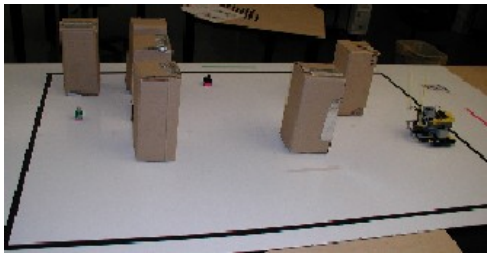
Figure 14: Arena Layout Patterns for Test Runs



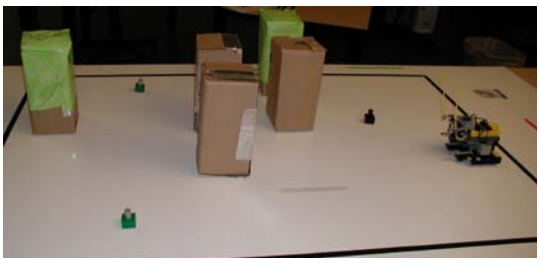
a) Simplistic, Static



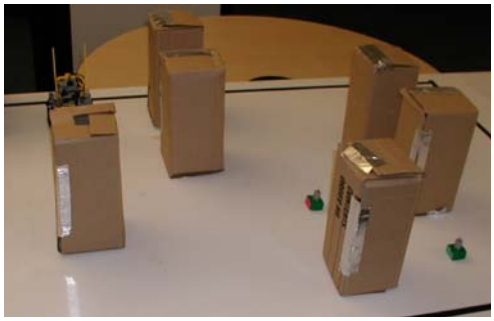
b) Simplistic, Dynamic*



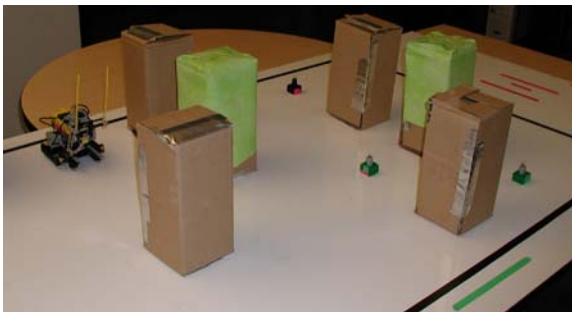
c) Box Canyon, Static



d) Box Canyon, Dynamic*



e) S-Curve, Static



f) S-Curve, Dynamic*

*Note: Green obstacles in dynamic arenas are unknown to the robot at the start of the run.

Figure 15: Project Arena Layouts

Classroom Application

Once it was determined that the deliberative robotic paradigm could be successfully implemented on the LEGO RCX, plans were made to incorporate material into SIUEs introductory artificial intelligence course. The course already uses the RCX to teach

introductory robotics but all of the laboratory assignments use the reactive paradigm. New instruction was designed to allow for both reactive and deliberative robotic assignments. The material was presented to a class of 23 students during the Spring 2003 semester.

Lectures and robotics projects

The robotic material involved five lectures and assignments (see Appendix C). The first lecture covered the basics of what a robot is, discussed the LEGO RCX platform, the IC software environment, and LEGO building components and techniques including sensors, motors, gearing, and structural considerations. Students were also given their first robotics assignment.

The second lecture covered reactive robotic architectures. The lecture included discussions on SENSE-ACT, emergent behavior, the pros and cons of reactive control, and behavior coordination. A class exercise, requiring 4 students, was also conducted. One student serves as the visual system, another decides the actions, and the other 2 each serve as a single arm. For this exercise, the student who declares the actions is blindfolded and the two students acting as arms are seated in front of a table. On the table are three boxes. The goal of the task is to have the students stack the three boxes on top of each other, in order of size. The action generator may only ask questions of the visual system and may only give commands to the arms. The intent is to provide students with an idea of how sensors, processors, and actuators are separate entities and some of the challenges of coordinating their actions to complete a goal. At the end of the lecture, the second robotics project was handed out.

The reactive robotic paradigm instruction continued during the third lecture. The topic of behavior coordination was completed. Potential fields, animal behaviors as models for reactive control, FSA diagrams and Stimulus-Response diagrams were discussed. At the end of lecture, the students openly discussed and completed a sample FSA problem in class.

The fourth lecture introduced deliberative robotic control. The lecture included a comparison between deliberative and reactive robotic control, SENSE-PLAN-ACT, world models, knowledge representation, the pros and cons of deliberative control, short-term memory versus long-term memory, and wavefront planners. There was also a brief discussion on hybrid control and different approaches to fusing reactive and deliberative parts. An in-class demonstration involved setting up a short maze of chairs in the classroom. A student volunteer was allowed to count steps and create a plan for following the maze. Then, the student was blindfolded and had to execute the plan. During execution, the maze was modified. A second individual and the instructor spotted the student to be sure no one was hurt. The demonstration highlighted the SENSE-PLAN-ACT process. In retrospect, the example may have been made safer by repeating the box stacking demonstration but with one student. The student would be allowed to see the location of the three boxes and make a plan for stacking. During execution, the student would be blindfolded and two boxes would be switched.

The final lecture focused on localization and the development of a deliberative robotic control. The localization information discussed both landmarking and deductive reckoning. The development topics included a walkthrough of how a deliberative architecture might be created. This project's robot was used as a model.

Students were divided into teams of two for the robotics projects. IC4 was used as a programming language. IC4 uses a runtime machine language module. It replaces the standard LEGO firmware with its own firmware that has a smaller memory footprint. This freed up program memory space for the deliberative program's world map. Since most of SIUE programming courses use the C programming language, the students were already familiar with the syntax.

The intent of the first project was to familiarize the students with the robot-building environment and garner further interest in robotics through a competition during demonstrations. Students were required to build a line-following robot and demonstrate that it could follow a meandering black line. Bonus points were given to the team that could successfully complete the line-following project in the least amount of time.

The second and third robotics projects were designed to mimic the foraging robots created for this project. The second involved development of a reactive robot and the third, a deliberative one. However, the scope of the effort was much reduced to ensure that the focus was on the topics being taught. This included making targets and obstacles colored tape on the arena floor. Thus, both could be sensed with a single, downward-looking light sensor. Since the targets were tape, they were no longer required to be returned to a starting position, simplifying the construction of the robot by not requiring a target trap and reducing programming complexity by not requiring a return trip. Given the localization capabilities of the robot, it was felt that a return trip would compound errors, reduce the students' satisfaction with their results, and provide nothing in terms of a better understanding of localization and planning that could not be gotten from the trip to the target.

Given the revised approach, students were assigned the task of designing, building, coding, and demonstrating a reactive foraging robot for the second project. The robot was not allowed to cross obstacle lines and had to provide an audible indication and stop when it had found a target. It should also be noted that the arena contained physical obstacles as well. This required the students to create bumpers with a touch sensor. The intent was to require them to become familiar with multithreading in the IC environment. Having two different sensors required them to monitor both as the robot moved about the arena.

For the third and final robotics project, students had to develop a deliberative robotic architecture that could successfully plan and navigate to a target. While students were told about the upcoming deliberative project and recommended to build a hardware architecture for the reactive assignment that could be reused, most had to rebuild the robot to meet the more stringent requirements required for localization. It should also be noted that the deliberative tests only involved one known target and included the dynamic additions of unknown obstacles and targets. When an unknown obstacle was encountered, the students' robots were required to stop, replan, and continue without passing through the new obstacle grid space. When a new target was encountered, the robots were required to audibly acknowledge the new target but could continue to follow the plan through the grid space to the original known target. As the robot localization was expected to be perfect, students were allowed to give their robot a maximum of three nudges to ensure it did not veer to far off course during the run. To encourage the students to consider as many of the causes of localization errors as possible, bonus points were given to the team that used the least amount of nudges.

Classroom results

The teams successfully built robots for all three of the projects. In most cases, it was a different robot for each project. While having students build the robots is good for learning first hand about the causes of physical errors and sensor limitations, building three different physical architectures over the course of five weeks is a waste of valuable time the students should be spending on understanding the concepts and implementing them via code. Many students felt frustrated at having to start over or modify what already existed between the reactive and deliberative foraging projects. Some students also mentioned some frustration with the programming language.

As mentioned above, the students were required to use IC4 to program their robots. The language was new to all of the students. However, they were familiar enough with C syntax to learn the basics of IC4 fairly quickly. Most were able to properly make use of multiple processes for the reactive and deliberative projects. The largest complaint from the students was the lack of a debugging environment similar to what they had available to them under Microsoft Visual C++.

Despite any complaints, all but a few students successfully demonstrated the ability to understand, design, and construct a deliberative robotic architecture using the LEGO RCX and standard LEGO components. In fact, during the deliberative demonstrations, at least four teams mentioned that they took it as a challenge to use as few nudges as possible, incorporating features such as accounting for gear slop when changing direction of motion. Three of these teams used only one nudge while the other used none. All but one team was

ultimately successful in planning and navigating their robot to the known target while avoiding all obstacles.

At the end of the robotics lectures and assignments, a feedback survey was handed out to all of the students. The results showed that a majority of the students felt that the lecture material and projects were worthwhile. They also stated that the deliberative project helped them better understand deliberative robotic control. In one question, the students were asked if a hybrid robotic control project would have been more appropriate. The intent was to ask the students if a hybrid control project should have been done in lieu of the individual reactive and deliberative projects. Responses indicated that the question was not phrased well, as many students stated that they felt there is little time for another project along with the other three. A more detailed summary of student responses may be found in Appendix D.

Conclusion

By using the LEGO RCX to develop a deliberative robot and allowing students to do the same, it has been shown that the RCX platform and standard off-the-shelf components are usable as an effective teaching tool for a introductory lesson in the deliberative robotic paradigm. Most of the failures and difficulties of the project were a result of the constructed environment and self-imposed complexities. In hindsight, the reactive portion of the project added little to the support of the final results. The initial thought was that a comparison of the two architectures would better explain the results. However, most people were at least somewhat familiar with the reactive paradigm or at least what it embodied. Thus, in the end, no direct comparison was really necessary. Omitting the reactive robot would have simplified

the arena and allowed more freedom in tailoring the hardware design for the deliberative architecture. That is not to say that the deliberative robot would have worked perfectly.

One of the results frequently pointed out is that there was loss of localization even with the added safeguards of reducing immeasurable errors such as accounting for gear slop and using a balanced construction. This loss of localization is beneficial in demonstrating to the students that the real world is a harsh place. Integrating a physical agent with a computer program into the real world is no easy task. However, the robot can do the job well enough to ensure that the basic concepts of what a deliberative robotic architecture is, is enforced in their minds.

Along those same lines, the complexity levied on the project robots is not required for student instruction. The revised, simpler assignment using colored tape was very effective and offered enough of a challenge within the allotted time. If the project was made more complex, such as by requiring the students to build a trap mechanism and drag a physical target around the arena, it would have detracted from the core lessons of designing a robot that could sense, plan, and act.

While the LEGO system can be designed to remove most immeasurable errors, deductive reckoning is only so reliable. Some external form of reference is required for improved localization. The use of sonar or a compass sensor would greatly improve the localization of the deliberative robot. However, this would be best left to a course in advanced robotics or one where building such a sensor is part of the instruction. Enough information exists on the web detailing construction methods and interface design into the RCX to construct these sensors. The other option is to use a more powerful robotic platform

better capable of supporting these advanced sensors. For the content of SIUEs current Introduction to Artificial Intelligence course, the standard LEGO suite is suitable.

Future Work

Over the course of this thesis project, a number of features arose that would have been nice to incorporate or may be useful to explore for future student projects. These are listed below as a reference for those interested in pursuing them.

1. To reduce the time that students spend building robots during projects, change the project format. The line following robot or a similar robot should remain to allow the students an easy introduction to the hardware and programming environment. However, the other projects should be combined. The assignment should consist of a hybrid control architecture presented in two phases. The first phase encompasses designing and building a purely deliberative control architecture. The second phase integrates reactive control into the existing deliberative robot. In this way, the students would build the second robot to the more stringent deliberative robotic control requirements. The lectures would be revised to provide deliberative control instruction before the reactive.
2. The current map is devised as a grid with obstacles and all other objects occupying the grid vertex. If obstacles representation is changed to the edges between grid vertices, then one-way paths could be developed. The drawback to this approach is that for each vertex additional memory would be required to retain the connection of each vertex.

3. Currently, when the deliberative robot finds a new target, it backs up, adds the new target location to the goal array and then develops an entirely new plan, including finding the closest target. The robot program should be modified to set the newly discovered target as the closest target and plan from there. Or, if a hybrid approach is to be used, allow the robot to capture the new target and immediately plan a path back to the start position.

4. Once the current deliberative robot has returned the last known target to the start position, it stops. The robot could be programmed to examine neighboring grid spaces along its planned path for unknown targets. Alternatively, the robot could maintain knowledge of which grid spaces it passed through for all target runs and, after it has captured the last known target, devise the most efficient path to explore all of the unknown grid spaces.

WORKS CITED

[1] Arkin, Ronald C. Behavior-Based Robotics. Cambridge: MIT Press, 1998.

[2] Baerveldt, Albert-Jan, Tommy Salomonsson, and Björn Åstrand. "Vision-Guided Mobile Robots for Design Competitions." IEEE Robotics & Automation Magazine. vol. 10, no. 2. June 2003: 38-44.

[3] Baum, Dave, et al. Extreme Mindstorms: An Advanced Guide to LEGO Mindstorms. Berkeley: Apress, 2000.

[4] Druin, Allison, and James Hendler, ed. Robots for Kids: Exploring New Technologies for Learning. San Francisco: Morgan Kaufmann Publishers, 2000.

[5] Fagin, Barry. "Ada/Mindstroms 3.0." IEEE Robotics & Automation Magazine. vol. 10, no. 2. June 2003: 19-24.

[6] Ferrari, Mario, and Giulio Ferrari. Building Robots with LEGO Mindstorms: The Ultimate Tool for Mindstorms Maniacs!. Ed. Ralph Hempel. Rockland: Syngress Publishing, Inc., 2002.

[7] [For Inspiration and Recognition of Science and Technology (FIRST)]. "Welcome to the FIRST LEGO League: Sports for the Mind." [Resource on-line]; available from http://www.usfirst.org/jrobtcs/flego.htm; Internet; accessed July 2003.

[8] Greenwald, Lloyd, and Joseph Kopena. "Mobile Robot Labs." IEEE Robotics & Automation Magazine. vol. 10, no. 2. June 2003: 25-32.

[9] Hougen, Dean F. "Project 2 -- Hybrid Deliberative/Reactive Systems." [Resource available on-line]; available from http://www.cs.ou.edu/~hougen/classes/Spring-2002/Robotics/materials/project2.html; Internet; accessed June 2003.

[10] Klassner, Frank, and Scott D. Anderson. "LEGO Mindstorms: Not Just for K-12 Anymore." IEEE Robotics & Automation Magazine. vol. 10, no. 2. June 2003: 12-18.

[11] Moh, Chang-Hue. "6.836 Final Project: Evolution in the Micro-Sense: An Autonomous Learning Robot." [Resource no longer available on-line]; previously available from http://www.pmg.lcs.mit.edu/~chmoh/836-project/final_report.pdf; Internet; accessed June 2003.

[12] Murphy, Robin. Introduction to AI Robotics. Cambridge: MIT Press, 2000.

[13] Piepmeier, Jenelle A., Bradley E. Bishop, and Kenneth A. Knowles. “Modern Robotics Engineering Instruction.” IEEE Robotics & Automation Magazine. vol. 10, no. 2. June 2003: 33-37.

[14] Resnick, Mitchel, et al. “Digital Manipulatives: New Toys to Think With.” Proceedings of the SIGCHI Conference on Human Factors in Computing Systems. [Association for Computing Machinery].18-23 April 1998. 281-287.

[15] Russell, Stuart, and Peter Norvig. Artificial Intelligence: A Modern Approach. Upper Saddle River: Pearson Education, Inc., 2003.

[16] Weinberg, Jerry, and Xudong Yu. “Robotics in Education: Low-Cost Platforms for Teaching Integrated Systems.” IEEE Robotics & Automation Magazine. vol. 10, no. 2. June 2003: 4-5.

APPENDIX A

WAVEFRONT PROPAGATION ALGORITHM

WAVEFRONT PROPAGATION ALGORITHM

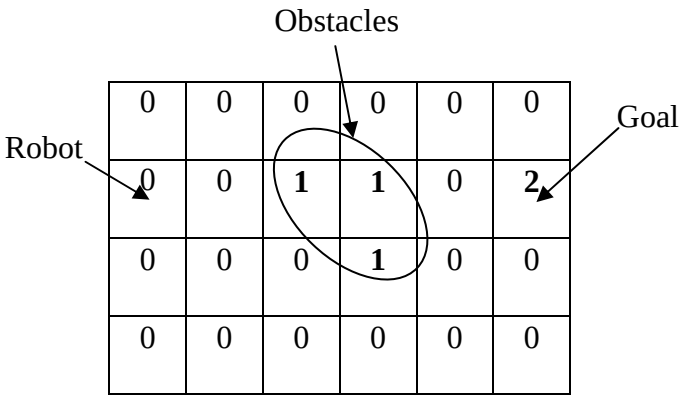
The wavefront propagation algorithm is a path-planning algorithm. It is most easily applied to maps using graph representations. Typically, the algorithm treats the grid as a conductive material. Heat radiates from the initial space to the goal space. Obstacles have a conductivity of 0. The conductivity of other items varies dependant upon the ability to traverse the area. The benefits of using the wavefront planner are that it is simplistic and not very memory or processor intensive. Its drawbacks are that it is not very efficient; every node needs to be visited to find the most optimal solution. Additionally, more than one path usually results [12].

This project uses a variant of the wavefront propagation planner. First, the goal with the shortest path between it and the robot is found. Then, before wave propagation, the obstacles and closest goal location are each assigned a unique value – 1 and 2, respectively. The wave then starts at the goal coordinates and propagates outward from that point. For this project, the robot could only move vertically and horizontally. Therefore, the wave propagation was restricted to the same directions. Each non-obstacle grid coordinate is assigned a value based upon distance from the goal location (usually in increments of 1). When wave propagation is complete, the robot's current position should have a value assigned to it. From there, the robot follows a path of decreasing value back to the goal position. The pseudocode for the project's wavefront algorithm is shown in Figure 16, below. An example follows in Figure 17, below.

```
1  WavefrontPropagation(Grid[ ])*
2          updates = TRUE
3          while updates = TRUE
4              updates = FALSE
5              for each Grid[row, col]
6                 if Grid[row, col] > 1
7                    if Grid[row-1, col] = 0 or
                        Grid[row-1, col] > Grid[row, col] + 1
8                     Grid[row-1, col] = Grid[row, col] + 1
9                     updates = TRUE
10                   if Grid[row+1, col] = 0 or
                        Grid[row+1, col] > Grid[row, col] + 1
11                    Grid[row+1, col] = Grid[row, col] + 1
12                    updates = TRUE
13                   if Grid[row, col-1] = 0 or
                        Grid[row, col-1] > Grid[row, col] + 1
14                    Grid[row, col-1] = Grid[row, col] + 1
15                    updates = TRUE
16                   if Grid[row, col-1] = 0 or
                        Grid[row, col-1] > Grid[row, col] + 1
17                    Grid[row, col+1] = Grid[row, col] + 1
18                    updates = TRUE
```

* Assumes *Grid*[ ] is pre-processed to include a goal coordinate set to 2, obstacles set to 1, and the remaining grid coordinates set to 0.

Figure 16: Wavefront Propagation Pseudocode



0	0	0	0	0	0
0	0	**1**	**1**	0	**2**
0	0	0	**1**	0	0
0	0	0	0	0	0

a) Pre-propagation

0	0	**6**	**5**	**4**	**3**
0	0	**1**	**1**	**3**	**2**
0	0	0	**1**	**4**	**3**
0	0	0	**6**	**5**	**4**

b) Mid-propagation

8	**7**	**6**	**5**	**4**	**3**
9	**8**	**1**	**1**	**3**	**2**
10	**9**	**8**	**1**	**4**	**3**
9	**8**	**7**	**6**	**5**	**4**

c) Post-propagation

8	**7**	**6**	**5**	**4**	**3**
9	**8**	**1**	**1**	**3**	**2**
10	**9**	**8**	**1**	**4**	**3**
9	**8**	**7**	**6**	**5**	**4**

d) One Possible Path Generated

Figure 17: Wavefront Algorithm Propagation Example

APPENDIX B

DIFFERENTIAL GEARING

DIFFERENTIAL GEARING

A differential gear relays motion from an input shell to the summation of two output axles. The purpose of the differential gear is to allow one wheel on the same axle as another to move at a different speed when turning. This is required, for the outer wheel in a radius turn travels further in the same amount of time as the inner wheel (see Figure 18, below). A single differential is typically used in vehicles equipped with two axles – one containing a steering mechanism and one that follows the turn (such as a car's front wheels and rear wheels, respectively). The differential is used on the axle that follows the turn. Without a differential, the wheels on the axle that follows the turn would drag causing the wheels to skid, and, in terms of robot localization, create immeasurable errors.

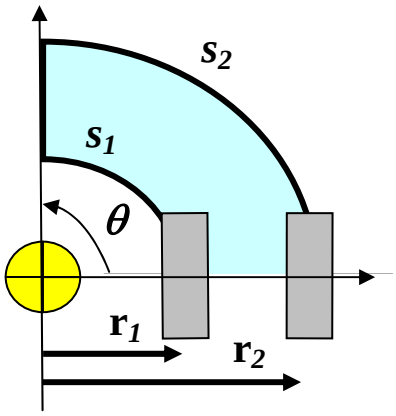


Figure 18: Radius Turn

The length of an arc, *s*, is equal to product of the angle of the turn, θ (in radians), and the radius of the wheel's turn, *r*. Mathematically,

$$s_n = \theta_{rad} \cdot r_n . \qquad \text{(Equation 3)}$$

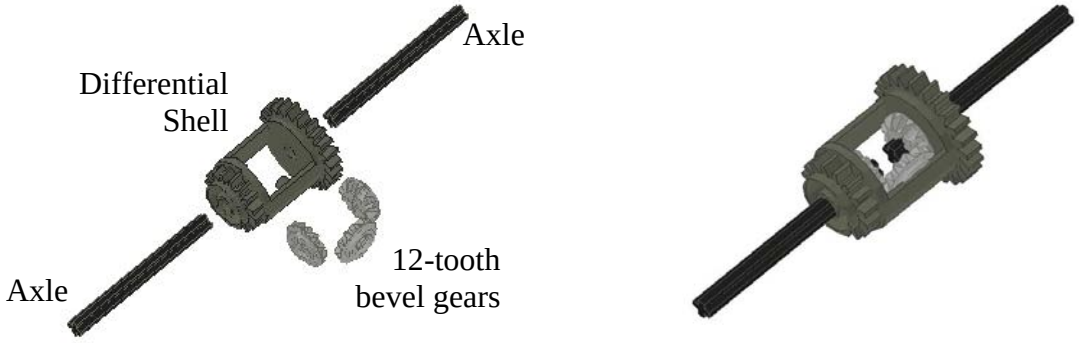
Rearranging Equation B.1, it is calculated that $\theta_{rad} = \frac{s_n}{r_n}$. Thus, $\theta_{rad} = \frac{s_1}{r_1}$ and $\theta_{rad} = \frac{s_2}{r_2}$. Since the angle, θ, is the same for both wheels, the two equations are equal to each other. Then, rearranging to put like terms on each side yields

$$\frac{r_2}{r_1} = \frac{s_2}{s_1} \qquad \text{(Equation 4)}$$

In summary, the ratio of the radius of the turn for each wheel is equal to the ratio of the distance traveled (arc length) for each wheel. As the outer wheel has a greater radius of turn, it must also have a larger distance to travel. Additionally, as it must travel a farther distance in the same amount of time as the inner wheel, it must travel at an increased velocity in order to do so. The later can be determined from the equation Velocity = Distance / Time. Since the time to complete the turn is equal for both wheels then it can be determined that the wheel traveling a farther distance must also be traveling at a greater velocity. In order for two wheels on the same axle to travel at different velocities, a differential is required.

The LEGO differential shell is used in conjunction with 3 12-tooth (12t) bevel gears and 2 axles (see Figure 19, below). The differential is driven through either the 24-tooth (24t) or 16-tooth (16t) gears that are part of the differential shell. When the differential shell turns, the center 12t bevel gear rotates with the shell. If both of the axles are unloaded (or equally loaded) then the central 12t gear drives the two outer 12t bevel gears to turn in the same

direction, at the same speed. Thus, turning both axles in the same direction, at the same speed. However, if the movement of one of the axles was stopped (being held, for instance) then the differential shell would still move but the central 12t gear would spin about the gear attached to the stopped axle. The motion of the central gear would then drive the remaining 12t gear even faster. Thus, one wheel would rotate *differentially* with respect to its counterpart.



a) Pre-assembled Differential b) Assembled Differential

Figure 19: LEGO Differential Gear Assembly

Mathematically, the angular velocity (speed and direction) of the differential shell is equal to the average angular velocities of the two axles as shown in Equation (4), above.

$$\omega_{shell} = \frac{(\omega_{Axle\_1} + \omega_{Axle\_2})}{2} \qquad \text{(Equation 5)}$$

where ω_{shell} is the angular velocity of the differential shell and ω_{Axle_1} and ω_{Axle_2} are the angular velocities of the two axles. Thus, if it is known that each wheel is turning at angular velocity *x*, then it is known that the differential shell is also turning at angular velocity *x*, since $\frac{(x+x)}{2} = x$. Similarly, if it is known that the differential shell is turning at *y*, then the

average angular velocity of the two axles must also be *y*. However, the angular velocities of the individual axles would be unknown. If the differential shell is stationary, it means one of two things. Either both of the wheels are stationary or both wheels are turning at the same speed but in opposite directions, as $\frac{(x+(-x))}{2}=0$.

While a single differential is useful for radius turns, it is difficult to apply it for assured straight-line motion. The reason for this is that, if at any time some force acts upon one axle and not equally upon the other (like dirt on the floor), the motion will be immediately translated through the differential gears and cause the uninhibited axle to turn even faster. With one wheel turning fast and another slow, the vehicle will turn from a straight path. Further, using a single differential forces the added complexity of an additional axle with a steering mechanism. A more effective approach for some robots is the use of two motors, each connected to a wheel and each having its own rotation sensor. Then, an error correction method like Proportional, Integral, and Derivative (PID)[9] could be used for error correction. Since the wheels are independent, they could also be used for turning. However, there is an easier way to assure straight-line motion and the ability to turn with a single axle – the dual-differential.

The dual-differential is a gearing of two differentials within the same gear train. One differential controls linear motion and the other differential controls turning with a separate motor driving the rotation of each differential. The wheel axles pass through the linear differential. The differential gears on the turn differential are connected to the wheel axles

[9] PID is an error correction method that provides a correction response based upon the amount of current error, the total amount of error over time, and the change in error.

through a series of idler gears[10] – an even number on one side, an odd number on the other. The difference is essential to ensure that when the turn differential is rotated, the motion is translated to the wheels with equal but opposite angular velocity. A diagram of a dual-differential configuration is below in Figure 20.

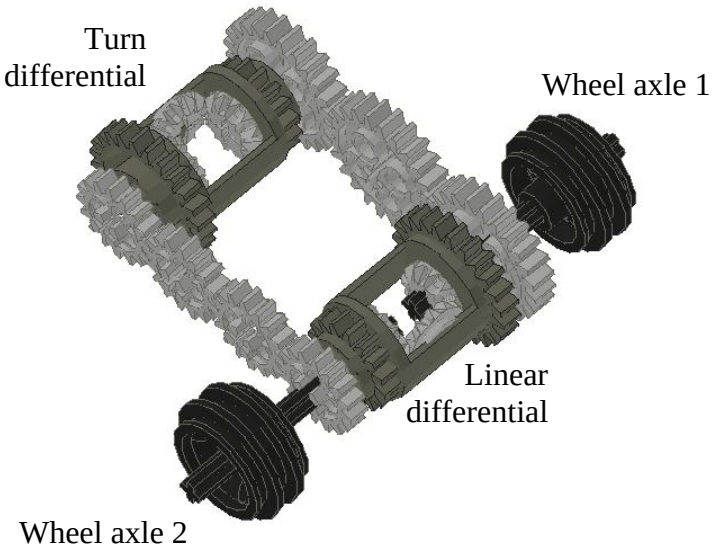


Figure 20: LEGO Dual-Differential

The benefit that the dual-differential offers is three-fold. First, if one differential shell is held steady (by braking the motor, for instance) and the other differential is rotated, equal power is assured to both wheels even if a force acts unequally upon the wheels. Second, both linear motion and turning can be achieved using the same axle. To avoid the errors and complications that a steering mechanism causes, a skid plate can be used on the back of the

[10] Idler gears are gears that do not modify the velocity/torque of the gear train but serve only to change the direction of motion.

robot. Third, if both differentials are rotated at the same time, then a radial turn can still be achieved.

When one differential shell is rotated and the other is held steady, equal power to both wheels is assured because the wheels are physically connected via the gear train that passes through each differential. For instance, if the linear differential is driven forward and the turn differential shell is held, then the situation is similar to a single differential. But, the idler gears would cause the two side 12t bevel gears inside the turn differential to rotate at equal speeds but in opposite directions. This, in turn, would cause the center 12t bevel gear to spin at the same speed as the two side bevel gears. With the turn differential held steady, the two side bevel gears are unable to spin at different speeds because they are both connected to the same center bevel gear. This ensures that the wheel axles spin at the same speed.

Further, if a portion of the dual-differential system is isolated and just the turn differential and its two axles are viewed, then according to Equation (5), the angular velocity of the shell must be 0.[11] Why then is it necessary to "hold" this other differential? As mentioned above, if an outside force acts upon a wheel/axle connected to a differential, then the differential just transfers more power to the wheel/axle that is free. If the opposite differential were not held, then this change would cause a difference in the speeds of the two wheel axles. When the two side bevel gears try to drive the same center bevel gear at different angular velocities, there is an imbalance. The difference in the angular velocities of the two side gears is now translated from the center bevel gear to the differential shell,

[11] Recall that the idler gears change the direction of motion between one wheel axle and the turn differential. Thus, the equation changes to be the difference between the angular velocity of two wheel axles, not the sum.

causing it to turn. In essence, the central bevel gear rotates at the speed of the fastest side bevel gear. The center gear then forces the shell to rotate around the slower bevel gear at the difference of angular velocity between the two side bevel gears. The exact motion of the wheels would be unknown, the same situation as if it were a single differential. However, by holding the turn differential shell, the center bevel gear cannot translate the power. This forces the linear differential to maintain equal power to both wheel axles.

The above holds true for both turning and linear motion. If the linear differential is rotated and the turn differential shell is held, then the motion translates through the bevel gears within the turn differential. As explained above, this would provide equal angular velocity to the wheel axles in the same direction as the linear differential shell is turned. Similarly, if the linear differential shell is held and the turn differential is rotated, then the motion from the turn differential is translated through the bevel gears within the linear differential shell. This situation provides the wheel axles with the same speed, but in opposite directions. If the direction of rotation of the turn differential shell were reversed, then the two wheels would also reverse direction. As the wheels are turning at equal speeds but in opposite directions, the robot turns in place.

By turning both differentials at the same time, a turn with greater than a 0° radius can be achieved. Above, it was explained that when one differential shell is rotated, equal power is translated to both wheel axles when the other differential is held steady. For rotational motion to the linear differential, the wheel axle rotation is in the same direction; for the turn differential, it is in the opposite direction. Therefore, mathematically, it can be written

$$\omega_{Axle\_1} = \omega_{Axle\_2} = \frac{\omega_{linear\_shell}}{2} \quad \text{(Equation 6)}$$

when the linear shell is rotated and the turn differential is held. Similarly,

$$\omega_{Axle\_1} = -\omega_{Axle\_2} = \frac{\omega_{turn\_shell}}{2} \quad \text{(Equation 7)}$$

when the turn differential is rotated and the linear differential is held. To calculate the effect on the wheel axles when both differentials are rotated, the above two equations for each respective wheel axle can be summed as follows:

$$\omega_{Axle\_1} = \frac{\omega_{linear\_shell} + \omega_{turn\_shell}}{2} \quad \text{(Equation 8)}$$

$$\omega_{Axle\_2} = \frac{\omega_{linear\_shell} - \omega_{turn\_shell}}{2} \quad \text{(Equation 9)}$$

With the above knowledge, the robot can be coded to rotate each differential appropriately. How finite of a control there is over the turn will depend largely upon the amount of motor speed control provided by the programming environment used. What is required is the desired turn angle, the inside turn radius, the width of the wheelbase (to calculate the outside turn radius), and the diameter of the wheel. In addition, a rotation sensor is required to be attached somewhere between each motor and differential gear train.

The goal is to calculate velocity ratios required of each motor to execute the desired turn. The exact velocity is not required unless there is a specific time in which the turn must be executed. Since the time to complete the turn for each wheel must be the same, the ratio of the two arc lengths, s_1 and s_2, is equal to the ratio of ω_{Axle_1} and ω_{Axle_2}. Per Equation (4), this is the ratio of the two turn radii, r_1 and r_2. Next, take the desired turn angle and a turn radii

and calculate the arc length using Equation (3). Since the ratios are all inter-related, only one arc length needs to be calculated. The next step is to calculate the circumference of each wheel. The number of wheel rotations required to execute the turn is the arc length divided by the circumference. The number of rotation sensor ticks per wheel rotations should be known. Multiply this value by the number of wheel rotations to complete the turn and this yields the number of ticks upon which the turn has been executed.

APPENDIX C

STUDENT ROBOTIC ASSIGNMENTS

STUDENT ROBOTIC ASSIGNMENTS

The following pages are copies of student robotic projects assigned as a part of SIUEs undergraduate Introduction to Artificial Intelligence course, CS 438, in Spring 2003. These assignments were developed as a part of this thesis project to determine if the results of this project could be successfully applied to a classroom environment. The goal was to enable students to develop deliberative robotic architectures using the LEGO RCX and commercial off-the-shelf LEGO components; thereby enhancing their understanding of the subject matter.

CS 438 – Artificial Intelligence
Interactive C Robotics Assignment 1 (RA1)

This assignment is designed to give you some experience with the Interactive C programming environment and building your own robot. You will build and program a robot that follows a wavering black line across a mat from a start to a finish point. Your robot should successfully manage the line in the most time efficient manner possible.

Items to Note:

- Your robot must fit and start completely within the start box on the mat.
- A part of the path has been taped because white lettering appears underneath. In the past, this has caused some robots to miss the line. As such, your light sensors need to be far enough up off of the mat as to avoid hitting the tape edges. Should anyone encounter trouble because of the tape, please inform Gary.
- The robots will be tested in EB 2029. The lights will be dimmed slightly to reduce glare off of the glossy mat. However, it is recommended that you program your robot to calibrate its light sensor beforehand and adjust threshold values accordingly. A marker will be placed on the room dimmer to indicate the light setting that will be used during the trials.
- The loop may be taken either direction – clockwise or counterclockwise.
- Your robot need not do anything special (stop, beep, whatever) once it enters the finish zone, as it will be difficult to tell the zone edges from a line that it was following.
- The mat will be available for use in EB 2029 for testing purposes. **Do not remove it from EB 2029.**

This assignment is due Wednesday, 02 April 2003 at 1:29PM and is worth 100 points. Any late submissions will receive a late penalty and be ineligible for bonus points. Turn in a hardcopy of your program (the *.ic file(s)) at class time on the due date. Name your program RA1-[*each teammate's initials*].ic and place it in the dropbox **under Assignment 6**. If your code spans multiple files, either zip them or place them all into one text file with the start and end of each individual file clearly marked. The assignment only needs to be submitted once by a single teammate. Be sure to place ALL teammate names in the comment blocks at the start of the program to ensure proper credit is received.

Each team will be given a single trial to complete the line following course. The robot **must** be programmed with a **1 second delay** after pressing the *Run* button. This will ensure that the robot is not accidentally thrown off course by moving while someone's hand is in the way. If the robot does not complete the course the first time, the team may elect to make a second attempt with a 5-point penalty. Only one additional attempt is allowed.

Robots that follow the path around and successfully complete it will receive a base score of 80 points. If the loop is successfully traversed, an additional 10 points will be given. The additional 10 points will be awarded based upon how well the robot is constructed and the logic in your robot's program.

The robot demonstrations / competition will occur in EB 2029 during the last half of class time on 02 April.

Bonus: A record will be kept of the time that it takes every robot to successfully complete the line following course. A bonus of 10 points each member will be given to the team whose robot has the best time score and can successfully navigate the loop. A bonus of 5 points each team member will be given for the fastest robot that was unable to navigate the loop BUT was able to complete the path in less time than the second fastest robot that did take the loop.

Please contact Gary Mayer, gmayer@siue.edu, with any questions regarding Interactive C or robot construction.

CS 438 – Artificial Intelligence
Interactive C Robotics Assignment 2 (RA2)

This assignment is designed to give you experience with a reactive robotic architecture. You will build and program a robot that emulates a scavenging animal. The robot will explore an area looking for "food" and avoiding obstacles (such as a wall). Your robot should successfully find the "food" in the most time efficient manner possible using a reactive robotic architecture.

Items to Note:

- The arena is a 40" x 60" space with a white background. You can think of this as a 4 x 6 grid of squares, each with 10" sides.
- Your robot must fit entirely within a single grid square. To do so, it must be no larger than 8" wide and deep. The grid space may be occupied by tape along their borders, making them less than the original 10" dimension.
- Black tape will be used to emulate obstacles and green tape will emulate food. The tape will be placed on the surface of the grid. (i.e. A light sensor needs to point downward.)
- Tape obstacles / food will outline a 10" x 10" square (one grid space).
- Physical obstacles may also be placed in the arena. (i.e. You need a bumper. If you run out of sensor ports, see the lecture notes on muxing touch sensors with a light sensor.) It will most likely not occupy an entire square.
- The robot may not cross a black line. It is understandable if a small portion of the robot crosses over but the robot as a whole must react to go around the black line as soon as it sees it.
- If the robot senses food, it should beep three times and stop. Only a single grid space will contain food.
- The arena will be available for use in EB 2029 for testing purposes. **Please do not remove it from EB 2029.** Tape will also be made available for you to test your robot with different obstacle / food configurations.
- The same (or nearly the same) environment will be used for the next assignment. The difference is that your robot will plan its trip. If you wish, you can try to build a robot model that will accomplish both the reactive and deliberative tasks. If you wish to do this, then build the model such that it is capable of traveling a straight line. (I'd suggest using a differential or two). You will also need to use the rotation sensors to determine the distance traveled from grid to grid for the next assignment. If you don't use the rotation sensors for this assignment then you can either incorporate them into the design and leave them unused or try to leave room for their addition later.

Each team will be given a single trial to complete the course. The robot **should** be programmed with a **1 second delay** after pressing the *Run* button. This will ensure that the robot is not accidentally thrown off course by moving while someone's hand is in the way. Optimally, the robot should find the food within 5 minutes, but no more than 20 minutes.

Robots must use a reactive architecture (direct *Sense* → *Act*) to complete the assignment. Specific locations for the start position, start direction, obstacles, and food will be established on the day of the demonstrations. All teams will demonstrate using the same layout. You will not be allowed to test against this layout. Remember, a reactive architecture's strength is handling the Open World assumption.

This assignment is due Wednesday, 16 April 2003 at 1:29PM and is worth 100 points. Any late submissions will receive a late penalty. Turn in a hardcopy of your program (the *.ic file(s)) at class time on the due date. Name your program RA2-[*each teammate's initials*].ic and place it in the dropbox **under Assignment 7**. If your code spans multiple files, either zip them or place them all into one text file with the start and end of each individual file clearly marked. The assignment only needs to be submitted once by a single teammate. Be sure to place **ALL** teammate names in the comment blocks at the start of the program to ensure proper credit is received.

Grading:

Logic and Coding	40 pts
Robot Physical Construction	25 pts
Overall Design (Integration)	10 pts
Finds food	25 pts
Total:	**100 pts**

The robot demonstrations will occur Thursday, 17 April, and Friday, 18 April. A sign-up sheet will be made available with specific times on these dates. To ensure fairness to those testing earlier, teams will be required to leave their robots when they turn in the assignment and/or upload the code from the dropbox before testing (instructor's option).

Please contact Gary Mayer, gmayer@siue.edu, with any questions regarding Interactive C or robot construction.

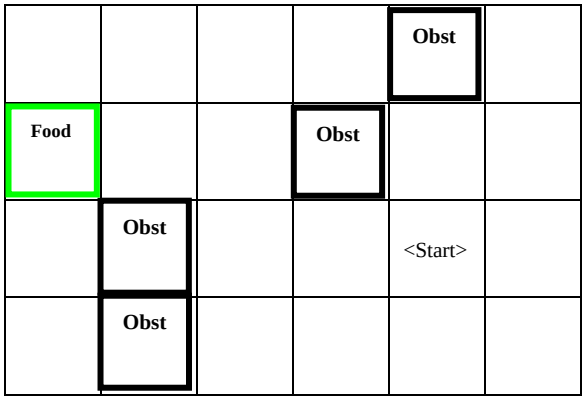


Figure 1 – **Example** of an Arena Layout

CS 438 – Artificial Intelligence
Interactive C Robotics Assignment 3 (RA3)

This assignment is designed to give you experience with a deliberative robotic architecture. You will build and program a robot that plans a route to a goal (or goals) and avoids known obstacles. Your robot should successfully find the goal in the most time efficient manner possible using a deliberative robotic architecture.

Items to Note:

- The arena is a 40" x 60" space with a white background. You can think of this as a 4 x 6 grid of squares, each with 10" sides.
- Your robot must fit entirely within a single grid square.
- Black tape will be used to emulate obstacles and green tape will emulate the goal. The tape will be placed on the surface of the grid. (i.e. A light sensor needs to point downward.)
- Tape obstacles / goal will outline a 10" x 10" square (one grid space).
- Physical obstacles may also be placed in the arena. (i.e. You need a bumper. If you run out of sensor ports, see the lecture notes on muxing touch sensors with a light sensor.) It will most likely not occupy an entire square.
- If the robot senses a goal, it should beep three times and stop. More than one grid space may contain a goal.
- The robot **may** pass through a grid containing a goal.
- The robot **may not** pass through a grid containing an obstacle. It is understandable if a small portion of the robot crosses over the line to properly sense it, but the robot as a whole must go around the grid space.
- The arena will be available for use in EB 2029 for testing purposes. **Please do not remove it from EB 2029.** Tape will also be made available for you to test your robot with different obstacle / goal configurations.
- To make the plan and execute it efficiently, your robot must know where it is within the arena. This can be accomplished through use of ded reckoning with the rotation sensors (counting how many "ticks" to the next grid space). However, this process can be severely hampered by immeasurable errors such as drift. To minimize these errors, you need to build your robot such that it can consistently travel a straight line and turn ninety degrees.
- As it is rather difficult to build a robot that will travel perfectly straight and turn a consistent ninety degrees all of the time, you may, at the instructor's discretion, nudge to robot back on course. *This will only be allowed twice during the entire run.* Do not mistake this as an opportunity for sloppiness. If it does not appear that your team put considerable effort into attempting to make it move straight and turn ninety degrees, points will e deducted.

Each team will be given a single trial to complete the course. The robot **should** be programmed with a **1 second delay** after pressing the *Run* button. This will ensure that the robot is not accidentally thrown off course by moving while someone's hand is in the way. The robot must reach the original goal within 10 minutes – including planning, any required re-plans due to plan failure, and actual movement.

Robots must use a deliberative architecture (*Sense* → *Plan* → *Act*) to complete the assignment. Specific locations for the start position, start direction, obstacles, and goal will be established on the day of the demonstrations. Your robot must be capable of having these locations manually entered at run-time. All teams will demonstrate using the same layout. You will not be allowed to test against this layout. Your robot must be capable of handling plan failure (a new goal or obstacle is found).

If a new goal is found, the robot must audibly signal that it recognizes the new goal. Then, it may proceed on its existing path to the original goal. If a new obstacle is found, the robot must stop and plan a new path from it's current location to the goal – around the new-found obstacle. Be sure to account for the robot being off-center of a grid space if this is important to the localization problem.

This assignment is due Wednesday, 30 April 2003 at 1:29PM and is worth 100 points. Any late submissions will receive a late penalty. Turn in a hardcopy of your program (the *.ic file(s)) at class time on the due date. Name your program RA3-[*each teammate's initials*].ic and place it in the dropbox **under Assignment 8**. If your code spans multiple files, either zip them or place them all into one text file with the start and end of each individual file clearly marked. The assignment only needs to be submitted once by a single teammate. Be sure to place **ALL** teammate names in the comment blocks at the start of the program to ensure proper credit is received.

Grading:

Logic and Coding	40 pts
Robot Physical Construction	25 pts
Overall Design (Integration)	10 pts
Finds goal	25 pts
Total:	**100 pts**

APPENDIX D

SUMMARY OF STUDENT FEEDBACK FORMS

SUMMARY OF STUDENT FEEDBACK FORMS

The following pages are copies of the Robotics Instruction Questionnaire that was distributed to the students after the robotics lectures were completed. From the 23 students taking the course, 19 responses were received. Within each question space, the average response score, median response score, and a summary of relevant comments are included. Notes regarding the student feedback are also included.

The overall feedback was very positive. Students were asked to provide responses on a scale of 1 to 5. One represented Strongly Disagree and 5 represented strongly agree, with 5 being the preferred response. The median of the response scores for all questions, except one, was 4.0. The one question was attempting to ask if having a hybrid robotic assignment would help with understand hybrid robotic architectures. Most students interpreted this as adding another lab into the whole robotic instruction versus looking at it from a strictly “would it beneficial to have a lab versus just lecture” point of view and marked it very negatively. The average for all scores, except for the same hybrid robotic control question, did not fall below 3.42. So, while both positive and negative comments were received for many responses, the overall indication from the class as a whole indicates a positive experience.

The major item drawn from the responses is that the amount of time spent building robots should be reduced. While construction may provide the students experience with the flaws of the mechanical world, too much (especially repetitively) drew away from the core lessons concerning the robotic paradigms and frustrated the students. A hybrid robotic control project, combining aspects of reactive and deliberative robotic paradigms and requiring only one robot to be built, may be the best solution. Further, a simple project

should remain as the first assignment to allow the students to gain familiarity with the hardware and programming environment.

Additional responses received indicate that some students thought that the course was too hardware-focused. The intent was to provide an overall view of robotics, which includes how hardware and software interface and the role that hardware (both the physical design and the sensors) can play when the robot interacts with the real world. It's this interaction with the real world that generates so many anomalies and errors that the programmer should be ready to compensate for as best as possible within the software. While less than 20% of the responses stated such an opinion, the fact that such comments were made may indicate that more emphasis needs to be placed on why understanding the hardware is important for robotics – whether designing the hardware or software.

CS 438 – Artificial Intelligence
Robotics Instruction Quesionnaire

This questionnaire is not an official SIUE class survey. However, it would be greatly appreciated if you would take the time to complete it. This survey's purpose is three-fold:

1) Help the instructor determine your evaluation of the Teaching Assistant's (TAs) performance in teaching you about robotics. Similarly, it will help the TA improve his teaching skills for the future.
2) Evaluate the usefulness of the information the TA has gathered while completing his thesis.
3) Provide feedback to the CS department chair to enable better integration between CS 438 and robotics courses that may be taught in the future.

For each question, please circle a number corresponding to your response to the statement.
1 – Strongly Disagree 2 – Disagree 3 – Undecided 4 – Agree 5 – Strongly Agree

Please include any comments in the space below each statement and / or on the back. If on the back, please reference the comment number. Please keep in mind that this questionnaire **refers only to the robotics portion of the course**.

1. The presentation material and in-class examples were appropriate and helped me better understand robotics.	**1 2 3 (4) 5** Avg: 4.0; Med: 4.0
Comments: - A bit too much mechanics.	
2. I required additional help from the TA outside of class. (**Y** or **N**) If Yes, the additional help provided by the TA was of value.	**1 2 3 (4) 5** Avg: 3.67; Med: 4.0
Comments:	
3. This course provided an appropriate amount of material on robotics.	**1 2 3 (4) 5** Avg: 3.78; Med: 4.0
Comments: - Way too much robotics. More interested in other AI topics. - Should be more robotics. - Good general overall view of mechanical and programming aspects. - Make last assignment easier for end of semester.	
4. Lab assignments are the best approach to determining a student's understanding of robotics. (Vice methods like quizzes, homework, or papers.)	**1 2 3 (4) 5** Avg: 3.89; Med: 4.0
Comments: - Assignments were a bit too much for time allotted. - Showed how to implement concepts; not just memorize them. - Learned a great deal and had fun doing it. - Had difficulty with the mechanics and building. Robots are finicky.	

5. Overall, The robotics labs assigned complimented the lectures and furthered my understanding of robotics. Specifically, …	**1 2 3 (4) 5** Avg: 4.16; Med: 4.0
5.a. Having a small project as the first assignment was very beneficial in allowing me to learn the software and experiment with the hardware prior to the next assignments.	**1 2 3 (4) 5** Avg: 4.42; Med: 4.0
5.b. The reactive robotic control assignment gave me a clear understanding of a reactive system's benefits and drawbacks.	**1 2 3 (4) 5** Avg: 4.21; Med: 4.0
5.c. The reactive robotic assignment was well worth doing.	**1 2 3 (4) 5** Avg: 3.74; Med: 4.0
5.d. The deliberative robotic control assignment gave me a clear understanding of a deliberative system's benefits and drawbacks.	**1 2 3 (4) 5** Avg: 4.16; Med: 4.0
5.e. The deliberative robotic control assignment was well worth doing.	**1 2 3 (4) 5** Avg: 3.79; Med: 4.0
5.f. From the last two assignments, I can clearly understand how a mixture of the two approaches would best serve most robotic platforms.	**1 2 3 (4) 5** Avg: 4.16; Med: 4.0
5.g. I do **not** feel an additional assignment is necessary to understand the benefits of a hybrid robot control architecture.	**1 (2) 3 4 5** Avg: 2.42; Med: 3.0
Comments: - While most responded that there are too many assignments; the original intent was not to ask if an assignment should be added, but if (regardless of time) one would be beneficial to understanding the material. - This much depth should only be used in an all-robotics course. - Computing power is so low and sensor error so bad that it detracts from designing good logic.	
6. The requirements for each lab assignment were clearly understood from the assignment sheet and explanation given in the classroom.	**1 2 3 (4) 5** Avg: 3.79; Med: 4.0
Comments: - Grading / points assignment needs revision.	
7. The size of each team was appropriate and enabled me to participate hands-on in each lab assignment.	**1 2 3 (4) 5** Avg: 4.0; Med: 4.0
Comments: - Pairs are probably the best for programming teams when code can't be split. - Teams are an unfair way to evaluate individuals. - Teammate never participated / was uncompromising.	
8. The software used to code the robot was appropriate for the assignments given and the hardware available.	**1 2 3 (4) 5** Avg: 3.63; Med: 4.0
Comments: - No debug function. [Note: latest version includes a "code check function"] - Extremely glad we used a C environment	

9. Having to build the robot gave me a better understanding of the importance hardware plays in robotic performance.	**1 2 3 (4) 5** Avg: 4.32; Med: 4.0
Comments: - Allowed more creativity. - Hardware played too much of a role. If hardware was messed up, robot couldn't complete task regardless of code.	
10. Having to build the robot provided a better understanding of sensor limitations.	**1 2 3 (4) 5** Avg: 4.11; Med: 4.0
Comments: - Greatly. - Had to calibrate sensor values. - Learned difficulty of [using] sensors.	
11. Having to build the robot was worthwhile in understanding overall robot design, including hardware-to-software integration.	**1 2 3 (4) 5** Avg: 4.16; Med: 4.0
Comments: - Learning about dual-differential seemed a bit much; though interesting. - Hands-on experience good source of knowledge.	
12. Overall, the robotics labs were at the appropriate difficulty level for the course.	**1 2 3 (4) 5** Avg: 3.68; Med: 4.0
Comments: - Too [strict] in grading. - Hardware too difficult. - Robotic assignments easier than first half of course. - Building robots took most of the time. - Too many aspects to the assignments to be done within allotted time.	
13. If an additional assignment is given requiring a robot to deliver mail to offices and travel in a model university (complete with a random placement of people), I would suggest the following robotic architecture: (Circle one and please explain in the comments section below). **REACTIVE HYBRID DELIBERATIVE**	
Comments: [Note: The intent of this question was to examine the students' understanding of the different robotic paradigms. The majority of the students answered the question correctly.]	
14. I was interested in robotics prior to taking this course.	**1 2 (3) 4 5** Avg: 3.55; Med: 3.50
15. This course enhanced my interest in robotics and I would like to learn more.	**1 2 (3) 4 5** Avg: 3.42; Med: 4.0
Comments: - I enjoyed the robot portion. - I can only take a class in robotics if I have a lot of time.	
16. Overall, I was very pleased with the robotics section of this course.	**1 2 3 (4) 5** Avg: 3.89; Med: 4.0

APPENDIX E

HARDWARE BUILDING INSTRUCTIONS

HARDWARE BUILDING INSTRUCTIONS

This is a detailed, visual description of how to build the robots used in this project. Given the reliance of the architecture on the dual-differential for accurate localization, it is the core of the overall design. The robot is built around the gear train and seeks to minimize immeasurable errors caused by an imbalanced design. Fittingly, the construction begins with the dual-differential. The next layer contains the gears between the motor and the differential. The third gear layer adds the motors and rotation sensors. Throughout, the remaining steps encapsulate the gear train and define a balanced support structure for the motors and RCX.

The diagrams were created in MLCAD version 3.0; created by Michael Lachmann. The software is free for download and may be found at http://www.lm-software.com/mlcad/.

During building, it may be true that a different piece may more efficiently substitute for one or two within the model. However, when the model was created, an attempt was made to use only a single set of pieces from the currently available, off-the-shelf LEGO Mindstorms sets.

To begin, Table 2, below, lists the parts required for construction. It was automatically generated by MLCAD. LEGO parts are used throughout a variety of different sets for different purposes. As such, the MLCAD descriptions may not be intuitive. However, the descriptions are typically useful enough to accurately identify a part and they do conform to LEGO nomenclature. For those unfamiliar with it, LEGO nomenclature uses the term “brick” to identify a block-type piece that is equal in height to 3 stacked “plates.” The number of studs on a part is then used to identify that part. For example, a “Brick 2 x 4” refers to a brick with 2 rows of studs and 4 columns – a total of 6 studs. Typically, the difference between a technic brick or plate and a normal brick or plate is the technic version has holes through the sides or top. When a number is given for an axle, that number refers to the axle’s length. So, a “Technic Axle 5” refers to an axle that is 5 studs long. On wheels, the numbers refer to millimeter diameters. Some of the wheels have this dimension stamped onto them. On angle connectors, the number is stamped onto the connector. The drawings that follow use the MLCAD descriptions for labels. In addition, other common abbreviations are used. For instance, gears are often abbreviated by using the number of gears followed by a “t” for teeth. Thus, a “24t” gear refers to a 24-tooth gear.

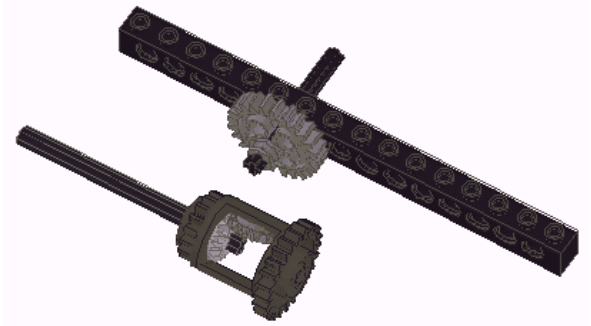
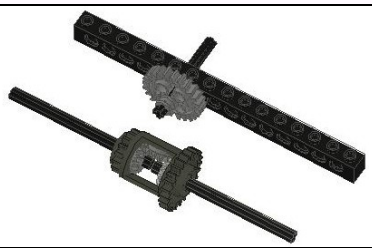
Table 3, below, provides step-by-step building instructions. Each step lists the parts required for the step on the right and provides a visual image of how the parts are assembled. While color does not usually matter, there are exceptions. For instance, technic pins. Technic pins are light-grey and are meant to freely rotate when inserted into brick holes. Frictions pins look just like technic pins but they are black. They are intended to hold pieces tightly together and, therefore, do not turn easily in the holes. Technic rubber bands are another example. Each color represents a different length and thickness.

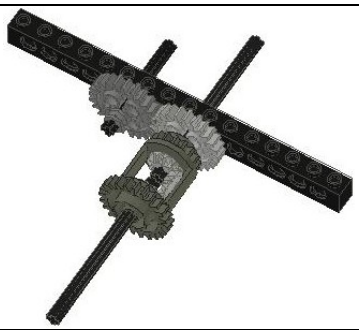
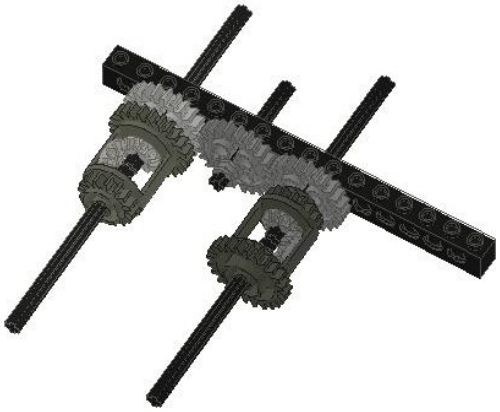
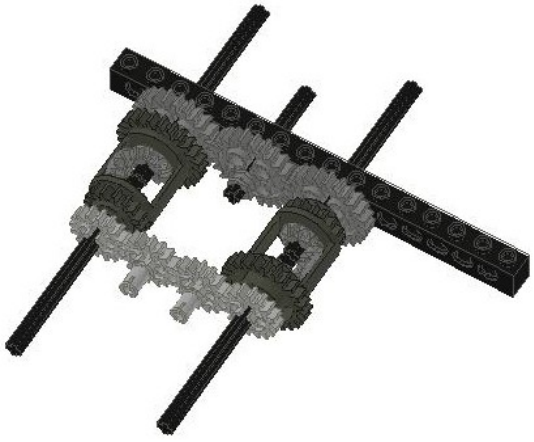
Table 2: LEGO Hardware Parts List

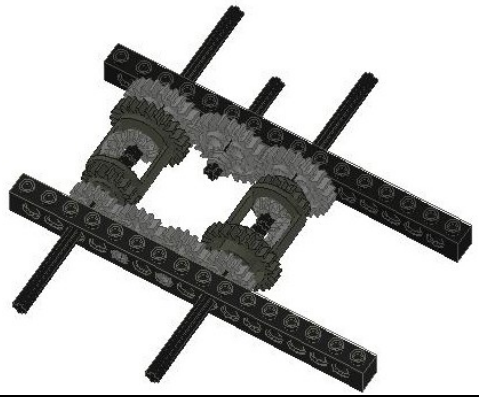
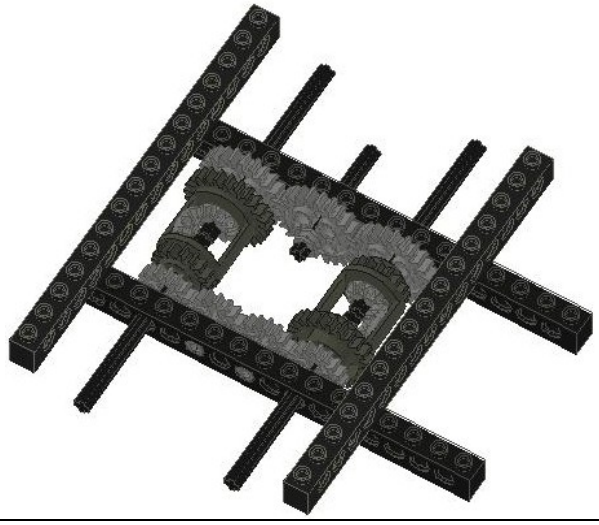
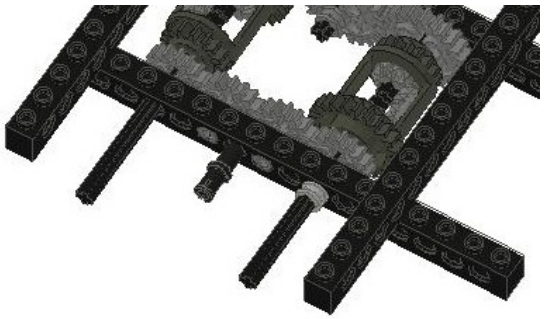
Quantity	Color	MLCAD Part Number	MLCAD Description
2	Black	6048.DAT	Arm Piece with Pin and 2 Fingers
1	Black	2654.DAT	Boat 2 x 2 Stud
5	Black	3004.DAT	Brick 1 x 2
1	Yellow	3004.DAT	Brick 1 x 2
3	Black	3003.DAT	Brick 2 x 2
3	Black	3001.DAT	Brick 2 x 4
1	Blue	2982C01.DAT	Electric Light Sensor (Complete Assembly Shortcut)
1	Light-Gray	884.DAT	Electric Mindstorms RCX (Complete Assembly Shortcut)
2	Black	4755.DAT	Electric Plate 1 x 2 with Contacts (Note: robot requires 4 connector wires)
2	Blue	2977C01.DAT	Electric Rotation Sensor (Complete Assembly Shortcut)
2	Light-Gray	2983.DAT	Electric Technic Micromotor Pulley
3	Light-Gray	71427C01.DAT	Electric Technic Mini-Motor 9V
1	Light-Gray	879.DAT	Electric Touch Sensor Brick 3 x 2 (Complete Assembly Shortcut)
4	Black	4221.DAT	Grab Jaw
2	Yellow	4220.DAT	Grab Jaw Holder
13	Light-Gray	3023.DAT	Plate 1 x 2
4	Light-Gray	32028.DAT	Plate 1 x 2 with Door Rail (Technic motor holder)
1	Yellow	3623.DAT	Plate 1 x 3
2	Light-Gray	3710.DAT	Plate 1 x 4
2	Light-Gray	3666.DAT	Plate 1 x 6
3	Light-Gray	3460.DAT	Plate 1 x 8
2	Light-Gray	4477.DAT	Plate 1 x 10
1	Blue	3022.DAT	Plate 2 x 2
3	Light-Gray	3022.DAT	Plate 2 x 2
4	Light-Gray	2420.DAT	Plate 2 x 2 Corner
4	Light-Gray	2817.DAT	Plate 2 x 2 with Holes
1	Light-Gray	3020.DAT	Plate 2 x 4
4	Yellow	3020.DAT	Plate 2 x 4
2	Light-Gray	3832.DAT	Plate 2 x 10
1	Light-Gray	3033.DAT	Plate 6 x 10
3	Black	3747.DAT	Slope Brick 33 3 x 2 Inverted
2	Black	3665.DAT	Slope Brick 45 2 x 1 Inverted
2	Yellow	3665.DAT	Slope Brick 45 2 x 1 Inverted
1	Black	3660.DAT	Slope Brick 45 2 x 2 Inverted
2	Light-Gray	32015.DAT	Technic Angle Connector #5
2	Black	3705.DAT	Technic Axle 4

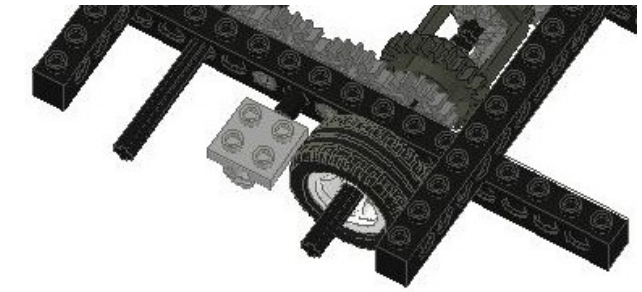
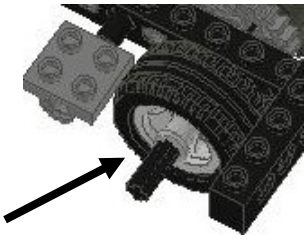
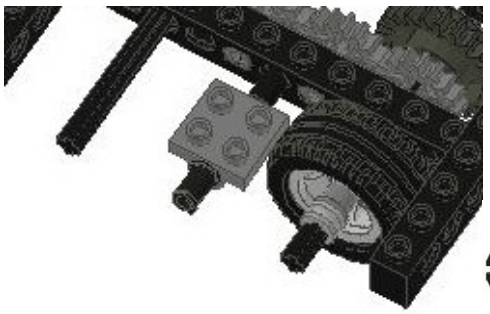
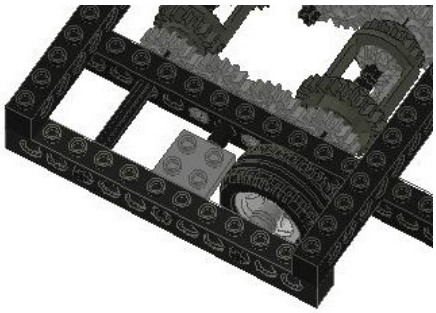
5	Black	32073.DAT	Technic Axle 5
1	Black	3706.DAT	Technic Axle 6
9	Black	3707.DAT	Technic Axle 8
2	Black	3737.DAT	Technic Axle 10
2	Black	3708.DAT	Technic Axle 12
2	Light-Gray	6538.DAT	Technic Axle Joiner
4	Light-Gray	3749.DAT	Technic Axle Pin
2	Green	32064.DAT	Technic Brick 1 x 2 with Axlehole
9	Black	3700.DAT	Technic Brick 1 x 2 with Hole
4	Yellow	3700.DAT	Technic Brick 1 x 2 with Hole
8	Black	3701.DAT	Technic Brick 1 x 4 with Holes
2	Green	3701.DAT	Technic Brick 1 x 4 with Holes
3	Black	3894.DAT	Technic Brick 1 x 6 with Holes
2	Black	3702.DAT	Technic Brick 1 x 8 with Holes
2	Black	3895.DAT	Technic Brick 1 x 12 with Holes
4	Black	3703.DAT	Technic Brick 1 x 16 with Holes
27	Light-Gray	3713.DAT	Technic Bush
9	Light-Gray	4265C.DAT	Technic Bush 1/2 Smooth
1	Light-Blue	32137.DAT	Technic Connector Block 3 x 2 x 2
2	Light-Gray	32039.DAT	Technic Connector with Axlehole
2	Dark-Gray	6573.DAT	Technic Differential New
2	Yellow	75.DAT	Technic Flex-System Hose
4	Light-Gray	3647.DAT	Technic Gear 8 Tooth
6	Light-Gray	6589.DAT	Technic Gear 12 Tooth Bevel
4	Light-Gray	4019.DAT	Technic Gear 16 Tooth
7	Light-Gray	3648.DAT	Technic Gear 24 Tooth
3	Light-Gray	6632.DAT	Technic Liftarm 1 x 3
2	Light-Gray	2825.DAT	Technic Liftarm 1 x 4
2	Black	6629.DAT	Technic Liftarm 1 x 9 Bent
2	Light-Gray	3673.DAT	Technic Pin
1	Yellow	32136.DAT	Technic Pin 3L Double
1	Dark-Gray	4274.DAT	Technic Pin 1/2
3	Black	6558.DAT	Technic Pin Long with Friction
7	Black	4459.DAT	Technic Pin with Friction
2	Light-Gray	3738.DAT	Technic Plate 2 x 8 with Holes
2	Light-Gray	6553.DAT	Technic Pole Reverser Handle
1	Blue	[created]	Technic Rubber Band
2	White	[created]	Technic Rubber Band
1	Yellow	[created]	Technic Rubber Band
1	Light-Gray	4185.DAT	Technic Wedge Belt Wheel
2	Black	6578.DAT	Tire 30.4 x 14 VR
2	White	2994.DAT	Wheel 30.4 x 14 VR

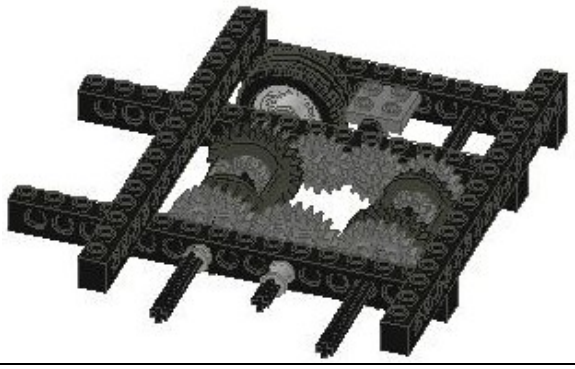
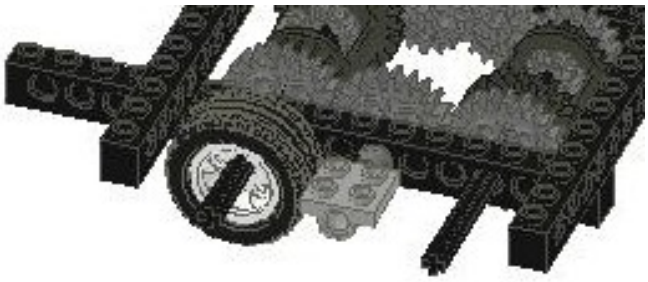
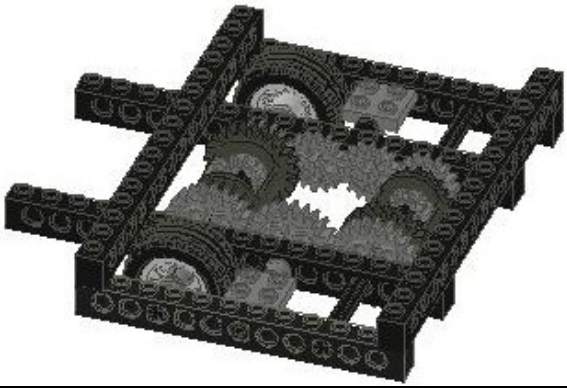
Table 3: Hardware Building Instructions

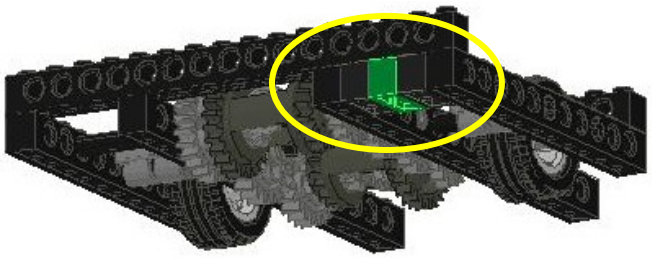
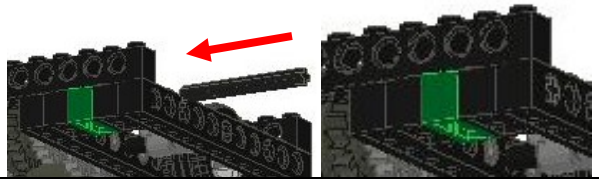
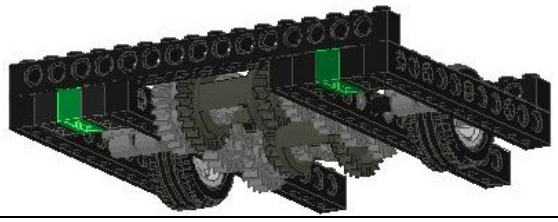
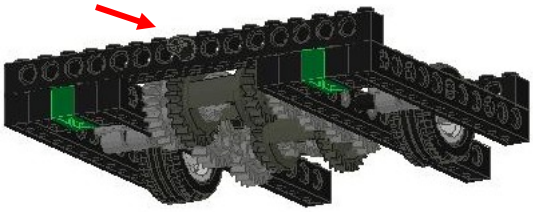
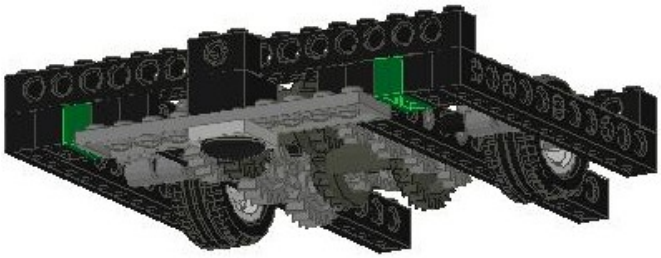
Step	Model	Parts Required / Instructions
1	 ½ bushing	Dual-Differential: 1 - Technic Axle 5 1 - Technic Bush ½ Smooth 1 - Technic Gear 24 Tooth 1 - Technic Brick 1 x 16 with Holes Slide a ½ bushing about 1 ½ bushing's width down a 5-axle. Place a 24t gear on the other end and insert the axle into the 6th hole (from the left) of the 1 x 16 technic brick.
2a	 12t bevel gear	1 - Technic Differential New 3 - Technic Gear 12 Tooth Bevel 2 - Technic Axle 8 Place a 12t bevel gear onto the small axle within the differential.
2b		Holding a second 12t bevel gear within the differential, slide an axle through the 2nd gear's hole.
2c		Repeat for the other side of the differential.

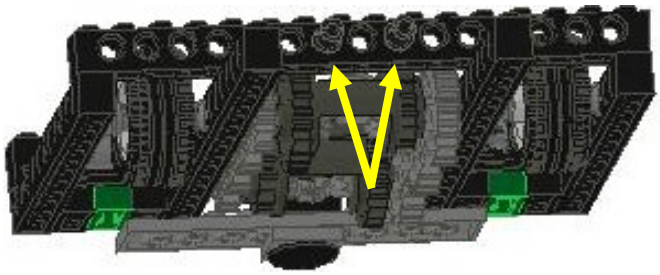
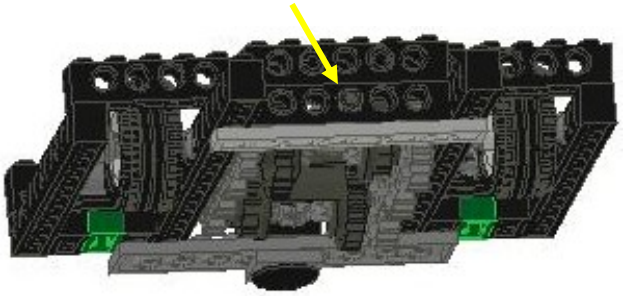
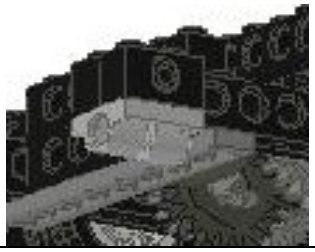
3		1 – Technic Gear 24 Tooth Note that the differential shell has two gears built onto the shell, a 24t gear and a 16t gear. Slide a 24t gear onto the axle closest to the 16t gear. Then, slide this axle into the 1 x 16 brick, to the right of the existing 24t gear, so that the teeth of the two 24t gears mesh.
4		<u>Turn Differential</u>: 1 - Technic Differential New 3 - Technic Gear 12 Tooth Bevel 2 - Technic Axle 8 1 - Technic Gear 24 Tooth Build a second differential using steps 2 a-c above. Then, slide a 24t gear onto the axle closest to the 24t gear on the differential shell. Finally, insert this axle into the 1 x 16 brick, to the left of the first 24t gear, so that the gear teeth mesh.
5	  Axle Pin	4 – Technic Gear 16 Tooth 2 – Technic Axle Pin Slide a 16t gear onto each differential axle. Then insert an axle pin into the 2 remaining 16t gears.

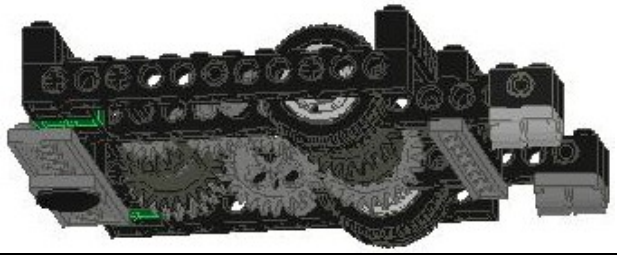
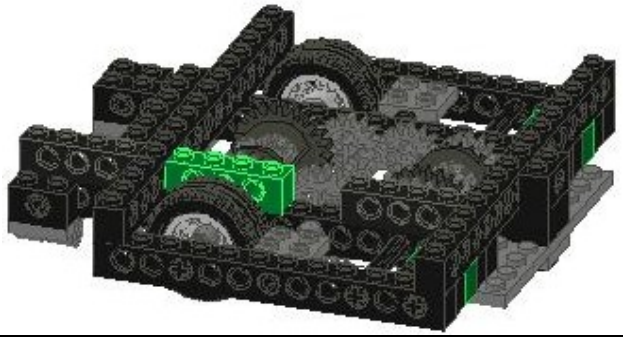
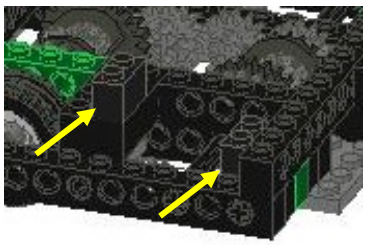
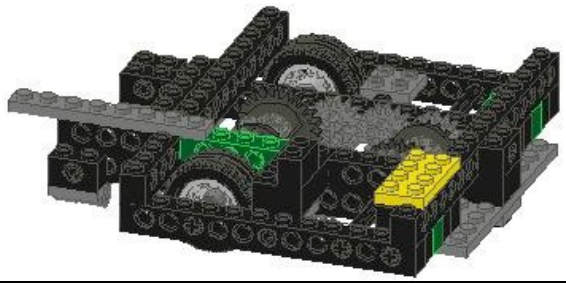
6		1 - Technic Brick 1 x 16 with Holes Slide a second 1 x 16 technic brick onto the two differential axles. Ensure that the holes align with the first 1 x 16 technic brick. As the 1 x 16 technic brick nears the differentials, snap the two 16t gears with axle pins into place, being sure that the teeth of the four 16t gears mesh with their neighbor's when the axle is fully inserted.
7		<u>Wheels and Support Frame</u>: 2 - Technic Brick 1 x 16 with Holes Place two 1 x 16 technic bricks onto the assembly as shown.
8	  Friction Pin - Long	1 - Technic Bush ½ Smooth 1 – Technic Pin Long with Friction Slide the ½ bushing onto the linear (right) differential as shown. Then, insert a long friction pin in the hole that is between the two center 16t gears. Note that the friction pin has one side with a single groove and the other has two. If you insert the side with two groves toward the 1 x 16 brick, only push the pin in far enough so it clicks into the first groove.

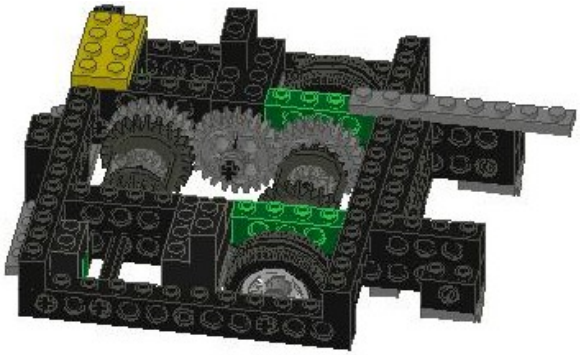
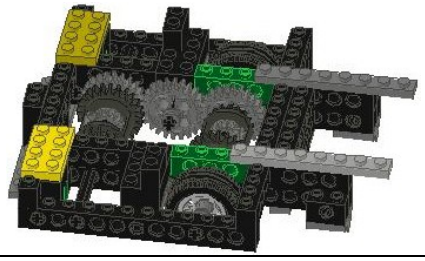
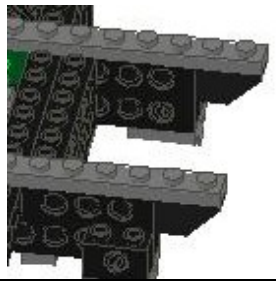
9	  Plate 2 x 2 with holes	1 – Plate 2 x 2 with Holes 1 – Tire 30.4 x 14 VR 1 – Wheel 30.4 x 14 VR Place the tire onto the wheel. Then, insert the wheel assembly onto the linear differential axle. Next, slide a 2 x 2 plate, stud side up, onto the long friction pin. Ensure the plate only goes as far as the first groove on the pin.
10	  Bushing	1 – Technic Bush Slide a full bushing onto the linear differential axle.
11	  Friction Pin	1 – Technic Pin with Friction 1 – Technic Bush ½ Smooth Insert a friction pin into the other hole on the 2 x 2 plate. Slide a ½ bushing onto the linear differential axle.
12		1 – Technic Brick 1 x 12 with Holes Slide the 1 x 12 technic brick over the axles and lock into place using the friction pin and the bottom of the two 1 x 16 technic bricks.

13		1 – Technic Bush 1 – Technic Bush ½ Smooth Rotate the model to work on the other axles. Slide a ½ bushing onto the linear differential axle (now on the left). Slide a full bushing onto the axle that passes through the central 24t gear.
14		1 – Plate 2 x 2 with Holes 1 – Tire 30.4 x 14 VR 1 – Wheel 30.4 x 14 VR Place the tire onto the wheel. Then, insert the wheel assembly onto the linear differential axle. Next, slide a 2 x 2 plate onto the center axle.
15		1 – Technic Bush Place a technic bushing onto the axle with the wheel.
16		1 – Technic Bush ½ Smooth 1 – Technic Pin with Friction Slide a ½ bushing over the wheel axle and insert a friction pin into the hole in the 2 x 2 plate.
17		1 – Technic Brick 1 x 12 with Holes Slide the 1 x 12 brick over the axles as shown. Secure it in place with the friction pin and the ends of the 1 x 16 bricks.

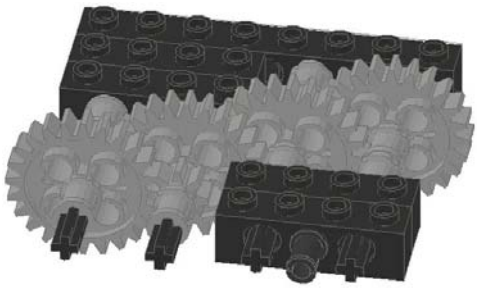
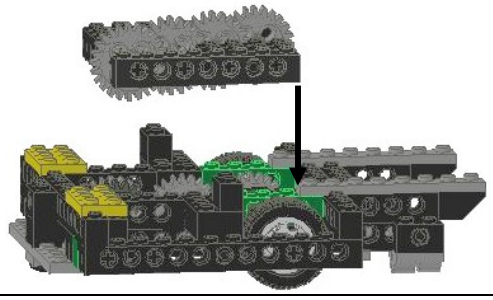
18		<u>Rear Support</u>: 2 – Technic Brick 1 x 2 with Hole 1 – Technic Brick 1 x 2 with Axlehole Place the three bricks in the positions shown. The holes in the three bricks should line up with the last hole in the 1 x 12 brick.
19		1 – Technic Axle 5 Slide the axle into the hole to secure the pieces. It should pass fully through the 1 x 12 brick, the three 1 x 2 bricks, and the main 1 x 16 brick.
20		2 – Technic Brick 1 x 2 with Hole 1 – Technic Brick 1 x 2 with Axlehole 1 – Technic Axle 5 Repeat Steps 18 and 19 to complete the rear support assembly on the other side.
21		1 – Technic Pin with Friction Insert the friction pin into the center hole of the rear 1 x 16 brick.
22		1 – Technic Brick 1 x 2 with Hole 1 – Brick 2 x 2 Place the 1 x 2 brick onto the friction pin. Connect the 2 x 2 brick to the 1 x 2 brick and the rear 1 x 16 brick, as shown.
23	  Boat 2 x 2 Stud	1 – Plate 2 x 10 1 – Plate 2 x 2 1 – Boat 2 x 2 Stud Center the 2 x 10 plate on the 2 x 2 brick and connect. The plate should connect with 1 pin into each 1 x 16 brick and the first 1 x 2 brick. Next, place the 2 x 2 plate onto the 2 x 10 plate, directly underneath the 2 x 2 brick. Finally, place the 2 x 2 boat stud onto the 2 x 2 plate.

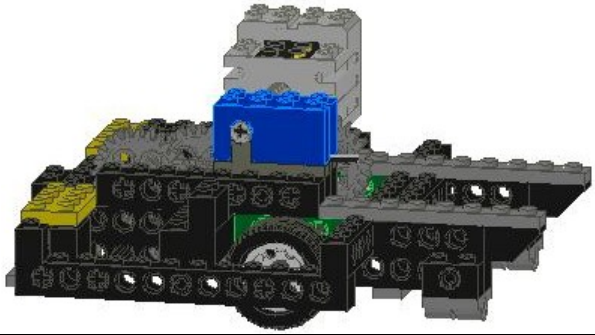
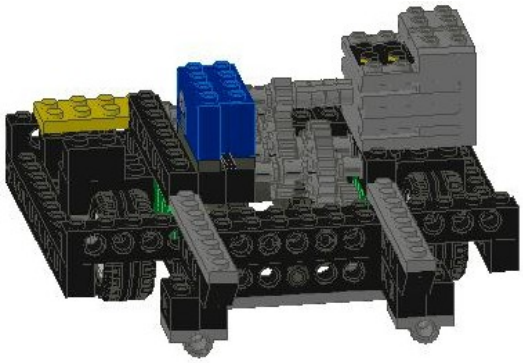
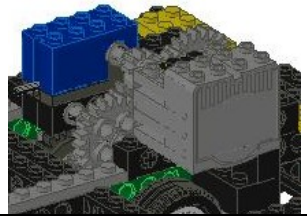
24		<u>Front Support and Kneeling Skids</u>: 2 – Technic Pin with Friction Rotate the model around to the front. Insert 2 friction pins into the forward 1 x 16 brick as shown
25		1 – Technic Brick 1 x 6 with Holes Slide the 1 x 6 technic brick onto the two friction pins. The brick should be centered between the two 1 x 16 technic bricks.
26		1 – Technic Brick 1 x 6 with Holes 1 – Plate 1 x 8 1 – Technic Pin ½ Place the 1 x 6 technic brick between the two 1 x 16 technic bricks and attach it to the bottom of the 1 x 6 technic brick added in Step 25. Secure the assembly using the 1 x 8 plate to connect the 1 x 6 technic brick and 1 x 16 technic bricks. Finally, insert the long end of the ½ pin into the center hole of the bottom 1 x 6 technic brick.
27		1 – Technic Pin with Friction Insert a friction pin into the 1st hole of the 1 x 16 technic brick (supporting the axles).
28		1 – Technic Brick 1 x 2 with Hole 1 – Plate 2 x 2 with Holes Place the 1 x 2 brick, stud-side up, onto the friction pin. Secure the pieces with the 2 x 2 plate.

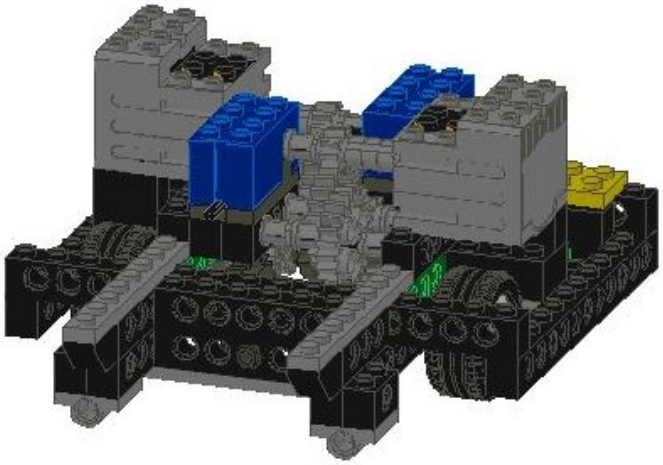
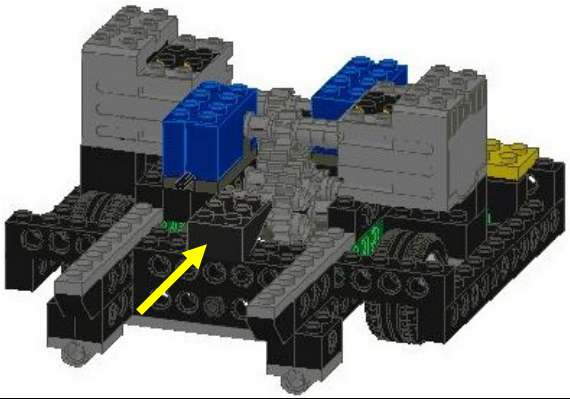
29		1 – Technic Pin with Friction 1 – Technic Brick 1 x 2 with Hole 1 – Plate 2 x 2 with Holes Repeat Steps 27 and 28 for the front of the other 1 x 16 technic brick.
30		<u>Motor and Gear Support</u>: 3 – Technic Brick 1 x 4 with Holes Place the three 1 x 6 technic bricks along the top of the 1 x 16 technic brick as shown.
31		1 – Brick 2 x 4 Place the 2 x 4 brick on top of the 2 x 2 plate so that it joins the plate and the 1 x 16 technic brick. Be sure that the 2 x 2 plate is studs-side up.
32		2 – Brick 1 x 2 Place one 1 x 2 brick on top of the 2 x 4 brick as shown. Place the other along side the rear 1 x 16 technic brick.
33		1 – Plate 1 x 8 1 – Plate 2 x 4 Place the 1 x 8 plate on top of the 1 x 16 technic brick so that 2 studs overhang from the front. Place the 2 x 4 plate along the back as shown.

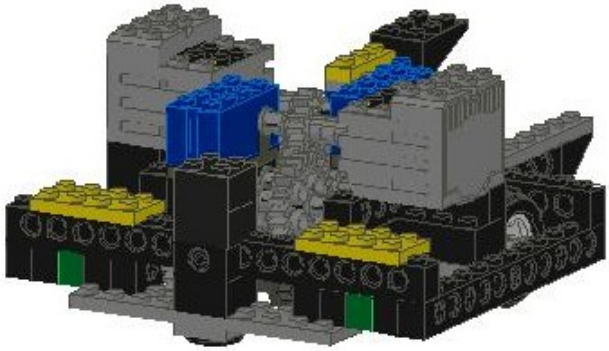
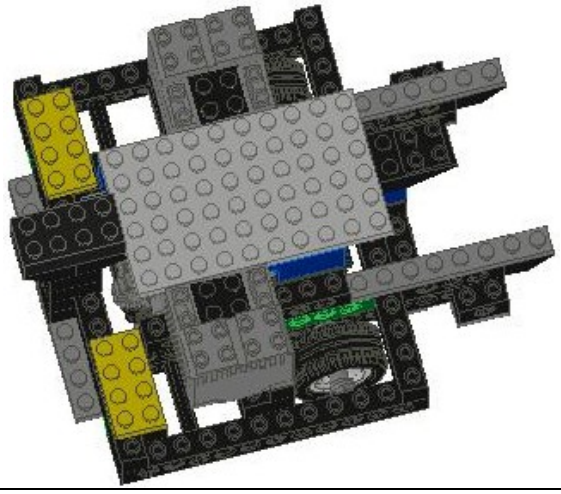
34		3 – Technic Brick 1 x 4 with Holes 1 – Brick 2 x 4 2 – Brick 1 x 2 Rotate the model to the other side and insert the three 1 x 4 bricks similar to Step 30. Attach the 2 x 4 brick and the two 1 x 2 bricks similar to Steps 31 and 32. However, note that the 1 x 2 brick on top of the 2 x 4 brick goes closer to the wheel on this side.
35		1 – Plate 1 x 8 1 – Plate 2 x 4 Repeat the placements of the 1 x 8 plate and the 2 x 4 plate for this side similar to Step 33.
36		2 – Slope Brick 45 2 x 1 Inverted Place the two inverted slope bricks onto the overhanging edges of the 1 x 8 plates attached in Steps 33 and 35.
37		<u>Main Gear Train Assembly</u>: 1 – Technic Brick 1 x 8 with Holes 1 – Technic Pin Long with Friction Place the short end of the long friction pin into the 2nd hole of the 1 x 8 technic brick, as shown.
38		2 – Technic Brick 1 x 4 with Holes Slide the center holes of the two 1 x 4 technic bricks onto the friction pin.
39		4 – Technic Axle 8 Place an axle into every other hole, as shown. The ends of the axles should pass through and be flush with the 1 x 8 technic brick.

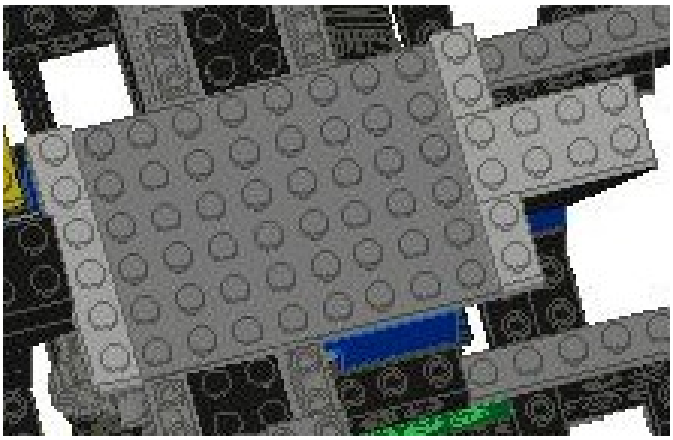
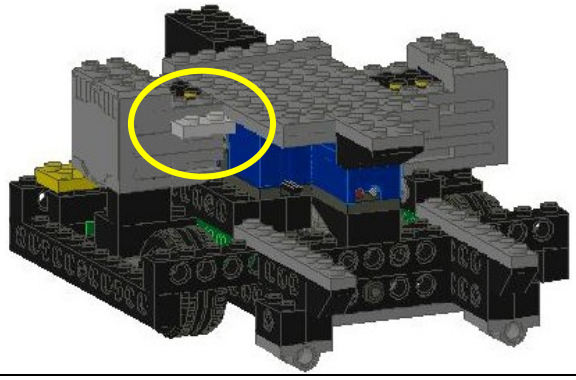
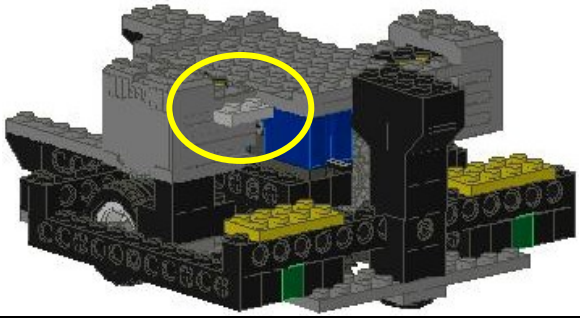
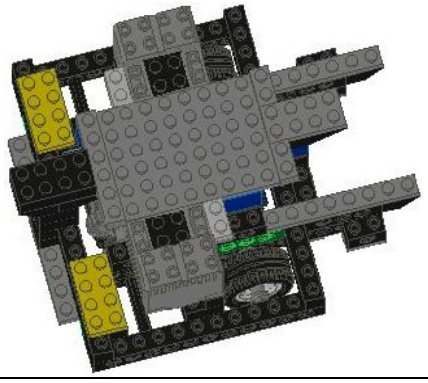
40		2 – Technic Bush Place a bushing onto the two longer pieces of axle.
41		1 – Technic Gear 24 Tooth 1 – Technic Gear 8 Tooth Slide the 24t gear onto the outermost , longer axle. Slide the 8t gear onto the axle next to it. Be sure that gear teeth mesh properly.
42		3 – Technic Bush 1 – Technic Gear 24 Tooth Place the 24t gear onto the axle with the 8t gear. Slide bushings onto the other 3 axles. Be sure that the bushings do not interfere with the 24t gear's rotation.
43		3 – Technic Bush 1 – Technic Gear 24 Tooth Place the 24t gear onto the axle to the left of the last 24t gear. Slide bushings onto the other 3 axles. Be sure that the bushings do not interfere with the 24t gear's rotation.
44		1 – Technic Gear 24 Tooth 1 – Technic Gear 8 Tooth 1 – Technic Brick 1 x 4 with Holes Place the 24t gear onto the last axle containing no gears. Place the 8t gear onto the axle next to it, being sure that the gears mesh properly. Slide the 1 x 4 technic brick onto the last two axles.

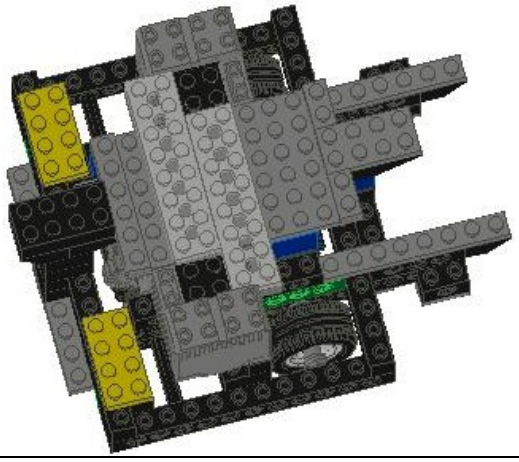
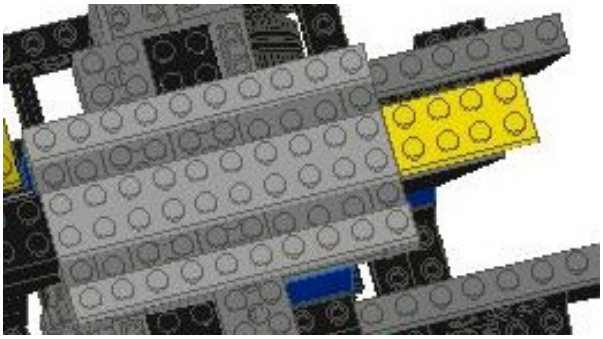
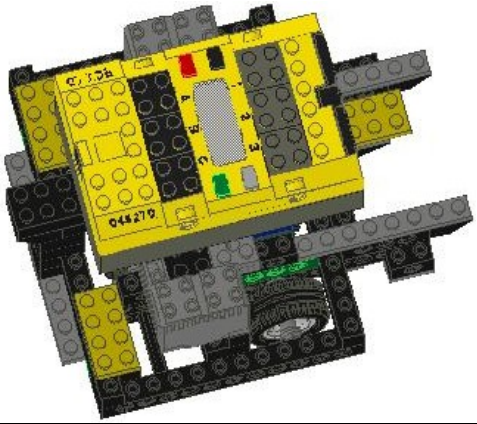
45		1 – Technic Brick 1 x 4 with Holes 1 – Technic Pin Long with Friction 2 – Technic Bush Slide the 1 x 4 technic brick onto the right-most axles and secure the two 1 x 4 bricks with the long end of the long friction pin. Place a bushings onto each of the other axles, in front of the 24t and 8t gears.
46		1 – Technic Brick 1 x 8 with Holes Complete the main gear train assembly by sliding the 1 x 8 technic brick onto the axles, locking it into place with the long friction pin.
47a		Insert the completed gear train assembly onto the frame.
47b		When properly inserted, the 1 x 8 technic bricks will rest along the 1 x 16 technic brick that support the axles, between the 1 x 8 and 2 x 4 plates placed in Steps 33 and 35. This will align the two outermost 24t gears to mesh with the 16t gears on the two differentials.
48		<u>Motor / Rotation Sensor Assembly</u>: 1 – Electric Technic Mini-Motor 9V 1 – Technic Axle Joiner 1 – Technic Axle 5 Place an axle joiner onto the 9V motor's shaft. Insert a 5 axle into the other end of the axle joiner.

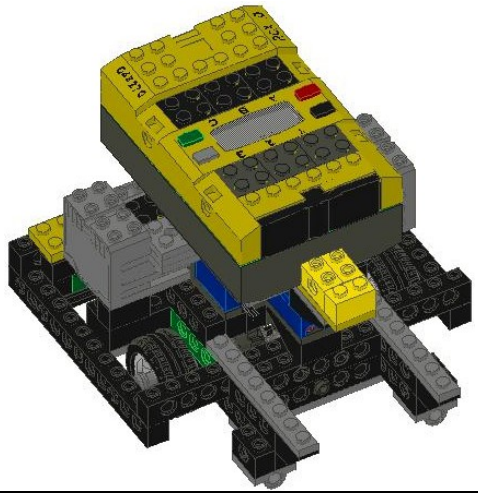
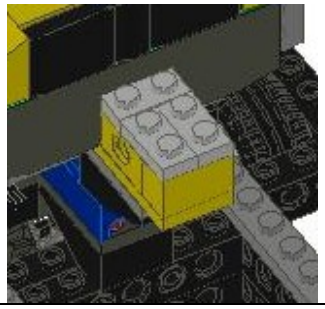
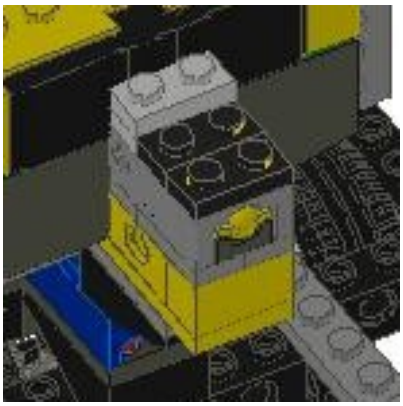
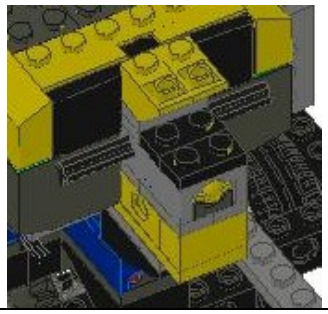
49		1 – Technic Gear 8 Tooth 1 – Technic Bush Slide an 8t gear onto the 5 axle. Next, slide a bushing onto the axle.
50		1 – Electric Rotation Sensor Place the rotation sensor onto the end of the axle in the direction shown.
51a		Insert the completed motor / rotation sensor assembly onto the model as shown.
51b		Note that when placed properly, the 8t gear on the motor shaft aligns with one of the mid-set 24t gears in the gear train. Further, the rotation sensor sits directly on top of two 1 x 4 technic gears used for the gear train assembly.
51c		Further, note how the 1 x 2 brick near the wheel supports the motor on its right side.

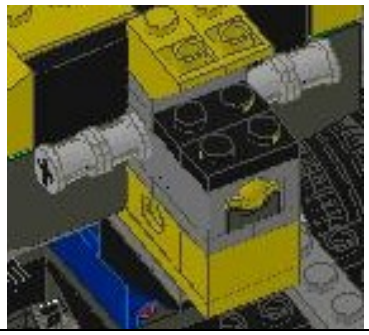
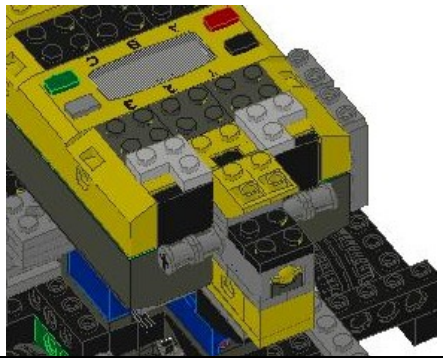
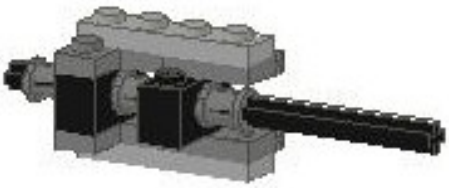
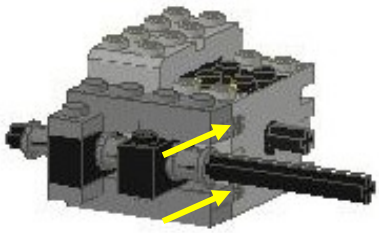
52		Repeat Steps 49 – 51 for the second motor / rotation sensor assembly.
53		<u>Light Sensor Assembly</u>: 1 – Slope Brick 33 3 x 2 Inverted Attach the 3 x 2 inverted slope brick to the front 1 x 6 technic brick. The slope should be facing backward, over the geartrain.
54		1 – Electric Light Sensor 1 – Plate 2 x 2 1 – Plate 1 x 2 Place the light sensor on top of the 3 x 2 inverted slope brick with the sensor facing away from the gear train. Place the 2 x 2 plate and the 1 x 2 plate on the light sensor as shown.
55		1 – Slope Brick 33 3 x 2 Inverted 1 – Plate 1 x 3 Place the 3 x 2 inverted slope brick on top of the forward edge of the light sensor with the slope going away from the gear train. Place the 1 x 3 plate on top of the 2 x 2 and 1 x 2 plates, as shown. The hole created on the other side of the 1 x 3 plate will be used to run the wire from the light sensor to the RCX.

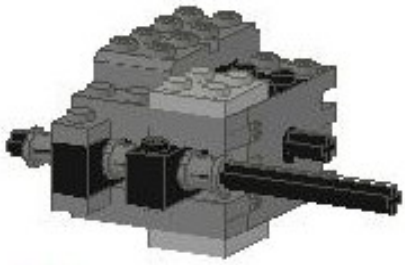
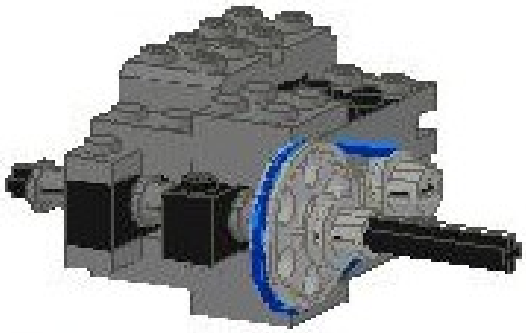
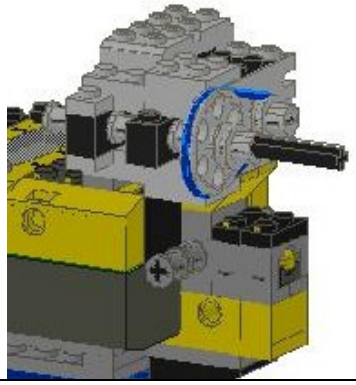
56		<u>RCX Support</u>: 2 – Brick 2 x 2 Stack the 2 x 2 bricks on top of one another. Place them at the rear of the model, on top of the middle two studs of the 1 x 16 technic brick and the 1 x 2 brick placed in Step 22.
57		1 – Slope Brick 33 3 x 2 Inverted 1 – Slope Brick 45 2 x 2 Inverted Place the 3 x 2 inverted slope brick on the forward edge of the 2 x 2 brick placed in the last step. The slope should be facing toward the gear train. Place the 2 x 2 inverted slope brick on the rear edge.
58		1 – Plate 6 x 10 1 – Brick 2 x 4 Looking down at the top of the model, place the 6 x 10 plate as shown. The plate should rest fully on top of the rotation sensors, the 1 x 3 plate placed in Step 55, and on the first row of studs of the 3 x 2 inverted slope brick placed in the last step. Then, place the 2 x 4 brick behind it, on top of the 2 x 2 inverted slope brick and remaining two rows of studs of the 3 x 2 inverted slope brick.

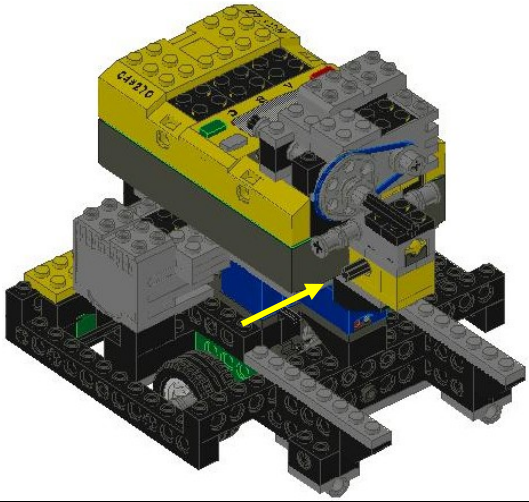
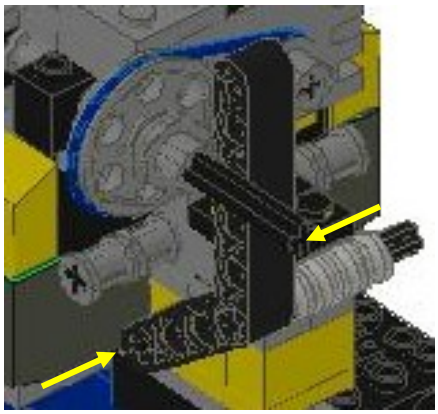
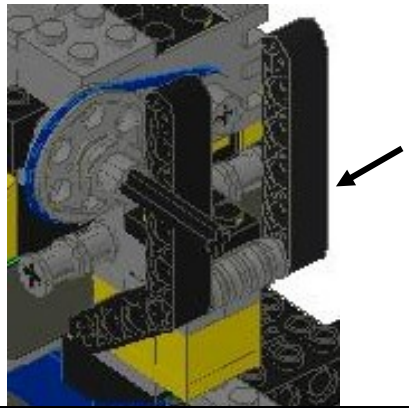
59		1 – Plate 2 x 4 2 – Plate 1 x 2 1 – Plate 1 x 6 Place the 2 x 4 plate on the 3 x 2 inverted slope brick that rests on top of the light sensor. One row should overlap onto the 6 x 10 plate placed in the last step. Place the two 1 x 2 plates to either side, on the forward edge of the 6 x 10 plate, as shown. Place the 1 x 6 plate along the trailing edge of the 6 x 10 plate.
60		1 – Plate 1 x 2 with Door Rail Insert the rail end of the door rail plate into the upper support slot on the right-hand side of one motor.
61		1 – Plate 1 x 2 with Door Rail Turn the model around and repeat for the other motor.
62		2 – Plate 1 x 2 From the top again, place a 1x 2 plate onto each door rail plate placed in Steps 60 and 61.

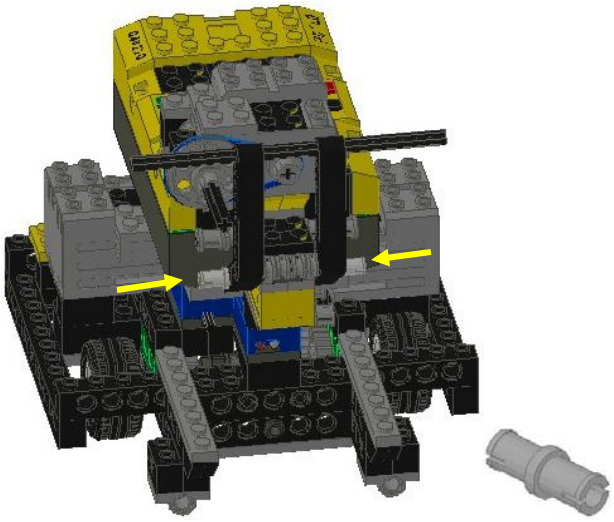
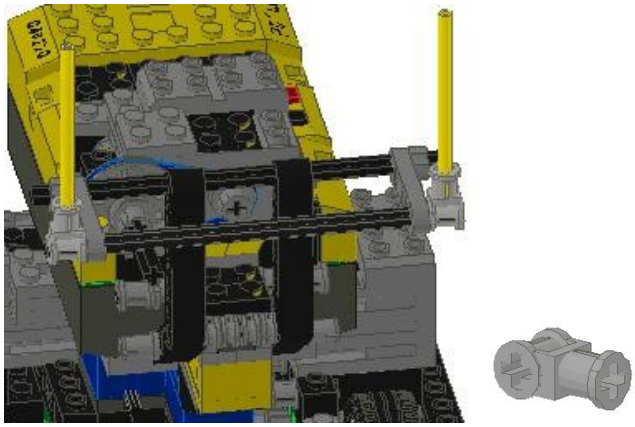
63		2 – Technic Plate 2 x 8 with Holes Place the two 2 x 8 plates across 6 x 10 plate, as shown. Each 2 x 8 plate should start from the edge of the 6 x 10 plate, in front of the center of a motor, and lock into the motor / door rail assembly on the other side.
64		2 – Plate 1 x 10 1 – Plate 2 x 10 1 – Plate 2 x 4 Place the 1 x 10 plates along the edges, as shown. The 2 x 10 should be placed down the center of the RCX support assembly. The 2 x 4 plate will cover the remaining three studs of the 2 x 4 plate placed in Step 59 and should overhang by 1 stud row.
65		1 – Electric Mindstorms RCX Place the RCX onto the model. It should rest evenly over the 2 x 10 plate.

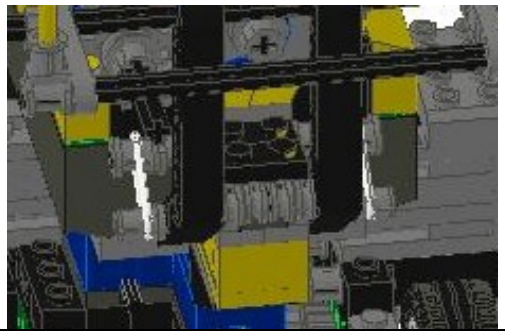
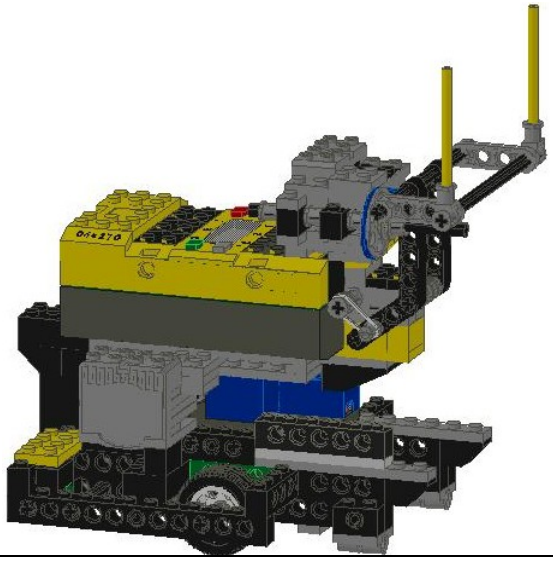
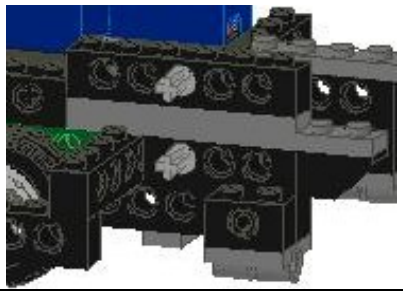
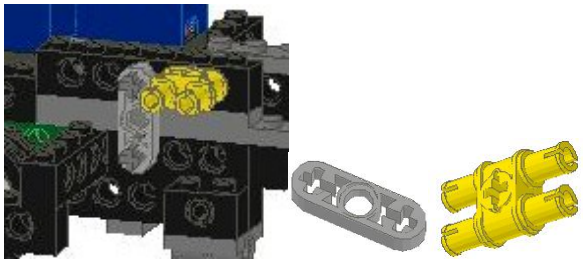
66		<u>Touch Sensor Assembly</u>: 2 – Technic Brick 1 x 2 with Hole 1 – Brick 1 x 2 Place the two 1 x 2 technic bricks with holes side by side up against the RCX. The holes should face to the sides. Place the 1 x 2 brick in front of them, as shown.
67		3 – Plate 1 x 2 Place the three 1 x 2 plates as shown to lock the 1 x 2 bricks from the previous step into place.
68		1 – Electric Touch Sensor Brick 3 x 2 2 – Plate 1 x 2 1 – Electric connector Place the touch sensor on top of the 1 x 2 plates installed in the last step. Stack the two 1 x 2 plates and place them on top of the touch sensor, against the RCX, as shown. Place one end of an electrical connector onto the touch sensor now.
69		2 – Slope Brick 45 2 x 1 Inverted 1 – Technic Axle 6 Place the two 2 x 1 inverted slope bricks on top of the 1 x 2 plate installed last step. The slopes should face forward. Insert a 5 axle into the axle hole in the touch sensor.

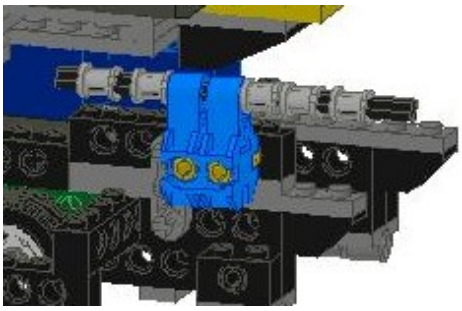
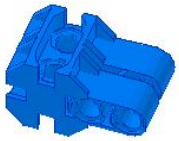
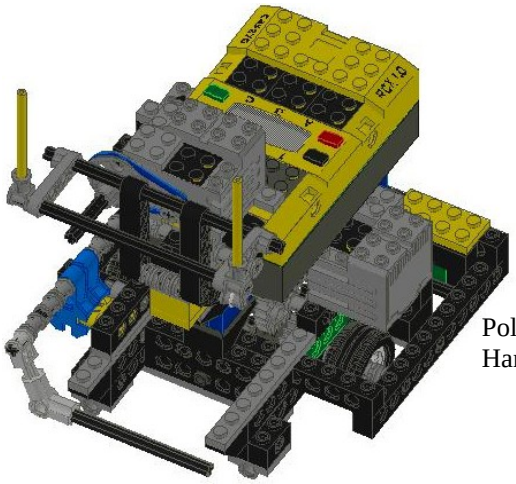
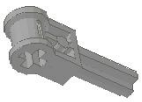
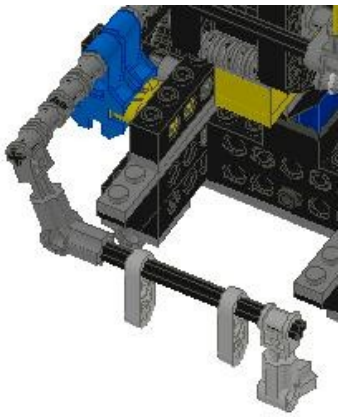
70		4 – Technic Bush Slide two bushings onto each end of the 5 axle installed last step.
71		<u>Trap Motor Assembly</u>: 2 – Plate 2 x 2 Corner Place the two 2 x 2 corner plates on top of the RCX as shown.
72		1 – Technic Axle 10 3 – Technic Bush 2 – Technic Brick 1 x 2 with Hole Assemble the bushings and 1 x 2 bricks on the 10 axle as shown.
73		2 –Plate 2 x 2 Corner 2 – Plate 1 x 2 with Door Rail Place the plates on the assembly as shown. A corner piece and a rail each go on top and on the bottom, in the same position.
74		2 – Plate 1 x 4 Attach the 1 x 4 plates to the top and bottom of the assembly, as shown.
75		1 – Electric Technic Mini-Motor 9V Slide the door rail plates of into the motor supports.

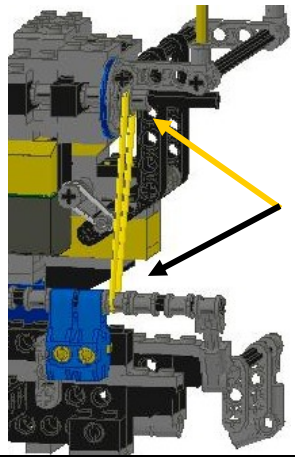
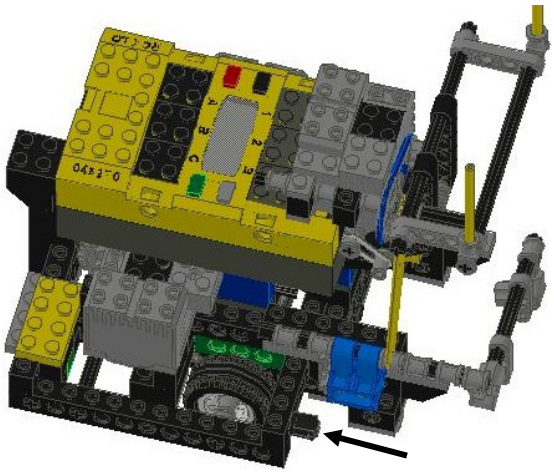
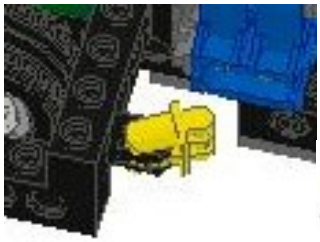
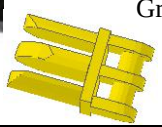
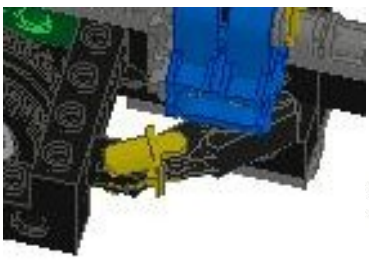
76		2 – Plate 2 x 2 Affix the motor to the assembly using the two 2 x 2 plates on the top and bottom, as shown.
77		1 – Technic Wedge Belt Wheel 2 – Electric Technic Micromotor Pulley 1 – Technic Rubber Band Blue Slide a wedge belt wheel onto the 10 axle and follow it with a micromotor pulley. Slide a micromotor pulley onto the motor shaft. Connect the micromotor pulley on the motor shaft and the wedge belt wheel with a blue technic rubber band.
78		Place the completed trap motor assembly onto the RCX, as shown. The curve on the bottom of the motor should rest between the two corner plates installed in Step 71 and the two 2 x 1 inverted slope bricks installed in Step 69 should support the front of the motor.

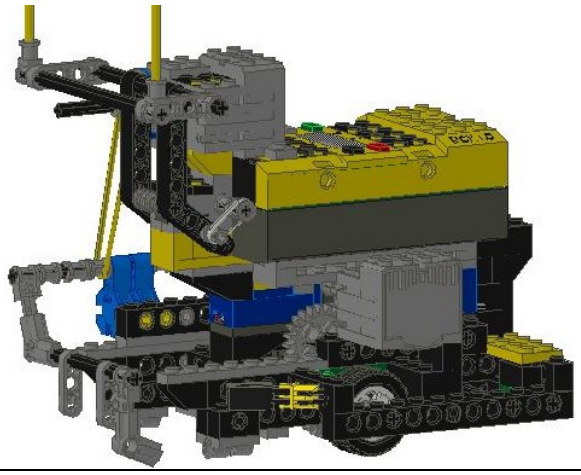
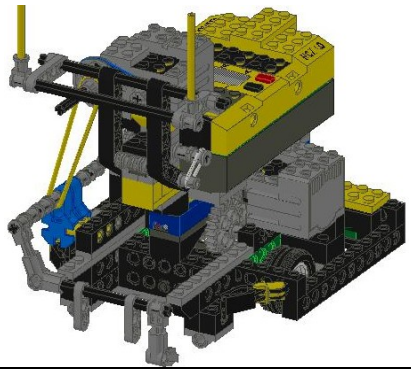
79		<u>Bumper Assembly</u>: 1 – Technic Axle 4 Insert the 4 axle into the holes in the 1 x 2 bricks installed in Step 66.
80		1 – Technic Lift Arm 1 x 9 Bent 1 – Technic Axle 4 4 – Technic Bush ½ Smooth Slide the lift arm onto one end of the 4 axle installed in the last step. The axle hole on the smaller end of the lift arm should be used. Place the 4 half bushings onto the center of the other 4 axle. Insert this axle into the hole at the corner of the bent lift arm.
81		1 – Technic Lift Arm 1 x 9 Bent Slide the technic lift arm onto the two 4 axles. Ensure that the four half bushings installed in the last step are in a position to fully depress the touch sensor button. It is usually most effective when the edges of two half bushing are centered on the touch sensor.

82	 Technic Pin	1 – Technic Axle 12 2 – Technic Pin Insert the 12 axle through the top two axle holes of the 1 x 9 technic lift arms. Center the axle and be sure that it rests on top of the 10 axle of the trap motor assembly. Next, place the two technic pins into the second hole of each technic lift arm, the hole right above the axle installed in Step 79.
83	 Liftarm 1 x 4	2 – Technic Liftarm 1 x 4 Slide a liftarm onto each end of the 12 axle, as shown. They should be at least a plate's width in from the ends.
84	 Connector	1 - Technic Axle 12 2 – Technic Connector with Axlehole 2 – Technic Flex-System Hose Slide the 12 axle through the end axleholes of the 1 x 4 liftarms installed in the last step. Place a connector with axlehole on each end of the 12 axle. Insert a flex-system hose into each connector.

85		2 – Technic Rubber Band White Place one rubber band on each side of the bumper, connecting the technic pin from Step 82 to the technic bush from Step 70.
86		<u>Trap Arm Assembly</u>: 1 – Plate 1 x 6 1 – Technic Brick 1 x 6 with Holes Place the 1 x 6 technic brick on top of the 1 x 6 plate. Install this assembly on the model's right side, as shown.
87		2 – Technic Axle Pin Insert an axle pin into the center hole of the 1 x 6 technic brick installed in the last step. Insert the other axle pin into the hole in the 1 x 4 technic brick immediately beneath.
88	 Liftarm 1x3 Double Pin	1 – Technic Liftarm 1 x 3 1 – Technic Pin 3L Double Slide the 1 x 3 liftarm over the two axle pins installed in the last step. Insert the 3L double pin into the two holes of the 1 x 6 technic brick as shown.

89	  Connector Block	1 – Technic Connector Block 3 x 2 x 2 1 – Technic Axle 10 5 – Technic Bush Place the connector block on the double pin installed in the step above. Slide the 10 axle through the top hole of the connector block, as shown. Place two bushings on the back of the axle and three on the front, as shown.
90	 Pole Reverser Handle  #5 Angle Connector 	1 – Technic Pole Reverser Handle 1 – Technic Angle Connector #5 1 – Technic Axle 8 Slide the pole reverser handle onto the front end of the 10 axle from the previous step. Next, place the #5 angle connector on the end of the pole reverser handle, as shown. Then, place the 8 axle into the other end of the #5 angle connector.
91		2 – Technic Liftarm 1 x 3 1 – Technic Pole Reverser Handle 1 – Technic Angle Connector #5 Slide the two liftarms onto the 8 axle from the previous step, as shown. Slide the pole reverser handle onto the 8 axle with the axle portion facing downward. Place the #5 angle connector onto the pole reverser handle, as shown.

92		1 – Technic Rubber Band Yellow Connect a yellow technic rubber band from the bushing on the trap arm to the micromotor pulley installed in Step 77.
93	  Arm Piece with Pin and 2 Fingers	<u>Side Guards</u>: 1 – Arm Piece with Pin and 2 Fingers Slide the armpiece pin into the first hole of the 1 x 16 technic brick. The slots between the two fingers should be sideways.
94	  Grab Jaw Holder	1 – Grab Jaw Holder Snap the three fingers of the grab jaw holder around the two fingers of the arm piece installed above.
95	  Grab Jaw	2 – Grab Jaw Snap the two grab jaws into the grab jaw holder installed in the last step.

96		1 – Arm Piece with Pin and 2 Fingers 1 – Grab Jaw Holder 2 – Grab Jaw Repeat Steps 93 – 95 for the other side.
97		<u>RCX Rear Support</u>: 3 – Plate 1 x 2 1 – Brick 1 x 2 2 – Technic Brick 1 x 2 with Hole 1 – Plate 2 x 4 Stack the three plates and three bricks as shown. Tie them into the RCX using the 2 x 4 plate as shown.
		The robot model is now complete.

The table below, Table 4, lists the port configurations from the RCX to the individual motors and sensors. If the motor turns the wrong way during operation, rotate one of the plugs connected to that motor 180 degrees.

Table 4: RCX Port Configuration

Port	**Motor / Sensor**
A	Trap Motor
B	Linear Motor (on the left, closest to the front)
C	Turn Motor
1	Turn Rotation Sensor (closest to the back)
2	Light Sensor / Touch Sensor (ganged)
3	Linear Rotation Sensor

APPENDIX F

CODE FOR THE REACTIVE ROBOTIC ARCHITECTURE

CODE FOR THE REACTIVE ROBOTIC ARCHITECTURE

The code for the reactive robotic architecture is split into multiple files with each file encompassing a specific behavior. A subsumption architecture is used to coordinate the individual behaviors.

```
////////////////////////////////////////////////////////////////////
// File:    Reactive.nqc
// Version: 1.1.1
// Author:  Gary R. Mayer
// Date:    24 August 2003
// Project: SIUE CIS 595 - Master's Thesis Project
//
// Title:   IMPLEMENTATION OF A DELIBERATIVE ROBOT CONTROL
//          ARCHITECTURE ON AN INEXPENSIVE ROBOT PLATFORM
//
// Description:   This document provides the main task control for a
// reactive robot control architecture. Individual robot behaviors
// are coded in their respective files.
//
// Arbitration between active states is managed by an arbitration
// subfunction within main()
//
// Logic is based upon the basic Finite State Acceptor Diagram and
// Suppression Arbitration Networks.
//
// The code is written in Not Quite C (NQC), a C-like programming
// language that resides on top of the RCX brick's default firmware.
//
// RCX firmware version 2.0;  NQC version 2.5 R1
////////////////////////////////////////////////////////////////////


#include "RGlobals.nqc"             // global constants and variables
#include "RForage.nqc"              // Forage behavior code
#include "RAcquire.nqc"             // Acquire-Target behavior code
#include "RReturn.nqc"              // Return-home behavior code
#include "RRelease.nqc"             // Release-target behavior code
#include "RAvoid.nqc"               // Avoid-obstacle behavior code


task main()
{
      int behavior_command;

      // clear display
      SetUserDisplay(0, 0);
```

```
// initialize input / output ports
Off(DRIVE_MOTOR);
Off(TURN_MOTOR);
SetPower(DRIVE_MOTOR, OUT_FULL);
SetPower(TURN_MOTOR, OUT_FULL);
SetSensorType(LIGHT_SENSOR, SENSOR_TYPE_LIGHT);
SetSensorMode(LIGHT_SENSOR, SENSOR_MODE_PERCENT);
SetSensor(DRIVE_ENCODER, SENSOR_ROTATION);
SetSensor(TURN_ENCODER, SENSOR_ROTATION);
SetTxPower(TX_POWER_LO);      // set IR output level
SetUserDisplay(0, 0);         // clear display

// Calibrate ambient and target light values.
// Use bumper on light sensor as user input.
PlaySound(SOUND_DOUBLE_BEEP);
while ( LIGHT_SENSOR < OBSTACLE_THRESHOLD );
while ( LIGHT_SENSOR > OBSTACLE_THRESHOLD );

PlaySound(SOUND_CLICK);
CalibrateLight();
gAmbientLevel =   gLightValue + LIGHT_BUFFER_AMB;
SetUserDisplay(gAmbientLevel, 0);

PlaySound(SOUND_DOUBLE_BEEP);
while ( LIGHT_SENSOR < OBSTACLE_THRESHOLD );
while ( LIGHT_SENSOR > OBSTACLE_THRESHOLD );

PlaySound(SOUND_CLICK);
CalibrateLight();
gTargetThreshold = gLightValue - LIGHT_BUFFER_TGT;
SetUserDisplay(gTargetThreshold, 0);

PlaySound(SOUND_DOUBLE_BEEP);
while ( LIGHT_SENSOR < OBSTACLE_THRESHOLD );
while ( LIGHT_SENSOR > OBSTACLE_THRESHOLD );

// Ensure disparity exists btwn ambient and target values.
if ( gTargetThreshold - gAmbientLevel < TARGET_DIFF )
{
      PlaySound(SOUND_DOWN);
      return;
}

Wait(WAIT_TIME);
gLightValue = gAmbientLevel;
SetUserDisplay(gLightValue, 0);

// Start default robot behaviors
start avoid_task;
start release_task;
start return_task;
start acquire_task;
start forage_task;
```

```
	// Arbitrate robot shared resources.
	while (1)
	{
		behavior_command = COMMAND_NONE;
		gLightValue = LIGHT_SENSOR;
		gDriveEncoderTicks = DRIVE_ENCODER;
		gTurnEncoderTicks = TURN_ENCODER;
		gIR_Message = Message();

		if ( gAvoidCommand != COMMAND_NONE )
		{
			behavior_command = gAvoidCommand;
			SetUserDisplay(1, 0);
		}

		else if ( gReleaseCommand != COMMAND_NONE )
		{
			behavior_command = gReleaseCommand;
			SetUserDisplay(2, 0);
		}

		else if ( gReturnCommand != COMMAND_NONE )
		{
			behavior_command = gReturnCommand;
			SetUserDisplay(3, 0);
		}

		else if ( gAcquireCommand != COMMAND_NONE )
		{
			behavior_command = gAcquireCommand;
			SetUserDisplay(4, 0);
		}

		else if ( gForageCommand != COMMAND_NONE )
		{
			behavior_command = gForageCommand;
			SetUserDisplay(5, 0);
		}

		ExecuteCommand(behavior_command);
	}

}  // end task main()

void ExecuteCommand(int& behavior_command)
{
	switch (behavior_command)
	{
	case COMMAND_STOP:
		Off(DRIVE_MOTOR);
		Off(TURN_MOTOR);
		Float(TRAP_MOTOR);
		break;
```

```
case COMMAND_FORWARD:
      Off(TURN_MOTOR);
      Float(TRAP_MOTOR);
      OnFwd(DRIVE_MOTOR);
      break;

case COMMAND_REVERSE:
      Off(TURN_MOTOR);
      Float(TRAP_MOTOR);
      OnRev(DRIVE_MOTOR);
      break;

case COMMAND_LEFT:
      Off(DRIVE_MOTOR);
      Float(TRAP_MOTOR);
      OnRev(TURN_MOTOR);
      break;

case COMMAND_RIGHT:
      Off(DRIVE_MOTOR);
      Float(TRAP_MOTOR);
      OnFwd(TURN_MOTOR);
      break;

case COMMAND_CAPTURE:
      Off(DRIVE_MOTOR);
      Off(TURN_MOTOR);
      OnFwd(TRAP_MOTOR);
      Wait(TRAP_TIME);
      Float(TRAP_MOTOR);
      break;

case COMMAND_RELEASE:
      Off(DRIVE_MOTOR);
      Off(TURN_MOTOR);
      OnRev(TRAP_MOTOR);
      Wait(TRAP_TIME);
      Float(TRAP_MOTOR);
      break;

case COMMAND_RESET_DRIVE_ENCODER:
      ClearSensor(DRIVE_ENCODER);
      gDriveEncoderTicks = DRIVE_ENCODER;
      break;

case COMMAND_RESET_TURN_ENCODER:
      ClearSensor(TURN_ENCODER);
      gTurnEncoderTicks = TURN_ENCODER;
      break;

case COMMAND_RESET_MESSAGE:
      ClearMessage();
      break;
```

```
	deafult:
		break;
	}
}


sub CalibrateLight()
{
	int i;
	int avg = 0;
	int num_times = 15;


	for ( i = 0; i < num_times; i++ )
	{
		avg = avg + LIGHT_SENSOR;
		Wait(WAIT_TIME);
	}

	avg = avg / num_times;

	gLightValue = avg;
}
```

```
////////////////////////////////////////////////////////////////////
// File:    RGlobals.nqc
// Version: 1.1.2
// Author:  Gary R. Mayer
// Date:    26 August 2003
// Project: SIUE CIS 595 - Master's Thesis Project
// Title:   IMPLEMENTATION OF A DELIBERATIVE ROBOT CONTROL
//          ARCHITECTURE ON AN INEXPENSIVE ROBOT PLATFORM
//
// Description:   This file provides the global constants and
// variables used throughout the reactive robot control
// architecture.
//
// RCX firmware version 2.0;  NQC version 2.5 R1
////////////////////////////////////////////////////////////////////

/* *** PREPROCESSOR DEFINITIONS *** */
/* Motor name assignments. */
#define    TRAP_MOTOR              OUT_A
#define    DRIVE_MOTOR             OUT_B
#define    TURN_MOTOR              OUT_C

/* Sensor name assignments. */
#define    TURN_ENCODER         SENSOR_1
#define    LIGHT_SENSOR         SENSOR_2
#define    DRIVE_ENCODER        SENSOR_3

/* Drive encoder ticks in 10" grid length. */
#define    GRID_LENGTH_TICKS         252

/* Turn encoder ticks in 90 degree turn.  */
#define    TURN_LENGTH_TICKS          64

/* Units of 10ms to wait to allow trap arm to raise or lower. */
#define    TRAP_TIME                 100

/* Units of 10ms to wait for arbitrator to manage command request. */
#define    WAIT_TIME                  45

#define    RESPONSE_TIME              50

/* Note: The rotation sensor (encoder) provides 16 ticks per    */
/* rotation. Both the drive and turn encoders are geared to the */
/* motor with a 1:1 ratio. The encoders are geared down to the  */
/* wheel axle with a 1:6 ratio. Thus, every revolution of the   */
/* wheel yields 96 ticks on the encoder.                        */

/* LIGHT VALUES (0 - 100 scale) */
/*   Higher values indicate brighter viewed source. */

/* Value indicates when a touch sensor is considered impacted. */
#define    OBSTACLE_THRESHOLD        100
```

```
/* Buffer for ambient threshold.                                  */
#define    LIGHT_BUFFER_AMB            5

/* Buffer for target threshold.                                   */
#define    LIGHT_BUFFER_TGT            5
/* Minimum difference allowed between ambient and target calibrations. */
#define    TARGET_DIFF                10

/* BOOLEAN CONSTANTS */
#define    TRUE                        1
#define    FALSE                       0

/* BEHAVIOR COMMANDS */
/* Output commands behaviors request of the arbitrator. */

/* Behavior has no desired output.            */
#define    COMMAND_NONE               -1

#define    COMMAND_STOP                0
#define    COMMAND_FORWARD             1
#define    COMMAND_REVERSE             2
#define    COMMAND_LEFT                3
#define    COMMAND_RIGHT               4
#define    COMMAND_CAPTURE             5
#define    COMMAND_RELEASE             6
#define    COMMAND_RESET_DRIVE_ENCODER 7
#define    COMMAND_RESET_TURN_ENCODER  8
#define    COMMAND_RESET_MESSAGE       9

/* IR MESSAGES */
#define    IR_MSG_MARCO                2
#define    IR_MSG_POLO                 5
#define    IR_MSG_ATHOME              10    /* Impacted Home      */


/* *** GLOBAL VARIABLES *** */

// Sensor variables.
int gLightValue;              // Light value of light/touch sensor.
int gAmbientLevel, gTargetThreshold;       // Threshold values.
int gDriveEncoderTicks = 0;                // Drive encoder reading.
int gTurnEncoderTicks = 0;
int gIR_Message;
int gAvoidCommand, gReleaseCommand, gReturnCommand, gAcquireCommand,
gForageCommand;
int gTargetTrapped = FALSE;           // Internal state of trap arm.
```

```
////////////////////////////////////////////////////////////////
// File:    RForage.nqc
// Version: 1.1.1
// Author:  Gary R. Mayer
// Date:    24 August 2003
// Project: SIUE CIS 595 - Master's Thesis Project
// Title:   IMPLEMENTATION OF A DELIBERATIVE ROBOT CONTROL
//          ARCHITECTURE ON AN INEXPENSIVE ROBOT PLATFORM
//
// Description:   This document provides a portion of code for a
// reactive robot control architecture.  Specifically, it implements
// a Forage behavior.  The details are explained below.
//
// Logic is based upon the basic Finite State Acceptor Diagram and
// Suppression Arbitration Networks.
//
// The code is written in Not Quite C (NQC), a C-like programming
// language that resides on top of the RCX brick's default firmware.
//
// RCX firmware version 2.0;  NQC version 2.5 R1
////////////////////////////////////////////////////////////////


task forage_task()
{
      int direction;
      int ticks;  // Amount to turn; modified randomly.

      gForageCommand = COMMAND_NONE;
      SetRandomSeed(Timer(1));

      // Repeat indefinitely.
      while (1)
      {
            // Randomly determine direction to turn.
            direction = Random(3) + 1;    // Random value of 1 - 4.

            if ( (direction == 2) || (direction == 3) )
            {
                  // Prepare for turn.
                  gForageCommand = COMMAND_RESET_TURN_ENCODER;
                  ticks = Random(TURN_LENGTH_TICKS/4) + 5;
                  Wait(WAIT_TIME);

                  if ( direction == 2 )
                  {
                        // Turn left.
                        gForageCommand = COMMAND_LEFT;
                        while ( gTurnEncoderTicks > -ticks );
                  }
```

```
                else
                {
                      // Turn right.
                      gForageCommand = COMMAND_RIGHT;
                      while ( gTurnEncoderTicks < ticks );
                }

                gForageCommand = COMMAND_STOP;
                Wait(WAIT_TIME);
          }


          // Move forward a random amount.
          gForageCommand = COMMAND_RESET_DRIVE_ENCODER;
          ticks = Random(GRID_LENGTH_TICKS/4) +
                         (GRID_LENGTH_TICKS/ 4);
          Wait(WAIT_TIME);
          gForageCommand = COMMAND_FORWARD;
          while ( gDriveEncoderTicks < ticks );
          gForageCommand = COMMAND_STOP;
          Wait(WAIT_TIME);
     }

}  // end forage_task()
```

```
////////////////////////////////////////////////////////////////
// File:    RAcquire.nqc
// Version: 1.1.2
// Author:  Gary R. Mayer
// Date:    26 August 2003
// Project: SIUE CIS 595 - Master's Thesis Project
// Title:   IMPLEMENTATION OF A DELIBERATIVE ROBOT CONTROL
//          ARCHITECTURE ON AN INEXPENSIVE ROBOT PLATFORM
//
// Description:   This document provides a portion of code for a
// reactive robot control architecture.  Specifically, it implements
// an Acquire-Target behavior.  The details are explained below.
//
// Logic is based upon the basic Finite State Acceptor Diagram and
// Suppression Arbitration Networks.
//
// The code is written in Not Quite C (NQC), a C-like programming
// language that resides on top of the RCX brick's default firmware.
//
// RCX firmware version 2.0;  NQC version 2.5 R1
////////////////////////////////////////////////////////////////


task acquire_task()
{
      int light_reading;
      int ticks;
      gAcquireCommand = COMMAND_NONE;

      // Repeat indefinitely.
      while (1)
      {
            // Get target.
            if ( gTargetTrapped == FALSE )
            {
                  light_reading = gLightValue;

                  // Target in trap; capture it.
                  if ( (light_reading < OBSTACLE_THRESHOLD) &&
                       (light_reading >= gTargetThreshold) )
                  {
                        gAcquireCommand = COMMAND_CAPTURE;
                        Wait(WAIT_TIME);
                        Wait(TRAP_TIME);

                        gAcquireCommand = COMMAND_STOP;
                        Wait(WAIT_TIME);

                        gAcquireCommand = COMMAND_RESET_MESSAGE;
                        Wait(WAIT_TIME);
```

```
      gTargetTrapped = TRUE;
      gAcquireCommand = COMMAND_NONE;
      Wait(WAIT_TIME);
}

// Target seen; move toward it.
else if ( (light_reading > gAmbientLevel) &&
        (light_reading < gTargetThreshold) )
{
      gAcquireCommand = COMMAND_FORWARD;

      while ( (light_reading > gAmbientLevel) &&
            (light_reading < gTargetThreshold) )
      {
            light_reading = gLightValue;
      }

      gAcquireCommand = COMMAND_STOP;
      Wait(WAIT_TIME);

      // Check if target was bypassed.
      if ( light_reading < gTargetThreshold )
      {
            // Look left.
            gAcquireCommand =
               COMMAND_RESET_TURN_ENCODER;
            Wait(WAIT_TIME);

            ticks = -TURN_LENGTH_TICKS / 2;
            gAcquireCommand = COMMAND_LEFT;
            while ( (gTurnEncoderTicks > ticks) &&
                  (gLightValue < gAmbientLevel) );
            gAcquireCommand = COMMAND_STOP;
            Wait(WAIT_TIME);
      }

      if ( gLightValue <= gAmbientLevel )
      {
            // Look right.
            gAcquireCommand =
              COMMAND_RESET_TURN_ENCODER;
            Wait(WAIT_TIME);

            ticks = TURN_LENGTH_TICKS;
            gAcquireCommand = COMMAND_RIGHT;
            while ( (gTurnEncoderTicks < ticks) &&
                  (gLightValue < gAmbientLevel) );
            gAcquireCommand = COMMAND_STOP;
            Wait(WAIT_TIME);
      }
}
```

```
                  // Target not found.
                  else
                  {
                        gAcquireCommand = COMMAND_NONE;
                  }
            }

            // Target is already held. Take no action.
            else
            {
                  gAcquireCommand = COMMAND_NONE;
            }
      }

}  // end acquire_task()
```

```
////////////////////////////////////////////////////////////////
// File:    RReturn.nqc
// Version: 1.1.2
// Author:  Gary R. Mayer
// Date:    27 August 2003
// Project: SIUE CIS 595 - Master's Project
// Title:   IMPLEMENTATION OF A DELIBERATIVE ROBOT CONTROL
//          ARCHITECTURE ON AN INEXPENSIVE ROBOT PLATFORM
//
// Description:   This document provides a portion of code for a
// reactive robot control architecture.  Specifically, it implements
// a Return-home behavior.  The details are explained below.
//
// Logic is based upon the basic Finite State Acceptor Diagram and
// Suppression ArbitrationNetworks.
//
// The code is written in Not Quite C (NQC), a C-like programming
// language that resides on top of the RCX brick's default firmware.
//
// RCX firmware version 2.0;  NQC version 2.5 R1
////////////////////////////////////////////////////////////////


task return_task()
{
      int light_reading;
      int count1, count2;
      int direction;
      int ticks;

      gReturnCommand = COMMAND_NONE;
      SetRandomSeed(Timer(1));

      // Repeat indefinitely.
      while (1)
      {
            // Take no action unless target is trapped.
            if ( gTargetTrapped == FALSE )
            {
                  gReturnCommand = COMMAND_NONE;
                  continue;
            }

            // Send message request to Home.
            SendMessage(IR_MSG_MARCO);
            Wait(RESPONSE_TIME);

            if ( gIR_Message == IR_MSG_POLO )
            {
                  PlaySound(SOUND_CLICK);

                  gReturnCommand = COMMAND_RESET_DRIVE_ENCODER;
                  ticks = Random(GRID_LENGTH_TICKS/3) + 20;
```

```
      Wait(WAIT_TIME);

      gReturnCommand = COMMAND_FORWARD;
      while ( gDriveEncoderTicks < ticks );
}

// No response received or response lost.
// Face a different direction or move.
else
{
      // Random value of 1 - 10.
      direction = Random(10) + 1;

      if ( direction == 3 )
      {
            // Turn right.
            gReturnCommand =
              COMMAND_RESET_DRIVE_ENCODER;
            ticks = Random(TURN_LENGTH_TICKS/4) + 5;
            Wait(WAIT_TIME);

            gReturnCommand = COMMAND_RIGHT;
            while ( (gTurnEncoderTicks < ticks) &&
                  (gIR_Message != IR_MSG_POLO) &&
                  (gIR_Message != IR_MSG_ATHOME) );
      }

      else if ( direction < 8 )
      {
            // Turn left.
            gReturnCommand = COMMAND_RESET_TURN_ENCODER;
            ticks = Random(TURN_LENGTH_TICKS/5) + 5;
            Wait(WAIT_TIME);

            gReturnCommand = COMMAND_LEFT;
            while ( (gTurnEncoderTicks > -ticks) &&
                  (gIR_Message != IR_MSG_POLO) &&
                  (gIR_Message != IR_MSG_ATHOME) );
      }

      else
      {
            // Move forward until robot hits something
            //   or sees a signal from Home.
            gReturnCommand = COMMAND_RESET_MESSAGE;
            Wait(WAIT_TIME);
            gReturnCommand = COMMAND_FORWARD;
```

```
                        while ((gLightValue < OBSTACLE_THRESHOLD) &&
                              (gIR_Message != IR_MSG_POLO) &&
                              (gIR_Message != IR_MSG_ATHOME) )
                        {
                              SendMessage(IR_MSG_MARCO);
                              Wait(WAIT_TIME);
                        }
                  }
            }

            gReturnCommand = COMMAND_STOP;
            Wait(WAIT_TIME);
            gReturnCommand = COMMAND_RESET_MESSAGE;
            Wait(WAIT_TIME);
      }

}  // end return_task()
```

```
////////////////////////////////////////////////////////////////////
// File:    RRelease.nqc
// Version: 1.1.1
// Author:  Gary R. Mayer
// Date:    26 August 2003
// Project: SIUE CIS 595 - Master's Project
// Title:   IMPLEMENTATION OF A DELIBERATIVE ROBOT CONTROL
//          ARCHITECTURE ON AN INEXPENSIVE ROBOT PLATFORM
//
// Description:   This document provides a portion of code for a
// reactive robot control architecture.  Specifically, it implements
// a Home behavior that also releases targets.  The details are
// explained below.
//
// Logic is based upon the basic Finite State Acceptor Diagram and
// Suppression Arbitration Networks.
//
// The code is written in Not Quite C (NQC), a C-like programming
// language that resides on top of the RCX brick's default firmware.
//
// RCX firmware version 2.0;  NQC version 2.2 R2
////////////////////////////////////////////////////////////////////


task release_task()
{
      gReleaseCommand = COMMAND_NONE;

      // Repeat indefinitely.
      while (1)
      {
            // Heard "At Home" message from RCX Home; drop target.
            if ( (gTargetTrapped == TRUE) &&
                 (gIR_Message == IR_MSG_ATHOME) )
            {

                  // Back away from Home a bit.
                  gReleaseCommand = COMMAND_STOP;
                  Wait(WAIT_TIME);
                  gReleaseCommand = COMMAND_RESET_DRIVE_ENCODER;
                  Wait(WAIT_TIME);
                  gReleaseCommand = COMMAND_REVERSE;
                  while ( gDriveEncoderTicks >
                          -(GRID_LENGTH_TICKS / 6) );
                  gReleaseCommand = COMMAND_FORWARD;
                  Wait(WAIT_TIME);
                  gReleaseCommand = COMMAND_STOP;
                  Wait(WAIT_TIME);

                  // Release target.
                  gReleaseCommand = COMMAND_RELEASE;
                  Wait(WAIT_TIME);
                  Wait(TRAP_TIME);
```

```
                    gReleaseCommand = COMMAND_RESET_DRIVE_ENCODER;
                    Wait(WAIT_TIME);
                    gReleaseCommand = COMMAND_REVERSE;
                    while ( gDriveEncoderTicks >
                            -(GRID_LENGTH_TICKS / 4) );
                    gReleaseCommand = COMMAND_STOP;
                    Wait(WAIT_TIME);

                    // Turn robot and prepare to find next target.
                    gReleaseCommand = COMMAND_RESET_TURN_ENCODER;
                    Wait(WAIT_TIME);
                    gReleaseCommand = COMMAND_LEFT;
                    while ( gTurnEncoderTicks >
                            -(TURN_LENGTH_TICKS * 2) );
                    gReleaseCommand = COMMAND_STOP;
                    Wait(WAIT_TIME);

                    gTargetTrapped = FALSE;

                    gReleaseCommand = COMMAND_RESET_MESSAGE;
                    Wait(WAIT_TIME);

                    gReleaseCommand = COMMAND_NONE;
              }

              // Do nothing until target is trapped.
              else
              {
                    gReleaseCommand = COMMAND_NONE;
                    Wait(WAIT_TIME);
              }
        }

}  // end release_task()
```

```
////////////////////////////////////////////////////////////////
// File:    RAvoid.nqc
// Version: 1.1.1
// Author:  Gary R. Mayer
// Date:    24 August 2003
// Project: SIUE CIS 595 - Master's Project
// Title:   IMPLEMENTATION OF A DELIBERATIVE ROBOT CONTROL
//          ARCHITECTURE ON AN INEXPENSIVE ROBOT PLATFORM
//
// Description:   This document provides a portion of code for a
// reactive robot control architecture.  Specifically, it implements
// an Avoid-obstacle behavior.  The details are explained below.
//
// Logic is based upon the basic Finite State Acceptor Diagram and
// Suppression Arbitration Networks.
//
// The code is written in Not Quite C (NQC), a C-like programming
// language that resides on top of the RCX brick's default firmware.
//
// RCX firmware version 2.0;  NQC version 2.4 R2
////////////////////////////////////////////////////////////////


task avoid_task()
{
      int ticks;
      int direction;

      gAvoidCommand = COMMAND_NONE;

      // Repeat indefinitely.
      while (1)
      {
            // Back away and turn if an obstacle is impacted.
            if ( gLightValue >= OBSTACLE_THRESHOLD )
            {
                  // Back away and turn around.
                  gAvoidCommand = COMMAND_STOP;
                  Wait(WAIT_TIME);
                  gAvoidCommand = COMMAND_RESET_DRIVE_ENCODER;
                  Wait(WAIT_TIME);
                  gAvoidCommand = COMMAND_REVERSE;
                  while ( gDriveEncoderTicks >
                          -(GRID_LENGTH_TICKS / 4) );
                  gAvoidCommand = COMMAND_STOP;
                  Wait(WAIT_TIME);

                  direction = Random(4) + 1;
                  gAvoidCommand = COMMAND_RESET_TURN_ENCODER;
                  Wait(WAIT_TIME);
                  ticks = Random(TURN_LENGTH_TICKS/2) +
                            (TURN_LENGTH_TICKS / 4);
```

```
                  if ( (direction == 1) || (direction == 3) )
                  {
                        // Turn left.
                        gAvoidCommand = COMMAND_LEFT;
                        while ( (gTurnEncoderTicks > -ticks) &&
                              (gLightValue < OBSTACLE_THRESHOLD) );
                  }

                  else
                  {
                        // Turn right.
                        gAvoidCommand = COMMAND_RIGHT;
                        while ( (gTurnEncoderTicks < ticks) &&
                              (gLightValue < OBSTACLE_THRESHOLD) );
                  }

                  gAvoidCommand = COMMAND_STOP;
                  Wait(WAIT_TIME);

                  // End behavior if obstacles successfully avoided.
                  if ( gLightValue < OBSTACLE_THRESHOLD )
                  {
                        gAvoidCommand = COMMAND_NONE;
                  }
            }

            else
            {
                  gAvoidCommand = COMMAND_NONE;
            }
      }

}  // end avoid_task()
```

```
////////////////////////////////////////////////////////////////
// File:    RHOME.nqc
// Version: 1.2.2
// Author:  Gary R. Mayer
// Date:    26 August 2003
// Project: SIUE CIS 595 - Master's Project
// Title:   IMPLEMENTATION OF A DELIBERATIVE ROBOT CONTROL
//          ARCHITECTURE ON AN INEXPENSIVE ROBOT PLATFORM
//
// Description:   This document provides the code for a reactive
// robotic system.  Specifically, the robot is provided a return
// home behavior that enables it to locate and maneuver towards a
// fixed location in the arena called HOME.  This code resides on
// the RCX unit that acts as HOME. It awaits an infrared (IR) signal
// from the robot indicating that it is looking for HOME. This
// signal is referred to as “Marco.” HOME then emits a response IR
// signal, “Polo,” to guide the robot toward it. HOME will then emit
// a different IR signal to inform the robot that it has reached
// HOME when the robot hits its bumper, triggering its touch sensor.
//
// Logic is based upon the basic Finite State Acceptor Diagram.
//
// The code is written in Not Quite C (NQC),a C-like programming
// language that resides on top of the RCX brick's default firmware.
//
// RCX firmware version 2.0;  NQC version 2.5 R1
////////////////////////////////////////////////////////////////


#define    BUMPER              SENSOR_1

#define    IR_MSG_MARCO              2
#define    IR_MSG_POLO               5
#define    IR_MSG_ATHOME            10

#define    PAUSE_TIME               30

task main()
{
      int response = 0;
      int echos = 3;

      //Set IR transmission power.
      SetTxPower(TX_POWER_HI);

      // Set bumper sensor.
      SetSensor(BUMPER, SENSOR_TOUCH);
```

```
	// Repeat indefinitely.
	while (1)
	{
		if ( Message() == IR_MSG_MARCO )
		{
			PlaySound(SOUND_DOUBLE_BEEP);

			while ( response < echos )
			{
				SendMessage(IR_MSG_POLO);
				Wait(PAUSE_TIME);
				response++;
			}

			response = 0;
			ClearMessage();
		}

		while (	BUMPER == true )
		{
			SendMessage(IR_MSG_ATHOME);
			PlaySound(SOUND_UP);
			Wait(PAUSE_TIME/2);
		}
	}

}  // end main()
```

APPENDIX G

CODE FOR THE DELIBERATIVE ROBOTIC ARCHITECTURE

CODE FOR THE DELIBERATIVE ROBOTIC ARCHITECTURE

The code for the deliberative robotic architecture is written in IC4, version 4.21, and contained in a single file.

```
////////////////////////////////////////////////////////////////////////
//     File:  Deliberative.ic
//   Author:  Gary R. Mayer
//     Date:  21 August 2003
//  Version:  1.0.0
//  Project:  SIUE CIS 595 - Master's Thesis Project
// Language:  Interactive C v4.2
//
// Description:  This program provides a deliberate robotic architecture
// for a LEGO Mindstorms RCX controller. The program maintains the
// robot's current position in an arena. The arena map is maintained as
// a grid and the program uses a wavefront algorithm to plan movement
// from the robot's current position to a desired goal position.
//
// Obstacles are initially hand designated, as are the goal position(s),
// starting location, and starting heading. At least one goal and the
// starting location must be specified.  When entering data, if a 0 is
// entered for the row number, it assumed that data entry for that object
// type is complete.  If the robot's sensors detect an inconsistency in
// the world map, then it will stop its current progress, update the map,
// and replan.
//
//  Note that it is not intended for the robot to travel diagonally.
////////////////////////////////////////////////////////////////////////

/* PREPROCESSOR CONSTANT DEFINITIONS */
#define    DEBUG                              /* LCD debug output. */
//#define    TRACE

#define    ROWS                         6     /* 4 working rows. */
#define    COLS                         8     /* 6 working columns. */
#define    STEPS                       19     /* Max anticipated steps. */
#define    MAX_GOALS                    5     /* Max number of goals */
#define    MAX_OBSTACLES               10

#define    ROW                          0     /* Row element ref. */
#define    COL                          1     /* Column element ref. */

#define    TRUE                         1
#define    FALSE                        0

#define    NORTH                        0
#define    EAST                         1
#define    SOUTH                        2
#define    WEST                         3
```

```
#define    TRAP_MOTOR                 1
#define    DRIVE_MOTOR                2
#define    TURN_MOTOR                 3

#define    TURN_ENCODER               1
#define    LIGHT_SENSOR               2     /* Touch sensor ganged. */
#define    DRIVE_ENCODER              3

#define    TRAP_SPEED                70
#define    DRIVE_SPEED              100
#define    TURN_SPEED               100

#define    GRID_LENGTH_TICKS        252
#define    TURN_LENGTH_TICKS         70
#define    DRIVE_TICKS_CORRECTION     8     /* Gear slop correction for
                                               Fwd->Rev and Rev->Fwd.  */
#define    TURN_TICKS_CORRECTION      3     /* Gear slop correction for
                                               changing turn direction.*/
#define    TRAP_TIME                1.0

#define    START                      0
#define    OBSTACLE                   1
#define    GOAL                       2

/* Drive and turn motors' direction last traveled. Used to */
/*   modify ticks traveled and compensate for gear slop.   */
#define    FORWARD                    0
#define    REVERSE                    1
#define    LEFT                       2
#define    RIGHT                      3

#define    OBSTACLE_THRESHOLD        70
#define    LIGHT_BUFFER_AMB          40
#define    LIGHT_BUFFER_TGT          50
#define    TARGET_DIFF               25

/* Path creation flags to determine if wavefront algorithm   */
/* should consider known goals as obstacles or consider them */
/* as empty space.                                           */
#define    SHORTEST_PATH              0
#define    AVOID_KNOWN_GOALS          1

/* Error codes. */
#define    ERROR_NO_PATH           -100     /* No path to goal       */
#define    ERROR_NO_GOALS          -110     /* No known goals remain */

/* GLOBAL VARIABLES */
int gGrid[ROWS][COLS];             // Arena (with boundary edges).
int gStartPosition[2];             // Start position robot returns to.
int gCurrentPosition[2];           // Robot's current position in arena.
int gHeading;                      // Robot's internal facing reference.
int gGoals[MAX_GOALS][2];          // Known goal locations in arena.
```

```
int gNumGoals = 0;                  // Number of goals in arena.
int gCurrentGoal = 0;               // Current goal being sought.
int gObstacles[MAX_OBSTACLES][2];   // Obstacle locations in arena.
int gNumObstacles = 0;              // Number of obstacles in arena.

int gPath[STEPS][2];                // Path to nearest goal.
int gHaveTarget = FALSE;            // Target is currently being held.

int gAmbient;                       // Arena ambient light value.
int gTargetThreshold;               // Target ambient light value.

int gMovingForward = FALSE;

int gLastDrive = FORWARD;
int gLastTurn = LEFT;

// Process identification numbers.
int gRetrieve_PID, gLook_PID;

/* Function: main() - primary program flow of control */
void main()
{
    // Clear display.
    printf("");

    // Calbrate light sensor to arena ambient average.
    while ( !prgm_button() );
    while ( prgm_button() );
    printf("CAML");
    beep();

    gAmbient = CalibrateLight() - LIGHT_BUFFER_AMB;

    #ifdef DEBUG
      printf("A%d", gAmbient);
    beep();
    #endif

    while ( !prgm_button() );
    while ( prgm_button() );

    // Calibrate target threshold to target light average.
    printf("CTGL");
    beep();

    gTargetThreshold = CalibrateLight() + LIGHT_BUFFER_TGT;

    #ifdef DEBUG
      printf("T%d", gTargetThreshold);
    beep();
    #endif
```

```
while ( !prgm_button() );
while ( prgm_button() );

// Ensure enough disparity exists between light readings.
if ( gTargetThreshold >= gAmbient - TARGET_DIFF )
  {
    printf("ERR");
    beep();
    beep();
    beep();
    beep();

    return;
}

// Enable encoders.
enable_bidir_encoder(DRIVE_ENCODER);
enable_bidir_encoder(TURN_ENCODER);
reset_encoder(DRIVE_ENCODER);
reset_encoder(TURN_ENCODER);

// Get obstacle positions from user.
GetPosition(OBSTACLE);

// Get goal positions from user.
GetPosition(GOAL);

// Get robot starting position and heading from user.
GetPosition(START);
GetStartHeading();

// Clear display.
printf("");

// Start the processes that monitor the sensors and
// path planning to the known targets.
gLook_PID = start_process(LookAhead());
gRetrieve_PID = start_process(RetrieveGoals());

// Wait until all known goals are found.
while ( (gNumGoals != 0) ||
       (gCurrentPosition[ROW] != gStartPosition[ROW]) ||
       (gCurrentPosition[COL] != gStartPosition[COL]) )
  sleep(0.25);

sleep (1.0);

// Kill all processes and signal completion.
kill_process(gLook_PID);
kill_process(gRetrieve_PID);
```

```
    if ( gHaveTarget == TRUE )
      {
        motor(TRAP_MOTOR, -TRAP_SPEED);
        sleep(TRAP_TIME);
        off(TRAP_MOTOR);

        gHaveTarget = FALSE;
    }

    beep();
    beep();

    return;

}  // end main()


/*********************************/
/**   MAJOR PROCESS FUNCTIONS   **/
/*********************************/
// Process: MonitorTouchSensor
// Monitor touch sensor(s) for possible failed plan
// caused by impact with unknown obstacles.
void LookAhead(void)
{
    int light_reading;

    while (1)
      {
        // Only seek new obstacles / goals while moving forward.
        // Input from turning will yield unexpected results.
        if ( (gNumGoals != 0) &&
            (gMovingForward == TRUE) )
          {
            light_reading = light(LIGHT_SENSOR);

            // Take another reading to allow sensor to settle.
            if ( light_reading < gTargetThreshold )
              //            if ( light_reading < gAmbient )
              {
                sleep(0.25);
                light_reading = light(LIGHT_SENSOR);
            }

            // Check if new obstacle was found.
            if ( light_reading < OBSTACLE_THRESHOLD )
              {
                // Stop current path following, update map, and
                // back robot up to last safe position.
                kill_process(gRetrieve_PID);
                brake(DRIVE_MOTOR);
                gMovingForward = FALSE;
                AddObstacle(gCurrentPosition[ROW], gCurrentPosition[COL]);
```

```
                BackUp();

                // Restart path following process (will force replan).
                gRetrieve_PID = start_process(RetrieveGoals());
            }

            // Check if light sensor found new goal.
            else if ( (gHaveTarget == FALSE) &&
                 ((gCurrentPosition[ROW] != gGoals[gCurrentGoal][ROW]) ||
                  (gCurrentPosition[COL] != gGoals[gCurrentGoal][COL])) &&
                 (light_reading <= gTargetThreshold) )
                {
                  // Stop current path following, update map, and
                  // back robot up to last safe position.
                  kill_process(gRetrieve_PID);
                  brake(DRIVE_MOTOR);
                  gMovingForward = FALSE;
                  AddGoal(gCurrentPosition[ROW], gCurrentPosition[COL]);
                  BackUp();

                  // Restart path following process (will force replan).
                  gRetrieve_PID = start_process(RetrieveGoals());
              }
        }
    }

    return;

}  // end LookAhead()


// Process: RetrieveGoals
// Retrieve known goals and return them to start location.
void RetrieveGoals(void)
{
    int num_steps;
    int light_reading;

    // Repeat indefinitely.
    while (1)
      {
        #ifdef TRACE
          printf("InRG");
        beep();
        sleep(1.5);
        #endif

        num_steps = ERROR_NO_PATH;
```

```
// Ensure path exists between current location and goal.
while ( num_steps == ERROR_NO_PATH )
  {
    // Find nearest goal.
    // If a target is held, the start position is the goal.
    if ( gHaveTarget == TRUE )
      {
        AddGoal(gStartPosition[ROW], gStartPosition[COL]);
        gCurrentGoal = gNumGoals - 1;
    }

    else
      gCurrentGoal = FindClosestGoal();

    #ifdef TRACE
      printf("CG%d", gCurrentGoal);
    tone(250.5, 0.5);
    sleep(3.0);
    #endif

    // Quit if no goals remain.
    if ( gCurrentGoal == ERROR_NO_GOALS )
      return;

    // Create a travel path.
    num_steps = CreatePath(gCurrentGoal, AVOID_KNOWN_GOALS);

    if ( num_steps == ERROR_NO_PATH )
      {
        // Attempt to create path Home failed.
        if ( gHaveTarget == TRUE )
          {
            // Try to create a path through known goals.
            num_steps = CreatePath(gCurrentGoal, SHORTEST_PATH);

            // If the second attempt at returning Home is invalid,
            //   robot is seperated from Home by obstacles.
            if ( num_steps == ERROR_NO_PATH )
              {
                // Signal error and set the number of remaining
                //   goals to 0 to cause main() to exit.
                printf("ERR");
                beep();
                beep();
                beep();
                sleep(2.0);
                gNumGoals = 0;

                return;
            }
        }
```

```
                // Attempt to create path to a goal failed.
                else
                  RemoveGoal(gCurrentGoal);        // Delete goal.
            }
        }

        // Follow travel path to goal.
        FollowPath(num_steps);

        // Deal with target at destination.
        if ( gHaveTarget == FALSE )
          {
            light_reading = light(LIGHT_SENSOR);

            // Capture target goal if it is there.
            if ( (light_reading < gTargetThreshold) &&
                //            if ( (light_reading < gAmbient) &&
                (light_reading > OBSTACLE_THRESHOLD) )
              CaptureTarget();

            else
              {
                // No target; signal plan failure.
                tone(90.0, 0.5);
                tone(10.0, 0.5);
            }
        }

        else
          {
            ReleaseTarget();

            // Announce target dropped and await signal to continue.
            // Note: Human can reposition robot to maintain localization.
            tone(5000.0, 0.5);
            tone(4000.0, 0.5);
            tone(3000.0, 0.35);
            tone(1500.0, 0.25);

            while ( !prgm_button() );
            while ( prgm_button() );

            sleep(1.0);
            gHaveTarget = FALSE;
        }

        // Remove goal from goal list.
        RemoveGoal(gCurrentGoal);
    }

    return;

}  // end RetrieveTargets()
```

```
/*************************************************/
/****         INITIALIZATION FUNCTIONS         ****/
/*************************************************/
// Provide an average of readings for the light sensor over time.
int CalibrateLight(void)
{
    int i;
    int num_samples = 15;
    float wait_time = 0.35;
    int amb = 0;

    for ( i = 0; i < num_samples; i++ )
      {
        amb = amb + light(LIGHT_SENSOR);
        sleep(wait_time);
    }

    amb = amb / num_samples;

    return amb;

}  // end CalibrateLight()


// Get position(s) of obstacle(s), goal(s) and start location. A selection
// of 0 for the row indicates data entry is complete for that data type.
void GetPosition(int type)
{
    int row = 999;            // row number
    int col = 999;            // column number
    int change = FALSE;       // The number should change.
    int select = FALSE;       // The number has been selected.
    int reset_zero;           // Force numbers to cycle between 0 and max
                              //   or 1 and max.

    if ( type == OBSTACLE )
      {
        reset_zero = TRUE;    // Obstacles are optional.
    }

    else
      {
        reset_zero = FALSE;   // A goal and start location must be input.
    }

    while ( row != 0 )
      {
        if ( reset_zero == TRUE )
          {
            row = 0;
        }
```

```
else
  {
    row = 1;
}

select = FALSE;
change = FALSE;

while ( select == FALSE )
  {
    if ( type == OBSTACLE )
      {
        printf("or%d", row);
    }

    else if ( type == GOAL )
        {
          printf("gr%d", row);
      }

      else
        {
          printf("Sr%d", row);
      }

    if ( view_button() )
      {
        change = TRUE;
    }

    if ( change == TRUE )
      {
        row++;

        if ( row > ROWS - 2 )
          {
            if ( reset_zero = TRUE )
              {
                row = 0;
            }

            else
              {
                row = 1;
            }
        }

        change = FALSE;
    }
```

```
    if ( prgm_button() )
      {
        select = TRUE;
    }

    sleep(0.5);
}

select = FALSE;
change = FALSE;
col = 1;

if ( row != 0 )
  {
    while ( select == FALSE )
      {
        if ( type == OBSTACLE )
          {
            printf("oc%d", col);
        }

        else if ( type == GOAL )
            {
              printf("gc%d", col);
          }

          else
            {
              printf("Sc%d", col);
          }

        if ( view_button() )
          {
            change = TRUE;
        }

        if ( change == TRUE )
          {
            col++;

            if ( col > COLS - 2)
              {
                col = 1;
            }

            change = FALSE;
        }

        if ( prgm_button() )
          {
            select = TRUE;
```

```
                    if ( type == GOAL )
                      {
                        reset_zero = TRUE;
                    }
                }

                sleep(0.5);
            }
        }

        if ( row != 0 )
          {
            if ( type == OBSTACLE )
              {
                AddObstacle(row, col);
            }

            else if ( type == GOAL )
                {
                  AddGoal(row, col);
              }

              else
                {
                  gStartPosition[ROW] = row;
                  gStartPosition[COL] = col;
                  gCurrentPosition[ROW] = row;
                  gCurrentPosition[COL] = col;
                  row = 0;
              }
        }
    }

    return;

}  // end GetPosition()


// Get starting heading.
void GetStartHeading(void)
{
    int select = FALSE;
    int change = FALSE;
    gHeading = NORTH;

    while ( select == FALSE )
      {
        if ( gHeading == NORTH )
          {
            printf("Hd N");
        }
```

```
        else if ( gHeading == SOUTH )
            {
              printf("Hd S");
          }

          else if ( gHeading == EAST )
              {
                printf("Hd E");
            }

            else
              {
                printf("Hd W");
            }

        if ( view_button() )
          {
            change = TRUE;
        }

        if ( change == TRUE )
          {
            gHeading = (gHeading + 1) % 4;
            change = FALSE;
        }

        if ( prgm_button() )
          {
            select = TRUE;
        }

        sleep(0.25);
    }

    return;

}  // end GetStartHeading()
```

```
/************************************************/
/****          CARTOGRAPHER FUNCTIONS          ****/
/************************************************/
// Generate map with current data.
void GenerateMap(void)
{
    // Loop variables.
    int row, col, ob;

    // Clear map of prior data.
    for ( row = 0; row < ROWS; row++ )
      {
        for ( col = 0; col < COLS; col++ )
          {
            if ( (row == 0) || (row == ROWS-1) ||
                (col == 0) || (col == COLS-1) )
              {
                gGrid[row][col] = 1;
            }

            else
              {
                gGrid[row][col] = 0;
            }
        }
    }

    // Add obstacles to map.
    for ( ob = 0; ob < gNumObstacles; ob++ )
      UpdateMap(gObstacles[ob][ROW], gObstacles[ob][COL], OBSTACLE);

    // Add current robot location to map.
    UpdateMap(gCurrentPosition[ROW], gCurrentPosition[COL], START);

    return;

}  // end GenerateMap()


// Update map with new information.
void UpdateMap(int row, int col, int identifier)
{
    if ( (row > 0) && (row < ROWS - 1) &&
        (col > 0 ) && (col < COLS - 1) &&
        ((identifier >= 0) && (identifier <= 2)) )
      {
        gGrid[row][col] = identifier;
    }

    return;

}  // end UpdateMap()
```

```
/************************************************/
/****           NAVIGATOR FUNCTIONS            ****/
/************************************************/
// Path planning using the wavefront algorithm.
// Returns number of steps to goal.
int CreatePath(int goal_num, int creation_flag)
{
    // Loop variables.
    int goal, row, col, facing;
    int step_position[2];

    // Monitor if a grid has been updated.
    int updates = TRUE;

    // Wavefront algorithm results.
    int num_steps = 0;

    #ifdef TRACE
      printf("InCP");
    beep();
    sleep(1.5);
    #endif

    // Generate clean map.
    GenerateMap();

    // Set grid for desired goal.
    SetGoal(goal_num, creation_flag);

    // Perform wavefront until no grid spaces are updated.
    updates = TRUE;

    while ( updates == TRUE )
      {
        updates = FALSE;

        for ( row = 1; row < ROWS-1; row++ )
          {
            for ( col = 1; col < COLS-1; col++ )
              {
                if ( gGrid[row][col] > 1 )
                  {
                    if ( gGrid[row-1][col] == 0 )
                      {
                        gGrid[row-1][col] = gGrid[row][col] + 1;
                        updates = TRUE;
                    }

                    if ( gGrid[row][col+1] == 0 )
                      {
                        gGrid[row][col+1] = gGrid[row][col] + 1;
                        updates = TRUE;
                    }
```

```
                    if ( gGrid[row+1][col] == 0 )
                      {
                        gGrid[row+1][col] = gGrid[row][col] + 1;
                        updates = TRUE;
                    }

                    if ( gGrid[row][col-1] == 0 )
                      {
                        gGrid[row][col-1] = gGrid[row][col] + 1;
                        updates = TRUE;
                    }
                }
            }
        }
    }

    gPath[0][ROW] = gCurrentPosition[ROW];
    gPath[0][COL] = gCurrentPosition[COL];
    num_steps = 0;

    // Ensure goal is reachable from current position.
    if ( gGrid[gCurrentPosition[ROW]][gCurrentPosition[COL]] == 0 )
      {
        printf("PERR");
        beep();
        beep();
        beep();
        sleep(2.0);

        return ERROR_NO_PATH;
    }

    step_position[ROW] = gCurrentPosition[ROW];
    step_position[COL] = gCurrentPosition[COL];

    while ( gGrid[step_position[ROW]][step_position[COL]] !=
gGrid[gGoals[gCurrentGoal][ROW]][gGoals[gCurrentGoal][COL]] )
      {
        num_steps++;
        facing = gHeading;   // Start search in direction robot is facing
                             //   to reduce turns.

        while (1)   // Loop exits when lower cost grid space is found.
          {
            if ( (facing == SOUTH) &&
                (gGrid[step_position[ROW]-1][step_position[COL]] <
                   gGrid[step_position[ROW]][step_position[COL]]) &&
                (gGrid[step_position[ROW]-1][step_position[COL]] > 1) )
              {
                step_position[ROW] = step_position[ROW] - 1;
                break;
            }
```

```
            else if ( (facing == EAST) &&
                   (gGrid[step_position[ROW]][step_position[COL]+1] <
                       gGrid[step_position[ROW]][step_position[COL]]) &&
                   (gGrid[step_position[ROW]][step_position[COL]+1] > 1) )
                {
                  step_position[COL] = step_position[COL] + 1;
                  break;
              }

              else if ( (facing == NORTH) &&
                   (gGrid[step_position[ROW]+1][step_position[COL]] <
                      gGrid[step_position[ROW]][step_position[COL]]) &&
                   (gGrid[step_position[ROW]+1][step_position[COL]] > 1) )
                  {
                    step_position[ROW] = step_position[ROW] + 1;
                    break;
                }

                else if ( (facing == WEST) &&
                   (gGrid[step_position[ROW]][step_position[COL]-1] <
                      gGrid[step_position[ROW]][step_position[COL]]) &&
                   (gGrid[step_position[ROW]][step_position[COL]-1] > 1) )
                    {
                      step_position[COL] = step_position[COL] - 1;
                      break;
                  }

                  // Check next direction.
                  facing = (facing + 1) % 4;
        }

        gPath[num_steps][ROW] = step_position[ROW];
        gPath[num_steps][COL] = step_position[COL];
    }

    #ifdef TRACE
      printf("EcCP");
    beep();
    sleep(1.5);
    #endif

    return num_steps;

}  // end CreatePath()
```

```
// Follow path to desired goal.
void FollowPath(int num_steps)
{
    int step, desired_facing;

    #ifdef DEBUG
      printf("Path");
    sleep(1.0);
    #endif

    for ( step = 1; step <= num_steps; step++ )
      {
        if ( gPath[step][ROW] == gCurrentPosition[ROW] - 1 )
          {
            desired_facing = NORTH;
        }

        else if ( gPath[step][ROW] == gCurrentPosition[ROW] + 1 )
            {
              desired_facing = SOUTH;
          }

          else if ( gPath[step][COL] == gCurrentPosition[COL] + 1 )
              {
                desired_facing = EAST;
          }

          else
            {
              desired_facing = WEST;
          }

        if ( desired_facing == ((gHeading + 3) % 4) )
          {
            TurnLeft();
        }

        else
          {
            while ( gHeading != desired_facing )
              {
                TurnRight();
            }
        }

        MoveForward();

        #ifdef DEBUG
          // Display [Row, Column, Step]
          // Note: Step counts down when moving toward a goal;
          //       up when moving toward Start.
          printf("%d%d%d", gCurrentPosition[ROW], gCurrentPosition[COL],
                 gGrid[gPath[step][ROW]][gPath[step][COL]]);
```

```
        beep();
        sleep(0.5);
        #endif
    }

    return;

}  // end FollowPath()


// Sets map to allow wavefront pathing to a single, desired goal.
// The flag variable determines if known goal positions are
//   considered obstacles or passable. Default is as obstacle.
void SetGoal(int goal_num, int flag)
{
    int g;
    int marker = OBSTACLE;

    if ( flag == SHORTEST_PATH )
      marker = 0;

    // All goals beside the one being sought are set per the flag.
    for ( g = 0; g < gNumGoals; g++ )
      {
        if ( g == goal_num )
          UpdateMap(gGoals[g][ROW], gGoals[g][COL], GOAL);

        else
          UpdateMap(gGoals[g][ROW], gGoals[g][COL], marker);
    }

    gCurrentGoal = goal_num;

    #ifdef DEBUG
      printf("g%d", goal_num);
    sleep(1.0);
    #endif

    return;

}  // end SetGoal()


// Find the goal closest to current location.
// Returns goal number of the closest goal.
int FindClosestGoal(void)
{
    int goal_num, g, steps;
    int min_steps = 999;

    #ifdef TRACE
      printf("InFG");
    beep();
```

```
    sleep(1.5);

    printf("nG %d", gNumGoals);
    beep();
    sleep(1.5);
    #endif

    if ( gNumGoals == 0 )
      {
        #ifdef TRACE
          printf("EcFG");
        beep();
        sleep(1.5);
        #endif

        return ERROR_NO_GOALS;
    }

    if ( gNumGoals == 1 )
      return 0;

    // Default; should no goals be reachable.
    goal_num = ERROR_NO_GOALS;

    for ( g = 0; g < gNumGoals; g++ )
      {
        steps = CreatePath(g, SHORTEST_PATH);

        if ( (steps != ERROR_NO_PATH) &&
             (steps < min_steps) )
          {
            min_steps = steps;
            goal_num = g;
        }
    }

    #ifdef TRACE
      printf("EcFG");
    beep();
    sleep(1.5);
    #endif

    return goal_num;

}  // end FindClosestGoal()
```

```
// Add a goal to the goal list.
void AddGoal(int goal_row, int goal_col)
{
    int g;

    // Ensure maximum number of goals isn't exceeded.
    if ( gNumGoals == MAX_GOALS )
      {
        printf("maxg");
        beep();
        beep();
        beep();
        sleep(2.0);

        return;
    }

    // Ensure goal coordinates are valid.
    if ( (goal_row <= 0) || (goal_row > ROWS - 2) ||
        (goal_col <= 0) || (goal_col > COLS - 2) )
      {
        return;
    }

    // Ensure goal doesn't already exist.
    for ( g = 0; g < gNumGoals; g++ )
      {
        if ( (gGoals[g][ROW] == goal_row) &&
            (gGoals[g][COL] == goal_col) )
          {
            return;
        }
    }

    // Enter new goal into goal list.
    gNumGoals++;

    gGoals[gNumGoals - 1][ROW] = goal_row;
    gGoals[gNumGoals - 1][COL] = goal_col;

    #ifdef DEBUG
      printf("g%d%d%d", gNumGoals - 1, goal_row, goal_col);
    tone(3500.0, 0.25);
    tone(5050.5, 0.25);
    sleep(2.0);
    #endif

    return;

}  // end AddGoal()
```

```
// Remove goal from goal list.
void RemoveGoal(int goal_num)
{
    if ( goal_num != gNumGoals - 1 )
      {
        gGoals[goal_num][ROW] = gGoals[gNumGoals - 1][ROW];
        gGoals[goal_num][COL] = gGoals[gNumGoals - 1][COL];
    }

    gGoals[gNumGoals - 1][ROW] = 0;
    gGoals[gNumGoals - 1][COL] = 0;

    gNumGoals = gNumGoals - 1;

    return;

}  // end RemoveGoal()


// Add obstacle to obstacle list.
void AddObstacle(int obstacle_row, int obstacle_col)
{
    int ob;

    // Ensure maximum number of obstacles isn't exceeded.
    if ( gNumObstacles == MAX_OBSTACLES )
      {
        printf("maxo");
        beep();
        beep();
        beep();
        sleep(2.0);

        return;
    }

    // Ensure obstacle coordinates are valid.
    if ( (obstacle_row <= 0) || (obstacle_row > ROWS - 2) ||
        (obstacle_col <= 0) || (obstacle_col > COLS - 2) )
      {
        return;
    }

    // Ensure obstacle doesn't already exist.
    for ( ob = 0; ob < gNumObstacles; ob++ )
      {
        if ( (gObstacles[ob][ROW] == obstacle_row) &&
            (gObstacles[ob][COL] == obstacle_col) )
          {
            return;
        }
    }
```

```
    // Enter new obstacle into obstacle list.
    gNumObstacles++;

    gObstacles[gNumObstacles - 1][ROW] = obstacle_row;
    gObstacles[gNumObstacles - 1][COL] = obstacle_col;

    #ifdef DEBUG
      printf("o%d%d%d", gNumObstacles, obstacle_row, obstacle_col);
    sleep(2.0);
    #endif

    return;

}  // end AddObstacle()


/************************************************/
/****             PILOT FUNCTIONS              ****/
/************************************************/
// Move the robot forward.
void MoveForward(void)
{
    int ticks = GRID_LENGTH_TICKS;

    if ( gHeading == NORTH )
      {
        if ( gCurrentPosition[ROW] == 1 )
          {
            return;
        }

        gCurrentPosition[ROW] = gCurrentPosition[ROW] - 1;
    }

    else if ( gHeading == EAST )
        {
          if ( gCurrentPosition[COL] == COLS - 2 )
            {
              return;
          }

          gCurrentPosition[COL] = gCurrentPosition[COL] + 1;
      }

      else if ( gHeading == SOUTH )
          {
            if ( gCurrentPosition[ROW] == ROWS - 2 )
              {
                return;
            }

            gCurrentPosition[ROW] = gCurrentPosition[ROW] + 1;
        }
```

```
        else
          {
            if ( gCurrentPosition[COL] == 1 )
              {
                return;
            }

            gCurrentPosition[COL] = gCurrentPosition[COL] - 1;
        }

    printf("fd");
    sleep(0.5);

    if ( gLastDrive == REVERSE )
      ticks -= DRIVE_TICKS_CORRECTION;

    reset_encoder(DRIVE_ENCODER);
    gMovingForward = TRUE;
    motor(DRIVE_MOTOR, DRIVE_SPEED);
    while ( read_encoder(DRIVE_ENCODER) < ticks );
    brake(DRIVE_MOTOR);

    gMovingForward = FALSE;
    gLastDrive = FORWARD;

    return;

}  // end MoveForward()


// Back robot up to last position.
void BackUp(void)
{
    int ticks = 0;

    if ( gHeading == NORTH )
      {
        gCurrentPosition[ROW] = gCurrentPosition[ROW] + 1;
    }

    else if ( gHeading == EAST )
        {
          gCurrentPosition[COL] = gCurrentPosition[COL] - 1;
      }

      else if ( gHeading == SOUTH )
          {
            gCurrentPosition[ROW] = gCurrentPosition[ROW] - 1;
        }
```

```
        else
          {
            gCurrentPosition[COL] = gCurrentPosition[COL] + 1;
        }

    #ifdef DEBUG
      printf("bu");
    sleep(1.0);
    #endif

    if ( gLastDrive == FORWARD )
      ticks = DRIVE_TICKS_CORRECTION;

    motor(DRIVE_MOTOR, -DRIVE_SPEED);
    while ( read_encoder(DRIVE_ENCODER) > ticks );
    brake(DRIVE_MOTOR);
    reset_encoder(DRIVE_ENCODER);

    gLastDrive = REVERSE;

    return;

}  // end BackUp()


// Turn the robot left.
void TurnLeft(void)
{
    int ticks = TURN_LENGTH_TICKS;

    gHeading = (gHeading + 3) % 4;

    #ifdef DEBUG
      printf("tl");
    #endif

    if ( gLastTurn == RIGHT )
      ticks -= TURN_TICKS_CORRECTION;

    reset_encoder(TURN_ENCODER);
    motor(TURN_MOTOR, TURN_SPEED);
    while ( read_encoder(TURN_ENCODER) < ticks );
    brake(TURN_MOTOR);

    gLastTurn = LEFT;

    return;

}  // end TurnLeft()
```

```
// Turn the robot right.
void TurnRight(void)
{
    int ticks = -TURN_LENGTH_TICKS;

    gHeading = (gHeading + 1) % 4;

    #ifdef DEBUG
      printf("tr");
    #endif

    if ( gLastTurn == LEFT )
      ticks += TURN_TICKS_CORRECTION;

    reset_encoder(TURN_ENCODER);
    motor(TURN_MOTOR, -TURN_SPEED);
    while ( read_encoder(TURN_ENCODER) > ticks );
    brake(TURN_MOTOR);

    gLastTurn = RIGHT;

    return;

}  // end TurnRight()


// Capture target at current location.
void CaptureTarget(void)
{
    #ifdef DEBUG
      printf("cap");
    #endif

    motor(TRAP_MOTOR, TRAP_SPEED);
    sleep(TRAP_TIME);
    off(TRAP_MOTOR);

    gHaveTarget = TRUE;

    return;

}  // end CaptureTarget()


// Release captured target.
void ReleaseTarget(void)
{
    #ifdef DEBUG
      printf("rel");
    #endif

    motor(TRAP_MOTOR, -TRAP_SPEED);
    sleep(TRAP_TIME);
```

```
    off(TRAP_MOTOR);
    return;
} // end ReleaseTarget( )
```