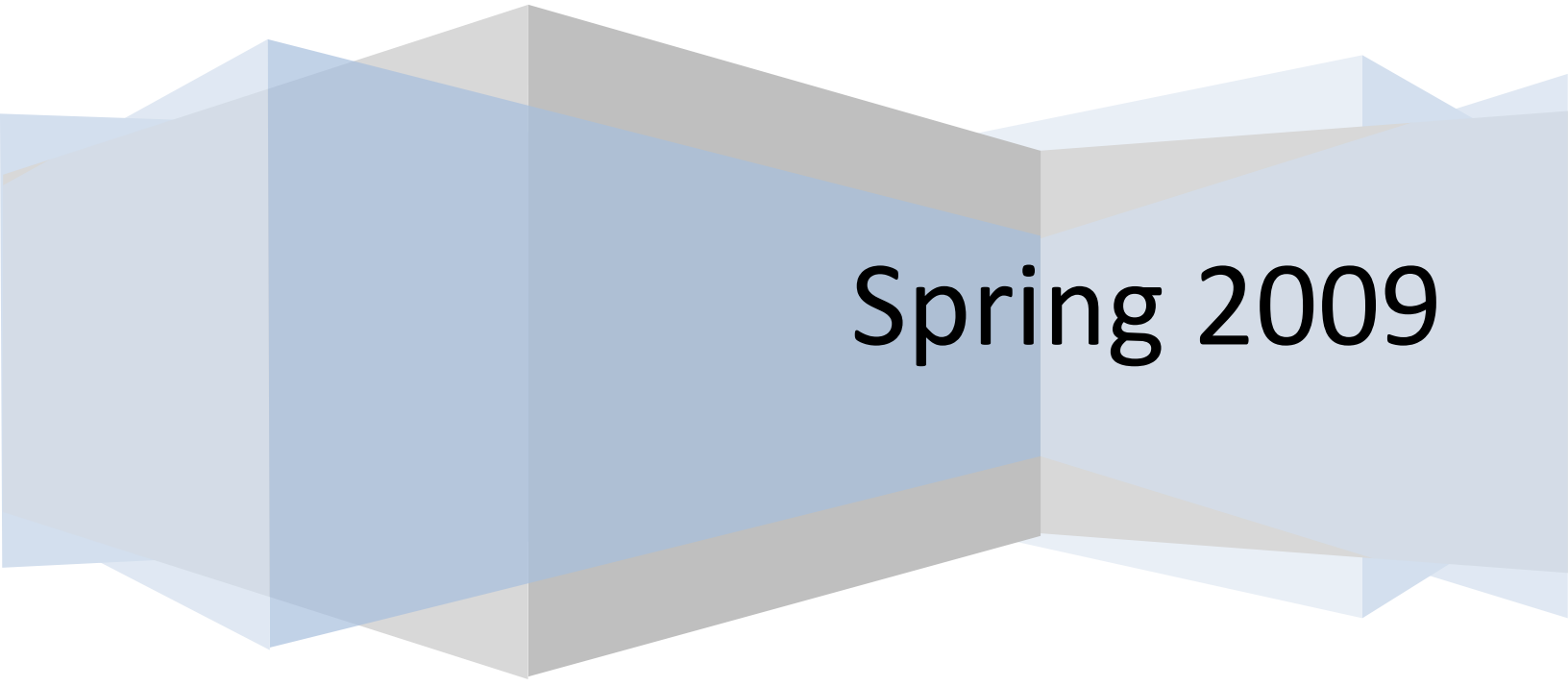
University of California, Berkeley, Department of Electrical Engineering and Computer Science

EE192 Progress Report

Team 2 – "FFHDAM"

Farzad Fatollahi-Fard, Hartej Dhami, Aman Mouhidin



Spring 2009

TABLE OF CONTENTS

I – Current State of Project

Current Hardware Configuration

Microcontroller

We are using the ADuC7026 Evaluation board as our controller for the car. The reason for doing so was because we wanted to have all features that the microcontroller has (the ADCs, Digital GPIO, etc) completely exposed so we wouldn't have write extra software to route the I/O through the programmable logic array. This allows us to directly use the built in PWM generator and provides us with more ADC conversion pins, if necessary.

Power

The power source for everything onboard the car is a single 4500-mAh, 7.2-V battery pack. The battery pack does not provide steady voltage as it drains; as such we have power circuitry in place to ensure stable voltage for all necessary components. The motor is not sensitive to voltage level (as long as it can provide enough power via more current), but all other components cannot tolerate fluctuating voltage. As detailed in the hardware documentation, we currently employ a DC-DC boost and voltage regulators to provide the micro controller board and rest of the circuitry with stable 5.0 Volts.

Communication

We are using the 'BlueSMiRF' Bluetooth adapter to remotely communicate and collect data from the micro controller board.

Sensors

The 75-kHz 100-mA_{RMS} track is sensed via a parallel RLC circuit that has a matching resonating frequency. The RLC circuit picks up the sinusoidal track signal, which we then amplify via two stages of op-amps. Instead of working with raw amplified signal, we have integrated low pass filter to our first op amp stage. This passes a level signal instead of a sinusoidal wave to the ADC converter on the micro controller. We have the circuitry for four magnetic sensors, but we only have three of the four connected, which are used in the front for our steering control. The three sensors allow us to compare left, right and center of the track values. The fourth sensor is planned for detecting crossings. The hardware is in place, but the software that will take advantage of the fourth sensor is not in place. We are using two quad LM6144 Op-Amps, which provide two Op-Amp stages for each magnetic sensor.

As already stated, the battery pack does not provide steady voltage over time; as such a constant motor PWM does not guarantee a constant speed. Therefore, we have installed a speed sensor. The speed sensor consists of an infrared diode and a Hamamatsu optical detector installed across a perforated disk which turns with the drive shaft. The optical detector is a smart detector in that it outputs a high and low signal when detecting the diode being off or on, instead of outputting a range of voltages. This saves us an ADC conversion. The sensor output is connected to a GPIO (General Purpose I/O) that triggers a timer on the controller board every time the pin goes high.

Software Modules

The software is responsible for generating PWMs for the motor and the servo. We use the micro controller's built in PWM generator for the motor. The minimum frequency the built in PWM (317.5 Hz) can provide is many times higher than the required frequency of 50Hz for the servo. Therefore we use a timer to generate servo PWM.

The software has a communication module which prints important system parameters on request via a serial connection. The communication module also allows us to remotely kill the motor PWM in case of an emergency.

The software is responsible for setting servo PWM based on sensor readings. The sensors are read on demand before making servo control decisions. The servo control decision is based on a look up table that sets servo angle after comparing the sensor readings to pre set thresholds. We set the thresholds based on calibration data we gathered from placing car at different distances parallel to track.

II – HARDWARE DOCUMENTATION

Motor Drive Circuitry

The Motor drive circuitry consists of a MOSFET driver along with logic gates. The logic gates provide protection against simultaneous operation of forward motor operation and motor braking operation.

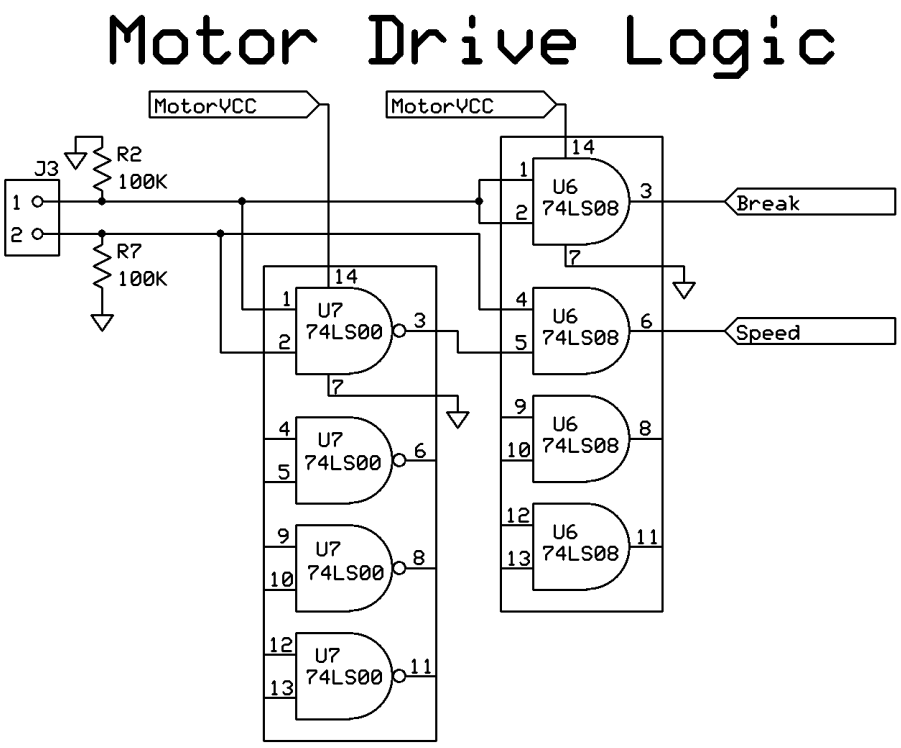


Figure 1: Logic to Ensure Speed and Break are never ON at the same time

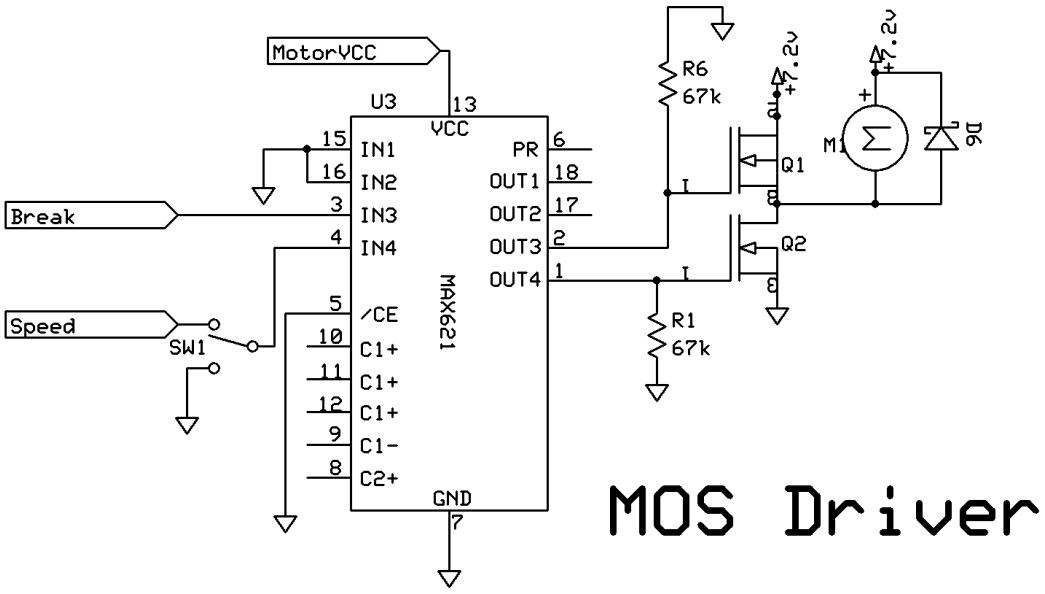


Figure 2: MAX621 MOSFET driver controlling the motor

Sensor Electronics

The track sensor network depends on Op-amps to amplify track signal. We use a two stage setup. The first stage amplifies and changes the signal to level shifted version using a low pass filter instead of a sinusoidal output. The second stage makes sure the output impedance is matched to the micro controller's input (low). The Hamamatsu speed encoder consists of a simple infra-red LED shining at the detector which reports a high or a low depending on whether it detects any light.

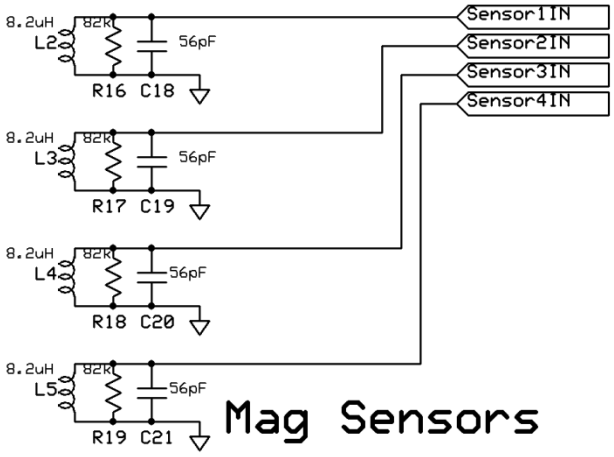


Figure 3: RLC circuit for each sensor

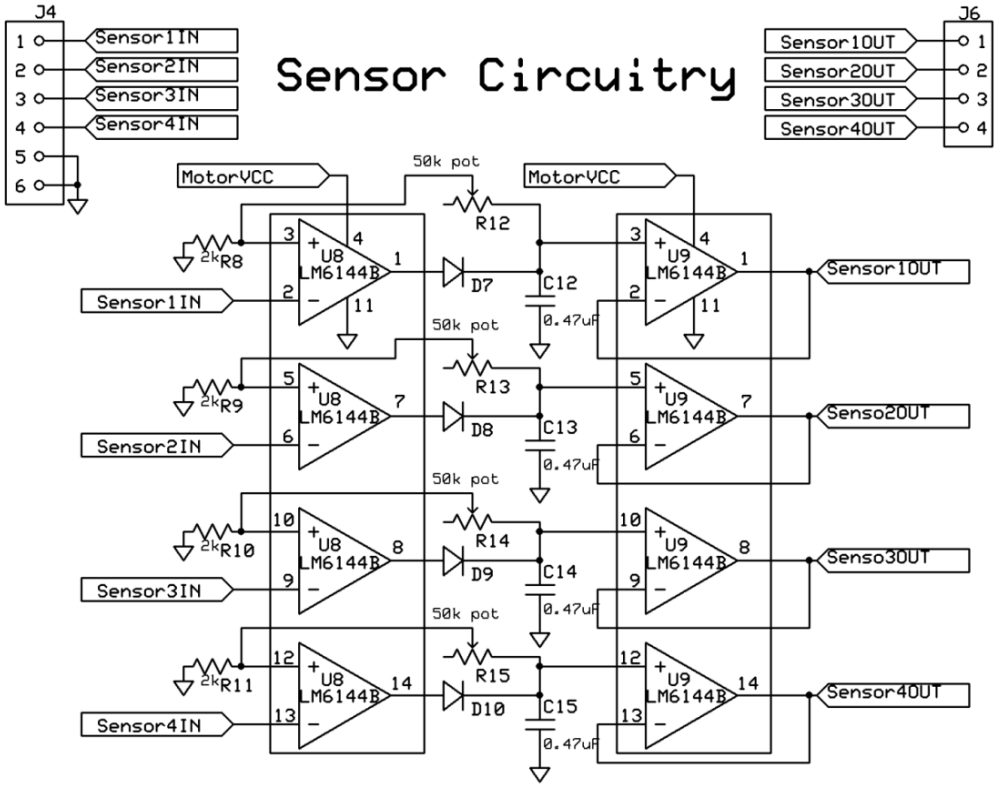


Figure 4: OP-Amp circuit amplifies signal picked by the RLC circuit

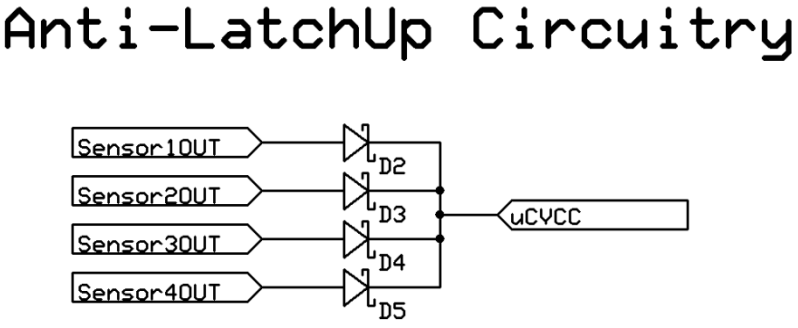


Figure 3: Each sensor output is connected to micro controller Vcc to prevent latch-up

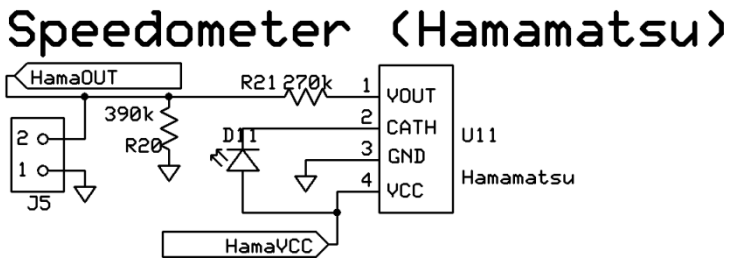


Figure 4: Speed Encoder circuitry

Power Supplies

The main power circuitry consists of a DC-DC boost converter that increases the incoming voltage to 12V. The 12V is then regulated by four voltage regulators that provide 5V output. Each voltage regulator serves a group of devices.

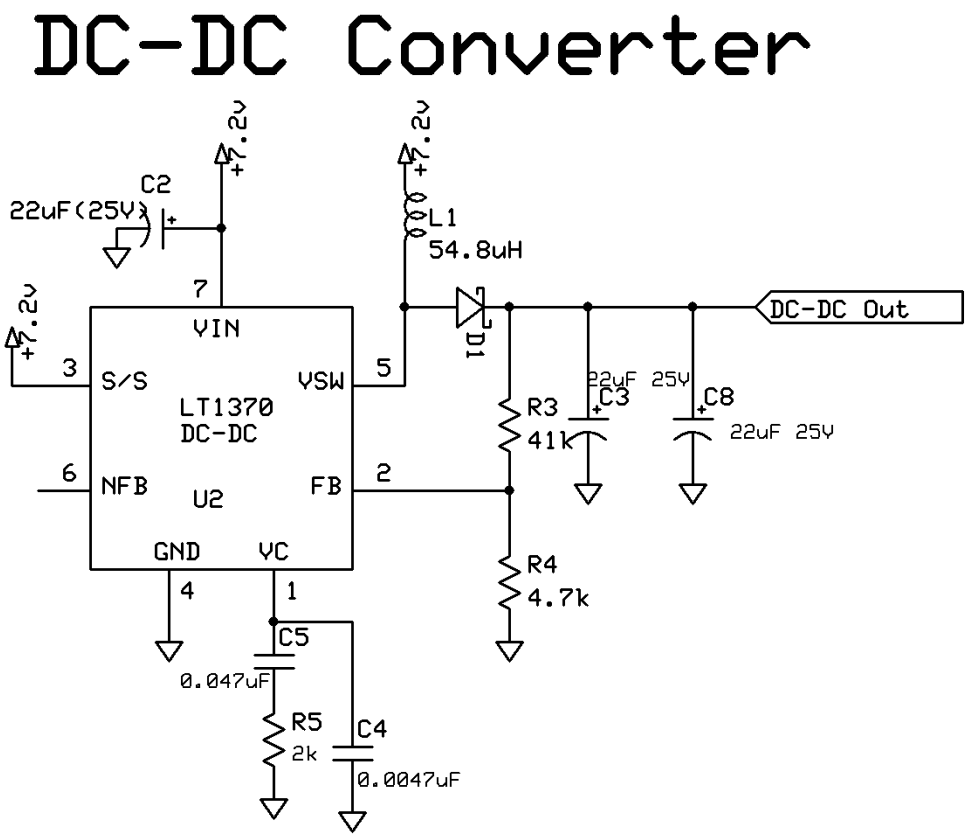


Figure 5 DC-DC boos converter uses switching to pump a low voltage to 12V

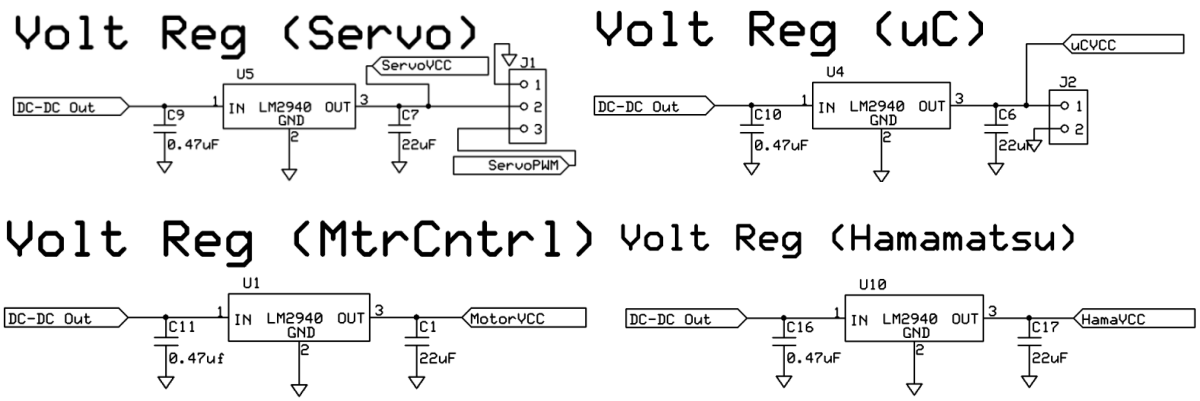


Figure 8: These Voltage Regulators convert incoming 12V to 5V

PCB Layout

The PCB layout ensures that all high current paths have proper trace width. Also as per recommended layout setup the DC-DC boost converter has short paths between V_{sw} and GND. Also the Motor and battery pack V_{cc} are close and capable of handling large currents.

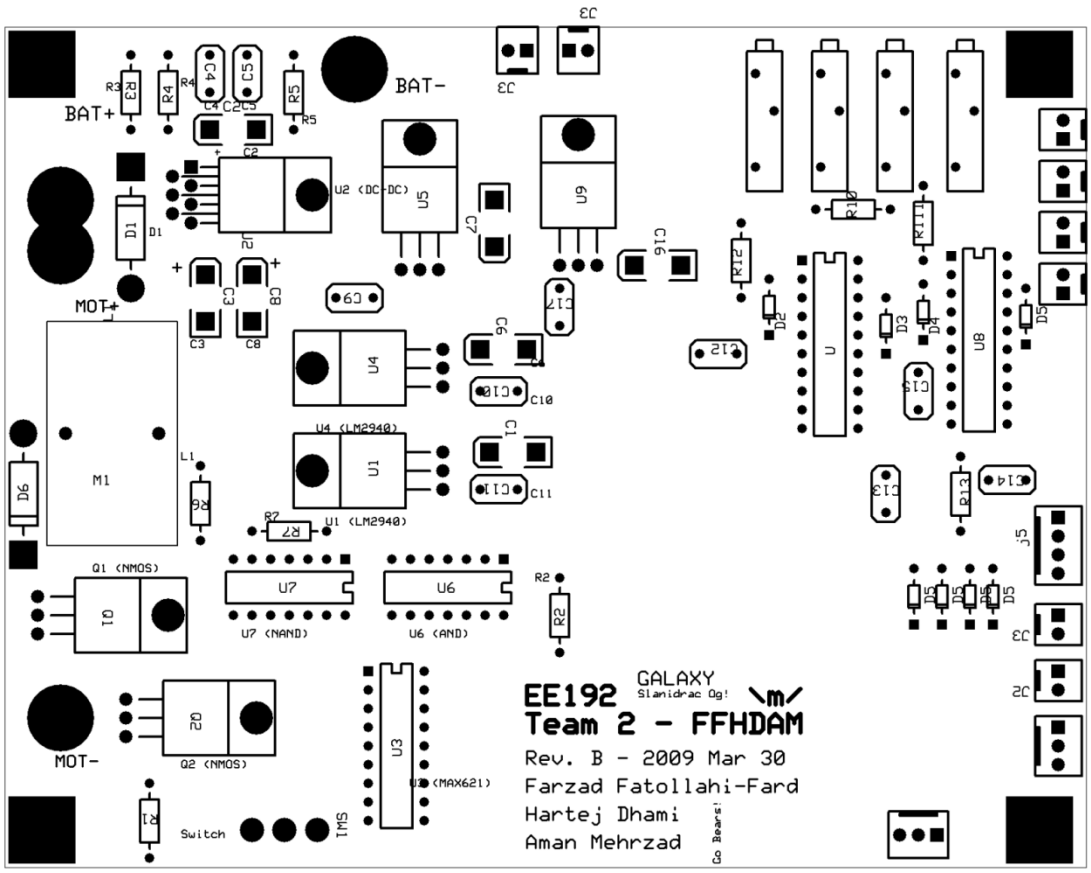


Figure 9: PCB Layout

III – Proposed Control Methods

Velocity Control

Current Control

We have a very simple control implemented. It communicates with steering control to determine when the car is in a straightaway or in a turn. If the steering controller says that the car is in a straightaway, the velocity controller sets the motors to a high PWM duty cycle (about a 30% duty cycle). If the steering controller says that the car is in a turn, the velocity controller sets the motor to a lower PWM duty cycle (between a 15% to 20% duty cycle). Also, if our sensors do not sense the track, the car is programmed to slow down, mainly to avoid crashing.

As you may have noticed, the values for our duty cycle are rather low. The reason for this is that we currently have no software to support the speedometer (the hardware has been implemented). As a result, we don't have any sensing solution for the speed. By keeping it at these low speeds, we can reliably keep it under control.

As a result of our lack of a speedometer, the speed of car is susceptible to the voltage of the battery. As the voltage decreases, the speed of the car decreases. This makes our current controller very unreliable.

Proposed Control

The controller we are looking to implement is a Proportional-Integral controller (PI controller). Using the speed determined by the speedometer, we find the error, Δ (the difference between the actual speed and out desired speed). We then feed this error through our PI controller, which has the following equation:

$$K_P \Delta + K_I \int \Delta \, dt$$

From this, we would determine the PWM duty cycle to drive the motor. This method would require us to run multiple simulations and calibration runs to determine the value of K_P and K_I. We also plan to slow down our desired velocity on turns to make the car much more stable. We're also planning on keeping the slow down when the car finds no track.

This solution is much more reliable than our current implementation. If the battery voltage decreases, it will automatically increase the PWM duty cycle to match our desired velocity. Also, since we have a direct gauge of our speed, we can also run our car faster.

Steering Control

Current Control

We have a very primitive steering controller. The controller uses only three of its four sensors: a left sensor, a right sensor, and a center sensor. It takes readings from all the sensors and takes the difference between the left and the right sensor readings. If the difference is close to zero and the center sensor senses the track, the steering controller determines that the car must go straight. If the difference is found to be positive, we turn right with four options; each slightly right than the previous. Choosing between these four options is determined by using the center sensor: if the center sensor reports high reading, the turn is less severe compared to if the center sensor reports a low reading. A similar scheme is implemented for turning left.

This implementation is good enough for our car to finish the Closed Loop Figure 8 in less than 10 seconds, but it doesn't provide the resolution needed for traversing a NATCAR track.

Proposed Control

With the left and the right sensors, we plan on using a linearized error signal. From the document posted on website titled *NATCAR Magnetic Sensor Modeling* by Darren Liccardo (Spring 2002), we've determined that we are going to use the following equation for our error:

$$error(s_{right}, s_{left}) = \frac{1}{s_{right}} - \frac{1}{s_{left}}$$

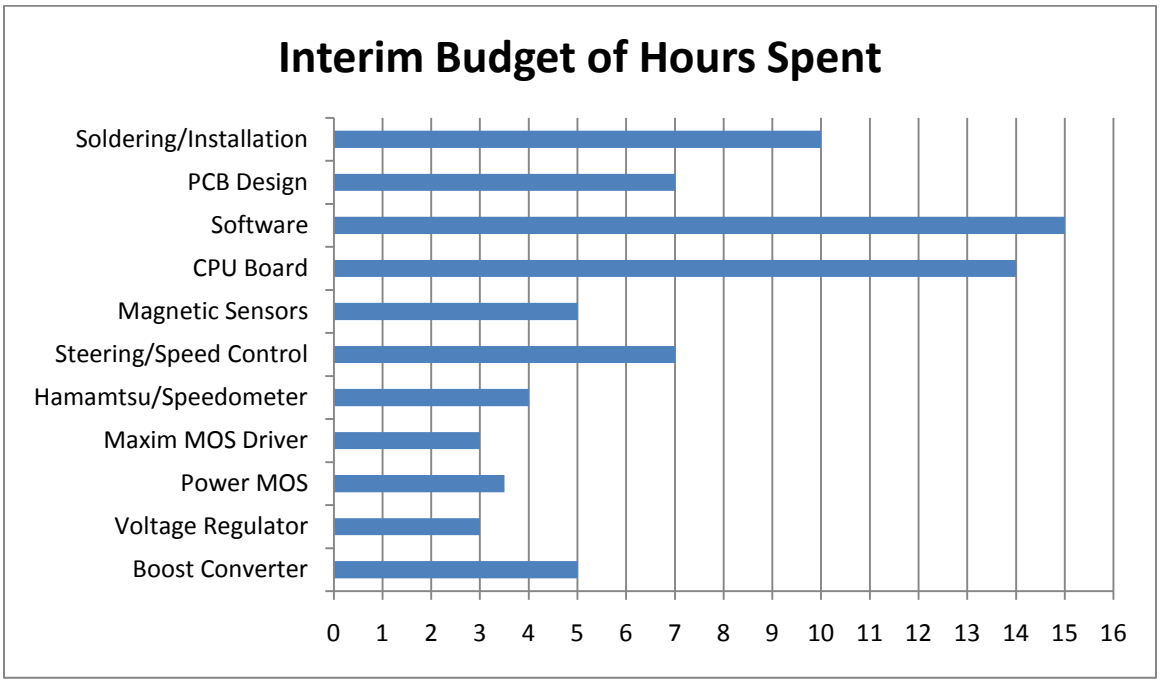
We plan on linearizing this equation into piece-wise regions (as is evident from the graph in the document on page 6) to make the calculation on our microcontroller simpler and faster. From this calculation, the car determines how far to turn and in what direction.

We plan using the center sensor for checking to make sure if the car is on the track itself.

We plan on using the fourth sensor for checking for crossings. If there is a crossing, we then use the reading from this sensor to ignore the noise of the other three sensors.

IV – Interim Budget

Mostly everything has been running very smoothly. Below, we have an estimate of how long we've spent on different portions of the project.



We had some trouble at first with the PCB tools. It took a while to learn how to use them and how to do a good layout. Software has taken a long time as well. This can be explained by the fact that there was that we had to familiarize ourselves with CPU and once we had written our code, it took a lot of hours to calibrate our code.

One major problem we had was that our first revision shorted right before a checkpoint for reasons we unbeknownst to us (we've checked the circuit many times and haven't found the short). This forced us to replace our DC-DC converter many times, as it burned out as a result of the short, which further damaged our PCB. With our second PCB, we've made sure to avoid any shorts by making sure no components pads touch when soldered. We also made sure that components with heat sinks don't touch traces, as that may have led to the shorting of the first PCB. As is evident in our new PCB, we have made the trace clearances much larger to hopefully avoid this problem.

We've also had some monetary costs. We ended up purchasing a new car chasis, which cost us $280. We also purchased our own mounting hardware to mount all our boards to the car, which cost us about $15.

V – Refined Proposal for Software Architecture

For the upcoming contest rounds we need to implement a few new software modules, as well as rework some of the older software modules.

One of the new modules we have to implement is the speed encoder module. The hardware is designed and implemented for the speed encoder. We need the software that will allow the speed encoder to count up a timer on the microcontroller on every positive transition. This setup will allow us to avoid servicing interrupts caused by the speed encoder. We will use polling to check and reset value of this timer at regular intervals using another interrupt that is already necessary (servo PWM generation interrupt). Writing code and debugging for this module should take us two to four hours of work.

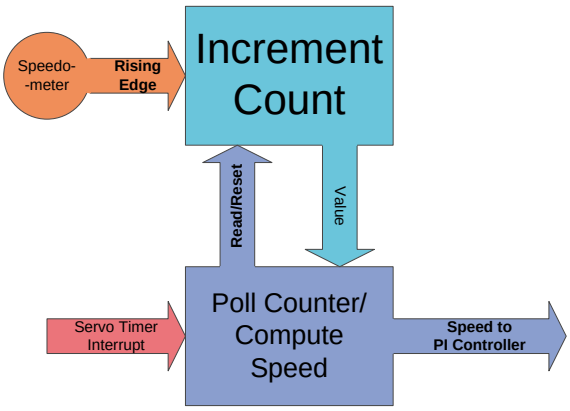


Figure 11: Speedometer software block diagram

In the current software modules, our PWM generation for the motor is based on integer division. Keeping the importance of latency in mind, we must remove this unnecessary division and put in its place a look up table with pre computed values. The only real advantage of using integer division is better resolution at controlling the motor PWM. This has been necessary so far without a speed sensor. Once the speed sensing is implemented, the feedback control system will ensure there is no need for precise PWM generation. A PI Control will provide necessary speed with just some look up table values for PWM generation. Another potential problem posed by this module is that changing the resolution provided by the look up table is going to be time consuming. It will require us to compute the values every time and enter them. We have planned on two approaches: First we will just use Microsoft Excel™ to pre-compute the look-up table values and use the ones we require. If during calibration runs we find ourselves frequently changing these values, we plan to use a scripting tool to auto-generate the look up table. The changing of the PWM generation to a LUT should take about one day of work with the generation of a Microsoft Excel™ file. If we need to use a scripting tool, that should take another two hours.

One of the major flaws with our current software setup is the ADC conversion timing. Currently the ADC conversion is on demand. Whenever the while loop of the micro controller comes around we do an ADC conversion. This means that a longer code will result in longer delay between ADC conversions. We did this thinking it will help us avoid servicing interrupts. But now we feel it is more important to have ADC conversions at regular intervals rather than saving on interrupts. Since most of the software for setting up and using ADC conversions is understood, the transition to an interrupt based setup should not take more than three to four hours of coding and debugging time.

The major software overhaul that we have anticipated will be in the control laws. We have a few different ideas that we think will improve our control algorithm.

One of the changes we plan to implement is to have a running average of last few ADC conversions to smooth out any noise. This will eliminate the twitching our car exhibits on straight ways. Since averaging will require us to use division, it is important that we don't make this change a burden on the computation unit. Therefore we will only average either last four or eight readings. This will allow us to use bit shifting for division, thus saving us valuable computation time.

Since we are changing our control algorithm to liberalize the inverse difference function, we will have to update the software to new look up table values. If this works, we expect to lose the middle sensor. This will greatly affect our latency. Latency will be one of the major metric that we will try to cut. We will also add look up table values that take into account an orthogonal sensor that reports crossings signals. This orthogonal sensor will act as a warning sensor. If this sensor reads high values, this means that the car is at risk of picking the wrong track. In such a scenario the software will ignore any sudden changes in the readings of horizontal sensors.

The last crucial update to the software module will be track learning. We have not decided how accurate the track learning we will implement. But the basic track learning that we have currently discussed entails recording major turns in the track. We want the car to go on the track at a slower speed and record all turns into an array that tags the turn severity with distance travelled. When the car has completed one lap, we will change the mode to reading the recorded values. During the second run, the car will look ahead at recorded values and check if there is a turn in the next three to four meters. If there is no turn, the car will speed up to maximum speed. Otherwise the car will stay within safe speed for cornering. A more elaborate method will allow the car to maximize cornering speed also. We have not discussed the algorithm to implement this yet.

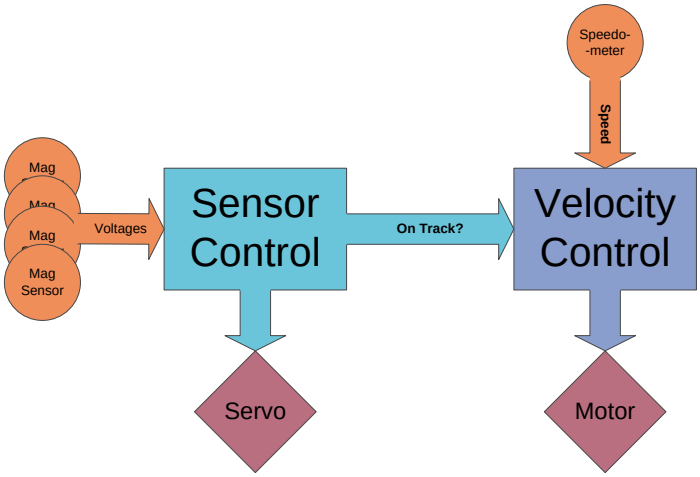


Figure 12: General Block Diagram

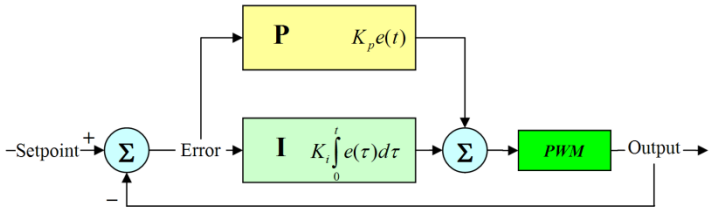


Figure 13: Motor control

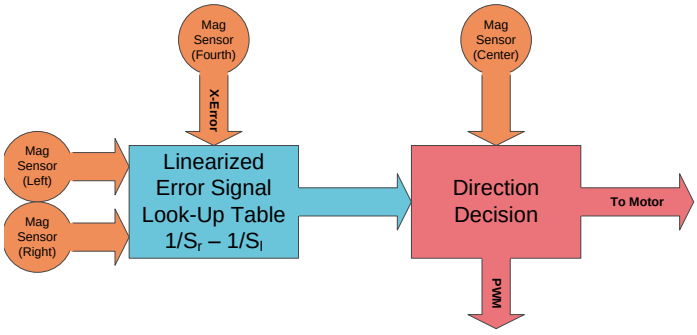


Figure 14: Servo Control

VI – Additional Resources Required

In order to complete this project, we are going to need a few more resources. The way we have currently implemented our sensor circuitry is by mounting a perf-board on our current PCB with all the sensor circuitry on it. This implementation is very vulnerable to noise and damage, as there are wires crisscrossing someone's hand may get caught in it and destroy it. As a result, we're going to need a new PCB to integrate our sensor electronics into our PCB so that we can avoid these problems.

Lab resources are a huge problem. At the beginning of the semester, it seemed like the lab all the tools and the equipment that we needed. However, as the semester has progressed, finding tools has become a feat in itself. A lot of the groups have decided it's a good idea to leave the tools they use on their desk rather than returning them to the front of the lab. We have to go on an epic scavenger hunt to find who stole the needle nose pliers, as an example. As a result of class policy, it's not a good idea to go rummaging in other students' stuff because we may inadvertently ruin their project. We think that there needs to be a weekly bench-top cleaning so those students don't horde tools.

Also, header pins are a nice thing to have out in the general population, as they are usually locked up in the cabinet. They are incredibly useful.