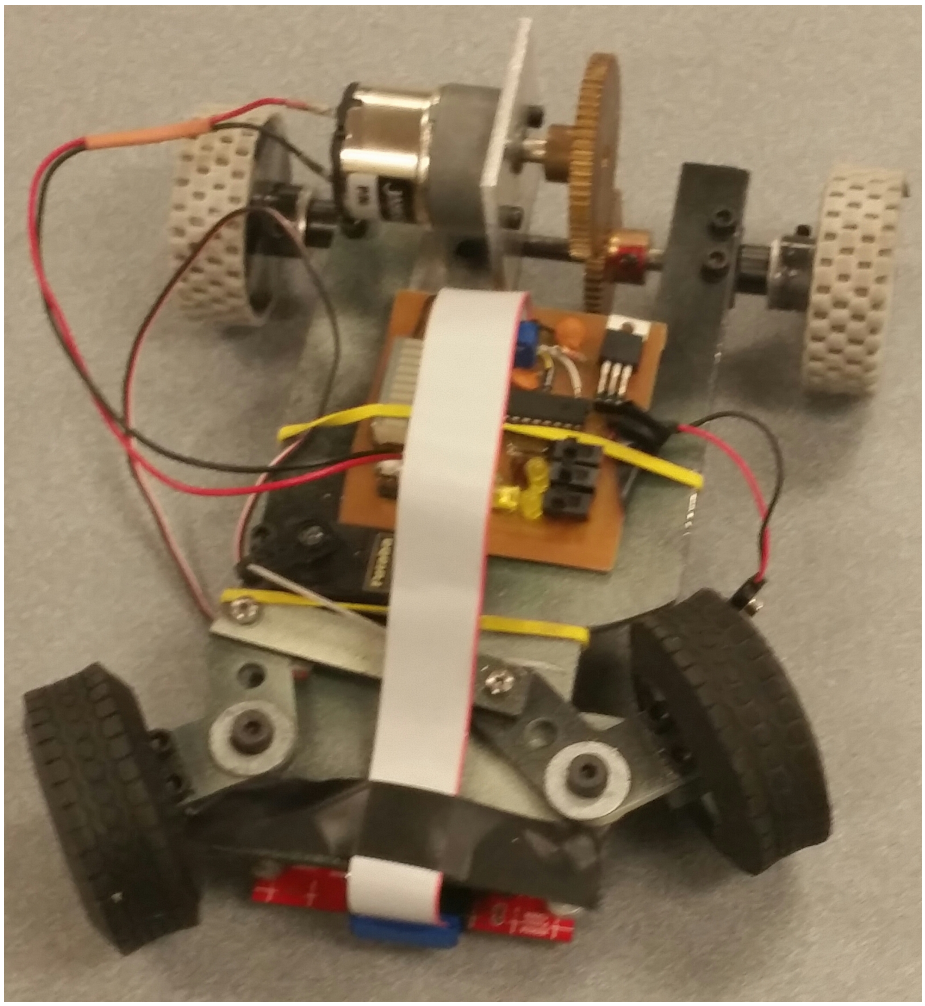

ENR 153 Control System

Jacob M. Kiggins, Timothy VanSlyke

2016

Contents

1 Introduction

1.1 Purpose of This Document

This Document is meant to act as a users manual for anyone attempting to setup and use an assembled control board, it is the reason most people will be reading this. In addition this document will contain all information pertaining to the concept, manufacture, assembly, and programming of the control board. This includes schematics, block diagrams, flowcharts, component requirements, and pretty much anything anyone would ever need to know. There is also information for anyone wishing to change the program or write an entirely new one from scratch.

1.2 Structure

This document is organized with the sections that will be referenced most often in the beginning and those less likely to be needed at the end. Most items pertaining to the general user will be in the first few sections. After that information useful to those who are manufacturing/populating the boards and of course programming and debugging procedures. Finally the last sections will be dedicated to anyone wishing to alter either the board or the program.

1.3 Background

ENR 153 is a mechanical design course in which students learn basic mechanical design principles. Students are required to modify the initial design of the vehicle and fabricate all major components themselves. At the end of the course students are given the task of building a car which is meant to drive around an oval track. This is where our control board comes in. The car is fitted with a Pololu QTR-8 line sensor, a servo for steering, and one or two motors (shared axle). The purpose of the control board is to connect to all of these components and use them to follow the oval shaped line.

The motivation to create this board stemmed from serious issues with the existing control system. Previously the students experienced more electrical and programming issues than mechanical. This was unacceptable since the course revolved around mechanical design. In addition the electronics required were expensive, bulky, and wasteful. It was obvious something had to be done.

1.4 Overview of the solution

Our solution needed to be simple, reliable, and rugged while maintaining a small form factor and improving upon the functionality of its predecessor. An emphasis was also put on identifying bad components which would cause the system to fail.

To accomplish this we designed a custom two layer PCB which was capable of providing power to the motors and interfaced with a micro-controller (ATMega328p) to handle the sensor, servo, and line following.

2 Layout/Operation of The Board

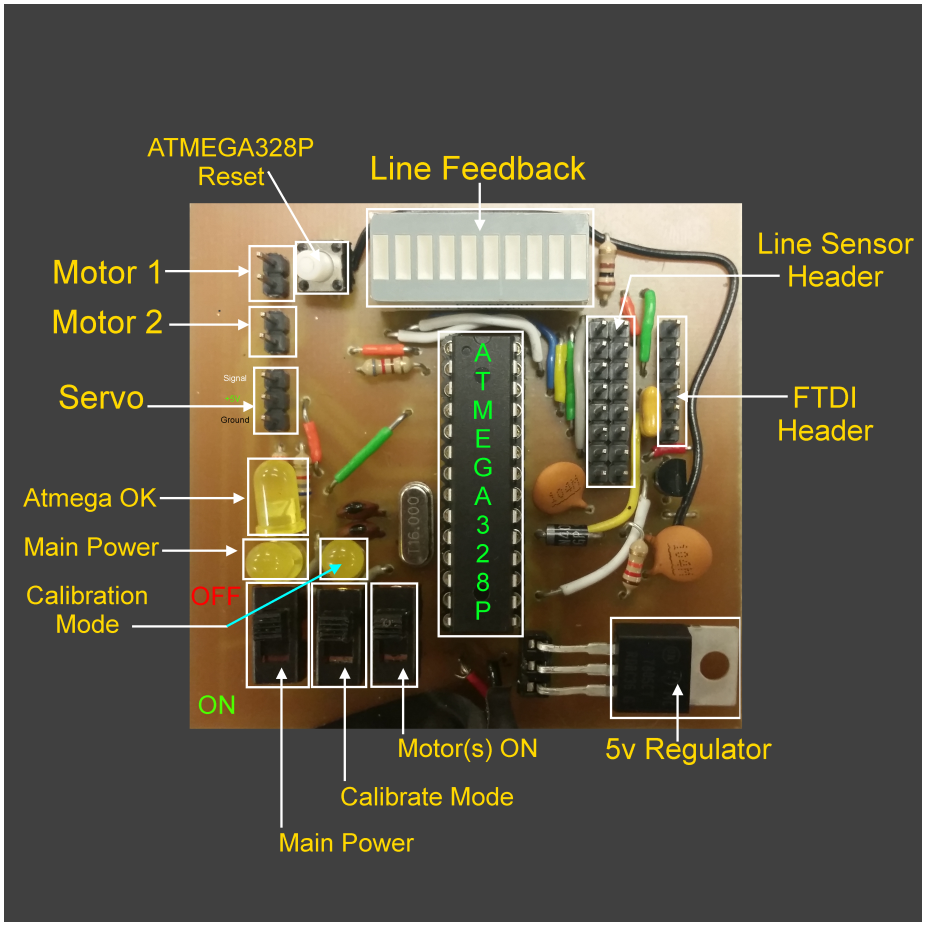

Figure 1: ENR 153 Board

2.1 On Board Components

Major components are highlighted an labeled in Figure 1. Most are self explanatory but descriptions are provided below

- ATMEGA328P Reset: This button resets the micro-controller and should (if everything is OK) result in the ATMega OK led blinking rapidly three times.
- Motor1 and Motor2: These are motor output ports, they provide full battery voltage and are reversible to allow for changing motor directions.
- Servo: A standard Servo header
- Atmega OK: blinks whenever the micro-controller is reset, assuming the chip is working
- Main Power: This led lights up when there is power to the servo, line sensor, micro-controller, and motor switch.
- Calibrate Mode: This switch will put the micro-controller in calibrate mode. While in this mode the line sensor values are normalized and the user should run the line sensor back and forth over the black line. The calibrated values are then displayed on the bar led where on represents black and off represents white.
- Motors On: This switch provides power to the motors, assuming the main power switch is also in the On position
- 5V regulator: A 7805CT voltage regulator
- FTDI Header: An FTDI cable plugs into this header and facilitates programming the ATMEGA328P from the Arduino IDE
- Line Sensor Header: 8x2 header, a ribbon cable connects this header to the QTR-8 line sensor

2.2 Off Board Components

This Module provides eight IR emitters/detectors (only 6 are used in this application)

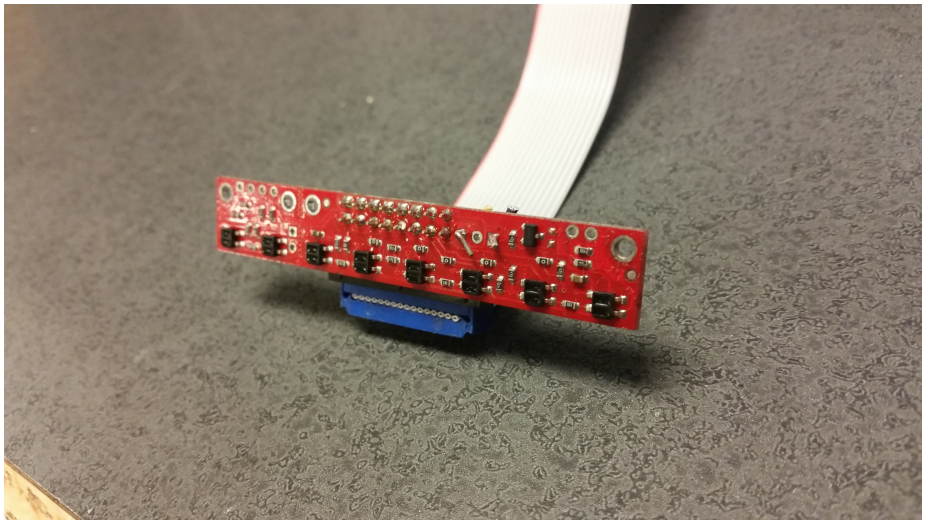

Figure 2: QTR-8 Line Sensor

The servo connects via the servo port

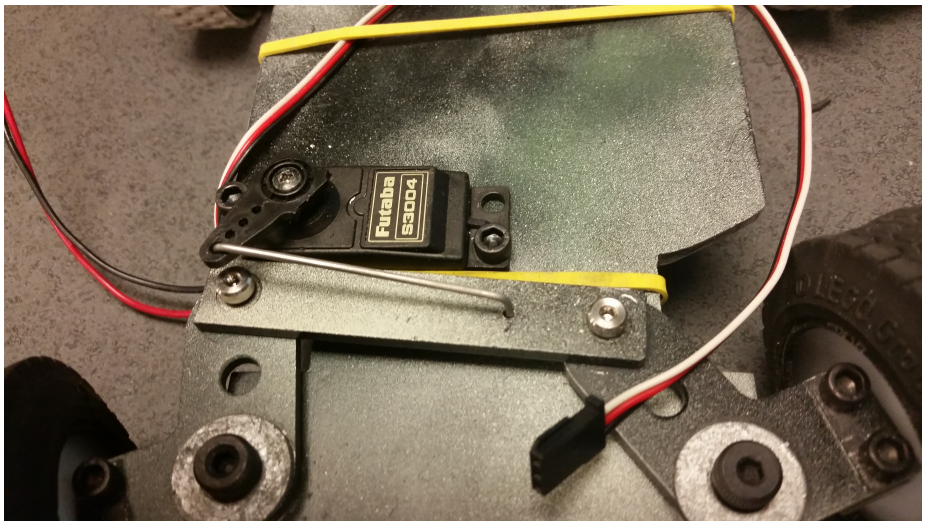

Figure 3: Steering Servo

The Motors connect to either motor port, connectors are intentionally reversible

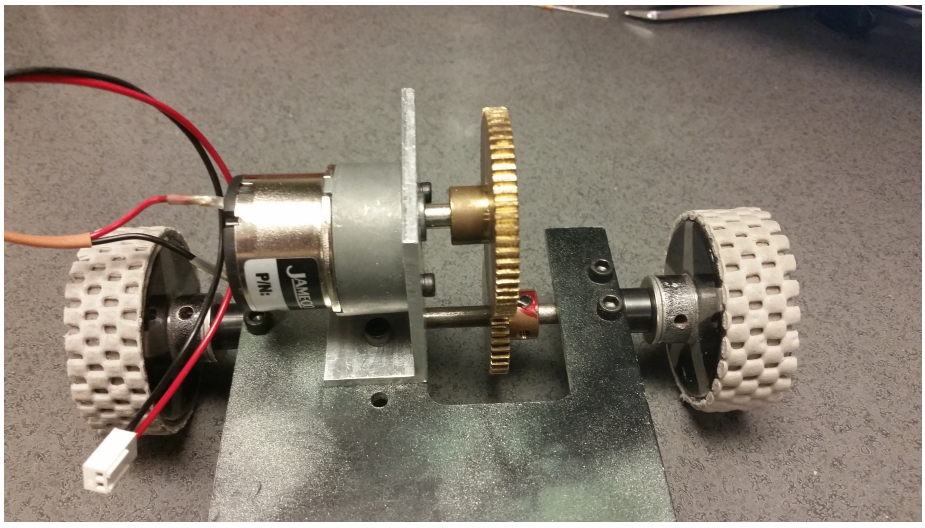

Figure 4: Drive System

2.3 Basic Operation

Hooking Up the board Each component that plugs into the board is documented above and connectors are used such that nothing may be plugged in the incorrect way.

Turn on the board by flipping the main power switch to the on position. at this point the program will load the last calibration from EEPROM and the servo should adjust itself based on the line sensor. You should always calibrate the line sensor if the board hasn't been used in a while or if you are experiencing poor performance.

To calibrate the board simply flip the calibration switch to the on position (the calibration led should light up) and run the line sensor back and forth over the black line. Once the sensors are calibrated you should see the bar LED light up when each individual sensor is over the black line, this gives you an idea how well the line sensor is functioning in its current orientation.

Once You are ready to run around the track make sure the calibrate switch is in the off position, and the servo is responding to movement over the line. Now simply flip the motor switch to the on position and watch the car go.

2.4 Troubleshooting

If you believe that the board is malfunctioning there are a few things you can do to further isolate the problem.

1. Make sure power is properly connected
2. Move the power switch to the on position, the main power LED should light up
3. Flip the power switch off, the LED should be off
4. Try wiggling the power connector and see if that has an effect, it may simply be a bad connection
5. With the power on hit the reset button, the ATMega OK led should blink quickly three times. If not the chip may be bad
6. Place the board in calibrate mode, the bar led should light up in response to the line sensor and the servo moves to the position specified by `SERVO_CENTER` in `definitions.h` (defaults to 90^o). If the servo and/or the line sensor behaves incorrectly the component(s) or their connection(s) may be bad.

If you make it through this troubleshooting list without abnormal behavior at any of the numbered items the problem is most likely not with the board. It should also be noted that some of if not all of the above issues could be caused by a low battery. Ensure you are always working with charged batteries.

3 Board Population and Schematics

The design of the board was done in EAGLE, which is installed on all Engineering Department computers. A majority of components on the board are passive elements (resistors, capacitors, ...) that act to either protect the onboard microcontroller, or improve its performance.

While somewhat dense, the board is fairly inexpensive and designed to be reusable when components fail. Even the

3.1 Information for Board Redesign

Main design constraints include:

- The board should be **easily** manufacturable in-house if need be.
  - Prototyping new designs is a must, and the ability to fabricate replacements on short notice keeps the supply healthy.
  - **No** surface-mount devices. Through-hole components only.
  - Wide traces connecting components. Thin traces are difficult to engrave and may peel off when exposed to heat (solder).
  - **Minimal** number of 2nd-layer (top layer) traces. Two layer boards add significant fabrication time.
  - Additionally, all 2nd-layer traces **must not** directly connect to any components (they should begin and terminate only on vias).
    - These last two constraints allow boards to be engraved on only one side so that all vias may be directly connected by wire.
- Components should be low cost, common, and easily replaced.
  - This board is designed to replace more expensive/wasteful alternatives.
- Size matters. The board should not exceed a length of 2.5" in any dimension.

It is also important to *document your design changes*. The current schematic in eagle is extensively annotated so that the design can be understood by future students.

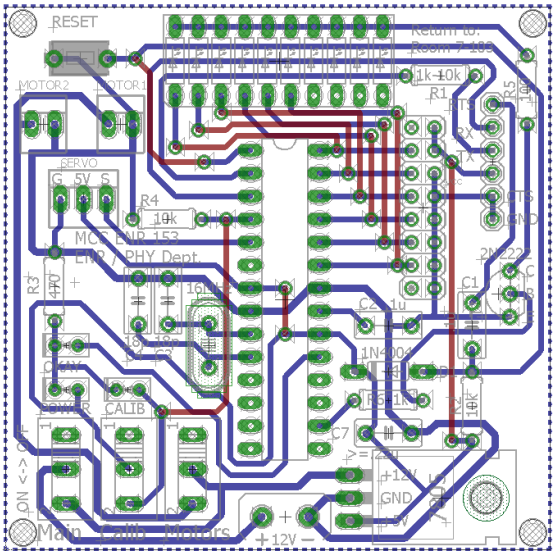

Figure 5: Original board layout.

Certain design changes are encouraged for future students. Particularly, reduction of the number of top-layer traces, simplification of the layout, and size reduction would all be welcome changes.

Additional components may be added to increase fault tolerance and can, consequently, actually end up simplifying the board. For example, the current-limiting resistor in series with the transistor gate actually reduces the number of vias on the board by allowing traces to travel through the gap underneath the resistor.

A worthwhile experiment, for example, would be to see if adding a series resistor to the analog pins cause any major issues in line sensor readings. If not, adding resistors could act to protect both the line sensor and Atmega chip, while providing an opportunity to eliminate the cramped top-layer traces that connect the analog pins to the LED array.

Finally, please avoid the following if possible:

- Routing between tightly packed pads, such as on a header.
- Using the AutoRouter as a first resort. While time-consuming, manual routing leads to better trace paths and component placement.
- Irregular trace paths. The final design should look professional.

- Thin routes on high-current lines. Line impedance causes poor performance and can lead to heating issues and even board failure.

4 Burning the Bootloader

To burn the bootloader to the Atmega328p we use Sparkfun's AVR pocket programmer. The hookup is very simple and Sparkfun provides a tutorial here. The basics however are outlined below.

Hookup the programmer and an Arduino board as show in Figure 6. You will need the following components.

- AVR pocket programmer
- 5x2 to 3x2 ribbon cable (included)
- Arduino UNO board
- Atmega328p chip to be programmed
- mini-USB to USB cable

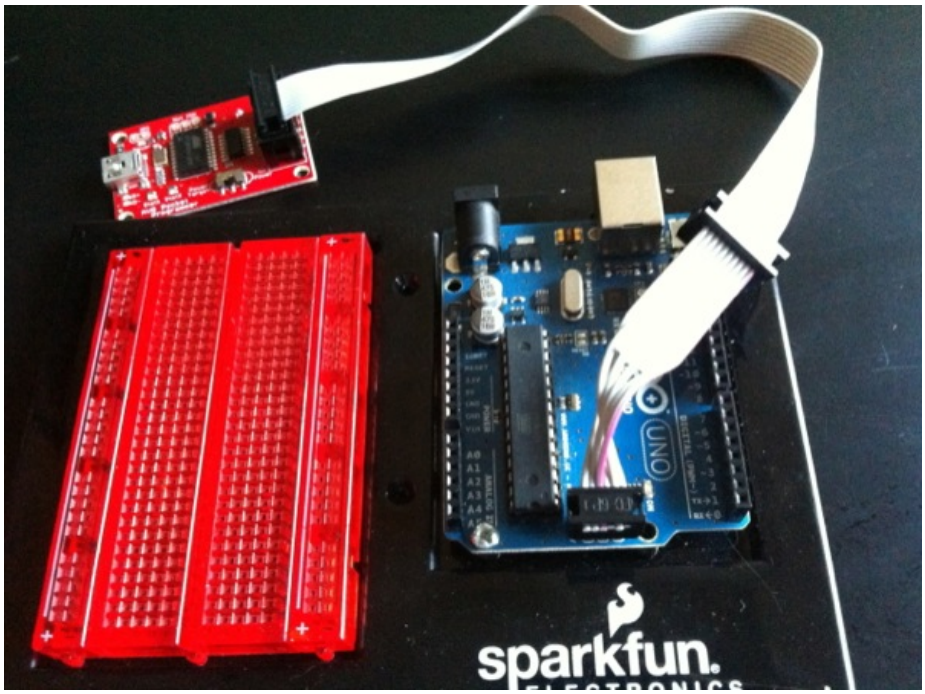

Figure 6: AVR pocket Programmer hooked up to an Arduino UNO

Once everything is hooked up, plug the AVR pocket programmer into your computer. Any driver issues can be solved by referencing Sparkfun's website (link above). To burn the bootloader open up the Arduino IDE click *Tools* → *BurnBootloader* Figure 7. If all goes well the chip is now ready to be pulled from the UNO board and plugged into the control board. Though you still need to upload the code via the FTDI cable.

NOTE: It is entirely possible to burn the boot loader and upload the ENR153 code while the chip is plugged into the UNO and hooked up to the AVR pocket Programmer. It is not possible at this time to include these steps in this document but, anyone willing to do a little research could easily figure it out.

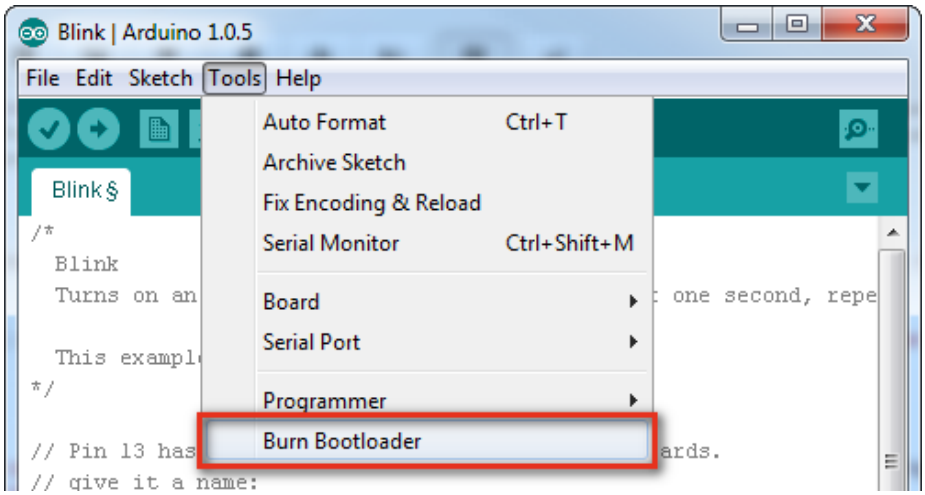

Figure 7: Burning the bootloader from the Arduino IDE

5 Uploading the program

For Issues With uploading please see subsection 5.1

The 153 control board code can be found on MCC's github page under the ENR153-line-follow repository. To download the repository simply open a bash or git-bash shell in the directory you want to clone to and type

```
# git clone https://github.com/monroecc/ENR153-line-follow.git
```

this will create a directory named ENR153-line-follow in the CWD. Alternatively you can click the Download As Zip button on the github repository main page (Figure 8).

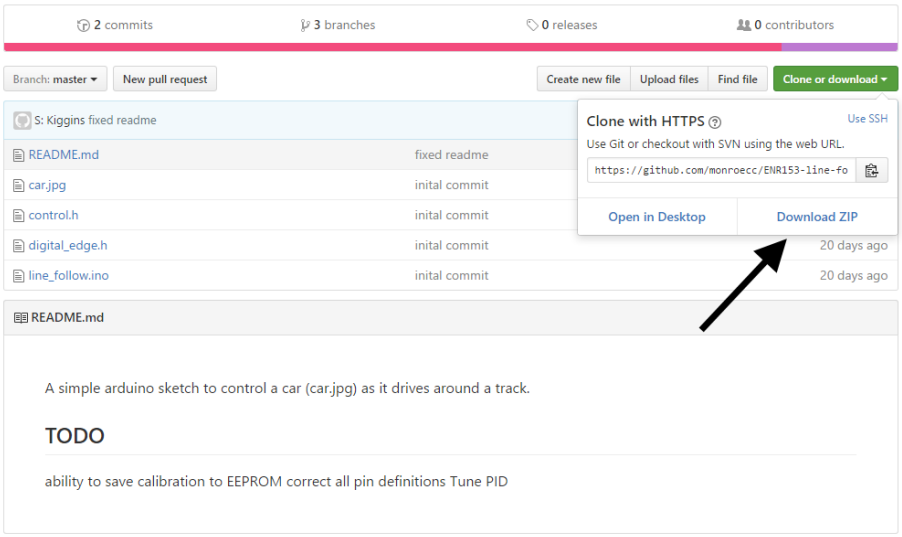

Figure 8: Git-hub Download as Zip

Once You download the repository you will by default be on the master branch, this is what you want in most cases. If you need to switch to a different branch simply open a bash shell and type

```
# git checkout <your branch>
```

. In most cases though, you can ignore that last bit and skip to the next step.

Extract the zip and open the directory, you should see 6 files. The only file you need to concern yourself with at this point is line-follow.ino file. Assuming you have the Arduino IDE installed double click on this file and let it open. Now you should find yourself an FTDI cable and plug it into the FTDI header on the board. Then plug the other end into a working USB port on your computer.

Your almost ready to upload to the controlf board, you need only to determine which is the correct COM port. The port you are looking for is never COM1, if there are no other ports refer to the uploading troubleshooting guide. If there are multiple COM ports not including COM1 you can try uploading to each, or open device manager and try and determine which is the FTDI cable. The port you are looking for will show up in device manager as "COM#" without any other writing before or after. Some devices show up as "COM# (description)", this will not happen with the FTDI cable. Once you have the correct port selected, simply press the upload button on the Arduino IDE.

5.1 Uploading Troubleshooting

NOTE: The FTDI cable will still show up as a device even if it's not connected to the chip on the other end, an upload error may be the result of a bad connection on the board, though it probably isn't.

If you see an error like the one in figure 9 please visit https://www.arduino.cc/en/Guide/Libraries this will show you how to install missing libraries.

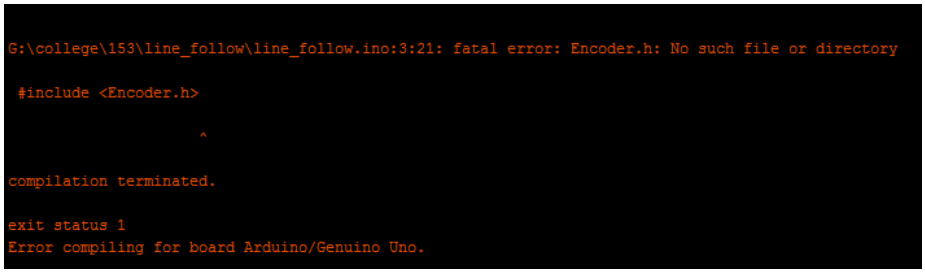

Figure 9: Missing Library Error

Common problems and Solutions

- Make sure the Arduino IDE is updated to the latest version
- Try running update driver software on the FTDI COM port from device manager ($Properties \Rightarrow Driver \Rightarrow UpdateDriver...$)
- Try a different computer or restart the one you are using. Sometimes Windows doesn't want to work and the fastest and easiest way to fix it is a restart
- If you can't determine the correct COM port try uploading to each COM port available individually until one works. Or unplug the FTDI cable and plug it back in, watch the ports listed in device manager to figure out which port changed.
- If there are COM ports listed in device manager that aren't listed in Arduino try restarting the IDE and/or unplugging and re-inserting the FTDI cable.
- if you continue to get upload errors try switching the programmer to AVR ISP ($Tools \Rightarrow Programmer \Rightarrow$ AVR ISP)
- Try using the same FTDI cable on a different board, or using a different board with the same FTDI cable, one of the two may be bad.

- Try switching out any replaceable components such as the regulator and the Atmega328p chip
- All other issues can most likely be resolved by referencing the Arduino troubleshooting page

Uncommon Problems if none of the above fix your problem and you are sure beyond any reasonable doubt that the issues you are having aren't caused by either the FTDI cable or The specific computer you are using the following may help

1. Sometimes there are manufacturing defects and more frequently there are mistakes made when soldering. Given how tightly packed everything is on this board a soldering mistake becomes even more likely. To test for this simply look at the bottom of the board and search for any unwanted solder bridges. You should also wiggle each joint around to make sure there is solid contact
2. Even if each solder joint feels solid it may still have a bad connection to the pad, use figure 10 as a reference and fix any bad solder joints
3. If a visual inspection is not enough continuity test is in order. Open up the Eagle board file for the control board and use the view tool to see where each trace goes. With a continuity tester make sure each trace goes between the proper pads, and make sure each pad is not connected to nearby pads.

Important: If you find a defective board SAVE IT! this could be a random thing or it could be an error that affects many boards. In either case the department may be able to get a refund and the board will be needed as proof. Alternatively, the board may simply be housing a damaged component which simply needs replacement.

Please return all damaged/lost boards to Don Howard in room 7-103.

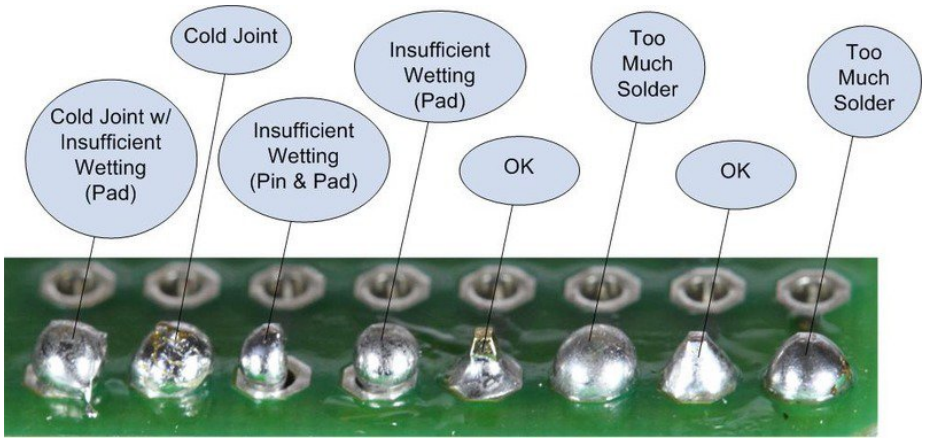

Figure 10: Examples of Good and Bad Solder Joints

6 In Depth Board Design Considerations

6.1 Components

Components were selected to balance quality and cost. Below is a list of components and why they were chosen

- Atmega328p - we need a replaceable micro-controller that was cheap and easy to program. Since this is used in the Arduino Uno it is very easy to program. Furthermore the Uno is open source and the eagle schematics are available through Arduino's website. With these schematics every minor detail and random capacitor value is right in front of you, not buried in 100 pages of a data sheet. It is also fairly cheap at $3 a chip.
- Bar LED - We needed a way to display what the chips is "seeing" through the line sensor. This would allow students to diagnose poor lighting, bad calibration, or a broken sensor. The 10 led bar was chosen for its convenience during board population.
- FTDI cable - an FTDI cable was chosen as opposed to an FTDI chip because these boards are only going to be programmed a few times at most, there was no need to complicate things with an on board chip. Also most (maybe all) FTDI chips are surface mount and that isn't something we would have been able to solder at MCC.
- 7805 regulator - This is a very common regulator and has proven its durability over time. It is also quite inexpensive.

All other components such as capacitors, resistors, and transistors were a direct results of the requirements imposed by the above components. These things are all pretty generic as well so as long as the values are correct, everything should work

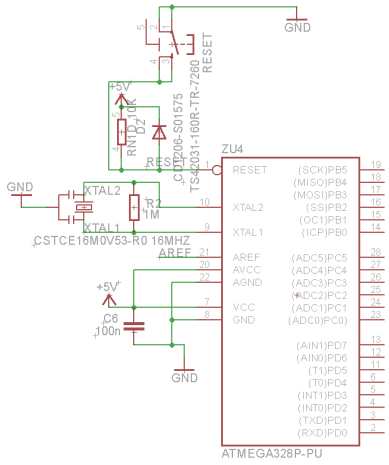

Figure 11: Minimum Requirements to run Atmega328p, our starting point

Why the Transistor? There are a limited number of digital pins on the Atmega328P, to leave the board flexible to additional features and due to routing simplification we decided to use the analog pins to read light values, and to display them on the bar led. The issue with that is having the led bar as part of the circuit when reading distorts the analog values read. To solve this we added a transistor between the bar led and ground, this allowed us to display for a time, then cut the bar led out of the circuit by turning off the transistor to allow for reading. A 10ms delay was necessary after turning the transistor off to allow everything to settle.

Completed Eagle sch and brd files can be found on the github page under the folder control_board.

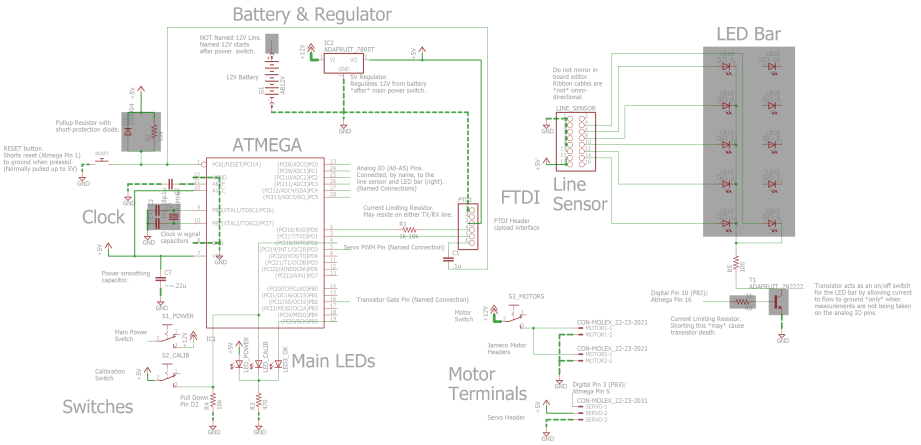

Figure 12: Final Schematic of the board.

7 Program Specifications

7.1 Basic Modification

The file definitions.h is provided to allow for simple modification of behavior of the line following. each "#define" is commented with its function. A snapshot can be seen in figure 13

```
// This is the position the servo will assume when the PID adjust is zero
#define SERVO_CENTER 90 //BOUNDS: [0, 180]

 //Limits the max deflection angle that can be set by the PID
#define MAX_ADJ_LEFT 51 //BOUNDS: [0, 90]
#define MAX_ADJ_RIGHT -32 //BOUNDS: [0, 90]

//PID control contants, be very careful adjusting these
#define P 0.23
#define I 0.1
#define D 0.0

//offset for EEPROM, can be used if the first 12 bytes are written too many times
#define EEPROM_OFF 0 * 12
```

Figure 13: definitions.h file

7.2 Advanced Modification

The program is divided into three main classes. The `PID` class which is initialized with constants via the `set_pid(float p, foat i, float d)` method.

This is a pretty straight forward implementation of a PID. The only aspect that may seem strange is the handling of the integral term, this is outlined in the comments.

The `analog_sensor` class, handles reading from an analog pin and scaling the value based on the calibration. This class also saves and loads individual calibration values (min and max) from EEPROM with the help of the EEPROMex library.

The `line` class uses both of the above classes and provides an interface for the main Arduino sketch to line follow, calibrate, and read/write operations to the EEPROM. In the Arduino Sketch an array of `analog_sensors` is created, and passed into the constructor for the line class. Then `line.load()` is called to read the calibration from EEPROM. After a few calls to `pinMode()` the program drops into the main loop, which uses the state of the calibrate switch to determine what case the program executes. If the board is put into calibration mode, `line.save()` can be called after calibration to save the new values to EEPROM.

To fully understand how the program functions and gain the necessary insight to modify or re-design, a person should read though the fully commented source code which can be found here.