Junior Design Final Report: Autonomous Navigation

Daniel Guillette

James Mason

Byron Marohn

Richard Rivera

New Mexico Institute of Mining and Technology

Electrical Engineering Junior Design – Team GRMM / Team D

May 4, 2011

Table of Contents

Table of Figures

Abstract

The goal of this paper is to outline the processes used, problems encountered and specific details of building a robot capable of autonomously navigating a series of waypoints in a courtyard at the New Mexico Institute of Mining and Technology. Of the many options available for completing this challenge, not all of the initial selections were incorporated due to R&D problems. However, the final product was ultimately successful though the use of a rotating array of sensors to take in information about its environment, exterior mounted IR sensors, front mounted immediate proximity violation detection sensors, a compass, and wheel encoders. The navigation strategy employed involved dead reckoning over small distances and constantly recalibrating the robot’s location against known landmarks. A power system using two sources and a tiered distribution system proved effective in providing all components plenty of power and minimized wiring problems. Through the proper incorporation of these systems and supporting software, the robot successfully navigated to all five waypoints and found 4 of the five waypoint objects while being under budget and on time. Given slightly more time, this approach is more than capable of completing every aspect of this challenge.

Index Terms—Autonomous Navigation, Control Systems, Localization, Robotics

Introduction

Vehicles that can autonomously drive themselves are becoming more real every day. There are many components necessary to build a robot capable of navigating on its own, like a solid chassis, an array of environmental detection sensors, and an intelligent logic system. The purpose of the robot presented here was to find and identify a diverse set of waypoints, using a number of components including object detection and localization sensors and a means of controlling the robot efficiently using a microcontroller. Of the many options available for completing this challenge not all of the initial selections were incorporated due to R&D problems.

What follows is an in-depth view of the challenge to overcome, the design of the robot, and how that design was implemented.

Objective

The purpose of this robotics project was to design a robot capable navigating through a series of waypoints in a courtyard at the New Mexico Institute of Mining and Technology, dubbed the “courtyard challenge.” Each waypoint was indicated by a red circle with a radius of approximately three meters. Within each waypoint there was an object that fit within a 50 cm x 50 cm x 50 cm box. The robot had to detect these objects based on color, visual contrast, temperature, ultrasonic radiation, and/or reflectivity. In addition, the robot needed to travel to a specified location given only the information relative to another waypoint. When an object is found, the robot had to approach it within one meter and perform some action to signify that the object was located. A picture of the courtyard layout is shown in Figure 1. The first object was randomly chosen the day of the challenge. The robot had to search for any identifiable object characteristics in the area, constrained to those listed above. The second waypoint contained an object that was dramatically colder than its surroundings. The third object was an ultra-sonic beacon. The fourth waypoint did not contain an object but was a specified location relative to the ultra-sonic beacon with a directional heading and a distance. The fifth object was also randomly chosen, similar to the first. It was located just outside the Figure 1's field of vision, on a concrete sidewalk. The total cost of all components on the robot was not to exceed $750.

Figure 1 – Courtyard Map

Provided Equipment

While many of the components needed to overcome this challenge will be detailed later, several basic parts were provided to use as the root of the system. These include a robot chassis with motors, a high performance 6.6v battery, h-bridge motor controllers, and a micro controller evaluation board. Figure 2 shows the provided battery, and Figure 3 shows the h-bridge motor controllers.

Figure 2 – Main Battery

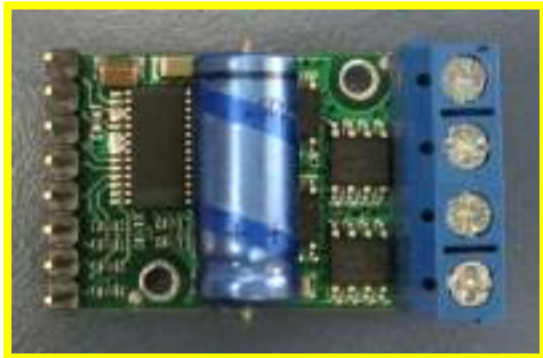

Figure 3 – H-Bridge Motor Controllers

The robot chassis provided for this project was the Dagu Wild Thumper 4WD All-Terrain Chassis. This chassis, shown in Figure 4, comes with four motors, four wheels, and a simple suspension system. Motors are connected together in parallel on each side; both front and back wheels on the left and right side of the robot spin together in unison. The motors on each side were driven by one of the high capacity h-bridges, which allowed the direction and speed of the connected motors to be controlled by the microcontroller. To turn, the motors on one side must be set to turn faster than the motors on the opposite side. This method of movement is sometimes called skid-steering.

Figure 4 – Chassis

The microcontroller used to control the system was the Motorola MC9S12 evaluation board, which comes equipped with an I2C interface, 8 analog to digital (A/D) converters, 8 PWM signal outputs, a real time clock, an LCD display, many general purpose input / output (IO) pins, and several general purpose switches / LEDs / buttons. The controller, shown in Figure 5, operates at a 24 MHz clock speed.

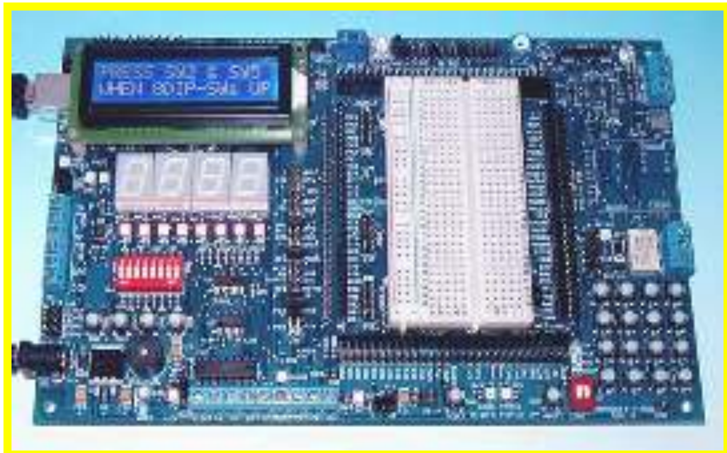

Figure 5 – Microcontroller

Hardware Approach

Sensors for the robot were located on a servo-controlled rotating head platform. This concept was inspired by Carnegie Mellon University's gimbal mounted laser that led them to victory in the DARPA Grand Challenges [1]. The sensors that were used to detect the object at each waypoint were mounted around the rotating gimbal like platform. A diagram of what this looked like is shown in Figure 6. This allowed the robot to search a wide area for a waypoint without physically moving the chassis. The servo used had a 180 degrees range of motion and high accuracy. With the accuracy of the servo, the robot was able to calculate the angle of each sensor with respect to the chassis, so that when a waypoint was located the chassis can turn to face it. The servo was also able to turn the spinning head so that any sensor was in the forward position, or any other desired direction, allowing the robot to track an object as it navigated through the course.

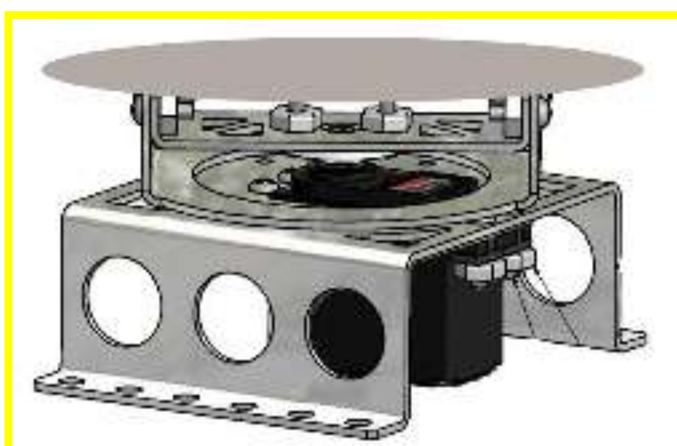

Figure 6 – Rotating Sensor Platform

Sensors

To obtain information about the world around it, a robot must have sensors. Of the many different types of sensors available, several have been identified as useful for this project. Among these are wheel encoders to measure distance traveled, and infrared sensors for temperature, reflectivity, and range. Other useful sensors include a camera for color recognition, a compass for navigation, bumpers for collision detection, and an ultrasonic sensor for finding the third waypoint.

Infrared Sensors

Infrared (IR) receivers for measuring distance were one of the most useful sensors available. Although they are not as accurate as lasers, they were perfect for the courtyard challenge. Lasers tend to be costly and large in size, which make them non-ideal for small form-factor robots. IR sensors, on the other hand, are small and cheap in comparison to lasers. To successfully use IR for navigation, both short-range and long-range IR sensors were used on the robot. The collaboration between the two sensor ranges allowed the robot to find objects at a distance while retaining high accuracy at close ranges. The short-range sensors also helped to detect immediate obstacles.

Figure 7 shows one of the Sharp GP2 series IR sensors chosen. These sensors were reasonably priced and had various range options allowing the tailoring of specific ranges to focus on. The Robot was outfitted with two 10 - 80 cm mid-range sensors and one 20 -150cm long-range IR sensor. Initially it was considered to place the two mid range sensors in the front of the robot to detect any upcoming obstacles. Instead two antenna type bumpers were placed in the front along with the single long-range IR sensor. The mid-range IR sensors were then relocated to the sides of the robot. This allowed detection of passing obstacles and even the waypoint objects, because the maximum range of these sensors was one meter, the maximum range at which the robot could “find” the object.

Figure 7 – Long Range IR Sensor

Camera

These days, a simple, small, and cheap digital surface mount camera can be easily obtained and programmed to take a low-resolution picture of the environment. This data can then be transferred to the microcontroller in red-green-blue (RGB) format via several parallel wires. Camera information could be used in many different ways, including color detection, line detection, localization and even obstacle avoidance. The camera was originally planned to be mounted on the rotating sensor platform. The goal was to have the camera take a few photos to capture the entire surrounding environment and not for it to run continuously. The biggest drawback to using a camera was the time that would have been required to process the picture. Implementation of this sensor using the slow 24 MHz clock on the MC9S12 would have been difficult. The camera chosen also would have to accommodate for variations in the amount of ambient light, including both sunny and cloudy days.

The camera selected was the ¼-Inch SOC VGA CMOS Digital Image Sensor, shown in Figure 8. This component offers built in ambient light filters, automatic exposure adjustments, and onboard image scaling down to 240 x 180 resolution, which would have been manageable on the MC9S12. The intended use for the camera was to identify colored and reflective objects within the waypoints. Ultimately, the camera was deemed to be unnecessary; all of the objects that were used in the challenge, including the colored object, were easily detectable by much simpler range-finding devices.

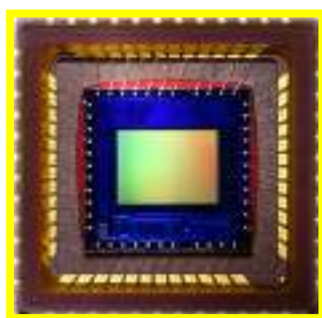

Figure 8 – Camera

Temperature Sensor

The ideal way to measure temperature is with a laser temperature reader. Sadly, laser temperature readers are expensive and large in size, making them infeasible for this project. The other two alternatives are short-range temperature probes and IR temperature sensors. An IR temperature sensor was the best choice for this project because it would be able to find the cold object remotely. If a suitable IR sensor could not be found, the last resort would have been to search around for objects and test them with a temperature probe.

The Melexis MLX90514ESF-AAA temperature sensor, shown in Figure 9, was selected to allow the robot to precisely read the temperature of remote objects. This sensor was selected for its small size, ease of integration through I2C, and its accuracy of 0.5° C. The sensor uses infrared radiated energy from objects to determine the temperature. It has a 90° field of view over which it averages all temperatures and before returning a value. To limit the field of view a cone was placed around the sensor to decrease its viewing angle and narrow the field of view enough to measure pin-point temperatures at a distance. Unfortunately, the temperature sensor was not installed on the robot due to complications discussed in the R & D section.

Figure 9 – Temperature Sensor

Ultrasonic Range Finder

Ultrasonic range finders are traditionally used for obtaining distance measurements and mapping. Since this sensor is being mounted on the servo head, it will do the majority of the distance measurements with only slight back up from the IR sensors. Figure 10 shows the LV-Maxsonar-EZ1. This range finder is a simple cheap ultrasonic range finder capable of finding objects within a range of 0 to 6.25 meters. The sensor can run off of either 3v or 5v power source and is highly power efficient, using only about 2mA. The sensor outputs a variety of different signals, including an analog voltage output. The sensor is also capable of performing a calibration on power up which helps remove false positives from stationary nearby objects like the ground. To make use of this calibration feature at many points throughout the course, the power to this sensor was hooked up through the relay on the MC9S12, allowing power to be cycled at important points throughout the course. This single sensor was one of the most important components for navigation and object detection. Through servo head mounting the sensor was able to be multi-purposed to scan large areas at fixed distances, to find the objects in the waypoints, and for object following.

Figure 10 – Ultrasonic Rangefinder

Ultrasonic Receiver

The ultrasonic receiver needed for this project was one that could measure signal intensity. The sensor was used solely for locating waypoint number 3, the ultrasonic beacon. This sensor needed to have a specific band-pass frequency characteristic, and output the intensity as a DC value.

In order to design and build a functioning ultrasonic receiver, it was important to understand the fundamental characteristics of such a device. The basic physical phenomenon behind ultrasonic transducers is "piezoelectricity." What happens is that certain crystals or ceramics generate a voltage when pressure is applied to them and they become slightly. Similarly, if you apply an electric voltage to the piezoelectric crystal, it will deform, increasing in size. The ultrasonic transducers used in this sensor and in the ultrasonic beacon work in this way. In the first step, transmission, a high frequency electric voltage is applied to a piezoelectric crystal causing it to deform rapidly and send out a pressure wave. These pressure waves come into contact with the transducer in the sensor and produce a very small electric voltage. In this way, ultrasonic transducers make it possible to send and receive transmission signals.

To detect the ultrasonic waypoint effectively, an ultrasonic receiver was built which used a two stage cascaded band-pass filter. This ultrasonic receiver was built with a pass-band centered on a 40 kHz frequency. Each cascading filter had a gain of 25, to boost the signal up to measurable levels. Due to battery constraints, the op-amps for the amplifiers were run one sided with rails at 0 v and 9 v. This ultimately resulted in an amplified waveform centered around 4.5 volts. After amplification this bias voltage was reduced to approximately one diode voltage drop using a capacitor. This signal was then rectified and fed across another capacitor with a time constant equal to 20 periods, allowing its voltage to approximately follow the magnitude of the amplified signal. The voltage on this capacitor was fed directly into the microcontroller's analog to digital converter for digitization. The final design of this circuit is shown in Figure 11.

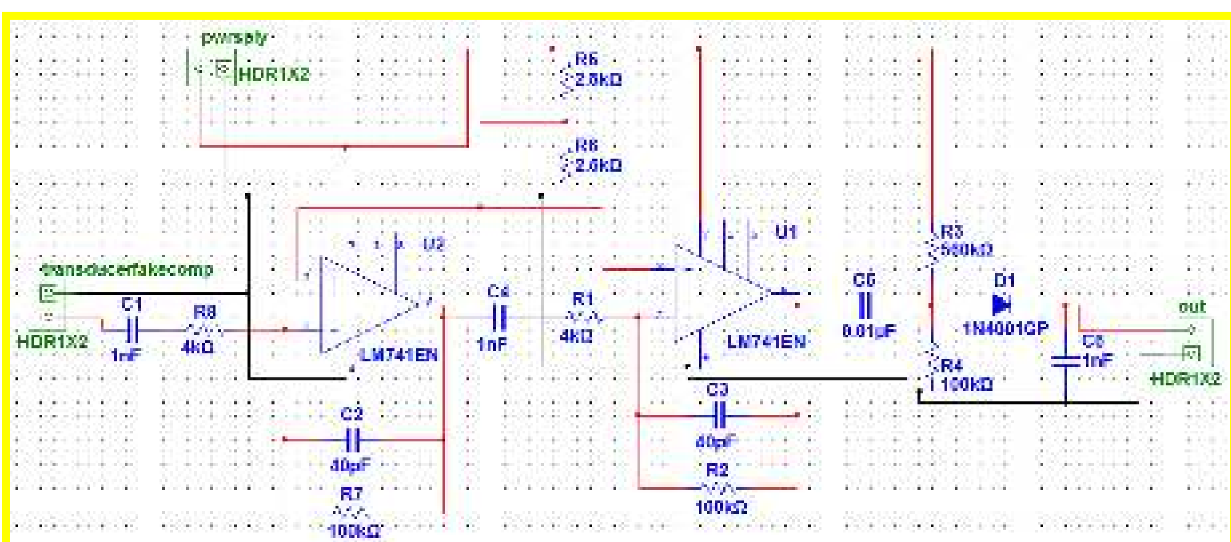

Figure 11 – Ultrasonic Receiver Circuit Schematic

With the use of National Instruments Ultiboard platform, it was possible to design a printed circuit board (PCB) and quickly manufacture it. The final design is shown in Figure 12. The design is single sided, which made it substantially easier to etch using the facilities at New Mexico Tech. Upon testing, this circuit was able to detect the ultrasonic beacon at a distance of up to approximately 10 meters, making it the most effective sensor on the robot.

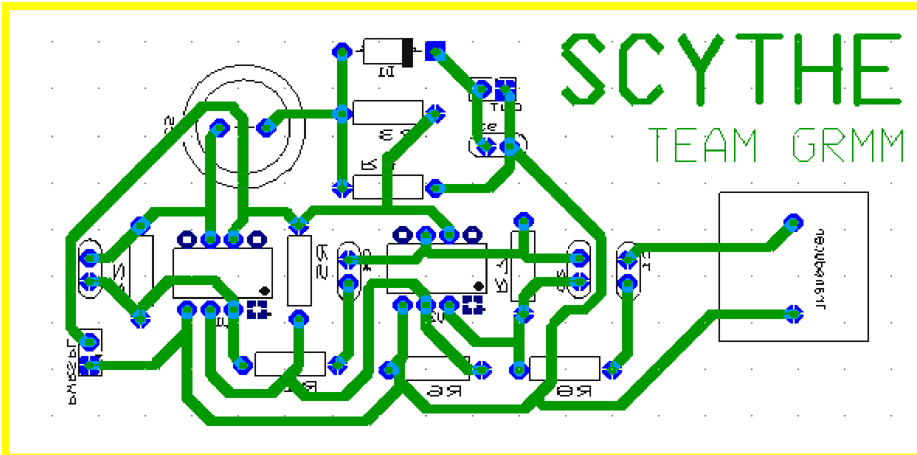

Figure 12 – Ultrasonic Receiver Board Layout

Compass

The addition of a compass greatly improved the level of control the robot had over its motors. Specifically, compass data was useful in turning, driving straight, and also in knowing the robot's current location.

Without a compass, turning would be based only on the speed of the motors and the amount of time they were left on, which could vary greatly as battery voltage decreases. By having a compass, the robot was able to turn exactly as far as was needed before stopping, regardless of the power remaining in the battery.

When driving straight, the compass helped to ensure that the robot was actually going in the proper direction on a course of a straight line, rather than in an arc. Even if all four motors were given the same voltage and current, it was not a reasonable expectation to have the robot to move in a perfectly straight line; wheel alignment or uneven terrain would causes discrepancies. With the information provided by the compass, the robot was able to slightly adjust the power given to each motor and correct its course without needing to stop.

When coupled with wheel encoders to measure the distance traveled, the compass helped to accurately judge the robot's location in 2D space. This was incredibly important in navigating the courtyard challenge, as the approximate location of every waypoint boundary circle was known ahead of time. Without the use of a compass, it would have been very difficult to find the location based area in waypoint 4.

Figure 13 - Compass

The compass used on the robot was a HMC6352 compass on a breakout board with four pins for power, ground and the I2C data and clock. Figure 13 shows this simple interface. The compass module had a 0.1 degree resolution and one degree repeatability. The micro-controller communicated with the compass over I2C and when asked for a heading, the compass returned a digital value from 0-3599 which indicated a heading of 0-359.9 degrees; with north equaling a value of zero degrees.

The compass was used for many operations, including travel from destination to destination along the course. These destinations generally were known locations like the circles for each waypoints or objects used to confirm the robots location. In these cases the headings were predefined and loaded into the program. Once arriving at the specific spot and detecting an object the robot would then determine the heading it needed to reach the object and turn towards it.

Immediate Proximity Violation Detectors (IPDV)

As a last line of defense, Two Lynxmotion Bumper Switches were located at the front of the robot to detect any possibly harmful objects that could make it past the ultrasonic and IR range finders. Each IPDV composes of a micro switch and a whisker/antenna made of plastic tubing.

The whisker shown in Figure 14, allows the bumper to detect objects within eighteen centimeters, effectively covering half of the width of the robot . With two IPDVs positioned at the center of the robot, covering both wheels, any imminent object can be detected. A spring connects the two bumpers together pulling them away from the switches. This allowed the robot to drive and turn in rough terrain without the unstable motions triggering the switches on and off at random, all the while still allowing the switches to be activated when objects comes in contact with them.

Figure 14 – Immediate Proximity Violation Detection Sensor

Encoders

One of the easiest ways to measure distance a robot has traveled is to attach accurate encoders to all four wheels. Since the course is relatively flat and the robot will be traveling slowly, wheel slippage was not be a significant issue. By putting encoders on every wheel, the distance traveled could be averaged to reduce error generated by any single wheel slipping.

Each wheel encoder is composed of a single QRD1114 Reflective Object Sensor and a black and white stripped disk, shown in Figure 15. The QRD1114 contains an infrared emitting diode and a NPN silicon phototransistor. The phototransistor measures the amount of energy radiated from the diode that reflects off a surface, thus determining contrast effectively.

Figure 15 – Wheel Encoder

A simple encoder wheel program generated the encoder disks. The program partitioned off the number of black stripes desired and placed them along the radius specified by the user. In an attempt to improve the accuracy of the QRD1114 sensors, the program was modified to allow users to change the black to white ratio on the wheel while maintaining the same number of stripes. This ability was needed because the QRD1114 sensors were reading oddly when viewing a black stripe. The problem was that the black stripes were receiving reflected energy from the white stripes, which when read, outputted a higher voltage than they should have. With a larger black area the encoders were able to measure the energy from the black more cleanly. These stripped wheels were then printed out on card stock and glued to a cardboard disk. The program used to generate the discs is provided on the accompanying DVD with the path:

Encoder Generator\Encoder Generator\bin\Release\Encoder Generator.exe

The QRD1114 sensors were mounted on the motors so that as the suspension moved the encoder wheel and the device stayed in line with it. The power and ground wires were connected to the buses of the proto-board on the microcontroller. Because the NPN transistors do not output a digital signal, an op-amp was used for each encoder to convert to a digital signal by simply railing high or low. The output of each transistor was fed into a negative input of the op-amp that was also located on the proto-board. The digital signal from each op-amp was then connected to the microcontroller to count the number of rising edges. Figure 16 shows the circuit used for each encoder.

By counting the rising edges of the op-amp, the robot was able to count how many times a stripe passed and thus how far it traveled.

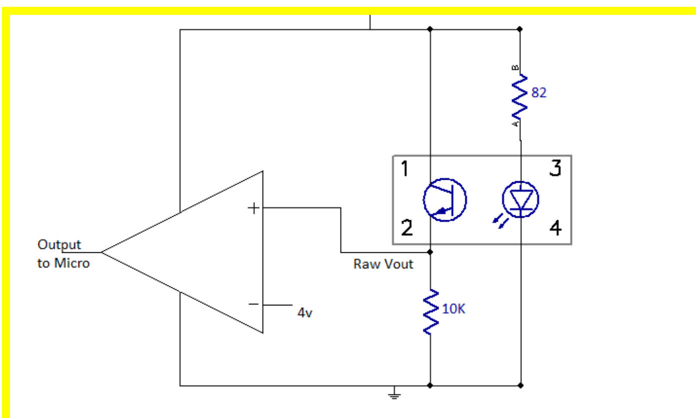

Figure 16 – Encoder Circuit

Power

Sources

To incorporate the power requirements of the array of devices built into the design, a three-tier power board system was devised. Each tier was interconnected to accommodate the required voltages of each device at that level. The voltage distribution across the robot consists of 9, 5, and 3-volt variations, utilizing the provided EP Buddy 6.6v battery and an additional 9v battery. To supply these voltages, linear regulators were incorporated into the power system to create the 5 and 3 volt sources.

From initial testing, it was found that the EP Buddy 6.6v battery was unable to power the microcontroller and motors at the same time due to the motors fluctuating current draw. This fluctuation caused the microcontroller to reset continuously. To solve this problem a 9-volt battery was added to power the microcontroller separately. Utilizing independent power systems, the Dragon-12 microcontroller was measured to draw 124mA and the motors/sensors to draw roughly around 1.8-3A. The total measured current draw from sensors was minimal compared to the motors. With the use of a 9v 200mAh battery and the EP Buddy 6.6v 2300mAh battery, the robot had 1.6 hours of computation time on the microcontroller and about 46 minutes of drive time with the motors/sensors. The supplied power was more than enough for the robot to be able to complete the course due to it only taking on average fifteen minutes to complete the course one time.

Tiered Distribution System

The power distribution system was split into three interconnected tiers to minimize the number of wires on the robot. Figure 17 shows the relative location of each tier; one tier each at the bottom, middle, and top.

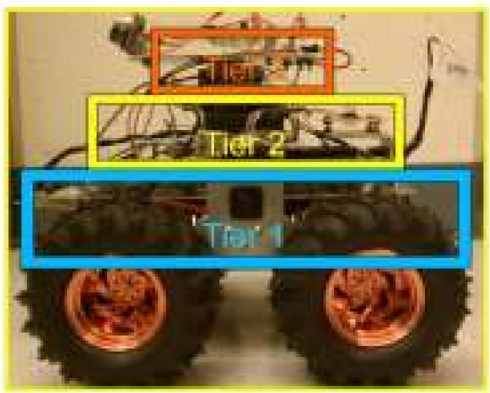

Figure 17 – Power Tiers

The first tier, or bottom level, of the robot's power system provided the initial connection from the two power sources, the 9v and EP Buddy 6.6v batteries, to the microcontroller and H-bridge motor controllers. This tier allowed for the high current draw of the motors to be isolated and provided a standalone source to the microcontroller. At this level a single throw double pole switch was also incorporated to allow the power of both sources to be turned on and off simultaneously. This switch also provided a safety measure for when the batteries are connected or disconnected and a fast and easy way to turn off the robot at anytime. However, from this hard power reset, the robot was tending to jump forward, applying a short burst of power to the motors. To eliminate this erratic behavior a simple AND gate was used along with pull-down resistors to wait for an enable line from the microcontroller to go high before passing any signal to the H-bridge motor controllers.

The second tier, or middle level, provided a central location to power the various hard mounted sensors on the chassis. These sensors included wheel encoders, infrared range sensors, bumper switches, the servo, and the compass. To accommodate the 5 and 3-volt requirements of these sensors, voltage regulators were incorporated off the EP Buddy battery in this tier. The board was etched and designed to adapt male/female style connectors, providing more reliable connections. These connectors also provided the option to interface new devices easily into the power system.

The third tier on the top pan supplied power to the temperature sensor, ultrasonic range finder, and camera. The built in power distribution board on the pan head allowed power to reach each device without the complication of numerous wires being cut, pulled, or shorted from the pan head rotating. Thus, a single power bus from the second layer fed this board. Although the 3-volt power supply was only for the camera which was not be implemented in the final design, it remained on the board in the event that future sensors required a 3-volt supply. Power for the ultrasonic receiver came from the 9-volt source in order to increase the rails and potential gain of the op-amps.

Software Approach

There were three major layers in the software required to complete this challenge. The fundamental first software layer was made up of the interface between the microcontroller and each individual sensor, like being able to read from the compass. On top of that framework was the intermediary layer, which interfaced several sensors and made them into useful subroutines that could perform basic tasks, like using the servo head to scan an area for an object. The highest software level called upon the middle layer subroutines in a sensible order to complete the course.

Sensor Integration

Compass

The compass was the most complicated device to interface with that was used in this project. The compass interfaces through a standard I2C bus, which was used both to control its operation and to read compass headings. To actually get the device to return data in a timely manner, the compass had to be put into a “continuous” mode by writing to its on board registers. After this, reading from the compass took no more than a few milliseconds. To obtain repeatable compass headings, the device had to be calibrated by sending it a command to start calibration, rotating the robot evenly for 20 seconds, and then sending it a command to stop calibration. A simple function was implemented to perform each of these three tasks in software.

Temperature Sensor

The temperature sensor, although unused in the final product, was fully implemented in software. This device was similar to the compass in that it also used a standard I2C bus, but was simpler to use in that it required no initial setup or calibration. A simple function was implemented to read the temperature from the device on the fly.

Bumpers

The bumpers were electrically simple devices which just switched from low to high when an obstacle was encountered. To read this into software, an input capture routine was used for each bumper to latch the rising edges into memory and register these “hits” into a global variable. After the collision was handled by the main program, the bumper global variables could be reset.

Encoders

The encoder subsystem was similar to the bumpers in that a simple high / low signal was passed into the microcontroller. Each encoder was assigned its own input capture routine, which just incremented a unique global variable on each rising edge. Each encoder had its own global variable, front right, front left, etc. When software actually read the distance traveled, a function was used to average all four values, discard the outlier, and average the remaining three counts. This setup ensured robustness against loss or damage to any one encoder on the robot.

IR Sensors

These sensors were simple devices that output a varying analog voltage based on how far away an object was. The biggest challenge with these devices was that the output was far from linear, making use of a simple regression equation impossible. Instead, many data points were taken for each individual sensor and were used to generate a piecewise linear model to closely approximate segments on the output curve. A function was then created to interpolate between raw analog to digital converted (ADC) values to highly linear digital output to the rest of the software. The raw data and subsequent linear model used to approximate the short-range IR sensors is shown in Figure 18.

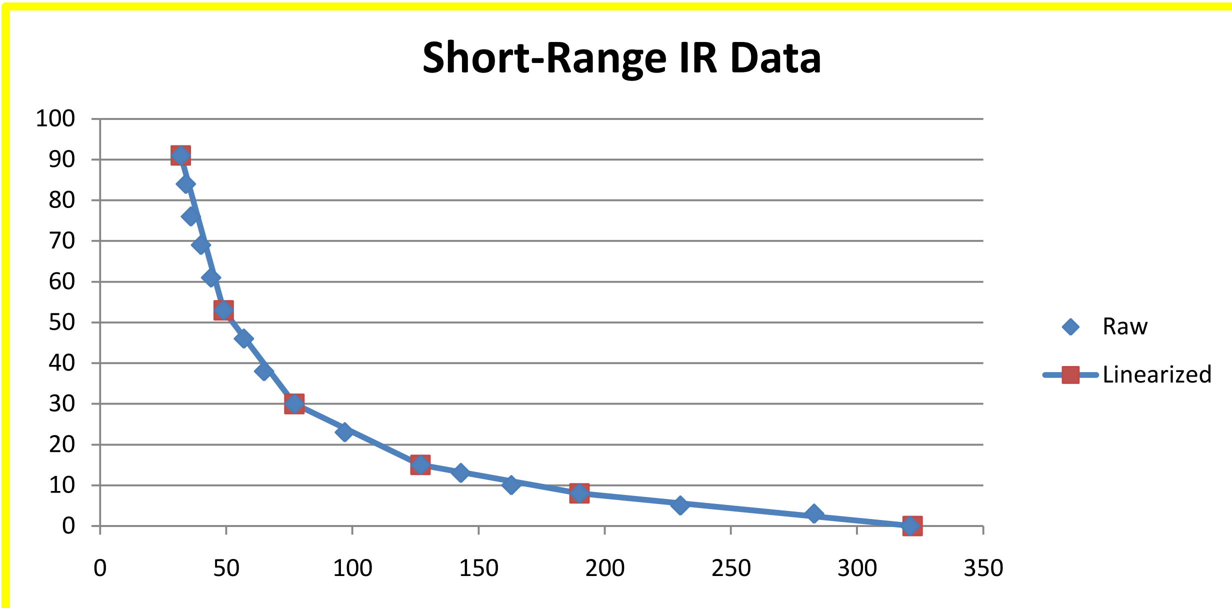

Figure 18 – Infrared Linearized Data

Ultrasonic Rangefinder

The ultrasonic rangefinder, like its infrared companions, output an analog voltage that varies with distance. Unlike the IR devices, however, this device was highly linear, and required very little work to convert values to centimeters.

Ultrasonic Receiver

This device was designed to output a DC analog voltage to the microcontroller, and it did so quite well. It was not necessary to try to convert analog values to any other units; relative maximum analog values were sufficient to find the direction of the ultrasonic beacon.

Speaker

Using the on-board speaker to generate sound was done through the use of a simple output compare subroutine which just toggled the speaker on and off at a set frequency. To change the tone of the device, the frequency was adjusted accordingly.

Servo

The servo is essentially controlled by a pulse width signal; setting the amount of time between falling and rising edges determines what angle the servo rotates to. Because the microcontroller only has an 8 bit resolution on its PWM subsystem, and output compare subroutine was used to generate a high precision signal allowing for sub degree accuracy (256 steps in 180 degrees).

Navigation

To navigate the course successfully, the robot had to know where it was at any point in time. To do this, a dead reckoning scheme was employed using the compass to obtain direction and the wheel encoders to measure distance traveled. Because neither the encoders or the compass were perfectly accurate, objects of known position were used to constantly re-zero the robot's internal position. Examples of the objects used include walls, railroad ties, light posts, and trees.

Localization & Obstacle Avoidance

To make use of the dead reckoning system, the robot made use of an internal x, y coordinate plane where the y axis points north and the x points east. Every movement the robot made automatically updated an internal position so that it constantly knew where it was at any time. Knowing this, the robot always knew how it needed to move to get to the destination coordinates. The calculations for this were relatively straightforward; by knowing the distance traveled and the angle with respect to the y axis, basic trigonometry could be used to solve for the change in x and y. With this information, the robot was able to find a path to any given location, regardless of objects encountered en-route.

Detailed Course Strategy

Each waypoint in the challenge features a unique challenge; for this reason, the robot's actions at each waypoint must also be unique. The next few sections cover exactly what the robot was programmed to do to complete the challenge.

Waypoint 1

1. Calibrate the compass
2. Adjust distance from the red barrier to 190 cm
3. Follow the barrier a distance of 11.3 m keeping the wall 190 cm away
4. Search for the object:
    a. Scan with the ultrasonic rangefinder, discarding any object more than 1.5 m away
    b. Move forward 1.3 m and repeat until the waypoint is found or the edge of the circle is reached
5. Path-find to a reasonable point on the other side of the circle
6. Resume following the wall until it ends

Figure 19 – Navigation Leg 1

Waypoint 2

1. Continue along the path until reaching the railroad ties
2. Back up 3.25 m
3. Repeat calibrating the compass on the nice level surface
4. Aim the ultrasonic rangefinder towards the start of the course
5. Drive towards waypoint 2 until the rangefinder picks up the light post
6. Adjust distance to light post to be 1.4 m.
7. Drive 3.7 m to the edge of the circle for waypoint 2
8. Aim the ultrasonic rangefinder towards the railroad ties
9. Drive straight through the circle until either the rangefinder picks up something less than 3.5 m away, the side IR sensors find something less than 90 cm away, or the bumpers encounter an object.
10. Path-find to the edge of waypoint 2
11. Aim the ultrasonic receiver back to the right and drive until it picks up the tree
12. Adjust distance to the tree to be 2.5 m

Figure 20 – Navigation Leg 2 Part 1

Figure 21 – Navigation Leg 2 Part 2

Waypoint 3

1. Drive down the diagonal path 20 meters
2. Aim ultrasonic rangefinder to the left and drive until the wildcard box is detected
3. Drive off a set distance into the grass and aim approximately towards waypoint 3
4. Search for the object:
    a. Scan with ultrasonic receiver and move 1.5 meters towards highest value
    b. Repeat until bumpers contact the beacon

Figure 22 – Navigation Leg 3 Part 1

Figure 23 – Navigation Leg 3 Part 2

Waypoint 4

1. Zero the robot's position
2. Path-find to predefined coordinates describing the relative location

Figure 24 – Navigation Leg 4

Waypoint 5

1. Travel in the general direction of waypoint 5 a distance of 12.5 m
2. Aim the ultrasonic rangefinder forward and drive towards the railroad ties
3. When the railroad ties are approximately 2.5 meters away, follow them until they end
4. Perform a similar search as waypoint 1, limiting the range to 4.5 meters

Figure 25 – Navigation Leg 5

Documentation

All of the code functions and subroutines used in this project are commented to be compliant with the Doxygen automatic documentation system. These comments made the code easier to debug while working on the project and make understanding the code very easy. To view the code documentation, open the file "Code Documentation\html\index.html" on the provided DVD.

Budget

As seen in Appendix A the total cost for the research and development of the robot was $784.70. This includes the cost for the camera and temperature sensor that were not ultimately used as well as the compass that was shipped over night after the first one stopped working. Removing these figures from the total, to determine the final operating price for the robot resulted in a cost of only $701.07 as shown in Appendix B. This is well under budget and allows for additional sensors to be added if desired in the future.

Overall Results

Development Challenges

Temperature Sensor

Although, the indoor testing of the sensor's functionality showed promise, moving the sensor outside caused it to behave poorly. Indoor testing showed that the sensor was able to detect a bucket of ice from about two meters away. This was because the room's temperature was constant and the cold bucket would drop the viewing temperature by at least 5° C. When outside, the cold bucket would only affect the observed average temperature by a maximum of 2° C (from one meter away or closer.) Only with the bucket immediately in front of the sensor would it drop the temperature by 5° C. A more significant problem was, that the temperature sensor would read that shaded areas of ground were colder than the cold bucket. The measured temperature of the shade was 10-15° C colder than that of the sunny areas and therefore the cold bucket could not affect the average enough to compete with the shade. Due to the terrible outdoor performance this sensor was removed from the final product. The cold bucket would instead be found using the working ultrasonic and IR range finders.

Range Finding

The original robot design used only IR sensors for detecting distances to objects. Two mid-range sensors were placed in the front facing slightly inward and one long-range sensor mounted on the pan head. During tests, though, it was found that the characteristics of these sensors were not as consistent as originally planned. The ranges of the IR sensors fell short of their described distance. Therefore, it was decided to incorporate an ultrasonic range finder into the design so that the robot could have a greater range and pinpoint accuracy for detecting objects. The ultrasonic range finder was able to provide a greater viewing distance than the IR sensors and had a linear output voltage to distance relationship, features to which the IR sensors did not have either of. The ultrasonic range finder replaced the long-range IR sensor on the pan head; which was moved to the front of the robot, below the IPVD sensors. The two mid-range IR sensors were then moved to the sides of the robot for side object detection.

Wheels

The wheels, motors, and the adapter between them was poorly designed and therefore was the source problem for several aspects of the robot, including accurate dead reckoning. In addition, the sticky grip of the wheels caused the motors to lock up and render the robot un movable in grassy area. This posed a relatively contradictive problem, being as how the chassis's and motors were built for off-road use. . To overcome the tire's excessive grip the PWM of the motors was set to almost max any time the robot moved in grass. When the motors did work properly, the adapter between wheel and motor became the focused problem. The wheel to shaft adapters were not very sturdy as they allowed the wheels to wobble as they drove. This made it very hard for the robot to drive straight, because the camber of the wheel was continually changing ultimately causing the wheels to be misaligned Near the end of the development cycle, the motors started leaking oil onto the shafts, which created the potential for the wheels to detach while driving. Although none of these problems could have foreseen, the team consensus was that chassis/motor platform was one of the weakest aspects of the robot.

Encoders

Unfortunately, the encoders relied on the ability to properly differentiate between black and white surfaces. This became a large problem because the QRD1114 sensors had to be within one centimeter of the disk, and as the wheel camber changed so did the distance between the sensor and stripped wheel. To correct for the wheel's camber movement, the QRD1114 was positioned such that it was almost touching the encoding wheel.

The presence of ambient light also affected how accurate the QRD1114 sensors were. The sunlight reflecting off the surface of the encoder wheels caused the transistor to output a high voltage for both black and white voltages. In a darkroom the QRD1114 had an output voltage of 4.88v for a white surface and an output of 0.5v for a black surface. With ambient light present the voltage for a white surface remained at around five volts but the voltage for a black surface ranged from three to five volts depending on its position. To help block out ambient light, a tube of black poster board was attached around each QRD1114. This tube is the same cover/shield mentioned above. Unfortunately, even with these conditions the QRD1114 only produced an output voltage of around 2.5v for a black surface.

To further improve the QRD1114's reading ability, an operation amplifier was used to convert the analog 2.5-5v output voltages for the transistor to a digital one or zero output. The output voltage was fed into the negative input of the op-amp and a four-volt reference was placed on the positive terminal. A five-volt potential was connected to the positive supply of the op-amp and ground connected to the negative supply. Using this set up, the output of the encoder system would be either a zero for white or a one for black.

Compass

The first compass used for the robot stopped working four weeks before the final demonstration. There were a few possible reasons for the compass to malfunction, but the official cause is still not known. The first possible reason for failure could be that the compass burned out due to a mix of 3v supply voltage with a 5v I2C line. The next possible reason could be that the compass altered its slave address. Further testing will determine the actual cause of the compass's malfunction. To prevent the second compass from malfunctioning the new compass was powered with 5v to compliment the 5v I2C line.

Another common problem encountered with the compass was that, while test running the robot's driving capabilities, the compass would occasionally lose its calibration leaving the robot completely blind in respect to which direction it was heading. To prevent this from happening, the robot was programmed to calibrate the compass upon being turned on and between certain waypoints.

Flash Memory

One of the major problems with such ambitiously complicated code was that the available EEPROM memory available on the Dragon12 microcontroller was limited to 3 kB. With a final code size of approximately 8.5 kB, not even the 8 kB of RAM was large enough to hold the program. To overcome this problem, the Dragon12 bootloader had to be used to load the program into flash memory. Flash memory on the MC9S12 can be as large as 128 kB using a paged memory model. Unfortunately, the bootloader only accepts s29 records, which the developing environment, CodeWarrior, was unable to generate natively. To perform this conversion, the free program SRecCvt.exe, provided by FreeScale, was used. In the end, using flash memory was far faster than EEPROM and performed perfectly for this project.

Status of Subsystems

Upon completion of the Junior Design Project, an array of subsystems were integrated into the robot and all designed hardware was completed. This allowed all the components to be mounted, powered, and working. A significant amount of time was put into creating a solid platform to allow for a simple transition into sensor integration and testing. The primary object of sensor integration was to enhance the efficiency and improve the robots functionality. In the process of integrating each sensor subsystem, alternate power and mounting locations for additional subsystems were added if needed. The ultrasonic receiver, range finder, compass, and infrared sensors were fully integrated and programmed into the software design. The camera required data processing and software integration, however, from an approach of dead reckoning and obstacle detection there was no need for the camera anymore, thus it was removed from the project. The completed software architecture provides a simplistic yet robust software approach, clear to any programmer. This simple programming logic allows the functionality of the robot to infer environmental information from the sensor data and incorporate it to navigate the courtyard. With a solid hardware platform and a simplistic software approach, the goal to achieve a fully autonomous robot only requires some fine tuning and testing.

Final Functionality

An autonomous system was developed to navigate the courtyard and achieve obstacle waypoints. The system consists of an array of sensors, a robust chassis, power supply, and microcontroller with pre-defined programming. At each waypoint significant progress was made, however, robot met varying degrees of success depending on where it was. At waypoint 1, the robot would initialize its location, travel to the waypoint, and locate the unknown object every run. Moving on to waypoint 2, the robot occasionally had issues locating its position off of the known light pole using the ultrasonic range finder, thus causing error in localization. This error was often caused by varying data measurements as the range finder interpreted range differently as the device was heated in the sun. However, even with this error in localization, the robot could navigate to waypoint 2 every run, but did not locate the object reliably. Waypoint 3 proved to be the easiest due to the large range of the ultrasonic receiver. Therefore, the robot located the object every time. Getting to this waypoint, however proved to be a challenge due to the error accumulated from waypoint 2. To solve any localization error accumulated, a wild card cardboard box was placed along the sidewalk. This provided another known location for the robot, allowing it to travel to waypoint 3 more effectively. With an approach of only dead reckoning in the grass, the robot was able navigate to waypoint 4 half the time. Compass calibrations fluctuated as wheel slip and inclined planes on the grass caused the robot to not drive straight. From waypoint 4, the robot continued on to waypoint 5, searching for the railroad ties as a guide. Error accumulated from driving on the grass, ultimately positioned the robot too far away to detect the railroad ties and hindered the success of reaching the final waypoint. However, during certain trials, error was minimal and the robot navigated the last two waypoints and found the object easily. The robot does not fully incorporate an autonomous navigation system, but it does accomplish the many challenges of navigation and object detection. The ability to navigate the courtyard and detect objects, with varying success, was an achievement in its own right.

Conclusion

Vehicles that can autonomously drive themselves are becoming more real every day. Components like a solid chassis, an array of environmental detection sensors, and an intelligent logic system facilitate the building of these machines. In this particular challenge, the designed robot needed to find and identify a diverse set of semi-randomly placed objects on a large course with a very limited budget. Many of the possible options available to perform this task were not possible due to cost, time, or processing requirements. After a few design iterations on the sensor subsystem array, it was decided that a simple ultrasonic rangefinder mounted on a rotating servo head platform could successfully detect nearly all of the objects and be cleverly used for localization purposes as well. One of the largest problems encountered was that of the terrible wheel connections, which continued to be a problem throughout the design process. Unfortunately, this was one of the few subsystems that was not within the scope of the design process and could not be entirely dealt with.

The final design of the robot used an ultrasonic rangefinder and an ultrasonic receiver mounted on a rotating pan head to detect the objects at each waypoint. A compass and four wheel encoders were used for navigation and localization, and immediate proximity violation detectors and infrared rangefinders were used to detect imminent objects. After all of these modifications were made to correct the problems encountered during the design process, the final robot was delivered under budget and on time. The robot successfully reached all five of the waypoints most of the time and was able to detect the majority of the target objects at each waypoint. With a little more time and a better chassis, this design could be used to successfully identify every object at every waypoint reliably.

Appendices

Appendix A – Research and Development Costs

Store	Item	Description	Qty	Unit Price	Total Price
Digikey	MT9V131C12STC	Camera	1	$21.20	$21.20
	LP2950ACZ-3.0/NOPB	3V Regulator	1	$1.12	$1.12
	MIC29150-5.0WT	5V Regulator	1	$3.13	$3.13
	QRD1114	IR reflective	4	$1.48	$5.92
Pololu	GP2Y0A21YK0F	10-80cm IR	2	$9.95	$19.90
	GP2Y0A02YK0F	20-150cm IR	1	$13.99	$13.99
		IR connector	3	$0.99	$2.97
	MLX90614ESF-AAA	Temp	1	$19.95	$19.95
Robot Shop	Parallax HMC6352	Compass	1	$29.99	$29.99
	Lynxmotion BMP-01	Bumpers	1	$10.00	$10.00
	LV-MaxSonar-EZ1	Ultra-Sonic	1	$24.95	$24.95
SparkFun	HMC6352	Compass	1	$34.95	$34.95
Andy		H. Shrink S	8	$0.25	$2.00
		H. Shrink L	1	$0.50	$0.50
		Transducer	1	$5.00	$5.00
		90 P. Header	2	$0.25	$0.50
		9v connector	3	$0.50	$1.50
		Q. OpAmp	1	$0.75	$0.75
		231 Cable	1	$2.50	$2.50
		Connectors			$24.15
		S. P. Header	2	$1.50	$3.00
Norton		Spacers	6	$0.20	$1.20
		Aluminum	1	$2.00	$2.00
Jorgensen	202-0009-02	Camera-Brd.	1	$9.99	$9.99
Provided	Dagu Wild Thumper 4WD	Chassis	1	$250.00	$250.00
	High-Power Motor Driver 18v15	H-Bridges	2	$39.95	$79.90
	2S1P LiFe Battery	6.6V-Battery	1	$28.85	$28.85
Richard		Servo	1	$15.00	$15.00
James	Dragon12-Plus Trianer	HCS12 Micro	1	$130.00	$130.00
Byron	AccuPower Ap200-1	9v	1	$10.95	$10.95
Shipping:					$28.84
			Total:		$784.70
			Expected Budget:		$750.00
			Remaining Budget:		**-$34.70**

Appendix B – Final Product Costs

Item	Qty	Unit Price	Total Price
QRD 1114	4	$1.48	$5.92
10-80cm IR	2	$9.95	$19.90
20-150cm IR	1	$13.99	$13.99
Bumpers	1	$10.00	$10.00
Ultra-Sonic	1	$24.95	$24.95
Compass	1	$34.95	$34.95
Hardware	1	$3.20	$3.20
Chassis	1	$250.00	$250.00
H-Bridges	2	$39.95	$79.90
6.6V Battery	1	$28.85	$28.85
Microcontroller	1	$130.00	$130.00
9v	1	$10.95	$10.95
Servo	1	$15.00	$15.00
Misc. Components			$44.62
Shipping:			$28.84
		Total:	**$701.07**

Table of Contents

/*Paper start*/

Abstract:

here is out shit. we succeeded. fuck yeah

Introduction:

here we talk about how awesome we are and how our robot is the shit.

Body:

*** thorough description of all sensor aspects**

- -Parts andDatasheet compilations
- -Relevent data curves
  - Chasis/motors
    - Dagu Wild Thumper
  - Brains:
    - MC9S12 Microcontroller
  - Movenment Control:
    - Motor controllers
      - H-Bridges
    - Wheel Encoders:
      - Qrd1113
      - Black and white wheel
    - Compass:
      - Honeywell HMC6352
  - Obstacle Detection Sensors:
    - Bumpers:
    - IR:
      - Sharp Gp2Y0A02YK
  - Challenge Sensors:
    - Temp

MLX90614
Camera
MT9V131C12STC
Ultrasonic Range Finder
LV-Maxsonar-EZ1
Ultrasonic Beacon
Self Built:
Schematic
PCB
BOM/Parts used
Power Distribution:
Two Different Boards:
Servo Power:
5 volt systems
3 volt systems
Other Power:
bumpers and shit
Motor Power:
ep buddy battery
Microcontroller power:
recharageable 9v battery

*** full circuit diagrams**

-Power Distribution Boards
-Flow diagram of connections to each thing
Hint: it all goes to the microcontroller lol except the motor battery

-Problems

R&D shit
ambient light
noise...

-Code:

shit tonnes of it

Cost Budget:

R&D Parts:
all parts +
mid range ir sensors
camera& break out board
Camera-MT9V131C12STC
fried compass.
Honeywell HMC6352
Actual Mounted parts:
Dagu Wild Thumper
2x H-Bridges Motor controllers
Microcontroller

Wheel Encoder sensors Qrd1113
Compass-Honeywell HMC6352
Bumpers
Front IR-Sharp Gp2Y0A02YK
Temp- MLX90614
Ultrasonic Range Finder-LV-Maxsonar-EZ1
Ultrasonic Beacon-Self Built
ep buddy battery
recharageable 9v battery
miscelanous:
paper
cone foil
heat shrink
wires
connectors

Power Budget:

what kind of run time we have on a single chage

Conclusion:

whoo our shit works. haha suck it blue!

Appendix:

* **list of figures**
    photos
    data curves
        IR
        US
        Encoders
        Comapss
        Temp
* **references**
    paper references
    photo references

* **floppy or CD with all source code and an electronic version of the report (preferably as MSWord, Wordperfect, or PDF).**

```
/**
 * @file compass.h
 * @brief
 *  HMC6352 compass code
 */

#define ADDRESS_COMPASS 0x42
#define COMPASS_BEARING_COMMAND 0x41

#define COMPASS_RAM_WRITE           'G'
#define COMPASS_RAM_OP_MODE_CTL     0x74
#define COMPASS_START_CALIBRATION   'C'
#define COMPASS_STOP_CALIBRATION    'E'

void compass_init(void);
void compass_calibrate(void);
unsigned int compass_get_bearing(void);

/**
 * Sets up the HMC6352 compass into continuous mode
 *
 * @todo
 *  Change the averaging amount
 */
void compass_init(void) {
    // Connect, write mode
    if (iic_start(ADDRESS_COMPASS) == 0) {
        // Send RAM write command
        if (iic_transmit(COMPASS_RAM_WRITE) == 0) {
            // Write to Operational Mode register
            if (iic_transmit(COMPASS_RAM_OP_MODE_CTL) == 0) {
                // Bit 6:5 - 11 - 20 Hz measurement rate
                // Bit 4   - 1  - Periodic Set / Reset
                // Bit 1:0 - 10 - Continuous Mode
                if (iic_transmit(0x72) == 0) {
                    iic_stop();
                    IBCR = IBCR | BIT_4;   //Set Tx/Rx to transmit to stop SCLK
                }
            }
        }
    }
}

/**
 * This function spins the robot around for 20 seconds while the compass
 * calibrates
 */
void compass_calibrate(void) {
    volatile unsigned int counter = 5000;

    ENABLE_MOTORS();
    if (iic_start(ADDRESS_COMPASS) == 0) {
        if (iic_transmit(COMPASS_START_CALIBRATION) == 0) {
            iic_stop();
            IBCR = IBCR | BIT_4;   //Set Tx/Rx to transmit to stop SCLK
        }
    }

    while (counter --) {
        update_motors(100, 100, FORWARD, BACKWARD, 4, CONCRETE);
        delay_1ms(4);
    }

    if (iic_start(ADDRESS_COMPASS) == 0) {
        if (iic_transmit(COMPASS_STOP_CALIBRATION) == 0) {
            iic_stop();
            IBCR = IBCR | BIT_4;   //Set Tx/Rx to transmit to stop SCLK
            success_action_location();
        }
```



```
    }

    counter = 50;
    while (counter --) {
        update_motors(0, 0, FORWARD, BACKWARD, 4, CONCRETE);
        delay_1ms(4);
    }
    DISABLE_MOTORS();
}

/**
 * Reads the bearing from the HMC6352 compass
 *
 * @return
 *  Compass bearing in 10ths of a degree between 0 and 3599 decimal
 */
unsigned int compass_get_bearing(void) {
    unsigned int bearing;

    if (iic_start(ADDRESS_COMPASS | BIT_0) == 0) {  // Connect, read mode
        iic_swrcv();                                // Switch to recieve mode

        // Must send a NAK after recieving the last byte
        bearing = (iic_recieve_m1() << 8) & 0xFF00;         // LSB
        bearing = bearing | (iic_recieve_last() & 0x00FF);  // MSB

        return bearing;
    }

    return 0;
}
```

```c
/**
 * @file iic.h
 * @brief
 *  IIC bus code
 */

void iic_init(void);
unsigned char iic_start(char address);
unsigned char iic_response(void);
void iic_stop(void);
unsigned char iic_transmit(char data);
char iic_recieve(void);
char iic_recieve_m1(void);
char iic_recieve_last(void);
void iic_swrcv(void);

/**
 * Power on the IIC subsystem @ 50kHz
 */
 void iic_init(void) {
    IBFD = 0x27;                            // Set to 50 KHz operation
    IBAD = 0x02;                            // Set slave address to 1
    IBCR = IBCR | BIT_7;                    // Enable the IIC bus
 }

/**
 * Initiate connection to device
 *
 * @param address
 * Address of the device to connect to
 * @return
 * Returns 0 if an ACK is recieved, 1 if a NACK is recieved
 */
unsigned char iic_start(char address) {
    while ((IBSR & BIT_5) != 0);            // Wait for IBB flag to clear
    IBCR |= BIT_4 | BIT_5;                  // Set XMIT and MASTER mode, start
    IBDR = address;                         // Send device address and R/W bit
    while ((IBSR & BIT_5) != BIT_5);        // Wait for IBB flag to set
    return iic_response();                  // Wait for response
}

/**
 * Wait for a response and stop the IIC bus if a NACK is recieved
 *
 * @return
 * Returns 0 if an ACK is recieved, 1 if a NACK is recieved
 */
unsigned char iic_response(void) {
    while ((IBSR & BIT_1) != BIT_1);        // Wait for IBIF to set
    IBSR = BIT_1;                           // Clear IBIF interrupt
    if ((IBSR & BIT_0) == BIT_0) {
        iic_stop();                         // Stop if NACK is recieved
        return 1;
    } else {
        return 0;
    }
}

/**
 * Release bus and send a stop signal (if transmitting)
 */
void iic_stop(void) {
    IBCR = IBCR & ~BIT_5;
}

/**
 * Send a byte of data and wait for a response
 *
 * @param data
```



```c
 * Data byte to transmit
 * @return
 * Returns 0 if an ACK is recieved, 1 if a NACK is recieved
 */
unsigned char iic_transmit(char data) {
    IBDR = data;                            // Write data to device
    return iic_response();                  // Wait for a response
}

/**
 * Receive a byte of data; next recieve will send an ACK
 *
 * @return
 * Returns the data byte recieved from the bus
 */
char iic_recieve(void) {
    while ((IBSR & BIT_1) == 0) ;           //Wait for IBIF to be set
    IBSR = BIT_1;                           //Clear IBIF
    return IBDR;                            //Return data from IBDR
}

/**
 * Recieve a byte of data; next recieve will send a NACK. Useful for the
 * second to last byte of data if device expects a NACK to end communication.
 *
 * @return
 * Returns the data byte recieved from the bus
 */
char iic_recieve_m1(void) {
    while ((IBSR & BIT_1) == 0) ;       //Wait for IBIF to be set
    IBSR = BIT_1;                       //Clear IBIF
    IBCR = IBCR | BIT_3;                //Set TXAK to not acknowledge
    return IBDR;                        //Return data from IBDR
}

/**
 * Recieve the last byte of data and stop the IIC bus
 *
 * @return
 * Returns the data byte recieved from the bus
 */
char iic_recieve_last(void) {
    while ((IBSR & BIT_1) == 0) ;       //Wait for IBIF to be set
    IBSR = BIT_1;                       //Clear IBIF
    IBCR = IBCR & ~BIT_3;               //Clear TXAK to acknowledge
    IBCR = IBCR & ~BIT_5;               //Clear MS/SL to generate stop
    IBCR = IBCR | BIT_4;                //Set Tx/Rx to transmit to stop SCLK
    return IBDR;                        //Return last data from IBDR
}

/**
 * Switches the IIC bus from transmit to receive mode
 */
void iic_swrcv(void) {
    volatile char nothing;

    IBCR = IBCR & ~BIT_4;               //Clear Tx/Rx to recieve and start SCLK
    nothing = IBDR;                     //Dummy read to move to initiate next
}
```

```c
/**
 * @file init.h
 * @brief
 *  Initialization code
 */

#define BIT_0   0x01
#define BIT_1   0x02
#define BIT_2   0x04
#define BIT_3   0x08
#define BIT_4   0x10
#define BIT_5   0x20
#define BIT_6   0x40
#define BIT_7   0x80

#define DISABLE_INTERRUPTS()    __asm(sei)
#define ENABLE_INTERRUPTS()     __asm(cli)

#define TRUE 1
#define FALSE 0

void delay_50us(int n);
void delay_1ms(int n);
void bus_clock_init(void);
void sci_init(void);
void ports_init(void);

/**
 * This function loops for approximately 50 us * n. The true number of cycles
 * consumed is 1196*n + 16
 *
 * @param n
 * Number of 50 microsecond intervals to delay
 */
void delay_50us(int n) {
    volatile int c;

    for (; n > 0; n --) {
        for (c = 131; c > 0; c --) ;
    }
}

/**
 * This function loops for approximately 1 ms * n using delay_50us. The true
 * number of cycles consumed is 23953*n + 12
 *
 * @param n
 * Number of 1 millisecond intervals to delay
 */
void delay_1ms(int n) {
    for (; n > 0; n --)
        delay_50us(20);
}

/**
 * Sets the main clock to 24 MHz
 */
void bus_clock_init(void) {
    CLKSEL &= ~BIT_7;
    PLLCTL |= BIT_6;
    SYNR = 0x05;
    REFDV = 0x01;
    while ((CRGFLG & BIT_3) == 0);
    CLKSEL |= BIT_7;
}

/**
 * Sets the serial port to 115200 Baud
 */
```



```c
void sci_init(void) {
    SCI0BDH = 0x00;     // 115200 Baud
    SCI0BDL = 0x0D;     // 115200 Baud
    SCI0CR1 = 0x00;
    SCI0CR2 = 0x0C;     // Enable trans/recieve
}

/**
 * Sets up general input / output ports
 * PORTA: Output on bit 0, 1, and 7
 * PORTB: Output to LEDs on all bits
 * PORTP: Turn off seven seg display using bits 0 - 4
 * PORTJ: Enable LEDs
 */
void ports_init(void) {
    DDRA = BIT_7 | BIT_1 | BIT_0;   // Make bit 0 and 1 output
    PORTA = 0x00;                   // Set A to 00000000

    /* Port B is used for its LEDs */
    DDRB |= 0xFF;
    PORTB = 0x00;

    /* Port P is used for PWM */
    DDRP |= 0x1F;               // Make [7:0] output
    PTP |= 0x0F;                // Turn off seven segs

    /* Enable LEDs */
    DDRJ |= 0x02;
    PTJ  &= 0xFD;
}
```

```c
/**
 * @file lcd.h
 * @brief
 *  LCD display code
 */

#define LCD_DAT PORTK   /* Port K drives LCD data pins, E, and RS */
#define LCD_DIR DDRK    /* Direction of LCD port */
#define LCD_E 0x02      /* LCD E signal */
#define LCD_RS 0x01     /* LCD Register Select signal */
#define CMD 0           /* Command type for put2lcd */
#define DATA 1          /* Data type for put2lcd */

void openlcd(void);
void puts2lcd(unsigned char *ptr);
void put2lcd(char c, char type);
unsigned long hex2bcd(unsigned int x);
void set_lcd(unsigned char * msg1, unsigned char * msg2);
void put_int_to_lcd(unsigned int raw_int);
void put_position_to_lcd(long x, long y);
void ftoa(unsigned char *buf, float f);

const char num_2_string[] = { '0', '1', '2', '3', '4', '5',
                        '6', '7', '8', '9', 'A', 'B',
                        'C', 'D', 'E', 'F' };

/**
 * Prepare the LCD display to display a message
 */
void openlcd(void) {
    LCD_DIR = 0xFF;     /* configure LCD_DAT port for output */
    delay_1ms(2);       /* Wait for LCD to be ready */
    put2lcd(0x28,CMD);  /* set 4-bit data, 2-line display, 5x7 font */
    put2lcd(0x0F,CMD);  /* turn on display, cursor, blinking */
    put2lcd(0x06,CMD);  /* move cursor right */
    put2lcd(0x01,CMD);  /* clear screen, move cursor to home */
    delay_1ms(2);       /* wait until "clear display" command is complete */
}

/**
 * Display a null terminated string on the LCD at the current position
 *
 * @param ptr
 * Pointer to a null terminated string
 */
void puts2lcd(unsigned char *ptr) {
    while (*ptr) {          /* While character to send */
        put2lcd(*ptr, DATA); /* Write data to LCD */
        delay_50us(1);      /* Wait for data to be written */
        ptr++;              /* Go to next character */
    }
}

/**
 * Send a command to the LCD display
 *
 * @param c
 * Data to send
 * @param type
 * Type of data being sent, either DATA or CMD
 */
void put2lcd(char c, char type) {
    char c_lo, c_hi;

    c_hi = (c & 0xF0) >> 2;             /* Upper 4 bits of c */
    c_lo = (c & 0x0F) << 2;             /* Lower 4 bits of c */

    if (type == DATA) {
        LCD_DAT |= LCD_RS;              /* select LCD data register */
```



```c
    } else {
        LCD_DAT &= (~LCD_RS);           /* select LCD command register */
    }

    if (type == DATA) {
        LCD_DAT = c_hi|LCD_E|LCD_RS;    /* output upper 4 bits, E, RS high */
    } else {
        LCD_DAT = c_hi|LCD_E;           /* output upper 4 bits, E, RS low */
    }

    LCD_DAT |= LCD_E;                   /* pull E signal to high */
    __asm(nop);                         /* Lengthen E */
    __asm(nop);
    __asm(nop);
    LCD_DAT &= (~LCD_E);                /* pull E to low */

    if (type == DATA) {
        LCD_DAT = c_lo|LCD_E|LCD_RS;    /* output lower 4 bits, E, RS high */
    } else {
        LCD_DAT = c_lo|LCD_E;           /* output lower 4 bits, E, RS low */
    }

    LCD_DAT |= LCD_E;                   /* pull E to high */
    __asm(nop);                         /* Lengthen E */
    __asm(nop);
    __asm(nop);
    LCD_DAT &= (~LCD_E);                /* pull E to low */

    delay_50us(1);                      /* Wait for command to execute */
}

/**
 * Converts a hex value to a BCD value
 *
 * @param x
 * Unsigned hex value to convert
 * @return
 * Unsigned long integer BCD representation of x
 */
unsigned long hex2bcd(unsigned int x) {
    unsigned int d4,d3,d2,d1,d0;
    d4 = x / 10000;
    x = x - (d4 * 10000);
    d3 = x / 1000;
    x = x - (d3 * 1000);
    d2 = x / 100;
    x = x - (d2 * 100);
    d1 = x / 10;
    x = x - (d1 * 10);
    d0 = x;
    return ((unsigned long) d4)*16*16*16*16 + ((unsigned long) d3)*16*16*16 +
           ((unsigned long) d2)*16*16 + ((unsigned long) d1)*16 +
           ((unsigned long) d0);
}

/**
 * Writes two null terminated strings to the LCD display. Takes at least 5 ms
 * to complete
 *
 * @param msg1
 * String to write to the first line, must be null terminated and less than
 * or equal to 16 characters long
 * @param msg2
 * String to write to the second line, must be null terminated and less than
 * or equal to 16 characters long
 */
void set_lcd(unsigned char * msg1, unsigned char * msg2) {
    openlcd();          //Initialize LCD display
    puts2lcd(msg1);     //Send first line
```

```
    put2lcd(0xC0, CMD); //Move cursor to 2nd row, 1st col
    puts2lcd(msg2);     //Send second line
}

/**
 * Displays an integer value in both hex and decimal on the LCD display
 *
 * @param raw_int
 * Unsigned integer to display
 */
void put_int_to_lcd(unsigned int raw_int) {
    static unsigned int last_value = 0;
    unsigned char msg1[11];             // Raw hex message
    unsigned char msg2[12];             // BCD message
    unsigned long bcd_val;              // BCD converted value

    // Only update the display if the value has changed
    if (last_value != raw_int) {
        last_value = raw_int;

        // Store BCD value in bcd_val
        bcd_val = hex2bcd(raw_int);

        // Create null terminated strings from the lookup table
        msg1[0] = 'H';
        msg1[1] = 'E';
        msg1[2] = 'X';
        msg1[3] = ' ';
        msg1[4] = '-';
        msg1[5] = ' ';
        msg1[6] = num_2_string[(raw_int >> 12)  & 0x0F];
        msg1[7] = num_2_string[(raw_int >> 8)   & 0x0F];
        msg1[8] = num_2_string[(raw_int >> 4)   & 0x0F];
        msg1[9] = num_2_string[(raw_int)        & 0x0F];
        msg1[10] = '\0';

        msg2[0] = 'D';
        msg2[1] = 'E';
        msg2[2] = 'C';
        msg2[3] = ' ';
        msg2[4] = '-';
        msg2[5] = ' ';
        msg2[6] = num_2_string[(bcd_val >> 16)  & 0x0F];
        msg2[7] = num_2_string[(bcd_val >> 12)  & 0x0F];
        msg2[8] = num_2_string[(bcd_val >> 8)   & 0x0F];
        msg2[9] = num_2_string[(bcd_val >> 4)   & 0x0F];
        msg2[10] = num_2_string[(bcd_val)       & 0x0F];
        msg2[11] = '\0';

        set_lcd(msg1, msg2);
    }
}

/**
 * Displays a coordinate on the LCD display
 * @param x
 * X coordinate to display
 * @param y
 * X coordinate to display
 */
void put_position_to_lcd(long x, long y) {
    unsigned char msg1[13];             // X coordinate
    unsigned char msg2[13];             // Y coordinate

    // Create null terminated strings from the lookup table
    msg1[0] = 'X';
    msg1[1] = ':';
    msg1[2] = ' ';
    if (x < 0) {
```



```
        msg1[3] = '-';
        x = x * (-1);
    } else {
        msg1[3] = ' ';
    }
    msg1[4] = num_2_string[(((unsigned long) x)     >> 28) & 0x0F];
    msg1[5] = num_2_string[(((unsigned long) x)     >> 24) & 0x0F];
    msg1[6] = num_2_string[(((unsigned long) x)     >> 20) & 0x0F];
    msg1[7] = num_2_string[(((unsigned long) x)     >> 16) & 0x0F];
    msg1[8] = num_2_string[(((unsigned long) x)     >> 12) & 0x0F];
    msg1[9] = num_2_string[(((unsigned long) x)     >> 8)  & 0x0F];
    msg1[10] = num_2_string[(((unsigned long) x)    >> 4)  & 0x0F];
    msg1[11] = num_2_string[(((unsigned long) x)        )  & 0x0F];
    msg1[12] = '\0';

    msg2[0] = 'Y';
    msg2[1] = ':';
    msg2[2] = ' ';
    if (y < 0) {
        msg2[3] = '-';
        y = y * (-1);
    } else {
        msg2[3] = ' ';
    }
    msg2[4] = num_2_string[(((unsigned long) y)     >> 28) & 0x0F];
    msg2[5] = num_2_string[(((unsigned long) y)     >> 24) & 0x0F];
    msg2[6] = num_2_string[(((unsigned long) y)     >> 20) & 0x0F];
    msg2[7] = num_2_string[(((unsigned long) y)     >> 16) & 0x0F];
    msg2[8] = num_2_string[(((unsigned long) y)     >> 12) & 0x0F];
    msg2[9] = num_2_string[(((unsigned long) y)     >> 8)  & 0x0F];
    msg2[10] = num_2_string[(((unsigned long) y)    >> 4)  & 0x0F];
    msg2[11] = num_2_string[(((unsigned long) y)        )  & 0x0F];
    msg2[12] = '\0';

    set_lcd(msg1, msg2);
}

/**
 * Source: http://www.edaboard.com/thread63677.html
\verbatim
    ****************************************************************************
    *      CONVERT A FLOATING POINT NUMBER TO STRING WITH 1 DECIMAL PLACE
    *
    * Description : This function converts a floating point to a null terminated
    *               string with 1 decimal place.
    *
    *               Examples:
    *
    *               float f = 9.567;
    *               ftoa(&s[0], f);     //s[]={'9','.','5', 0}
    *               float f = -0.189;
    *               ftoa(&s[0], f);     //s[]={'-', '0', '.', '1', 0}
    * Arguments   : 'unsigned char* buf'    is the pointer to the string holding
    *                                       the conversion result
    *               'float f'               is the input floating point
    * Returns     : Returns the string with unsigned char* buf pointing to.
    * Notes       : This routine modified from itoa10() in ..\sample\misc folder
    *               of ht-picc
    *               If more decimal places required, modify the last section of
    *               this code
    *               Range of f in (-3,276.7, 3,276.7)
    *               This function does print result like 0.0.
    ****************************************************************************
\endverbatim
 *
 * @param buf
 * Pointer to the buffer to populate - must be long enough!
 * @param f
 * Floating point number to convert to a string
```

```
 */
void ftoa(unsigned char *buf, float f) {
    unsigned int rem;
    unsigned char *s,length=0;
    int i;

    i = (int)((float)f*10);

    s = buf;
    if (i == 0){        //print 0.0 with null termination here
        *s++ = '0';
        *s++ = '.';
        *s++ = '0';
        *s=0;           //null terminate the string
    } else {
        if (i < 0) {
            *buf++ = '-';
            s = buf;
            i = -i;
        }
        //while-loop to "decode" the long integer to ASCII by append '0',
        // string in reverse manner
        //If it is an integer of 124 -> string = {'4', '2', '1'}
        while (i) {
            ++length;
            rem = i % 10;
            *s++ = rem + '0';
            i /= 10;
        }
        //reverse the string in this for-loop, string became {'1', '2', '4'}
        // after this for-loop
        for(rem=0; ((unsigned char)rem)<length/2; rem++) {
            *(buf+length) = *(buf+((unsigned char)rem));
            *(buf+((unsigned char)rem)) = *(buf+(length-
                                ((unsigned char)rem)-1));
            *(buf+(length-((unsigned char)rem)-1)) = *(buf+length);
        }

        /* Take care of the special case of 0.x if length ==1*/
        if(length==1) {
            *(buf+2) = *buf;
            *buf = '0';
            *(buf+1) = '.';
            *(s+2)=0;           //null terminate
        } else {
            *(buf+length) = *(buf+length-1);
            *(buf+length-1)='.';
            *(s+1)=0;           //null terminate
        }
    }
}
```

```
/**
 * @file main.c
 * @brief
 *  Main program for robot navigation and control
 *
 * @mainpage Junior Design Team GRMM Autonomous Navigation Code
 *
 * @section motors Motors
 * − Right PWM − PP5
 * − Left PWM − PP4
 * − Right Directional − PA0
 * − Left Directional − PA1
 * − Motor enable − PA7
 *
 * @section icoc Input Capture / Output Compare Devices
 * − Front Right Encoder − PT0
 * − Front Left Encoder − PT1
 * − Back Right Encoder − PT2
 * − Back Left Encoder − PT3
 * − Servo − PT4
 * − Speaker (Internal) − PT5
 * − Right Bumper − PT6
 * − Left Bumper − PT7
 *
 * @section iic Serial Devices
 * − Compass − SCL / SDA
 * − Temperature Sensor − SCL / SDA
 *
 * @section analog Analog Devices
 * − Longrange IR sensor − PAD11
 * − Shortrange IR sensor 1 − PAD12
 * − Shortrange IR sensor 2 − PAD13
 * − Sonar rangefinder − PAD14
 * − Ultrasonic reciever − PAD15
 */

// If this is uncommented, many functions are slower and more verbose on the LCD
//#define DEBUG_MODE 1

#include <hidef.h>
#include <math.h>
#include <stdio.h>
#include <termio.h>
#include "derivative.h"
#include "vectors12.h"
#include "init.h"
#include "lcd.h"
#include "iic.h"
#include "motors.h"
#include "rangefinders.h"
#include "temp.h"
#include "compass.h"
#include "navigation.h"
#include "search.h"

interrupt void RTI_isr(void);
void RTI_init(void);
void start(void);

unsigned char count;
unsigned char current_duty;

/**
 * Startpoint for the program, initializes subsystems and calls an appropriate
 * function based on the dip switches
 */
void main(void) {
    unsigned long dummy;
```



```
    DISABLE_INTERRUPTS();

    /* Initialize subsystems */
    bus_clock_init();
    #ifdef DEBUG_MODE
        // Speeds up printf, not used anywhere in final code
        sci_init();
    #endif
    ports_init();
    iic_init();
    motors_init();
    servo_encoders_speaker_bumpers_init();
    rangefinders_init();
    compass_init();
    RTI_init();

    /* If switch 0 is set, calibrate the compass */
    if ((PTH & BIT_0) == BIT_0) compass_calibrate();

    ENABLE_IR();
    ENABLE_INTERRUPTS();

    /* Start at absolute location 0,0 */
    pos_x = 0;
    pos_y = 0;

    // If switch 1 is set, drive up the sidewalk 500 clicks to calibrate
    // encoders
    if ((PTH & BIT_1) == BIT_1) {
        DISABLE_IR();
        dummy = travel_distance(500, 3500, CONCRETE);
    }

    // If switch 2 is set, start navigating from the beginning
    if ((PTH & BIT_2) == BIT_2) start();

    while (1) {
        __asm(wai);
    }
}

/**
 * The RTI function is used only for debugging − displays specific information
 * on the LCD display depending on the state of the dip switches
 */
interrupt void RTI_isr(void){
    /* Switch 7         = Compass Heading
     * Switch 6         = Front IR distance in CM
     * Switch 5         = Sonar sensor distance in CM
     * Switch 4 & 3     = Right IR distance in CM
     * Switch 4         = Ultrasonic reciever 10 bit ADC value
     * Switch 3         = Left IR distance in CM
     */
    if ((PTH & BIT_7) == BIT_7) {
        put_int_to_lcd(compass_get_bearing());
    } else if ((PTH & BIT_6) == BIT_6) {
        put_int_to_lcd(IR_FRONT_CM());
    } else if ((PTH & BIT_5) == BIT_5) {
        put_int_to_lcd(SONAR_CM());
    } else if ((PTH & (BIT_4 | BIT_3)) == (BIT_4 | BIT_3)) {
        put_int_to_lcd(IR_RIGHT_CM());
    } else if ((PTH & BIT_3) == BIT_3) {
        put_int_to_lcd(IR_LEFT_CM());
    } else if ((PTH & BIT_4) == BIT_4) {
        put_int_to_lcd(ULTRASONIC_RAW_INT());
    }

  // Clear interrupt
    CRGFLG = BIT_7;
```

```c
}

/**
 * All code for actually navigating the course, from the start point by workman
 * to (theoretically) the object at waypoint 5
 */
void start(void){
    // Dummy variable avoids annoying warnings from the compiler
    unsigned long dummy;
    unsigned int i;
    unsigned int bearing;
    unsigned long travel_return;
    unsigned char loop_counter;

    CALIBRATE_SONAR();
// **** First waypoint, navigate from the start to the end of the pink rail ****
    // Follow the wall until just before the edge of the circle
    dummy = follow_wall(2500, 190, cm_to_ticks(1128), CONCRETE);

    // Re adjust position against the wall
    dummy = follow_wall(2500, 190, 0, CONCRETE);

    // Zero out the robot's position
    pos_x = 0;
    pos_y = 0;

    // Search in a straight line inside the circle for the waypoint
    for (i = 0; i < 6; i ++) {

        if (use_sensor(sonar_search, 150, CONCRETE) == 1) {
            //Found the waypoint, stop searching
            break;
        }
        if (travel_distance(cm_to_ticks(130), 3470, CONCRETE) != 0) {
            // Encountered an object enroute, back up a little bit and
            // resume searching
            dummy = travel_distance(-20, get_angle(0), CONCRETE);
        }
    }

    // Go to the other side of the circle
    pathfind_to_location(cm_to_ticks(-142), cm_to_ticks(805), CONCRETE);

    // Go to the end of the pink wall
    dummy = follow_wall(2500, 195, 1500, CONCRETE);

// *** Second waypoint, go from the end of workman to the diagonal sidewalk ***
    // Go to the railroad ties
    dummy = travel_distance(1500, 3500, CONCRETE);

    // Go backwards from the railroad ties
    dummy = travel_distance(cm_to_ticks(-325), 3500, CONCRETE);

    // Calibrate the compass again
    compass_calibrate();

    // Aim servo toward empty space
    ENABLE_SERVO();
    servo_position = 127;
    delay_1ms(1000);
    DISABLE_SERVO();

    // Calibrate the sonar sensor
    CALIBRATE_SONAR();

    // Find the light post and adjust position against it
    dummy = drive_bearing_until_sonar(650, 1000, 275, 0, 500, CONCRETE);
    dummy = travel_distance((((int) cm_to_ticks(SONAR_CM()))
                            - ((int) cm_to_ticks(140))), 1750, CONCRETE);
```



```c
    // Travel to the edge of the circle
    dummy = travel_distance(cm_to_ticks(366), 660, CONCRETE);

    // Aim the servo across the circle and calibrate
    ENABLE_SERVO();
    servo_position = -127;
    delay_1ms(1000);
    DISABLE_SERVO();
    CALIBRATE_SONAR();

    // Zero the position
    pos_x = 0;
    pos_y = 0;

    if (drive_bearing_until_sonar(650, cm_to_ticks(580), 350, 90, 200,
                                  CONCRETE) != 0) {

        if ((IR_FRONT_CM() < 20) || (bumper_right_state != BUMPER_CLEAR) ||
            (bumper_left_state != BUMPER_CLEAR) || (ir_left < 90) ||
            (ir_right < 90)) {

            // Encountered an object enroute, probably the waypoint
            // If the IR sensors found it, aim the servo at the object
            if (ir_left < 90) {
                success_action_location();
                ENABLE_SERVO();
                servo_position = -127;
                delay_1ms(1000);
                DISABLE_SERVO();
            } else if (ir_right < 90) {
                success_action_location();
                ENABLE_SERVO();
                servo_position = 127;
                delay_1ms(1000);
                DISABLE_SERVO();
            }
            success_action_object();
            // If the object was encountered by the bumpers or the front IR,
            // move backwards
            if ((ir_left > 90) && (ir_right > 90)) {
                dummy = travel_distance(-25, get_angle(0), CONCRETE);
            }
        } else {
            // Found something with the ultrasonic, need to approach it and
            // signal
            success_action_location();
            dummy = travel_distance(sonar_temp, 3500, CONCRETE);
            success_action_object();
            dummy = travel_distance(-25, get_angle(0), CONCRETE);
        }
        // Go to the edge of the circle
        pathfind_to_location(cm_to_ticks(526), cm_to_ticks(275), CONCRETE);
    }

    // Re align robot
    turn_to_bearing(650, CONCRETE);

    // Aim servo towards tree
    ENABLE_SERVO();
    servo_position = 127;
    delay_1ms(1000);
    DISABLE_SERVO();

    // Drive until find tree and move a set distance away from it
    dummy = drive_bearing_until_sonar(650, 500, 300, 0, 50, CONCRETE);
    dummy = travel_distance((((int) cm_to_ticks(SONAR_CM())) -
                            ((int) cm_to_ticks(250))), 1750, CONCRETE);
```

```
// ************ Third waypoint, go down the sidewalk to the beacon ************
    // Disable the IR sensors because they pick up grass
    DISABLE_IR();

    // Travel down the sidewalk
    dummy = travel_distance(cm_to_ticks(2000), 1120, CONCRETE);

    // Aim servo towards box
    ENABLE_SERVO();
    servo_position = -127;
    delay_1ms(1000);
    DISABLE_SERVO();

    // Find the box
    dummy = drive_bearing_until_sonar(1120, cm_to_ticks(3000), 400, 0, 500,
                                    CONCRETE);

    // Disable the sonar sensor because it interferes with the ultrasonic
    // reciever
    DISABLE_SONAR();

    // Travel near the ultrasonic beacon and point towards it
    dummy = travel_distance(grass_cm_to_ticks(1050), 2300, GRASS);
    dummy = travel_distance(grass_cm_to_ticks(100), 1800, GRASS);

    // Find the ultrasonic beacon
    for (loop_counter = 0; loop_counter < 20; loop_counter ++) {
        bearing = ultrasonic_search(100);
        if (bearing == 0xFFFF) break;
        // Approach the object
        travel_return = travel_distance(grass_cm_to_ticks(150), bearing, GRASS);
        if (travel_return != 0) {
            // Reached an object while traveling, probably the waypoint
            success_action_object();
            dummy = travel_distance(grass_cm_to_ticks(-25), get_angle(0),
                                    GRASS);
            break;
        }
    }

// ************ Fourth waypoint, go to the relative location for #4 ************
    // Go to the relative location for waypoint 4
    pos_x = 0;
    pos_y = 0;
    pathfind_to_location(grass_cm_to_ticks(1300), grass_cm_to_ticks(-1440),
                         GRASS);
    success_action_object();

// *********** Final waypoint, go from location 4 to the object at 5 ***********
    // Travel in the gerenal direction of waypoint 5
    dummy = travel_distance(grass_cm_to_ticks(1220), 500, GRASS);

    // Re-enable the sonar sensor so it can find the railroad ties
    ENABLE_SONAR();

    // Center Servo
    ENABLE_SERVO();
    servo_position = OFFSET_SONAR;
    delay_1ms(1000);
    DISABLE_SERVO();

    // Approach the railroad ties
    dummy = drive_bearing_until_sonar(3200, grass_cm_to_ticks(1000), 250, 0,
                                  300, GRASS);
    // Follow railroad ties
    dummy = follow_wall(3250, 250, grass_cm_to_ticks(0), GRASS);
    dummy = travel_distance(grass_cm_to_ticks(960), 500, GRASS);
    dummy = travel_distance(grass_cm_to_ticks(50), 1300, GRASS);
```



```
    // Search in a straight line inside the circle for the waypoint
    for (i = 0; i < 4; i ++) {
        if (use_sensor(sonar_search, 450, GRASS) == 1) {
            //Found the waypoint, stop searching
            break;
        }
        if (travel_distance(grass_cm_to_ticks(400), 1300, GRASS) != 0) {
            // Encountered an object enroute, back up a little bit and
            // resume searching
            dummy = travel_distance(-20, get_angle(0), GRASS);
        }
    }
    dummy ++;
}

// Set the RTI to 8.192 ms
void RTI_init(void) {
    UserRTI = (unsigned short) &RTI_isr;
    RTICTL = 0x70;      // 8.192 ms rate
    CRGINT |= BIT_7;    // Enable RTI
    CRGFLG = 0x80;      // Clear interrupt
}
```

```
/**
 * @file motors.h
 * @brief
 *  Motor and servo control code
 */

#define FORWARD             0xFF
#define BACKWARD            0x00
#define RDUTY               PWMDTY5
#define LDUTY               PWMDTY4
#define RDIR                PORTA_BIT0
#define LDIR                PORTA_BIT1

#define SERVO_PULSE_LOW     0
#define SERVO_PULSE_HIGH    1
#define SERVO_PERIOD        60000

#define SERVO_CENTER_COUNT  4370
#define SERVO_STEP          20

#define MOTOR_MIN_GRASS_PWM         100
#define MOTOR_MIN_CONCRETE_PWM      75
#define MOTOR_MAX_PWM       255

#define BUMPER_CLEAR        0x00
#define BUMPER_HIT          0x01

#define ENABLE_MOTORS()     PORTA_BIT7=0xFF
#define DISABLE_MOTORS()    PORTA_BIT7=0x00;RDUTY=0x00;LDUTY=0x00
#define ENABLE_ENCODERS()   TIE|=(BIT_0|BIT_1|BIT_2|BIT_3)
#define DISABLE_ENCODERS()  TIE&=~(BIT_0|BIT_1|BIT_2|BIT_3)
#define ENABLE_SERVO()      TIE|=BIT_4
#define DISABLE_SERVO()     TIE&=~BIT_4
#define ENABLE_SPEAKER()    TIE|=BIT_5;TCTL1=(TCTL1&~(BIT_3|BIT_2))|BIT_2
#define DISABLE_SPEAKER()   TIE&=~BIT_5;TCTL1=(TCTL1&~(BIT_3|BIT_2))|BIT_3
#define ENABLE_BUMPERS()    TIE|=(BIT_6|BIT_7)
#define DISABLE_BUMPERS()   TIE&=~(BIT_6|BIT_7)

#define OFFSET_TEMP -35
#define OFFSET_ULTRASONIC -35
#define OFFSET_SONAR 0

#define GRASS 0
#define CONCRETE 1

void motors_init(void);
void servo_encoders_speaker_bumpers_init(void);
interrupt void encoder_fr_isr(void);
interrupt void encoder_fl_isr(void);
interrupt void encoder_br_isr(void);
interrupt void encoder_bl_isr(void);
interrupt void servo_isr(void);
interrupt void speaker_isr(void);
interrupt void bumper_right_isr(void);
interrupt void bumper_left_isr(void);
void update_motors(unsigned char LS, unsigned char RS, unsigned char LD,
                   unsigned char RD, unsigned char step, unsigned char surface);
void success_action_location(void);
void success_action_object(void);

/**
 * Signed variable indicating the angle of the servo
 * Centered = 0
 * Maximum = 127
 * Minimum = -128
 */
char servo_position;

unsigned long ticks_fr;
```



```
unsigned long ticks_fl;
unsigned long ticks_br;
unsigned long ticks_bl;
unsigned char bumper_right_state;
unsigned char bumper_left_state;
unsigned int speaker_step;

/**
 * Signed variable indicating the angle of the servo
 * Centered = 0
 * Maximum = 127
 * Minimum = -128
 */
char servo_position;

unsigned long ticks_fr;
unsigned long ticks_fl;
unsigned long ticks_br;
unsigned long ticks_bl;
unsigned char bumper_right_state;
unsigned char bumper_left_state;
unsigned int speaker_step;

/**
 * Enables PWM at 18750 Hz on channel PP4 and PP5
 */
void motors_init(void) {
    PWME |= BIT_4 | BIT_5;                      // Enable PWM on Channel 4 and 5
    PWMCTL = 0x00;                              // 8 bit mode
    PWMPOL |= BIT_4 | BIT_5;                    // High Polarity
    PWMCAE &= ~(BIT_4 | BIT_5);                 // Left aligned
    PWMCLK |= BIT_4 | BIT_5;                    // Use clock A
    PWMPRCLK = (PWMPRCLK & 0xF0) | 0x07;        // Set A prescaler to 7
    PWMSCLA = 0x05;                             // Set SA prescaler to 5
    PWMDTY4 = 0x00;
    PWMDTY5 = 0x00;
    PWMPER4 = 0xFF;                             // 255
    PWMPER5 = 0xFF;                             // 255
}

/**
 * Sets up all of the input capture / output compare subsystems
 */
void servo_encoders_speaker_bumpers_init(void) {
    /* Set the global variables */
    ticks_fr = 0;
    ticks_fl = 0;
    ticks_br = 0;
    ticks_bl = 0;
    servo_position = 0;
    bumper_right_state = BUMPER_CLEAR;
    bumper_left_state = BUMPER_CLEAR;

    TSCR1 = 0x80;               // Enable Timer
    TSCR2 = 0x03;               // Overflow = 21.8453 ms, no overflow interrupt

    /*  Encoders input capture on PT0 - PT3
        Servo output compare on PT4
        Speaker output compare on PT5
        Bumpers input capture on PT6 - PT7
    */
    TIOS = BIT_4 | BIT_5;

    /* Output compare - set channels 4 initially and toggle channel 5 */
    TCTL1 = 0x0B;
    TCTL2 = 0x00;

    /* Input capture - rising edge channels 0 - 3, 6, 7 */
    TCTL3 = 0x50;
```

```
    TCTL4 = 0x55;

    /* Clear all the interrupt flags */
    TFLG1 = 0xFF;

    /* Set up interrupts */
    UserTimerCh0 = (unsigned short) &encoder_fr_isr;
    UserTimerCh1 = (unsigned short) &encoder_fl_isr;
    UserTimerCh2 = (unsigned short) &encoder_br_isr;
    UserTimerCh3 = (unsigned short) &encoder_bl_isr;
    UserTimerCh4 = (unsigned short) &servo_isr;
    UserTimerCh5 = (unsigned short) &speaker_isr;
    UserTimerCh6 = (unsigned short) &bumper_right_isr;
    UserTimerCh7 = (unsigned short) &bumper_left_isr;

    /* Disable all the interrupts */
    TIE = 0x00;
}

interrupt void encoder_fr_isr(void) {
    ticks_fr ++;
    TFLG1 = BIT_0; // Clear Interrupt
}

interrupt void encoder_fl_isr(void) {
    ticks_fl ++;
    TFLG1 = BIT_1; // Clear Interrupt
}

interrupt void encoder_br_isr(void) {
    ticks_br ++;
    TFLG1 = BIT_2; // Clear Interrupt
}

interrupt void encoder_bl_isr(void) {
    ticks_bl ++;
    TFLG1 = BIT_3; // Clear Interrupt
}

/**
 * Rotates the servo to servo_position
 */
interrupt void servo_isr(void) {
    static unsigned char servo_status;
    static unsigned int servo_last_high_count;

    if (servo_status == SERVO_PULSE_LOW) {
        /* Output currently low - after time elapses, bring output high */
        TC4 = TC4 + (SERVO_PERIOD - servo_last_high_count);
        TCTL1 |= BIT_0 | BIT_1;
        servo_status = SERVO_PULSE_HIGH;
    } else {
        /* Output currently high - after time elapses, bring output low */
        servo_last_high_count = (unsigned int) (SERVO_CENTER_COUNT
                                              + SERVO_STEP * servo_position);
        TC4 = TC4 + servo_last_high_count;
        TCTL1 = (TCTL1 | BIT_1) & ~BIT_0;
        servo_status = SERVO_PULSE_LOW;
    }

    TFLG1 = BIT_4; // Clear Interrupt
}

interrupt void speaker_isr(void) {
    TC5 = TC5 + speaker_step;
    TFLG1 = BIT_5; // Clear Interrupt
}

interrupt void bumper_right_isr(void) {
```



```
    bumper_right_state = BUMPER_HIT;
    //DISABLE_MOTORS();
    TFLG1 = BIT_6; // Clear Interrupt
}

interrupt void bumper_left_isr(void) {
    bumper_left_state = BUMPER_HIT;
    //DISABLE_MOTORS();
    TFLG1 = BIT_7; // Clear Interrupt
}

/**
 * Updates the left and right motor speeds gracefully to prevent rapid changes
 *
 * @param RS
 *  Desired speed of the right motor
 * @param LS
 *  Desired speed of the left motor
 * @param RD
 *  Desired direction of the right motor; use the constant FORWARD or BACKWARD
 * @param LD
 *  Desired direction of the left motor; use the constant FORWARD or BACKWARD
 * @param step
 *  The amount that the duty register is allowed to change each cycle; a higher
 *  value will make changes faster. Set to 255 for instant reaction.
 * @param surface
 *  The surface the robot is expected to be driving on, either GRASS or CONCRETE
*/
void update_motors(unsigned char LS, unsigned char RS, unsigned char LD,
                   unsigned char RD, unsigned char step, unsigned char surface) {
    unsigned char min_pwm;

    if (surface == CONCRETE) {
        min_pwm = MOTOR_MIN_CONCRETE_PWM;
    } else {
        min_pwm = MOTOR_MIN_GRASS_PWM;
    }

    if ((RS < (unsigned char) RDUTY) || ((PORTA & BIT_0) != (RD & BIT_0))) {
        if (((short) RDUTY - (short) step) < min_pwm) {
            RDUTY = 0;
            RDIR = RD;
        } else if ((((short) RDUTY - (short) step < RS)) &&
                    ((PORTA & BIT_0) == (RD & BIT_0))) {
            RDUTY = RS;
        } else {
            RDUTY -= step;
        }
    } else if (RS > ((unsigned char) RDUTY)) {
        if (((unsigned short) RDUTY + (unsigned short) step) > MOTOR_MAX_PWM) {
            RDUTY = MOTOR_MAX_PWM;
        } else if (((unsigned short) RDUTY + (unsigned short) step) < min_pwm) {
            RDUTY = min_pwm;
        } else if (((unsigned short) RDUTY + (unsigned short) step) > RS) {
            RDUTY = RS;
        } else {
            RDUTY += step;
        }
    }

    if ((LS < ((unsigned char) LDUTY)) || (((PORTA & BIT_1) != (LD & BIT_1)))) {
        if ((short) LDUTY - (short) step < min_pwm) {
            LDUTY = 0;
            LDIR = LD;
        } else if ((((short) LDUTY - (short) step < LS) &&
                    ((PORTA & BIT_1) == (LD & BIT_1))) {
            LDUTY = LS;
        } else {
            LDUTY -= step;
```

```
        }
    } else if (LS > (unsigned char) LDUTY) {
        if (((unsigned short) LDUTY + (unsigned short) step) > MOTOR_MAX_PWM) {
            LDUTY = MOTOR_MAX_PWM;
        } else if (((unsigned short) LDUTY + (unsigned short) step) < min_pwm) {
            LDUTY = min_pwm;
        } else if (((unsigned short) LDUTY + (unsigned short) step) > LS) {
            LDUTY = LS;
        } else {
            LDUTY += step;
        }
    }
}

/**
 * One long beep from the speaker, usually identifing a location or important
 * action.
 */
void success_action_location(void) {
    speaker_step = 2000;
    ENABLE_SPEAKER();
    delay_1ms(500);
    DISABLE_SPEAKER();
}

/**
 * A beep with varying tone signaling that the robot has found the target object
 */
void success_action_object(void) {
    speaker_step = 1000;
    ENABLE_SPEAKER();
    for (speaker_step = 1000; speaker_step < 3000; speaker_step += 2) {
        delay_1ms(1);
    }
    for (speaker_step = 3000; speaker_step > 1000; speaker_step -= 2) {
        delay_1ms(1);
    }
    DISABLE_SPEAKER();
}
```

```
/**
 * @file navigation.h
 * @brief
 * Navigation algorithms
 */

#define CW 0
#define CCW 1
#define PI 3.14159

unsigned long drive_bearing_until_sonar(unsigned int bearing,
                                        unsigned int max_distance,
                                        unsigned int sonar_max,
                                        unsigned int ir_max,
                                        unsigned int min_count,
                                        unsigned char surface);
unsigned long follow_wall(unsigned int wall_bearing,
                          unsigned int target_wall_distance,
                          unsigned long target_distance,
                          unsigned char surface);
unsigned long go_to_location(long x, long y, unsigned char surface);
unsigned long travel_distance(long input_distance,
                              unsigned int bearing,
                              unsigned char surface);
unsigned long get_encoder_average(void);
void turn_to_bearing(unsigned int target_bearing, unsigned char surface);
unsigned int get_angle(int relative_bearing);
unsigned int get_distance_to_bearing(unsigned int bearing,
                                     unsigned char surface);
void pathfind_to_location(unsigned long targetx,
                          unsigned long targety,
                          unsigned char surface);

/* Absolute position of the robot */
long pos_x;
long pos_y;

/* Last rangefinder reading accessible after a navigation function exits */
unsigned int sonar_temp;
unsigned int ir_right;
unsigned int ir_left;

/**
 * Drives in a specified direction a until the sonar picks something up below
 * a specified distance.
 * @param bearing
 * Compass bearing to drive towards
 * @param max_distance
 * Maximum number of encoder ticks to travel if nothing is detected
 * @param sonar_max
 * Maximum value the sonar sensor can return before an object is recognized
 * @param ir_max
 * Maximum value of the side IR sensors before an object is recognized
 * @param min_count
 * Number of 2 ms hits in a row before an object is officially detected.
 * @param surface
 * The surface the robot is expected to be driving on, either GRASS or CONCRETE
 * @return
 * Zero if no object is found, number of encoder ticks traveled otherwise
 */
unsigned long drive_bearing_until_sonar(unsigned int bearing,
                                        unsigned int max_distance,
                                        unsigned int sonar_max,
                                        unsigned int ir_max,
                                        unsigned int min_count,
                                        unsigned char surface) {
    unsigned int current_bearing;
    int theta;
    unsigned char rotate_direction;
```



```
    unsigned char left_speed;
    unsigned char right_speed;
    unsigned int count;
    unsigned long traveled_distance;

    #ifndef DEBUG_MODE
        unsigned int counter;
    #endif

    turn_to_bearing(bearing, surface);

    /* Clear the encoders */
    count = 0;
    ticks_fr = 0;
    ticks_fl = 0;
    ticks_br = 0;
    ticks_bl = 0;
    bumper_right_state = BUMPER_CLEAR;
    bumper_left_state = BUMPER_CLEAR;
    ENABLE_ENCODERS();
    ENABLE_BUMPERS();
    ENABLE_MOTORS();

    sonar_temp = 0;

    do {
        #ifdef DEBUG_MODE
            put_int_to_lcd(SONAR_CM());
            traveled_distance = get_encoder_average();
        #else
            // Delay for 2 ms while checking encoders
            counter = 20;
            while (counter --) {
                delay_50us(2);
                traveled_distance = get_encoder_average();
                if (traveled_distance >= max_distance) break;
            }
        #endif

        /* Stop if an object is encountered */
        if (traveled_distance >= max_distance) {
            #ifdef DEBUG_MODE
                set_lcd((unsigned char *) "Distance\0", (unsigned char *) "\0");
            #endif
            break;
        }
        if (IR_FRONT_CM() < 20) {
            #ifdef DEBUG_MODE
                set_lcd((unsigned char *) "Front IR\0", (unsigned char *) "\0");
            #endif
            break;
        }
        if ((bumper_right_state != BUMPER_CLEAR) ||
            (bumper_left_state != BUMPER_CLEAR)) {
            #ifdef DEBUG_MODE
                set_lcd((unsigned char *) "Bumpers\0", (unsigned char *) "\0");
            #endif
            break;
        }

        /* Get the current direction of the robot */
        current_bearing = compass_get_bearing();

        /* Calculate which direction to turn and how much */
        theta = ((int) bearing) - ((int) current_bearing);
        if (theta < 0) theta += 3600;
        if (theta <= 1800) {
            rotate_direction = CW;
        } else {
```

```
            /* Recalculate theta in the opposite direction */
            theta = ((int) current_bearing) − ((int) bearing);
            if (theta < 0) theta += 3600;
            rotate_direction = CCW;
        }

        left_speed = MOTOR_MIN_GRASS_PWM;
        right_speed = MOTOR_MIN_GRASS_PWM + 20;

        // Adjust the speeds to maintain compass bearing
        if (theta < 10) {
            // Within compass tolerance, no need for dramatic adjustments
            update_motors(left_speed, right_speed,
                          FORWARD, FORWARD, 1, surface);
        } else {
            // Outside tolerance, need to correct course
            if (rotate_direction == CW) {
                update_motors(left_speed + 30, right_speed,
                              FORWARD, FORWARD, 5, surface);
            } else {
                update_motors(left_speed, right_speed + 20,
                              FORWARD, FORWARD, 5, surface);
            }
        }

        sonar_temp = SONAR_CM();
        ir_right = IR_RIGHT_CM();
        ir_left = IR_LEFT_CM();

        if ((ir_left < ir_max) || (ir_right < ir_max)) {
            count += 5;
        } else if ((sonar_temp < sonar_max) && (sonar_temp > 40)) {
            count ++;
        } else {
            count = 0;
        }
    } while ((count < min_count) && (traveled_distance < max_distance));

    update_motors(0, 0, FORWARD, FORWARD, 200, surface);

    DISABLE_MOTORS();
    DISABLE_ENCODERS();
    DISABLE_BUMPERS();

    #ifdef DEBUG_MODE
        if (count >= min_count) {
            set_lcd((unsigned char *) "Count\0", (unsigned char *) "\0");
        }
    #endif

    /* Update robot global position */
    if (bearing > 2700) {
        theta = 3600 − bearing;
        pos_x −= (long) (((double) traveled_distance) *
                          sin(((double) theta) * PI / 1800));
        pos_y += (long) (((double) traveled_distance) *
                          cos(((double) theta) * PI / 1800));
    } else if (bearing > 1800) {
        theta = bearing − 1800;
        pos_x −= (long) (((double) traveled_distance) *
                          sin(((double) theta) * PI / 1800));
        pos_y −= (long) (((double) traveled_distance) *
                          cos(((double) theta) * PI / 1800));
    } else if (bearing > 900) {
        theta = 1800 − bearing;
        pos_x += (long) (((double) traveled_distance) *
                          sin(((double) theta) * PI / 1800));
        pos_y −= (long) (((double) traveled_distance) *
                          cos(((double) theta) * PI / 1800));
```



```
    } else {
        theta = bearing;
        pos_x += (long) (((double) traveled_distance) *
                          sin(((double) theta) * PI / 1800));
        pos_y += (long) (((double) traveled_distance) *
                          cos(((double) theta) * PI / 1800));
    }

    delay_1ms(300);
    if (traveled_distance < max_distance − 2) {
        // Didn’t reach the destination
        return traveled_distance;
    }

    return 0;
}

/**
 * Attempts to follow a wall on the left side of the robot for a set number of
 * encoder ticks at a set distance.
 * @param wall_bearing
 * The compass bearing where the wall is expected to be initially
 * @param target_wall_distance
 * The distance in centimeters that the robot should be away from the wall
 * @param target_distance
 * The max number of encoder ticks parallel to the wall to travel
 * @param surface
 * The surface the robot is expected to be driving on, either GRASS or CONCRETE
 * @return
 * Zero if no object is found, number of encoder ticks traveled otherwise
 */
unsigned long follow_wall(unsigned int wall_bearing,
                          unsigned int target_wall_distance,
                          unsigned long target_distance,
                          unsigned char surface) {
    unsigned long traveled_distance;
    unsigned int wall_dist;
    int theta;
    unsigned char counter;
    unsigned int drive_bearing, current_bearing;
    unsigned char rotate_direction;
    char right_offset, left_offset;

    // Center the sonar sensor
    servo_position = 0;
    ENABLE_SERVO();
    delay_1ms(1000);
    DISABLE_SERVO();

    // Turn to face the wall
    turn_to_bearing(wall_bearing, surface);

    // Move the correct distance away from the wall
    ENABLE_MOTORS();
    do {
        delay_1ms(10);
        wall_dist = SONAR_CM();

        if (wall_dist > target_wall_distance) {
            update_motors(MOTOR_MIN_GRASS_PWM, MOTOR_MIN_GRASS_PWM,
                          FORWARD, FORWARD, 4, surface);
        } else {
            update_motors(MOTOR_MIN_GRASS_PWM, MOTOR_MIN_GRASS_PWM,
                          BACKWARD, BACKWARD, 4, surface);
        }
    } while ((wall_dist < target_wall_distance − 3)
             || (wall_dist > target_wall_distance + 3));

    // Stop the motors
```

```
    update_motors(0, 0, FORWARD, FORWARD, 200, surface);
    DISABLE_MOTORS();

    // Aim the servo to the right
    servo_position = -128;
    ENABLE_SERVO();
    delay_1ms(1000);
    DISABLE_SERVO();

    // Turn perpendicular to the wall
    drive_bearing = wall_bearing + 900;
    if (drive_bearing >= 3600) drive_bearing -= 3600;
    turn_to_bearing(drive_bearing, surface);

    // Clear the encoders
    ticks_fr = 0;
    ticks_fl = 0;
    ticks_br = 0;
    ticks_bl = 0;
    bumper_right_state = BUMPER_CLEAR;
    bumper_left_state = BUMPER_CLEAR;

    ENABLE_ENCODERS();
    ENABLE_BUMPERS();
    ENABLE_MOTORS();

    // Travel along the wall
    do {
        // Delay for 2 ms while checking encoders
        #ifdef DEBUG_MODE
            put_int_to_lcd(wall_dist);
        #endif

        counter = 20;
        while (counter --) {
            delay_50us(2);
            traveled_distance = get_encoder_average();
            if (traveled_distance >= target_distance) break;
        }

        // Stop if an object is encountered
        if (traveled_distance >= target_distance) break;
        if (IR_FRONT_CM() < 20) break;
        if (((bumper_right_state != BUMPER_CLEAR) ||
            (bumper_left_state != BUMPER_CLEAR))) break;

        // Get the current direction of the robot
        current_bearing = compass_get_bearing();

        // Calculate which direction to turn and how much
        theta = ((int) drive_bearing) - ((int) current_bearing);
        if (theta < 0) theta += 3600;
        if (theta <= 1800) {
            rotate_direction = CW;
        } else {
            // Recalculate theta in the opposite direction
            theta = ((int) current_bearing) - ((int) drive_bearing);
            if (theta < 0) theta += 3600;
            rotate_direction = CCW;
        }

        wall_dist = SONAR_CM();

        if (wall_dist > (target_wall_distance + 200)) break;
        if (wall_dist > target_wall_distance) {
            if (rotate_direction == CCW) {
                // Compass agrees with ultrasonic
                if (theta > 150) {
                    // Compass massively agrees,
```



```
                    // need to correct the course a lot
                    left_offset = -20;
                    right_offset = 10;
                } else {
                    // Compass agrees slightly, need to correct the course some
                    left_offset = 0;
                    right_offset = 15 + (char) (theta / 10);;
                }
            } else {
                // Compass disagrees with ultrasonic
                if (theta > 150) {
                    // Too much deviation away from the compass heading; ignore
                    // the ultrasonic and re-correct course
                    left_offset = (char) (theta / 10);
                    right_offset = 0;
                } else {
                    // Compass disagrees only slightly, can turn a bit more
                    left_offset = 0;
                    right_offset = 15 - (char) (theta / 10);
                }
            }
        } else {
            if (rotate_direction == CW) {
                // Compass agrees with ultrasonic
                if (theta > 150) {
                    // Compass massively agrees,
                    // need to correct the course a lot
                    left_offset = 20;
                    right_offset = -20;
                } else {
                    // Compass agrees slightly, need to correct the course some
                    left_offset = 22 + (char) (theta / 10);
                    right_offset = 0;
                }
            } else {
                // Compass disagrees with ultrasonic
                if (theta > 150) {
                    // Too much deviation away from the compass heading; ignore
                    // the ultrasonic and re-correct course
                    left_offset = 0;
                    right_offset = (char) (theta / 10);
                } else {
                    // Compass disagrees only slightly, can turn a bit more
                    left_offset = 15 - (char) (theta / 10);
                    right_offset = 0;
                }
            }
        }
        update_motors(160 + left_offset, 190 + right_offset,
                      FORWARD, FORWARD, 1, surface);
    } while (traveled_distance < target_distance);

    update_motors(0, 0, FORWARD, FORWARD, 200, surface);

    DISABLE_MOTORS();
    DISABLE_ENCODERS();
    DISABLE_BUMPERS();

    #ifdef DEBUG_MODE
        put_int_to_lcd((unsigned int) traveled_distance);
        delay_1ms(2000);
    #endif


    delay_1ms(300);
    if (traveled_distance < target_distance - 2) {
        // Didn't reach the destination
        return traveled_distance;
    }
```

```c
    return 0;
}

/**
 * Travels to an absolute position in the courtyard
 * @param x
 * X position (increases going east)
 * @param y
 * Y position (increases going north)
 * @param surface
 * The surface the robot is expected to be driving on, either GRASS or CONCRETE
 * @return
 * 0 on success, 1 if an object was encountered enroute
 */
unsigned long go_to_location(long x, long y, unsigned char surface) {
    unsigned long xdist, ydist;
    unsigned long travel_dist;
    unsigned int theta;

    if ((pos_x <= x) && (pos_y <= y)) {
        /* North & East */
        xdist = x - pos_x;
        ydist = y - pos_y;
        theta = (unsigned int) (atan(((double) xdist) /
                    ((double) ydist)) * 1800.0 / PI);
    } else if ((pos_x <= x) && (pos_y > y)) {
        /* South & East */
        xdist = x - pos_x;
        ydist = pos_y - y;
        theta = 900 + (unsigned int) (atan(((double) ydist) /
                        ((double) xdist)) * 1800.0 / PI);
    } else if ((pos_x > x) && (pos_y <= y)) {
        /* North & West */
        xdist = pos_x - x;
        ydist = y - pos_y;
        theta = 2700 + (unsigned int) (atan(((double) ydist) /
                        ((double) xdist)) * 1800.0 / PI);
    } else {
        /* South & West */
        xdist = pos_x - x;
        ydist = pos_y - y;
        theta = 1800 + (unsigned int) (atan(((double) xdist) /
                        ((double) ydist)) * 1800.0 / PI);
    }
    travel_dist = sqrt((xdist * xdist) + (ydist * ydist));

    return travel_distance(travel_dist, theta, surface);
}

/**
 * Travels up to a maximum distance in a specified direction
 * @param input_distance
 * The maximum distance that the robot will attempt to travel, in ticks
 * @param bearing
 * The compass direction to drive towards
 * @param surface
 * Surface the robot is driving on, either GRASS or CONCRETE
 * @return
 * 0 on success, number of ticks traveled if an object was encountered enroute
 */
unsigned long travel_distance(long input_distance,
                              unsigned int bearing,
                              unsigned char surface) {
    int theta;
    unsigned long traveled_distance;
    unsigned long max_distance;
    unsigned char direction;
    unsigned int current_bearing;
```



```c
    unsigned char counter;
    unsigned char rotate_direction;
    unsigned char right_speed;
    unsigned char left_speed;

    /* Figure out if the robot should move backwards or forwards */
    if (input_distance < 0) {
        direction = BACKWARD;
        max_distance = (unsigned long) ((-1) * input_distance);
    } else {
        direction = FORWARD;
        max_distance = input_distance;
    }

    /* Initially point toward the destination */
    turn_to_bearing(bearing, surface);

    /* Clear the encoders */
    ticks_fr = 0;
    ticks_fl = 0;
    ticks_br = 0;
    ticks_bl = 0;
    bumper_right_state = BUMPER_CLEAR;
    bumper_left_state = BUMPER_CLEAR;
    ENABLE_ENCODERS();
    ENABLE_BUMPERS();
    ENABLE_MOTORS();

    do {
        // Delay for 2 ms while checking encoders
        counter = 20;
        while (counter --) {
            delay_50us(2);
            traveled_distance = get_encoder_average();
            if (traveled_distance >= max_distance) break;
        }

        /* Stop if an object is encountered */
        if (traveled_distance >= max_distance) break;
        if (IR_FRONT_CM() < 20 && (direction == FORWARD)) break;
        if (((bumper_right_state != BUMPER_CLEAR) ||
            (bumper_left_state != BUMPER_CLEAR)) &&
            (direction == FORWARD)) break;

        /* Get the current direction of the robot */
        current_bearing = compass_get_bearing();

        /* Calculate which direction to turn and how much */
        theta = ((int) bearing) - ((int) current_bearing);
        if (theta < 0) theta += 3600;
        if (theta <= 1800) {
            rotate_direction = CW;
        } else {
            /* Recalculate theta in the opposite direction */
            theta = ((int) current_bearing) - ((int) bearing);
            if (theta < 0) theta += 3600;
            rotate_direction = CCW;
        }

        /* Step the speed based on how far away the location is */
        if ((max_distance <= 20) && (traveled_distance > (max_distance - 20))) {
            if (direction == FORWARD) {
                left_speed = MOTOR_MIN_GRASS_PWM;
                right_speed = MOTOR_MIN_GRASS_PWM + 20;
            } else {
                left_speed = MOTOR_MIN_GRASS_PWM + 5;
                right_speed = MOTOR_MIN_GRASS_PWM;
            }
        } else {
```

```c
            if (direction == FORWARD) {
                left_speed = 160;
                right_speed = 160 + 30;
            } else {
                left_speed = 160 + 8;
                right_speed = 160;
            }
        }

        // Adjust the speeds to maintain compass bearing
        if (theta < 10) {
            // Within compass tolerance, no need for dramatic adjustments
            update_motors(left_speed, right_speed,
                          direction, direction, 1, surface);
        } else {
            // Outside tolerance, need to correct course
            if ((rotate_direction == CW && direction == FORWARD) ||
                (rotate_direction == CCW && direction == BACKWARD)) {
                update_motors(left_speed + 30, right_speed,
                              direction, direction, 5, surface);
            } else {
                update_motors(left_speed, right_speed + 20,
                              direction, direction, 5, surface);
            }
        }
    } while (traveled_distance < max_distance);

    update_motors(0, 0, FORWARD, FORWARD, 200, surface);

    DISABLE_MOTORS();
    DISABLE_ENCODERS();
    DISABLE_BUMPERS();

    /* Update robot global position */
    if (bearing > 2700) {
        theta = 3600 - bearing;
        pos_x -= (long) (((double) traveled_distance) *
                          sin(((double) theta) * PI / 1800));
        pos_y += (long) (((double) traveled_distance) *
                          cos(((double) theta) * PI / 1800));
    } else if (bearing > 1800) {
        theta = bearing - 1800;
        pos_x -= (long) (((double) traveled_distance) *
                          sin(((double) theta) * PI / 1800));
        pos_y -= (long) (((double) traveled_distance) *
                          cos(((double) theta) * PI / 1800));
    } else if (bearing > 900) {
        theta = 1800 - bearing;
        pos_x += (long) (((double) traveled_distance) *
                          sin(((double) theta) * PI / 1800));
        pos_y -= (long) (((double) traveled_distance) *
                          cos(((double) theta) * PI / 1800));
    } else {
        theta = bearing;
        pos_x += (long) (((double) traveled_distance) *
                          sin(((double) theta) * PI / 1800));
        pos_y += (long) (((double) traveled_distance) *
                          cos(((double) theta) * PI / 1800));
    }

    #ifdef DEBUG_MODE
        //put_int_to_lcd(traveled_distance);
        put_position_to_lcd(pos_x, pos_y);
        delay_1ms(2000);
    #endif

    delay_1ms(300);
    if (traveled_distance < max_distance - 2) {
        // Didn't reach the destination
```



```c
        return traveled_distance;
    }

    return 0;
}

/**
 * Gets the average number of encoder ticks since the last time the encoders
 * were reset. Drops any one encoder value that is strongly deviant from the
 * average.
 * @return
 * Number of encoder ticks traveled
 */
unsigned long get_encoder_average(void) {
    unsigned long average = (ticks_fr + ticks_fl + ticks_br + ticks_bl) >> 2;

    if ((ticks_fr > (average + (average >> 3))) ||
        (ticks_fr < (average - (average >> 3)))) {
        return ((ticks_fl + ticks_br + ticks_bl) / 3);
    } else if ((ticks_fl > (average + (average >> 3))) ||
        (ticks_fl < (average - (average >> 3)))) {
        return ((ticks_fr + ticks_br + ticks_bl) / 3);
    } else if ((ticks_br > (average + (average >> 3))) ||
        (ticks_br < (average - (average >> 3)))) {
        return ((ticks_fr + ticks_fl + ticks_bl) / 3);
    } else if ((ticks_bl > (average + (average >> 3))) ||
        (ticks_bl < (average - (average >> 3)))) {
        return ((ticks_fr + ticks_fl + ticks_br) / 3);
    } else {
        return average;
    }
}

/**
 * Turns the robot to face a specified bearing
 * @param target_bearing
 * The compass direction to turn to
 * @param surface
 * The surface the robot is expected to be driving on, either GRASS or CONCRETE
 */
void turn_to_bearing(unsigned int target_bearing, unsigned char surface) {
    unsigned int current_bearing;
    int theta;              /* Magnitude of the angle needed to turn */
    unsigned char RD, LD;   /* Directions for each side to turn */
    unsigned char duty;

    ENABLE_MOTORS();

    /* Loop until within 1 degree of the target */
    do {
        delay_1ms(50);
        ENABLE_MOTORS();

        do {
            #ifdef DEBUG_MODE
                put_int_to_lcd(current_bearing);
            #endif

            delay_1ms(5);
            current_bearing = compass_get_bearing();

            theta = ((int) target_bearing) - ((int) current_bearing);
            if (theta < 0) theta += 3600;
            if (theta <= 1800) {
                // Need to turn clockwise
                LD = FORWARD;
                RD = BACKWARD;
            } else {
                // Need to turn counterclockwise
```

```
                // Recalculate theta in the opposite direction
                theta = ((int) current_bearing) - ((int) target_bearing);
                if (theta < 0) theta += 3600;
                LD = BACKWARD;
                RD = FORWARD;
            }

            if (surface == GRASS) {
                if (RDUTY == 0 || LDUTY == 0) {
                    update_motors(200, 200, LD, RD, 255, surface);
                } else {
                    duty = ((unsigned char)
                            (0.0555556 * ((double) theta))) + 125;
                    update_motors(duty, duty, LD, RD, 2, surface);
                }
            } else {
                if (theta > 900) {
                    update_motors(200, 200, LD, RD, 2, surface);
                } else if (theta > 450) {
                    update_motors(150, 150, LD, RD, 2, surface);
                } else if (theta > 200) {
                    update_motors(115, 115, LD, RD, 2, surface);
                } else {
                    update_motors(MOTOR_MIN_CONCRETE_PWM,
                                  MOTOR_MIN_CONCRETE_PWM, LD, RD, 2, surface);
                }
            }
        } while (theta > 20);

        update_motors(0,0,FORWARD, FORWARD, 255, surface);
        DISABLE_MOTORS();
        delay_1ms(100);

        current_bearing = compass_get_bearing();
        theta = ((int) target_bearing) - ((int) current_bearing);
        if (theta < 0) theta += 3600;
        if (theta > 1800) {
            // Recalculate theta in the opposite direction
            theta = ((int) current_bearing) - ((int) target_bearing);
            if (theta < 0) theta += 3600;
        }

    } while (theta > 20);
}

/**
 * Calculates the angle with respect to magnetic north from a relative angle
 * @param relative_bearing
 * Relative angle to convert
 * @return
 * Compass bearing with respect to magnetic north
 */
unsigned int get_angle(int relative_bearing) {
    unsigned int current_bearing = compass_get_bearing();
    relative_bearing += current_bearing;
    if (relative_bearing >= 3600) {
        return relative_bearing - 3600;
    } else if (relative_bearing < 0) {
        return relative_bearing + 3600;
    } else {
        return relative_bearing;
    }
}

/**
 * Rotates the robot to face a compass bearing and measures the distance to the
 * nearest object with the ultrasonic rangefinder
 * @param bearing
 * The compass direction to look for an object
```



```
 * @param surface
 * The surface the robot is expected to be driving on, either GRASS or CONCRETE
 * @return
 * Approximate number of encoder ticks to the object on success,
 * 0xFFFF on failure
 */
unsigned int get_distance_to_bearing(unsigned int bearing,
                                     unsigned char surface) {
    ENABLE_SERVO();

    servo_position = OFFSET_SONAR;
    turn_to_bearing(bearing, surface);
    delay_1ms(500);

    DISABLE_SERVO();
    return SONAR_TICKS();
}

/**
 * Repeatedly attemps to travel to a specific absolute location and will attempt
 * to find a path around any object encountered enroute.
 * @param targetx
 * X coordinate to go to (East)
 * @param targety
 * Y coordinate to go to (North)
 * @param surface
 * The surface the robot is expected to be driving on, either GRASS or CONCRETE
 */
void pathfind_to_location(unsigned long targetx,
                          unsigned long targety,
                          unsigned char surface) {
    unsigned long travel_return;

    while (go_to_location(targetx, targety, surface) != 0) {
        // Back up 25 clicks, shouldn't fail
        travel_return = travel_distance(-25, get_angle(0), surface);
        // Turn right, move 60 clicks
        travel_return = travel_distance(60, get_angle(-900), surface);
        if (travel_return > 0) {
            // Turn left failed
            travel_return = travel_distance((-1)*((long) travel_return),
                                            get_angle(0), surface);
            // Back at start, facing left -> try to go right instead
            travel_return = travel_distance(60, get_angle(1800), surface);
        }
    }
}
```

```c
/**
 * @file rangefinders.h
 * @brief
 *  Infrared and ultrasonic rangefinding code
 */

/** Number of ticks in 1 cm */
#define CM_TO_TICKS 0.568932
#define GRASS_CM_TO_TICKS 0.665

//0.5463
/** Potentiometer connected to PAD07 − 10 bit (unsigned int) */
#define POT_RAW_INT() ((((((unsigned int) ATD0DR0H) << 2) & 0x03FC) |
                            ((((unsigned int) ATD0DR0L) >> 6) & 0x0003)) / 2)
/** Potentiometer connected to PAD07 − 8 bit (unsigned char) */
#define POT_RAW_CHAR() ATD0DR0H

/** IR0 (longrange) connected to PAD02 − 10 bit (unsigned int) */
#define IR_FRONT_RAW_INT() ((((((unsigned int) ATD1DR0H) << 2) & 0x03FC) |
                            ((((unsigned int) ATD1DR0L) >> 6) & 0x0003)) / 2)
/** IR0 (longrange) connected to PAD02 − 8 bit (unsigned char) */
#define IR_FRONT_RAW_CHAR() ATD1DR0H

/** IR1 (shortrange) connected to PAD03 − 10 bit (unsigned int) */
#define IR_LEFT_RAW_INT() ((((((unsigned int) ATD1DR1H) << 2) & 0x03FC) |
                            ((((unsigned int) ATD1DR1L) >> 6) & 0x0003)) / 2)
/** IR1 (shortrange) connected to PAD03 − 8 bit (unsigned char) */
#define IR_LEFT_RAW_CHAR() ATD1DR1H

/** IR2 (shortrange) connected to PAD04 − 10 bit (unsigned int) */
#define IR_RIGHT_RAW_INT() ((((((unsigned int) ATD1DR2H) << 2) & 0x03FC) |
                            ((((unsigned int) ATD1DR2L) >> 6) & 0x0003)) / 2)
/** IR2 (shortrange) connected to PAD04 − 8 bit (unsigned char) */
#define IR_RIGHT_RAW_CHAR() ATD1DR2H

/** Sonar connected to PAD05 − 10 bit (unsigned int) */
#define SONAR_RAW_INT() ((((((unsigned int) ATD1DR3H) << 2) & 0x03FC) |
                            ((((unsigned int) ATD1DR3L) >> 6) & 0x0003)) / 2)
/** Sonar connected to PAD05 − 8 bit (unsigned char) */
#define SONAR_RAW_CHAR() ATD1DR3H

/** Ultrasonic reciever connected to PAD06 − 10 bit (unsigned int) */
#define ULTRASONIC_RAW_INT() ((((((unsigned int) ATD1DR4H) << 2) & 0x03FC) |
                            ((((unsigned int) ATD1DR4L) >> 6) & 0x0003)) / 2)
/** Ultrasonic reciever connected to PAD06 − 8 bit (unsigned char) */
#define ULTRASONIC_RAW_CHAR() ATD1DR4H

/** IR distance from the front bumper in cm (unsigned int) */
#define IR_FRONT_CM() ir_long_cm(IR_FRONT_RAW_INT())
/** IR distance from the front bumper in encoder ticks (unsigned int) */
#define IR_FRONT_TICKS() cm_to_ticks(IR_FRONT_CM())

/** IR distance from the left wheels in cm (unsigned int) */
#define IR_LEFT_CM() ir_short_cm(IR_LEFT_RAW_INT())
/** IR distance from the left wheels in encoder ticks (unsigned int) */
#define IR_LEFT_TICKS() cm_to_ticks(IR_LEFT_CM())

/** IR distance from the right wheels in cm (unsigned int) */
#define IR_RIGHT_CM() ir_short_cm(IR_RIGHT_RAW_INT())
/** IR distance from the right wheels in encoder ticks (unsigned int) */
#define IR_RIGHT_TICKS() cm_to_ticks(IR_RIGHT_CM())

/** Sonar distance from the front bumper in cm (unsigned int) */
#define SONAR_CM() sonar_cm(SONAR_RAW_INT())
/** Sonar distance from the front bumper in encoder ticks (unsigned int) */
#define SONAR_TICKS() cm_to_ticks(SONAR_CM())

#define ENABLE_SONAR()      PORTE&=~(BIT_2)
#define DISABLE_SONAR()     PORTE|=BIT_2
```



```c
#define CALIBRATE_SONAR()   DISABLE_SONAR();delay_1ms(500);
                            ENABLE_SONAR();delay_1ms(1000)

#define IR_ENABLED      1
#define IR_DISABLED     2
#define ENABLE_IR()     ir_state=IR_ENABLED
#define DISABLE_IR()    ir_state=IR_DISABLED

/**
 * Data structure representing a datapoint for a rangefinding device. Expressed
 * as an input analog value and the corresponding value in centimeters. Used to
 * accurately convert data with a piecewise linear map
 */
struct datapoint {
    unsigned int adcval;
    unsigned int cm;
} ;

void rangefinders_init(void);
unsigned int ir_long_cm(unsigned int ir_val);
unsigned int ir_short_cm(unsigned int ir_val);
unsigned int sonar_cm(unsigned int sonar_val);
int cm_to_ticks(int cm);

unsigned char ir_state;

/**
 * Sets up both A/D converters. The first reads the potentiometer continuously,
 * and the second reads PAD11:15 continuously in 10 bit left justified mode.
 */
void rangefinders_init(void) {
    /* Enable the relay and leave it on */
    DDRE = BIT_2;
    PORTE &= ~(BIT_2);

  /* ADC0 − Potentiometer */
    ATD0CTL2 = 0x80;    // Power on
    ATD0CTL3 = 0x08;    // 1 Conversion
    ATD0CTL4 = 0x05;    // 10 bit mode

    /*  DJM     − 0     − Left justified
        DSGN    − 0     − Unsigned data
        SCAN    − 1     − Continuous conversion
        MULT    − 0     − Sample multiple channels
                − 0     − Unused
        [2:0]   − 111   − Channel 7 − Potentiometer */
    ATD0CTL5 = 0x27;

  /* ADC1 − Sensors */
    ATD1CTL2 = 0x80;    // Power on
    ATD1CTL3 = 0x28;    // 5 Conversions
    ATD1CTL4 = 0x05;    // 10 bit mode

    /*  DJM     − 0     − Left justified
        DSGN    − 0     − Unsigned data
        SCAN    − 1     − Continuous conversion
        MULT    − 1     − Sample multiple channels
                − 0     − Unused
        [2:0]   − 011   − Start at channel 3 */
    ATD1CTL5 = 0x33;
}

/**
 * Converts a 10 bit analog value from the longrange IR sensor to a distance in
 * cm using a piecewise linear map
 * @param ir_val
 * 10 bit ADC value from the longrange IR sensor
 * @return
 * 0xFFFF if nothing is in range, distance in cm rounded down otherwise */
```

```
unsigned int ir_long_cm(unsigned int ir_val) {
    unsigned char i;
    struct datapoint data[5] = {29, 103,
                                57, 164,
                                85, 194,
                                134, 217,
                                268, 245};

    if (ir_state == IR_DISABLED) return 0xFFFF;

    // i < number of data points in array - 1
    for (i = 0; i < 4 ; i ++) {
        if ((ir_val >= data[i].adcval) && (ir_val < data[i+1].adcval)) {
            return 255 - (unsigned int)(((((float)(data[i+1].cm - data[i].cm)) /
                    ((float)(data[i+1].adcval -
                    data[i].adcval))) * ((float)(ir_val - data[i].adcval))) +
                    ((float)(data[i].cm)));
        }
    }

    return 0xFFFF;
}

/**
 * Converts a 10 bit analog value from etiher shortrange IR sensor to a distance
 * in cm using a piecewise linear map
 * @param ir_val
 * 10 bit ADC value from either shortrange IR sensor
 * @return
 * 0xFFFF if nothing is in range, distance in cm rounded down otherwise */
unsigned int ir_short_cm(unsigned int ir_val) {
    unsigned char i;
    struct datapoint data[6] = {32, 164,
                                49, 202,
                                77, 225,
                                127, 240,
                                190, 247,
                                322, 255};

    if (ir_state == IR_DISABLED) return 0xFFFF;

    // i < number of data points in array - 1
    for (i = 0; i < 5 ; i ++) {
        if ((ir_val >= data[i].adcval) && (ir_val < data[i+1].adcval)) {
            return 255 - (unsigned int)(((((float)(data[i+1].cm - data[i].cm)) /
                    ((float)(data[i+1].adcval -
                    data[i].adcval))) * ((float)(ir_val - data[i].adcval))) +
                    ((float)(data[i].cm)));
        }
    }

    return 0xFFFF;
}

/**
 * Converts a 10 bit analog value from the ultrasonic rangefinder to a distance
 * in cm
 * @param sonar_val
 * 10 bit ADC value from the ultrasonic rangefinder
 * @return
 * 0xFFFF if nothing is in range, distance in cm rounded down otherwise */
unsigned int sonar_cm(unsigned int sonar_val) {
    if (sonar_val > 180) {
        return 0xFFFF;
    }
    return ((unsigned int) (((float) (sonar_val - 1)) * 2.54));
}

/**
```



```
 * Converts a centimeter value to a value in ticks on concrete
 * @param cm
 * Centimeter value to convert
 * @return
 * Distance in concrete encoder ticks */
int cm_to_ticks(int cm) {
    return ((int) (((double) cm) * CM_TO_TICKS));
}

/**
 * Converts a centimeter value to a value in ticks on grass
 * @param cm
 * Centimeter value to convert
 * @return
 * Distance in grass encoder ticks */
int grass_cm_to_ticks(int cm) {
    return ((int) (((double) cm) * GRASS_CM_TO_TICKS));
}
```

```c
/**
 * @file search.h
 * @brief
 * Search algorithms
 */

unsigned int temperature_search(unsigned int min_deviation);
unsigned int ultrasonic_search(unsigned int min_val);
unsigned int sonar_search(unsigned int max_dist);
unsigned int get_bearing_from_servo(char servo_val, char sensor_offset);
unsigned char use_sensor(unsigned int (*sensor_func)(unsigned int),
                         unsigned int sensor_param, unsigned char surface);

/**
 * Locates the coldest object in range
 * @param min_deviation
 * Minimum difference (in 0.02 degree C increments) that the lowest temperature
 * must be from the average to be considered a good match
 *
 * @return
 * 0xFFFF on failure, compass bearing on success
 */
unsigned int temperature_search(unsigned int min_deviation) {
    unsigned int temp;
    unsigned long total_temp = 0;
    unsigned int min_temp = 0;
    char min_pos = 0;

    ENABLE_SERVO();
    servo_position = -128;
    delay_1ms(1500);

    while (servo_position < 127) {
        temp = temp_read(0x07);
        if (((temp > 0) && (temp < min_temp)) || (min_temp == 0)) {
            min_temp = temp;
            min_pos = servo_position;
        }
        #ifdef DEBUG_MODE
            put_temp_to_lcd(temp);
        #endif
        total_temp += temp;
        delay_1ms(100);
        servo_position ++;
    }

    servo_position = 0;
    delay_1ms(1500);
    DISABLE_SERVO();

    /* Disregard the min temp if it is too close to the average temperature */
    if (min_temp > ((total_temp / 255) - min_deviation)) return 0xFFFF;

    #ifdef DEBUG_MODE
        ENABLE_SERVO();
        servo_position = min_pos;
        delay_1ms(1500);
        success_action_location();
        servo_position = 0;
        delay_1ms(1500);
        DISABLE_SERVO();
    #endif

    return get_bearing_from_servo(min_pos, OFFSET_TEMP);
}

/**
 * Locates the strongest direction of ultrasonic radiation
 * @param min_val
```



```c
 * Minimum 10 bit ADC value that could be considered a worthwhile signal
 *
 * @return
 * 0xFFFF on failure, compass bearing on success
 */
unsigned int ultrasonic_search(unsigned int min_val) {
    unsigned int magnitude;
    unsigned int max_magnitude = 0;
    char max_pos = 0;

    ENABLE_SERVO();
    servo_position = -128;
    delay_1ms(1500);

    while (servo_position < 127) {
        magnitude = ULTRASONIC_RAW_INT();
        if (magnitude > max_magnitude) {
            max_magnitude = magnitude;
            max_pos = servo_position;
        }
        #ifdef DEBUG_MODE
            put_int_to_lcd(magnitude);
        #endif
        delay_1ms(40);
        servo_position ++;
    }

    servo_position = 0;
    delay_1ms(1500);
    DISABLE_SERVO();

    /* Disregard any magnitude less than the cutoff value */
    if (max_magnitude < min_val) return 0xFFFF;

    #ifdef DEBUG_MODE
        ENABLE_SERVO();
        servo_position = max_pos;
        delay_1ms(1500);
        success_action_location();
        servo_position = 0;
        delay_1ms(1500);
        DISABLE_SERVO();
    #endif

    return get_bearing_from_servo(max_pos, OFFSET_ULTRASONIC);
}

/**
 * Locates the closest object within range
 * @param max_dist
 * Maximum distance (in cm) to consider an object. Objects further than this
 * will be ignored
 *
 * @return
 * 0xFFFF on failure, compass bearing on success
 */
unsigned int sonar_search(unsigned int max_dist) {
    unsigned int dist;
    unsigned int min_dist = 0;
    char min_pos = 0;

    ENABLE_SERVO();
    servo_position = -128;
    delay_1ms(1500);

    while (servo_position < 127) {
        dist = SONAR_RAW_INT();
        if (((dist > 0) && (dist < min_dist)) || (min_dist == 0)) {
            min_dist = dist;
```

```
            min_pos = servo_position;
        }
        delay_1ms(30);
        servo_position ++;
    }

    sonar_temp = min_dist;
    servo_position = 0;
    delay_1ms(1500);
    DISABLE_SERVO();

    /* Disregard any object further than the specified max distance */
    if (sonar_cm(min_dist) > max_dist) return 0xFFFF;

    ENABLE_SERVO();
    servo_position = min_pos;
    delay_1ms(1500);
    success_action_location();
    servo_position = 0;
    delay_1ms(1500);
    DISABLE_SERVO();

    return get_bearing_from_servo(min_pos, OFFSET_SONAR);
}

/**
 * Converts a servo angle to an absolute compass bearing
 * @param servo_val
 * Servo position to convert
 * @param sensor_offset
 * Servo value that points the particular sensor straight forward
 *
 * @return
 * Compass bearing towards the servo position
 */
unsigned int get_bearing_from_servo(char servo_val, char sensor_offset) {
    int target_bearing;

    target_bearing = ((int) compass_get_bearing()) +
                     (((((int) servo_val) - ((int) sensor_offset)) * 92) / 13);
    if (target_bearing >= 3600) {
        return (unsigned int) (target_bearing - 3600);
    } else if (target_bearing < 0) {
        return (unsigned int) (target_bearing + 3600);
    } else {
        return (unsigned int) target_bearing;
    }
}

/**
 * Attempts to find an object of interest using a specific sensor function
 * Function first scans the area with the sensor; if something is found, it
 * rotates the robot to face the object. Next, the sonar sensor is used to
 * attempt to determine how far away the object is. The robot will then attempt
 * to approach the object up to that distance, and will signal success if it
 * enconters the waypoint. This function will retry up to three times, and will
 * stop at any point if no object of interest is detected.
 *
 * @param sensor_func
 * Function pointer to the sensor_search function to use. Can be
 * temperature_search, ultrasonic_search, or sonar_search
 * @param sensor_param
 * Function specific parameter to pass to the sensor_func
 * @param surface
 * Surface the robot is driving on, either GRASS or CONCRETE
 * @return
 * 1 if the waypoint was found, 0 on failure
 */
unsigned char use_sensor(unsigned int (*sensor_func)(unsigned int),
```



```
                        unsigned int sensor_param, unsigned char surface) {
    unsigned int bearing;
    unsigned int dist;
    unsigned long travel_return;
    unsigned char loop_counter;
    unsigned long dummy;

    // Search up to three times with the specific sensor
    for (loop_counter = 0; loop_counter < 3; loop_counter ++) {
        bearing = sensor_func(sensor_param);
        if (bearing == 0xFFFF) {
            return 0; // Exit the function if an object was not found
        }
        // Get distance to object
        dist = get_distance_to_bearing(bearing, surface);
        if (dist < 80) {
            success_action_object();
            dummy = travel_distance(-25, get_angle(0), surface);
            return 1;
        } else {
            if (dist == 0xFFFF) { // Check if distance was found successfully
                // Could not determine distance ->
                // set distance to a small fixed value
                dist = 30;
            } else if (dist >= 300) {
                dist = 300;
            }
            // Approach the object
            travel_return = travel_distance(dist, bearing, surface);
            if (travel_return != 0) {
                // Reached an object while traveling, probably the waypoint
                success_action_object();
                dummy = travel_distance(-25, get_angle(0), surface);
                return 1;
            }
        }
    }return 0;
}
```

```c
/**
 * @file temp.h
 * @brief
 *  MLX90614 temperature sensor code
 */

#define ADDRESS_TEMP 0xB4

unsigned char PEC_cal(unsigned char pec[]);
unsigned int temp_read(unsigned char address);
float temp_to_c(unsigned int raw_temp);
float temp_to_f(unsigned int raw_temp);
void put_temp_to_lcd(unsigned int raw_temp);

/**
 * Calculates the packet error code, based on the sample code supplied at:
 * http://www.melexis.com/Sensor-ICs-Infrared-and-Optical/
 * Infrared-Thermometers/MLX90614-615.aspx
 *
 * @todo figure out how this function works
 */
unsigned char PEC_cal(unsigned char pec[]) {
    unsigned char crc[6];
    unsigned char Bitposition=47;
    unsigned char shift;
    unsigned char i;
    unsigned char j;
    unsigned char temp;

    do {
        crc[5] = 0;             // Load CRC value 0x000000000107
        crc[4] = 0;
        crc[3] = 0;
        crc[2] = 0;
        crc[1] = 0x01;
        crc[0] = 0x07;
        Bitposition = 47;       // Set maximum bit position at 47
        shift = 0;

        // Find first 1 in the transmitted bytes
        i=5;                    // Set highest index (package byte index)
        j=0;                    // Byte bit index, from lowest
        while ((pec[i] & (0x80 >> j)) == 0 && (i > 0)) {
            Bitposition --;
            if (j < 7) {
                j ++;
            } else {
               j = 0x00;
               i --;
            }
        } // End of while, and the position of highest "1" bit is in Bitposition

        // Shift CRC value left with "shift" bits
        shift = Bitposition - 8;    // Get shift value for CRC value
        while (shift) {
            for (i = 5; i < 0xFF; i --) {
                // Check if the MSB of the byte lower is "1"
                // So that "1" can shift between bytes
                if ((crc[i - 1] & 0x80) && (i > 0)) {
                    // Yes - current byte + 1
                    temp = 1;
                } else {
                    // No - current byte + 0
                    temp = 0;
                }
                crc[i] <<= 1;
                crc[i] += temp;
            }
            shift --;
```



```c
        }

        // Exclusive OR between pec and crc
        for(i = 0; i <= 5; i ++) {
            pec[i] ^= crc[i];
        }
    } while (Bitposition > 8);

    return pec[0];
}

/**
 * Reads the specified memory address from the MLX90614 temperature sensor
 *
 * @param address
 * Address to read, can be one of the following:
 * 0x06 - TA (Ambient temperature sensor)
 * 0x07 - TOBJ1 (First IR temperature sensor)
 * 0x08 - TOBJ2 (Second IR temperature sensor)
 *
 * @return
 * Success - Value returned by the MLX90614
 * Failure - 0
 */
unsigned int temp_read(unsigned char address) {
    unsigned int temp;

    if (iic_start(ADDRESS_TEMP) == 0) {     // Connect, write mode
        if (iic_transmit(address) == 0) {   // Send command
            IBCR |= BIT_2;                  // Repeated start condition
            IBDR = ADDRESS_TEMP | BIT_0;    // Connect, read mode
            if (iic_response() == 0) {      // Wait for an ACK
                iic_swrcv();                // Switch to recieve mode
                temp = iic_recieve() & 0x00FF;                          // LSB
                temp = temp | ((iic_recieve_last() << 8) & 0xFF00);     // MSB

                return temp;
            }
        }
    }

    return 0;
}

/**
 * Converts a raw temperature reading to a celsius value
 *
 * @param raw_temp
 * Raw temperature hex value to convert
 * @return
 * Floating point temperature in celsius
 */
float temp_to_c(unsigned int raw_temp) {
    return (((float) (raw_temp - 0x27AD)) * 0.02 - 70.01);
}

/**
 * Converts a raw temperature reading to a fahrenheit value
 *
 * @param raw_temp
 * Raw temperature hex value to convert
 * @return
 * Floating point temperature in fahrenheit */
float temp_to_f(unsigned int raw_temp) {
    return (((float) (raw_temp - 0x27AD)) * 0.02 - 70.01) * 9.0 / 5.0 + 32.0;
}

/**
 * Displays a temperature value in fahrenheit from the MLX90614 on the
```

```
 * LCD display
 *
 * @param raw_temp
 * Raw temperature value from the IIC interface
 */
void put_temp_to_lcd(unsigned int raw_temp) {
    // Lookup table of string characters for conversion
    const char num_2_string[] = { '0', '1', '2', '3', '4', '5',
                                  '6', '7', '8', '9', 'A', 'B',
                                  'C', 'D', 'E', 'F' };
    static unsigned int last_value = 0;
    unsigned char msg1[11];             // Raw hex message
    unsigned char msg2[11];             // BCD message

    // Only update the display if the value has changed
    if (last_value != raw_temp) {
        last_value = raw_temp;

        // Create null terminated strings from the lookup table
        msg1[0] = 'H';
        msg1[1] = 'E';
        msg1[2] = 'X';
        msg1[3] = ' ';
        msg1[4] = '-';
        msg1[5] = ' ';
        msg1[6] = num_2_string[(raw_temp >> 12)  & 0x0F];
        msg1[7] = num_2_string[(raw_temp >> 8)   & 0x0F];
        msg1[8] = num_2_string[(raw_temp >> 4)   & 0x0F];
        msg1[9] = num_2_string[(raw_temp)        & 0x0F];
        msg1[10] = '\0';

        ftoa(msg2, temp_to_f(raw_temp));

        set_lcd(msg1, msg2);
    }
}
```