CS 430: Senior Project
Robotics

Students
Nathan Woods, Jennings Anderson
Shae Tiegen, Forrest Laskowski

Professor
Steve Harper

Term
Spring 2012

Abstract

"Robo"

Carroll College 2012
Mathematics, Computer Science, & Engineering

Contents

1 Introduction

1.1 Overview

After procuring robots from MSU's robotic department, we familiarized ourselves with the IntelliBrain2 Micro Controller from Ridgesoft mounted atop the Ridgesoft Deluxe Educational Robot:

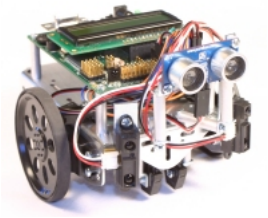


Figure 1: IntelliBrain2 Micro Controller on the standard Ridgesoft base

This small robot allowed us to learn the basics before moving onto a larger frame. This robot ran with two independent continuous rotation servos (controlling each wheel independently), a range finding sensor, and two light sensors pointed downwards to help it follow a line.

Next, we moved onto a larger, faster robot mounted on an Axial Remote Control Rock Crawling car base:

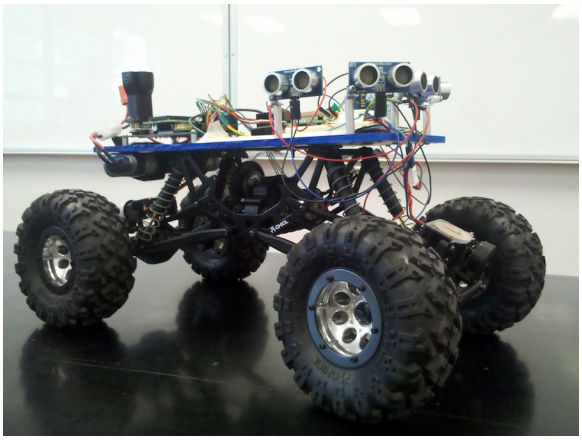


Figure 2: Axial RC car mounted with IntelliBrain2 Controller

We modified the base to include 4 wheel steering. We then mounted a GPS, 5 range finding sensors and a single WheelWatcher Incremental Quadrature Encoder for navigation and speed control.

1.2 This Documentation

The purpose of this documentation is to explain the process and steps we took to learn, build, and program our self-navigating robot. We will introduce the development environment, discuss the specific functions of the robot and describe the running process of the robot, as the end user would see.

1.3 Project Goals

1. Grasp the concepts of introductory robotics.
2. Learn to translate a theoretical environment to a running physical machine.
3. Enhance our understanding of the Java Language.
4. Learn about external Java libraries.
5. Understand how Java can be compiled to a specific platform and handle physical sensor input.
6. Learn advanced robotic algorithms - separate from theoretical Computer Science algorithms.

2 Development Environment

The IntelliBrain2 can be programmed in Java. A specific RoboJDE library is required at compile to interface with the board. There are numerous options to turn Java code into running code on the robot that we will explain here.

2.1 Requirements

First, download the development software from the Ridgesoft Website:

http://ridgesoft.com/robojde/robojde.htm

The software, API, required libraries, and physical wiring diagrams can be found on this website.

There is an included development application, *RoboJDE*, that uses RoboJDE project files $(.rjp)$ to program the robot. This editor is great for getting started, but is quickly outgrown. The software comes with many example files and as long as the host computer has a Serial Port (We used an OptiPlex GX620), then there is no additional software required to get the first programs loaded to the robot.

This environment, however, is all plain text with no syntax coloring or error detection. Moreover, We found Eclipse to be the best editor because it can be configured to automatically compile to the robot using the rsload script.

2.2 RSLoad Script

Included in downloads for all platforms is a script called rsload. rsload is a command line utility for loading code to the robot. The best documentation on it can be found here:

http://ridgesoft.com/robojde/2.0/docs/RoboJDEGuide.pdf

Some important arguments to know for rsload are:

-**port** Specifies the port the robot is connected to, typically COM1 on Windows or /dev/ttys0 on Linux, but if using a USB adapter, this could be /dev/tty.usbserial or COM5. Understanding particular drivers and the OS is important here.

-**run** If included, the robot will automatically run when load is complete.

-**bank** Specify to load to either Flash or Ram; we always loaded to Flash and in some cases, it was required to use this argument.

-**verbose** Good for debugging purposes.

The ultimate required argument is the name of the class that includes main. An example call:

```
rsload -bank Flash -port /dev/tty.usbserial -run Controller
```

This command will load the Controller class to the robot over the USB to Serial adapter. When load is complete, it will run the program. It should be noted that in this example, Controller.class is located in the same directory as rsload.

2.3 Individual Load Scripts

For greater compatibility, we wrote a customizable load script that would compile and then load a specific file, given the file path and the name of the file containing the function, $main$.

```
#Instructions:
#Update both filePath and fileName variables to represent where the files are.
#These should all be relative to Dropbox.  (Note in eclipse cases, compiled files
#may be found in the /bin/ directory of the project.  However, these should be
#pointed to the java file for compiling.
#Don't put a '/' on the end of the filePath:
#The fileName is the main controlling class.
#Warning! Every .java file in the path BETTER compile.
#This is an example:
#filePath=/home/robotics/Dropbox/Robo/Development/Forrest
#fileName=MusicTest
################################################################
####################### WHAT YOU SHOULD EDIT ###################
filePath=/home/robotics/Dropbox/Robo/Development/Master/Robot/src
#Note, don't end with /
fileName=Controller #Note, no .java or .class! This has 'main function'
######################## END WHAT YOU SHOULD EDIT ###############
################################################################
clear
echo "Welcome to the CS 430 Load Script"
echo "This will fail if RoboJDE is running"
echo ""
echo "Copying to Temp Build Path to not upset Eclipse..."
rm -f /home/robotics/Dropbox/Robo/Development/TempLoadNoTouchie/*
cp $filePath/* /home/robotics/Dropbox/Robo/Development/TempLoadNoTouchie/
```

```
echo "Files Copied To Temp Build Path:"
cd /home/robotics/Dropbox/Robo/Development/TempLoadNoTouchie
ls -al
echo "Press Enter to Compile"
read nothing
#Let's compile it first:
echo "Compiling..."
javac -classpath "/home/robotics/RoboJDE/RoboJDE.jar" *.java
echo "Compilation Complete"
echo "Make sure Robot is in Bootstrap Mode, Turn ON then press STOP,
                                                    Then Press Enter"
read nothing
echo "Initiating Robot Loader:"
sh /home/robotics/RoboJDE/rsload -port /dev/ttyS0 -bank flash $fileName
echo ""
echo "Press Enter to Quit"
read nothing
```

This script was written for the robotics user account on an OptiPlex GX620 running Ubuntu 11.10 with the robot connected to the serial port. It will copy the required files to a temporary directory, compile them and then load them to the robot.

The main purpose of a script like this is to abstract the robot development to simple java files in a single directory. The script takes care of including the necessary files for interfacing with the board, so the user can write simple Java code to feed this script.

Next, we take this concept one step further by integrating with Eclipse.

2.4 Integrating with Eclipse

Eclipse is a robust open source development environment that allows for much customization. The two requirements for this project are:

1. Include the RoboJDE.jar archive in the compile path
2. Write an external build tool to automatically call the rsload script.

Once a project is created in Eclipse, preferences can be set by right-clicking on the project root. Ensure here that the Java Build Path includes the RoboJDE.jar file. It is convenient to put a copy of this file in the workspace directory so that it can be referenced locally.

There are explicit instructions for setting up an eclipse external tool in the read me, but I have included a screen shot here for further description:

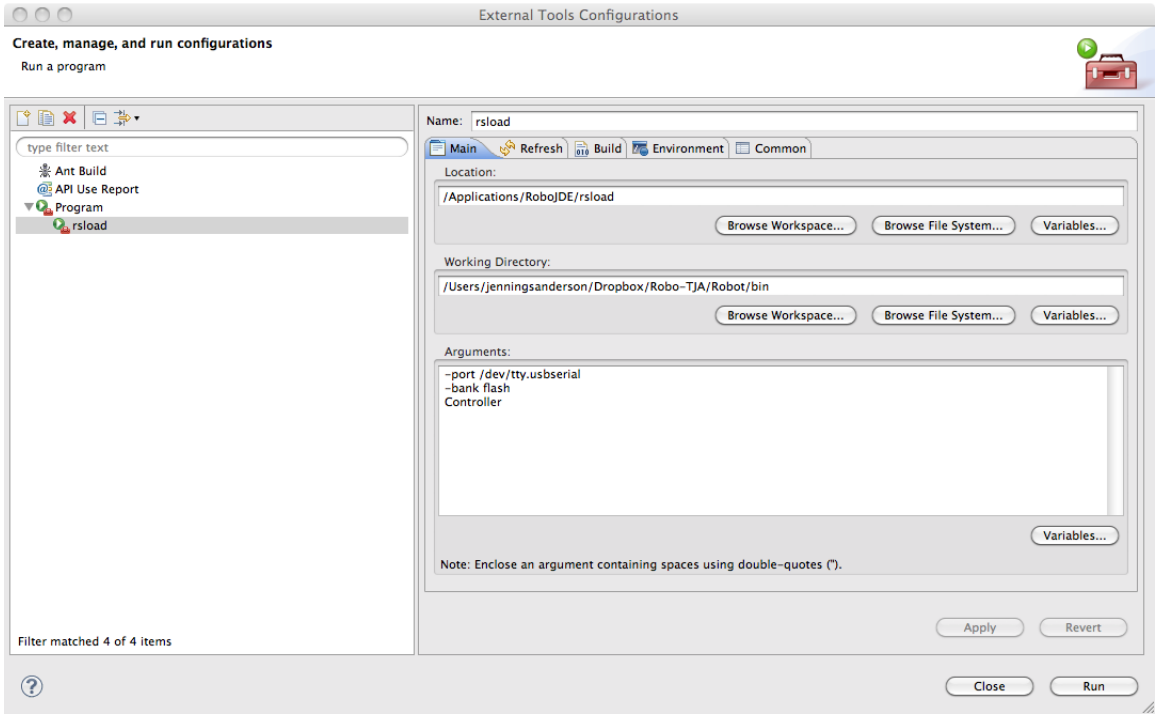


Figure 3: Eclipse External Tool Configuration

This tool configuration differs from the suggested configuration with static references for convenience. Note that the Working Directory references the bin directory, where Eclipse puts the compiled class files. The name of the class with the function, $main$ in it is $Controller$.

2.5 Source Version Control

The importance of a steadfast source version control method should not be over looked; unfortunately, we did so for months until turning to git and github.com. We recommend to spend the time in the beginning familiarizing yourself with the processes of branching, forking, and merging with the github interface.

We found an open source utility, EGit installed through the Eclipse market place, that allowed us to integrate the Eclipse environment with github. It should be known and user be warned that github integration with Eclipse is very powerful and will overwrite the local workspace, so one must be comfortable with the github commands and Egit interface before fetching or pushing from the remote repository.

Originally, we used Dropbox, which worked as a great backup tool at the end of the day, but hard for real-time development. This is because Dropbox is not true version control software.

SVC Anecdote on Dropbox

While Dropbox is a simple, convenient, and magical application in today's computing, it is not a substitute for source version control. We initially used Dropbox to sync files across all of our computers for team development. The problem here is that changes propagate immediately. Backups are kept in the depths of Dropbox's servers, but are tedious to scan through and recover. However, the real kicker occurs when Dropbox meets DeepFreeze, the Enterprise software that Carroll uses to fight malicious intentions. If Dropbox is installed on a user's profile, it will by default sync all files to a folder on the local hard drive. When the computer restarts, changes to the C drive are erased and Dropbox is fooled into believing the user deleted all files in their Dropbox directory. This change is then propagated to the rest of the team, so come Monday morning, all code is erased across every team member's computer. *Thanks, Forrest*.

3 Physical Robot Design

Here we describe the physical attributes of the robot, after introducing each part, the next section will cover the logic that governs each physical interface.

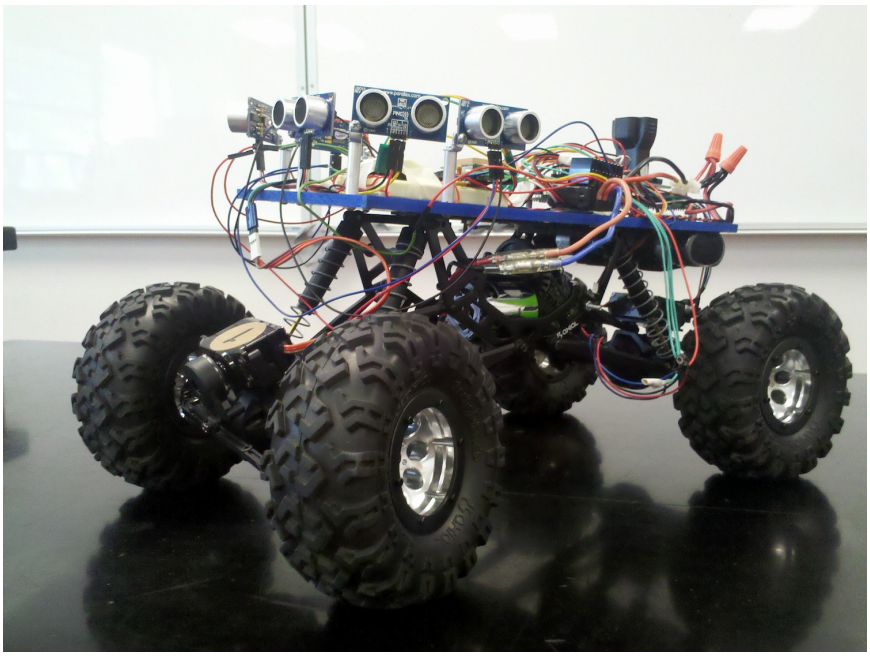
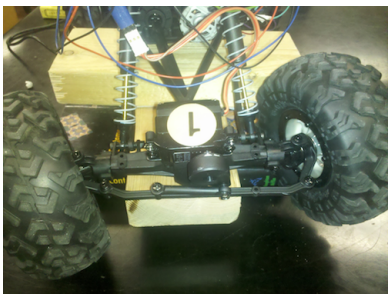


Figure 4: Final Robot Design

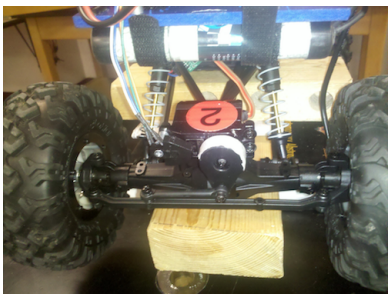
3.1 Description

The Robot is about 18" long, 10" tall, and 11" wide. It has 4 wheels of 5" diameter. and runs off a 3000 mAh Ni-MH battery. The RC car body is designed by Axial Racing (www.axialracing.com).

Borrowing the parts from a second RC car, we gave our robot 4-wheel steering by substituting the rear axle with a second car's front steering axle.



(a) Front Steering



(b) Rear Steering

A central motor, controlled by an Axial AE-2 ESC drives all four wheels. This is mounted

in the center of the body and is directly linked to the WheelWatcher chip which records how fast the motor is spinning.

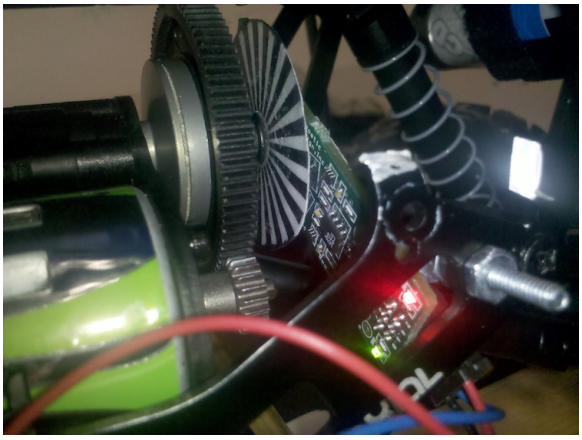


Figure 5: Nubotics' WheelWatcher Incremental Quadrature Encoder mounted next to motor

As the motor turns, the black and white disk spins over light sensors on the WheelWatcher chip. With each change in color, a counter is incremented. This counter can then be converted to rotations per minute, or RPM.

On the top of the robot are 5 range finding sensors that operate by sonar:

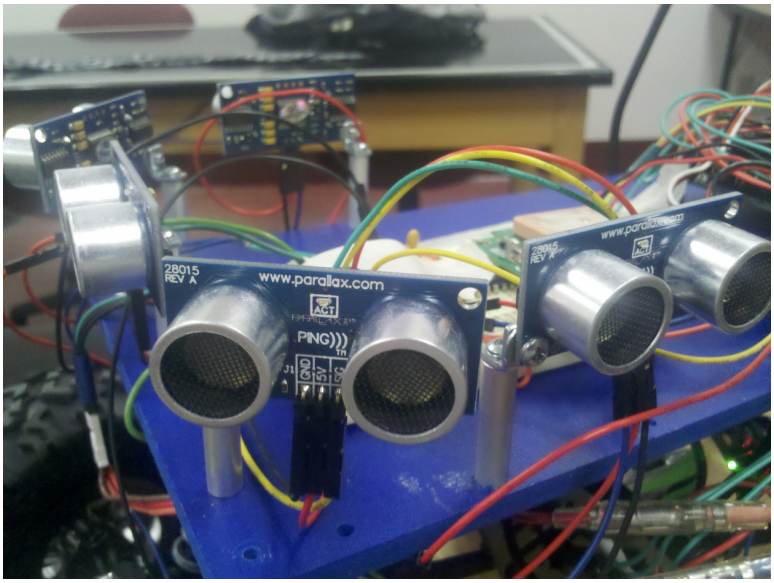


Figure 6: Ping))) Sensors

These 5 sensors point directly ahead, 45° left, right, and 90° left, right. They operate via sonar and require special attention to timing as the command tells it when to send a 'ping' and when to read the ping. If not a enough or too much time elapses between the send and read command, the data is corrupted.

Ultimately, the main controlling board is the IntelliBrain2 by Ridgesoft:

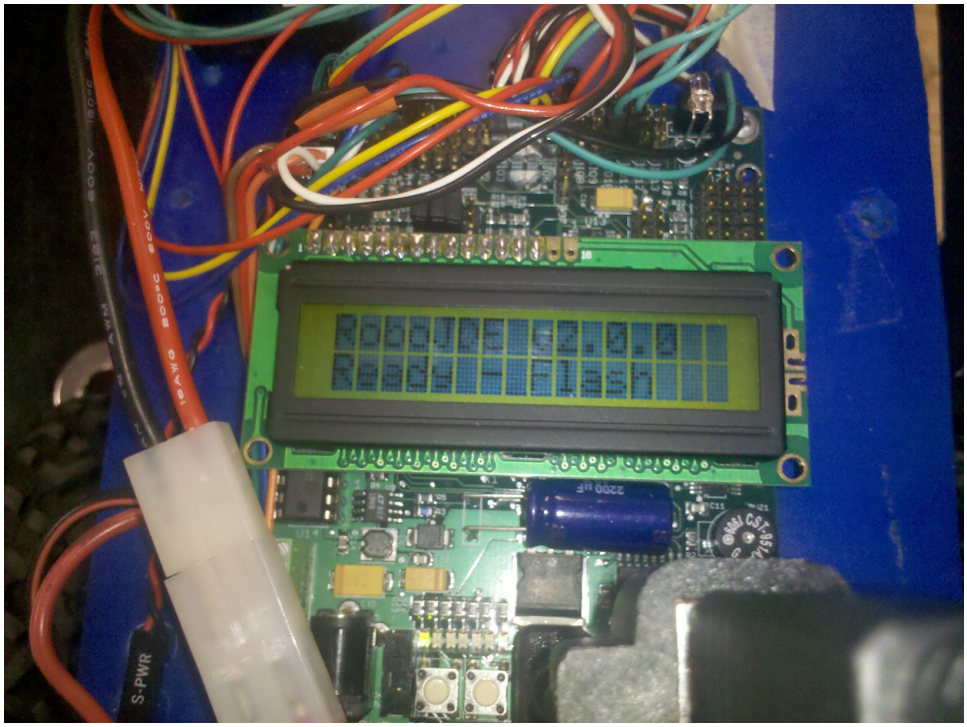


Figure 7: The 14.7 MHz IntelliBrain2 MicroController

This is a Java programmable board that calls a specific API to interface with the physical inputs. It connects to a computer via a serial port. In this figure, we have a USB adapter plugged into the serial port.

4 Robot Logic

There are 2 main obstacles that we overcame in this project:

1. Difference between Theory and Application
2. CPU Clock Speed

The difference between theory and application deals with the inconsistencies in a physical environment. On the white board and in the computer, issuing commands such as *turnRight(35°)* then *goForward(10 inches)* would move the robot 10 inches at 35°from the current location.

In a physical environment, this is never the case, especially when working with a remote control vehicle. The driving surface may be inconsistent, making the wheels only turn 30°and the battery may be low, so the motor may not drive at the proper speed, making the car only go 9 inches. A series of this mistakes and the robot will lose its heading completely.

Similarly, if a range finding sensor misfires and returns false data, decision making processes are misinformed. On the white board, a command such as *getDistance()* can be trusted, but in reality, these functions must be checked and double checked for errors before they can be trusted by the robot.

In previous Computer Science courses, we were never limited by the speed of the CPU. Sure it may take time to compute *factorial(16984)*, but the impact of this time has never had high consequences.

For this robot, we have become very aware of exactly how long it takes the CPU to complete a cycle. This has two ramifications:

1. How long will a set of commands tie up the CPU? *If it takes too long to interpret sensor data, then the robot may hit the wall before it realizes the wall is there.*
2. How fast is the CPU? *If a particular sensor is running substantially faster than the CPU, can the robot even interpret that data?*[1]

We have to consistently work to overcome the difference between theory and application in robotics, but seem to have worked out a few tricks to function thus far. In terms of physical CPU limitations, we separated sensors and logic into various threads with specific priorities.

[1]See the section on Remote Control

4.1 Threads

The IntelliBrain2 MicroController has the ability to spawn multiple process threads. As such, we have separated our robots sensory inputs and logic into different threads to optimize driving abilities. These threads (in order of priority) are:

Remote Handles checking for the remote to turn on (wireless kill switch)

Sonar Controls the 5 range finding sensors on the front.

Brain Governs all decision making

Engine Controls the motor servo.

Steering Controls the front and rear steering servos.

GPS Controls the Global Positioning System chip that is mounted on the top of the robot.

Debugging Handles debugging data if requested - has minimum priority.

This design is based off the general principle of ranking importance as: *1) Sensory Input 2) Decision-Making 3) Motor Control* (Note, the GPS is excluded in this because it is rarely used).

4.1.1 Challenges

4.2 Sonar & Range Finding

The Robot has 5 range finding sensors that operate via sonar. That is, they emit a sound wave and measure the time it takes for the wave deflect off an object and return. Since the speed of sound is constant, the distance it traveled in the measured time can be determined. In order to get 5 sensors active, a relay had to be added to the circuit to overload the digital input ports on the IntelliBrain2.

As a result, when sonar is running, there is a ticking noise as the relay switches back and forth. The logical obstacles we overcame with the sonar sensors was timing. There is a window of time between when the sensor has received the signal and the sensor trashes the signal. If you ask for the distance too soon after telling the sensor to *ping()* (emit a sound wave), there will be no data to receive. Too long and the data is no longer valid or trashed by the sensor.

Here is a diagram of the sensor we used and its function:[2]

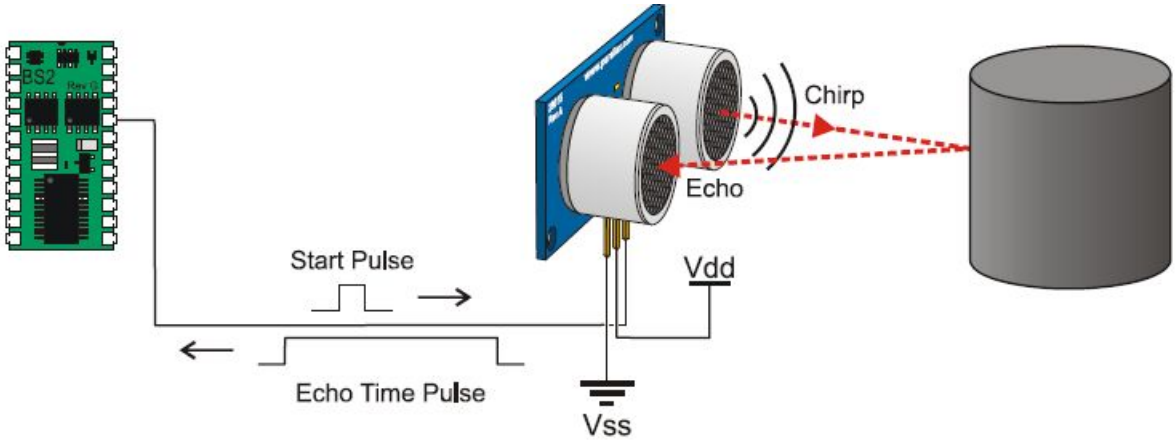


Figure 8: Parallax Ping))) Sensor

The timing issue made us shy away from using Threads at first because we could not control the timing very accurately if the processor was determining which processes got to run when; however, we ultimately decided to run the sensors iteratively as a single thread with hard coded 200 millisecond delays between thg ping and the retrieval of information from the sensor.

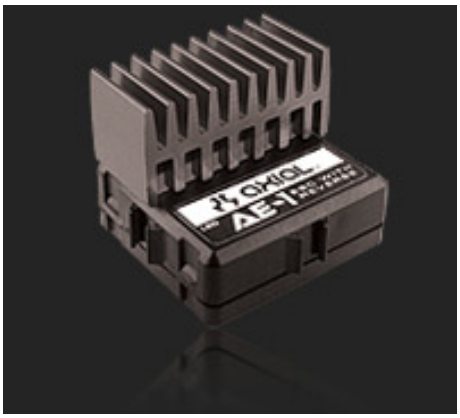
The two side sensors (East and West) are connected to the relay and take turns as active distance sensors.

[2]Image from http://www.generationrobots.com/site/medias/Principe-fonctionnement-Sonar-Ping-Parallax.JPG

4.3 Engine & Motor

One of the more difficult parts of the robot to program, the motor, an Axial 27T electric motor is controlled by the Axial AE-1 ESC. These are the two parts:[3]

(a) 27T Electronic Motor



(b) Axial AE-1 ESC

The AE-1 is connected to one of the MicroController's servo ports and we initialize it as a *continuous rotation servo*. This means that it takes integer arguments for the command *setPower(int n)* between -16 and 16 (negative numbers representing reverse).

Additionally, active braking is available, so if the controller gets a command for the opposite direction, it will brake and bring the motor to a stop.

The motor control is the ultimate showcase of theoretical versus physical programming. The values passed to *setPower()* never represent the same speed. There are too many factors: battery level, how long its been running, what speed it was going, driving surface, etc.

To remedy this, we wrote a separate driver called Engine that uses the Nubotic Wheel Watcher chip[4] to count the rotations of the motor. This is calculated to the number of rotations-per-minute (RPM) so that we can receive active feedback on the current wheel speed.

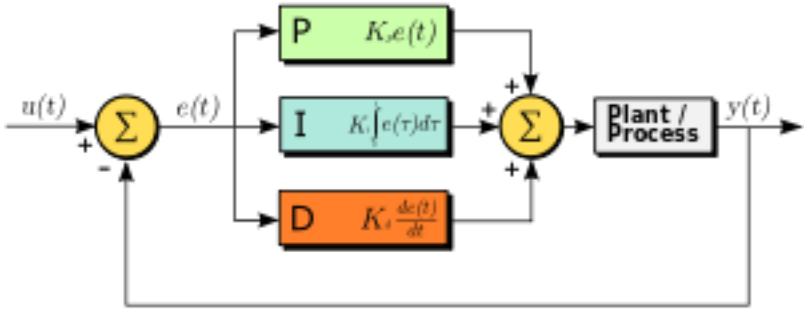
From here, we can attempt the general motor control logic:

1. Set a desired velocity; if not 0, set the motor power to a specific range that somewhat corresponds to that velocity.
2. Get the current RPM of the motor
3. If the RPMs are higher than desired, decrease motor power.
4. If the RPMS are too low, increase motor power if its not already at maximum level.
5. Go To Step 2.

[3]Images from: http://axialracing.com/ftp/ax10rtr/
[4]Image in previous section

To implement this motor logic, we used a proportional-integral-derivative (PID) Controller.

The PID is the most commonly used feedback controller. It works to minimize error by adjusting input variables.[5]



[5]Image and Description from Wikipedia: http://en.wikipedia.org/wiki/PID_controller

4.4 Remote Control

In a major addition to the robot, we incorporated the original RC car remote to override the robot logic for motor control. This was done to serve two purposes: 1) act as a kill switch and 2) provide the ability to manually drive the robot to train a Neural Network. We played with this idea early on, but abandoned it due to difficulty in obtaining clean data from the remote.

Upon revisiting the idea, we determined the original problem was a limitation in the physical clock speed of the Micro Controller. The IntelliBrain2 has a clock speed of 14.7 MHz. The frequency of the remote is 27 MHz. Therefore, no matter how many consecutive clock cycles we gave the remote sensor, it was incapable of accurately sampling the change in signal.

To remedy this issue, our team member pursuing a Physics degree created an RC circuit (a type of Low-Pass filter) for the remote sensor that cleaned up the signal and allows for meaningful sampling of the analog input.

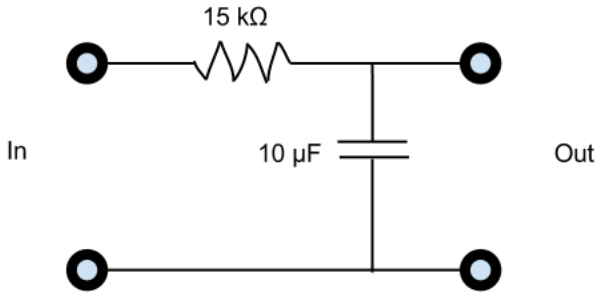


Figure 9: Nate's Low-Pass Filter Circuit Diagram

We should note that this current circuit is an improvement on the first attempt which utilized much larger capacitors that *featured* a multiple second delay on steering and throttle input.

Remote Input

After averaging the 3 sample points (for consistency), we can obtain nearly real-time steering and throttle input. Because we must wait for the capacitor to charge/discharge, there is some lag in control.[6]

The impact of this feature is a physical kill switch: Turn the remote on and the running logic stops and the remote takes over.[7]; turn off the remote, and the running logic resumes.

[6] After all this, we've succeeded in building a simple RC car (worse than the one we started with)
[7] Also, the *Mario* theme song plays

5 Robot User Manual

6 Conclusion

6.1 Strengths

6.2 Weaknesses

7 Appendix

7.1 Source Code