Carnegie Mellon University

Design, Construction, and Evaluation of a Ballbot with a Spherical Induction Motor

by

Greg Seyfarth

A thesis submitted in partial fulfillment for the
degree of Master of Science

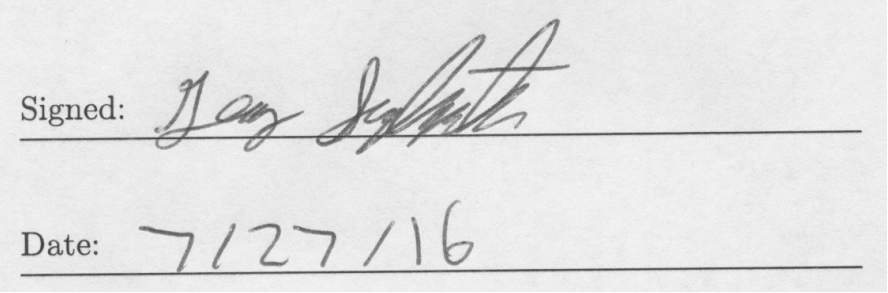
in the
Robotics Institute
Carnegie Mellon School of Computer Science

August 2016
CMU-RI-TR-16-44

Declaration of Authorship

I, Greg Seyfarth, declare that this thesis titled, 'Design, Construction, and Evaluation of a Ballbot with a Spherical Induction Motor' and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.



"If you can't control your peanut butter, you can't expect to control your life."

Bill Watterson

Carnegie Mellon University

Abstract

Robotics Institute
Carnegie Mellon School of Computer Science

Master of Science

by Greg Seyfarth

Ballbots are human-sized, dynamically stable mobile robots that balance on top of a single spherical wheel. They were originally developed to address fundamental problems with locomotion in statically stable mobile robots - statically stable mobile robots must have wide bases and low acceleration to avoid tipping over when moving. This results in slow, clunky, and fat robots. Dynamically stable mobile robots, such as ballbots, avoid this problem by balancing actively. Traditional drive mechanisms for ballbots have been mechanically complex, involving either omniwheels or timing belts and steel rollers. This work details the design, construction, and evaluation of a ballbot driven with a spherical induction motor (SIM). The SIM is extremely mechanically simple, with only the rotor moving with respect to the stators; when used as the drive of a ballbot, it results in a mobile robot with just two moving parts. This work reports both feasibility experimental results for SIMbot (spherical induction motor robot) as well as power efficiency results for SIMbot as compared to the original inverse mouseball drive-based (IMB) ballbot. We show with experimental evidence that SIMbot has comparable performance to the IMB-based ballbot but is, in general, 2-3 times less efficient.

Acknowledgements

This work was made possible by many people. I want to acknowledge Ralph Hollis, my advisor, for all of his help and support. I also want to acknowledge Michael Shomin for his help and guidance. Lastly, I want to acknowledge Dani Solomon and Pickles for their unending emotional support.

Contents

List of Figures

Dedicated to Pickles

Chapter 1

Introduction

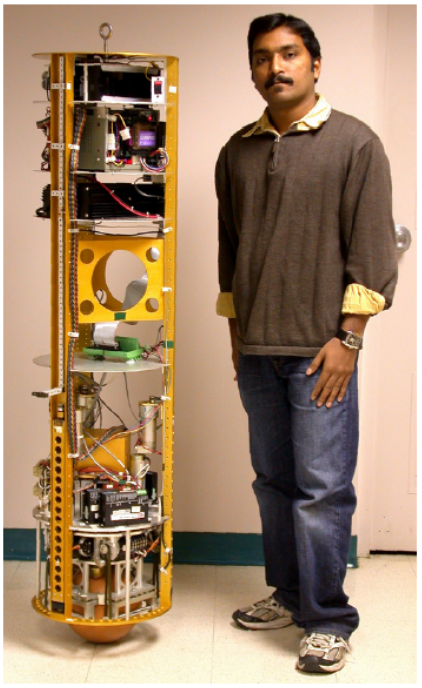


FIGURE 1.1: IMB-based Ballbot built at CMU

As interest in mobile robots for service and domestic use grows, the exact form factor of a *general purpose* mobile robot remains an open question. Since one would likely want such a general purpose mobile robot to operate in environments which were designed by and for humans, a human form factor seems to be one obvious answer. In practice, this presents many engineering complexities. Modern legged humanoid robots have many degrees of freedom and are extremely mechanically, electrically, and algorithmically complex. A wheeled robot with three or more points of contact on the floor is much simpler and much more feasible, although these types of robots come with a different set of problems. This kind of mobile robot, which does not have to expend energy to stay upright, is called *statically stable*. A tall, slender, statically stable wheeled robot designed to have a human form factor must have a very wide base or risk tipping over when accelerating or attempting to pick up a heavy object. If the vector sum of gravity and the robot's acceleration vector passes outside of its wheel base, this kind of mobile robot will experience a torque which causes it to tip over. As a result, modern statically stable mobile robots must have a wide wheel base and limited acceleration. This makes them non-ideal for cluttered

human environments, as their large footprints take up a lot of space on the floor and cause difficulty in fitting through tight spaces such as doors.

An alternative to static stability is dynamic stability. A dynamically stable mobile robot actively balances and must expend energy to stay upright but does not need a large wheelbase to avoid falling over. This makes it possible to build a tall, slender mobile robot with a human form factor that has a small footprint and is still able to accelerate quickly and be agile.

Ballbots are dynamically stable mobile robots that balance on top of a single spherical wheel. They were originally introduced in 2005 specifically to address the problems with statically stable mobile robots outlined above [5]. A photo of the original Ballbot developed at Carnegie Mellon University (CMU) can be seen in Figure 1.1. In addition to the primary benefits of a tall, slender form factor, a small footprint, and fast and graceful motion, ballbots also benefit from omnidirectionality; natural physical compliance - bumping into a ballbot simply results in the ballbot rolling away; and the ability to use their own weight to apply forces to objects and people by leaning.

1.1 Ballbot drive mechanisms

Over the years, two primary types of drive mechanisms have been used in ballbots - the inverse mouseball drive (IMB) and omni-wheel based drive mechanisms. The main focus of this work is the demonstration of a novel third type of drive mechanism - a spherical induction motor.

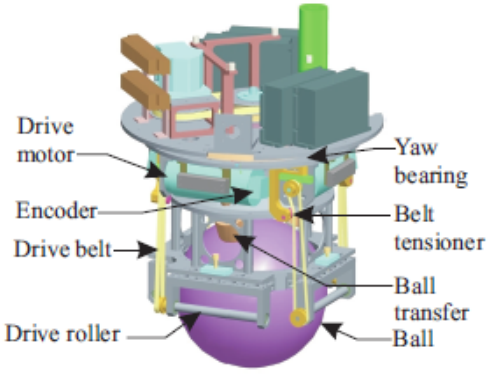


FIGURE 1.2: Inverse mouseball drive (IMB) [1]

1.1.1 Inverse mouseball drive

The CMU Ballbot relies on an inverse mouseball drive, or IMB. The IMB is shown in Figure 1.2. Four timing belts connect dc motors to stainless steel rollers. The stainless steel rollers squeeze the polyurethane covered ball and transmit the motor rotation to the ball. This drive system has proven very successful over the years, but suffers from two main disadvantages. The first is that it is reasonably mechanically complex; for instance, the timing belts can

wear and require maintenance. The second is that it has excessive friction. The rollers that are parallel to the direction of motion of the ball must slip along the ball's surface.

1.1.2 Omniwheel drive systems

The second primary type of drive system used in ballbots is an omniwheel-based system. In this drive system, typically three omniwheels support the weight of the robot and rest on top of the ball. The first use of omniwheels in a ballbot that the author is aware of is BallIP, shown in Figure 1.3 [2]. Rezero is a second example of an omniwheel-based system and was developed shortly after BallIP at ETH Zurich [6].

Omniwheel drive systems have their own set of advantages and disadvantages. They do not suffer from the same friction problem as the IMB and do not need timing belts. On the other hand, the omniwheels themselves are extremely mechanically complex and consist of hundreds of tiny moving parts. Furthermore, the omniwheels must be of a certain minimum size to have the load capacity to support the entire weight of the robot in designs with omniwheels on the top of the ball. This makes it difficult to obtain a sufficient speed reduction between the motors and the ball, which in turn often results in the use of gears. Note that the IMB benefits from a large gear reduction between the motors and the ball due to the presence of the stainless steel rollers. Gears introduce friction and backlash.

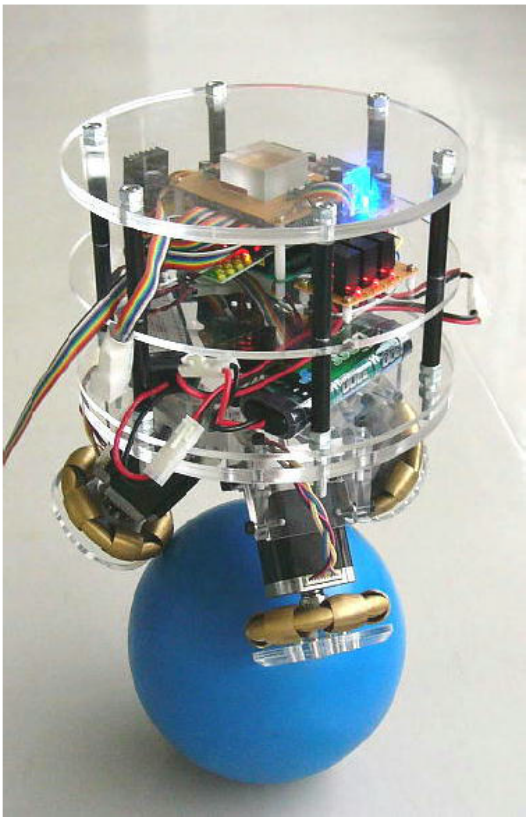


FIGURE 1.3: BallIP, an omniwheel-based ballbot. [2]

Additionally, the body of the robot can turn freely with respect to the ball; this introduces an additional control challenge as the robot must use its drive system to maintain a constant heading. In the IMB, the body does not turn freely with respect to the ball and a separate yaw mechanism is used to control the heading of the robot.

1.1.3 Spherical induction motor

The primary focus of this work explores the application of a novel spherical induction motor developed previously in the Microdynamic Systems Laboratory as the drive system of a ballbot [3]. A short overview is provided here to motivate the primary research question, but a more detailed and technical overview can be found in Section 2.

The spherical induction motor (SIM) is shown in Figure 1.4. It consists of a hollow steel ball with an outer copper shell which rests on a set of six passive nylon ball transfers inside the motor frame. Six stators are positioned around the circumference of the ball. Each stator is driven by a custom three-phase motor driver which controls the amount of force applied to the ball by the stator.

The SIM is extremely mechanically simple and the motivation for using it as the drive system for a ballbot is to create a high-performance mobile robot with just two moving parts.

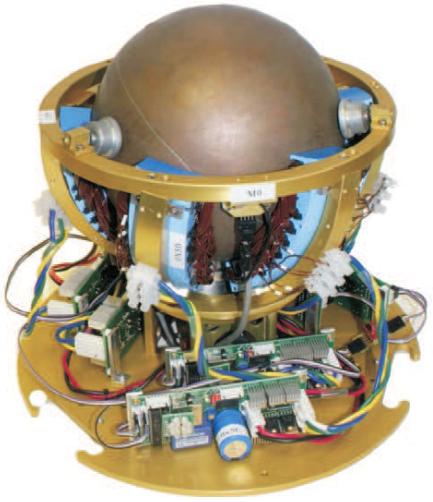


FIGURE 1.4: Spherical induction motor (SIM) [3]

1.2 Research question

The goal of this research is to address how a ballbot driven with a spherical induction motor differs from a ballbot driven with an IMB with respect to power efficiency as well as performance.

Our working hypothesis is that SIMbot's performance will generally fall below that of the IMB-driven ballbot.

It is known from prior work that the SIM is only capable of generating 8 Nm; this limitation was primarily due to the current carrying capability of the drive electronics. New drive electronics were designed (see Sec. 2.2) and able to demonstrate up to roughly 14 Nm with the primary limitation being the battery voltage, but this still falls significantly below the 40 Nm that the IMB-based ballbot is able to generate. Furthermore, the step response of the SIM with respect to a torque input is on the order of 100 ms, which is expected to degrade control performance [3]. This rise time is due to the inductance in

the stator windings. The SIM also suffers from stator end-effects - the magnetic field at the ends of the stators cannot be tightly controlled.

From an efficiency perspective, the SIM benefits from much improved friction relative to the IMB as the SIM has no rollers which squeeze the ball. This may also improve the performance of the controller. Additionally, the SIM is direct drive, so there is no power loss due to gearing or a transmission.

To investigate these trade-offs, two sets of experiments were designed and carried out. The first set was intended to demonstrate the feasibility of a ballbot driven with a spherical induction motor. These are detailed in Section 4.1. The second set was designed to address the performance and efficiency questions directly. The set of experiments detailed below were originally proposed by Ralph Hollis. For the experiments described below which directly compare SIMbot to the IMB Ballbot, additional mass will need to be added to SIMbot to ensure a fair comparison between the two drive systems as SIMbot is lighter than the IMB Ballbot.

1.2.1 Efficiency while balancing

A dynamically stable mobile robot must expend energy to stay upright - minimizing this energy expenditure is important as the robot will likely spend much of its time simply balancing in one place in normal operation. In theory, the energy expenditure of a ballbot should approach zero under the condition of no external disturbances while balancing; in practice, of course, the fact that the robot is balancing about an unstable equilibrium combined with inevitable error and noise in the orientation estimate will lead to appreciable energy expenditure while balancing. To compare the energy expenditure of SIMbot and the IMB-based ballbot while balancing, the current drawn by the motors and the voltage across the motors will be measured while the robots are operating.

1.2.2 Peak acceleration

A unique advantage of ballbots is that they are naturally physically compliant; if a person walks into a ballbot, the ballbot simply rolls away. In order to maintain balance under such a disturbance, the ballbot needs to be able to accelerate quickly enough to keep the ball under the center of mass of the body. In order to evaluate the peak acceleration of SIMbot relative to the IMB-based ballbot, an Imada force gauge will be used to measure impulses applied to the two robots. Energy efficiency as well as performance will be evaluated during this experiment.

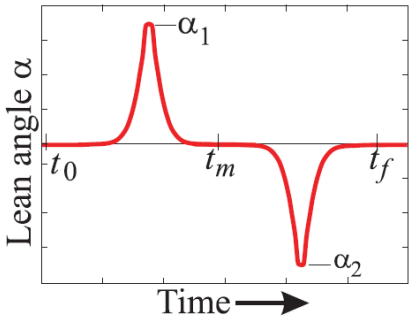


FIGURE 1.5: An open loop hyperbolic secant trajectory [1]

1.2.3 Point-to-point motions

A basic task for robots is to move from point to point. To evaluate energy efficiency and performance in doing so, hyperbolic secant lean angle trajectories of the form

$$\alpha(t) = \alpha_1 \operatorname{sech}(k\frac{2t - t_m - t_0}{t_m - t0}) + \alpha_2 \operatorname{sech}(k\frac{2t - t_f - t_m}{t_f - tm}) \tag{1.1}$$

where $t_m = (t_f + t_0)/2$ and k and α_1 and α_2 are constants. A plot of this trajectory can be seen in Figure 1.5. These lean angle trajectories will be commanded in an open-loop style with no feedback based on ball position. This will allow for an accurate comparison between the two robots.

1.2.4 Top speed

A final metric for characterizing the performance of the IMB-based ballbot and SIMbot is top speed. The IMB-based ballbot has already demonstrated speeds comparable to human walking. Trajectories with gentle accelerations and over long distances ($>$ 4 m) will allow for the measurement of the top speed of both robots.

1.3 Related work

A number of ballbots have been built and tested over the past ten years. The original was the IMB-based ballbot built at CMU [5]. See [1] for the most recent published overview of this ballbot. BallIP was constructed by Masaaki Kumagai at Tohoku Gakuin University in Sendai, Japan. It was the first example of an omniwheel based design [2]. Rezero, another omniwheel-based design, was built at ETH Zurich [6].

The SIM itself was originally introduced in 2013 [7]. Prior to its development, the sensing scheme used to track the ball velocity was developed in [8], and a linear induction motor was developed in [9]. Vector control methods for linear induction motors were investigated and developed in [10]. A larger, more capable SIM, intended for driving a ballbot, was developed in [3].

Prior to the development of the SIM, a number of spherical motors had been developed, but none of them had the torque or speed characteristics necessary for driving a ballbot [11] [12] [13]. To the best of the author's knowledge, this work outlines the first exploration of a ballbot driven with a spherical induction motor.

1.4 Chapter 1 Conclusion

In this section, the main types of ballbot drive mechanisms were introduced. The two primary types of drive mechanisms - omniwheel-based systems and the IMB - both have proven successful over the years, but each is reasonably mechanically complex and has room for improvement. A new drive system, the SIM, was briefly introduced and its potential advantages and disadvantages were discussed. It is hypothesized that the performance of the SIM-based ballbot, or SIMbot, will fall below that of the IMB-based ballbot due to lower torque, stator end-effects, and a relatively large rise time with respect to torque control. It is also hypothesized that the efficiency of SIMbot may be higher than that of the IMB-based ballbot due to lower friction in the drive system. A list of experiments designed to evaluate the performance of a ballbot driven with the SIM were presented. The experiments are designed to test top speed, peak acceleration, and power efficiency while balancing and performing point-to-point motions.

Chapter 2

Spherical Induction Motor

2.1 Spherical Induction Motor Overview

This section outlines the design, construction, modeling, and control of the spherical induction motor. The particular SIM described in this section was originally presented in [3], and the details of the SIM are included here for completeness.

2.1.1 Design and construction

A picture of the SIM is shown in Fig. 2.1. Note that this particular SIM was constructed and described in [3]. The SIM consists of the ball, or rotor, the six stators, the supporting frame, and a set of three optical gaming mouse sensors which are used to track the angular velocity of the rotor.

The ball consists of a hollow soft steel 6.35 mm thick core plated with a 1.3 mm copper shell. The hollow soft steel core was constructed by oven brazing together two steel hemispheres and then electroplating the outside of the resulting sphere with solder. The copper hemispheres were similarly electroplated on their interiors. The two copper hemispheres were then shrink-fitted over the steel core by cooling the core,

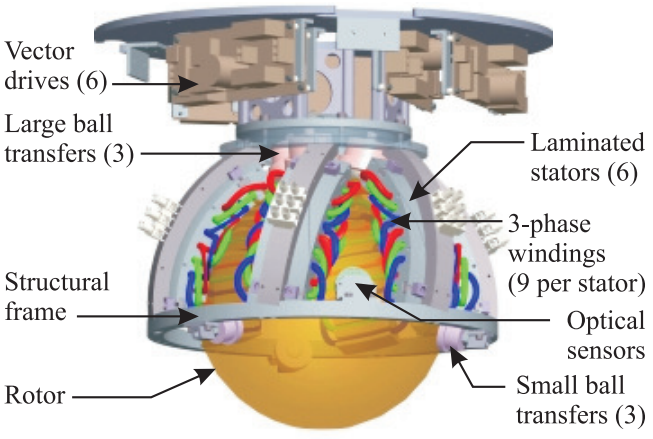


FIGURE 2.1: Spherical induction motor (SIM) CAD model from [4] but originally appearing in [3].

(a) Figure A (b) Figure B

FIGURE 2.2: On the left: The crack in the ball. On the right: Demonstration of closing the crack by hand

(a) Figure A (b) Figure B

FIGURE 2.3: On the left: The ball with the top hemisphere removed and set aside. On the right: The top of the ball after having the hemisphere removed.

heating the copper hemispheres
to just below the melting point
of the solder, and then clamping the hemispheres onto the steel core in an oven in order to reflow the plated solder and fuse the copper and steel together.

The six stators were constructed from stacks of laser cut electrical steel bonded together with 3M Scotchcast. They were designed to have a nominal 1 mm spacing from the rotor. Each stator has 12 slots which are wound with 19 AWG double polyimide insulated wire in a three-phase, four-pole delta winding scheme. Note that the stators are skewed at 10° from the rotor polar axis; this allows the SIM to produce torques around all three axes, while biasing it towards motion in the x- and y-axes.

The supporting frame was constructed from 7075-T6 aluminum. It also has mounting points for the six nylon ball transfers used to support the rotor as well as the three optical gaming mouse sensors. The main mounting deck has mounting points for the six vector drivers. Each vector driver runs a variant of a sensorless field oriented controller, which is described in more detail in Section 2.1.4.

2.1.2 Ball failure

In the course of this work, the ball used by SIMbot was discovered to have cracked along the great circle joining the two copper hemispheres. A photo of the crack is shown in Fig. 2.2(a). It was hoped that the copper hemispheres were securely bonded to the iron core by the solder in between the core and the hemispheres, but it was possible to expand and contract the crack by hand - see Fig. 2.2(b).

In order to diagnose the problem, this particular ball was split open with a screw driver. One of the copper hemispheres was securely bonded to the iron core, but the other was not and was removed. A photo of the copper hemisphere and ball with the hemisphere removed can be seen in Fig. 2.3(a). A noticeable amount of rust can be seen towards the top of the ball, as well as a liquid-like grey substance on roughly one half of the hemisphere and the core in Fig. 2.3(b).

2.1.3 Modeling

The modeling scheme for the SIM comes from [3] and was originally developed, though for a different arrangement of stators, in [7]. The SIM is modeled as having six forces applied to the rotor, with each force originating at the center of a stator. The forces are assumed to be parallel to each stator. It is possible [3] to derive a linear mapping from the six stator forces F to the torque on the ball τ. Specifically, let $F \in \mathbb{R}^6$ be the vector consisting of the six stator forces, and let $\tau \in \mathbb{R}^3$ be the three-dimensional torque applied to the ball. Then we can derive a matrix M_a such that

$$M_a F = \tau \tag{2.1}$$

To apply a particular desired torque τ_d to the ball, we use the pseudo-inverse of M_a, denoted M_a^+, to obtain the vector F with the shortest L_2 norm that satisfies $M_a F = \tau_d$. This makes sense in practice as we could reasonably assume that the power drawn by the motor is proportional to the length of the force vector F.

The rotor angular velocity is sensed by the three mouse sensors mounted about the equator of the rotor. This particular scheme and modeling approach was originally developed in [8] and re-used again in [3] and [7]. The three mouse sensors sense surface velocities v_x^i and v_y^i for i $= 1, 2, 3$. We can similarly [3] derive a linear mapping from the angular velocity of the rotor $\omega \in \mathbb{R}^3$ to the six surfaces velocities contained in a vector $V \in \mathbb{R}^6$, resulting in

$$M_s\omega = V \tag{2.2}$$

where M_s is the sensor matrix. Since $M_s \in \mathbb{R}^{6x3}$, this system is overconstrained and multiplying the pseudo inverse of the sensor matrix, denoted M_s^+, with the measured surface velocities results in a least-squares solution for the rotor angular velocity.

2.1.4 Control scheme

The six stators are each force-controlled by an associated vector driver. The vector drivers run a variant of a sensorless field oriented controller previously developed in [10]. At a high level, field oriented control works by splitting the magnetic field generated by the stators into two components, a q component and a d component, which separately control the torque and the magnetic field induced in the rotor. Modulating the q component then allows for straightforward force (or torque, in a traditional single-axis motor) control, and the force generated by the stators generally follows

$$F \sim I_q I_d \tag{2.3}$$

where I_q and I_d are the commanded currents along the q and d axes. The d component of current is generally referred to as the magnetizing current and generates zero torque but must be present for the motor to work - this results in relatively high power consumption even at zero torque.

2.2 Vector Drivers

The previously described control scheme is implemented on a circuit board, referred to as a vector driver. A high-level block diagram of the vector driver can be seen in Fig. 2.4. Block diagrams with much more detail can be found in sections 2.2.1.1 and 2.2.1.2. The microcontroller (a dsPIC33FJ64MC202-E/SO) implements the control scheme at 10 kHz. It receives current measurements for the three phases, labeled I_R, I_S, and I_T, for phases R, S, and T, respectively, via three Hall effect current sensors (Allegro ACS758). These current readings are pushed through

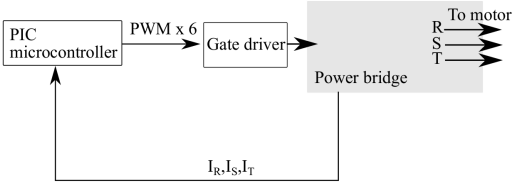


FIGURE 2.4: Block diagram of entire vector driver

the control scheme on the microcontroller, which then sets the duty cycles for three complementary PWM signals (six total PWM outputs) labeled 1H, 1L, 2H, 2L, 3H, and 3L. These signals are inverted relative to the desired MOSFET states; a high PWM signal from the microcontroller corresponds to a MOSFET being non-conductive. The PWM signals are input into a three-phase gate driver which works in conjunction with a charge pump to apply the PWM signals to the six MOSFET gates. The six MOSFETs are wired in a standard three-phase power bridge configuration. A shunt resistor on the low side of the power bridge provides over-current monitoring. The gate driver has an over-current protection pin which will halt the gate signals if the voltage applied to this pin is over a certain threshold (generally about 0.4 V).

There have been three different designs of the vector drivers in use by the SIM.

The first version was designed by Masaaki Kumagai, a professor from Tohoku Gakuin University in Japan who worked in the Microdynamics Systems Laboratory for a year. In addition to all of his work on the SIM, Masaaki also constructed BallIP [2], shown in Fig. 1.3. This version of the vector driver worked reliably, but was not designed to carry high currents. This limits the amount of torque that the SIM can produce. A photo of this version of the vector driver can be seen in Fig. 2.5.

The second version was designed by Ankit Bhatia, a masters student working in the Microdynamics Systems Lab, at the end of 2014. The major difference between this version and Masaaki's version is the use of heavier 2 oz copper and much wider power bridge traces; both of these differences are intended to facilitate higher currents. Other small improvements include on-board rotor flux angle LED indicators, higher current Hall effect sensors, and a 12 V dc-dc converter that accepts inputs of up to 140 V.

FIGURE 2.5: Vector driver designed by Masaaki Kumagai

The main problem with this version was electrical noise, presumably from the MOSFET switching. The noise was so extreme that it consistently caused the microcontroller to reset itself at currents past roughly 5 A. A plot of the microcontroller supply voltage, measured directly at the supply and ground pins, can be seen

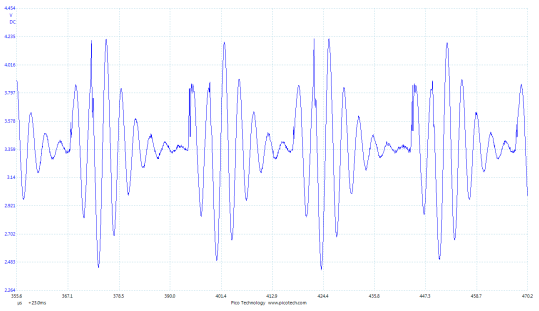


FIGURE 2.6: Noise on the supply line of the PIC microcontroller with motor running at a DC current of 5.8A. The nominal PIC supply voltage is 3.3V.

in Fig. 2.6. A number of passive filter solutions were attempted but none were found which worked reliably. After several months of work on this version of the board, it was finally decided to split the control and power portions of the board into two boards for isolation purposes. In addition to electrical isolation, a two board solution also improves modularity - the control and power boards are all interchangeable and any failures are better isolated.

The third version of the board was designed by the author and Olaf Sassnick, a visiting student from Austria. The author designed and laid out the control board, and Olaf Sassnick designed and laid out the power board. After an initial build, the author then made a number of changes to the power board to improve ringing and fix other signal conditioning issues, resulting in a version 3A.

2.2.1 Vector Driver Rev 3A Overview

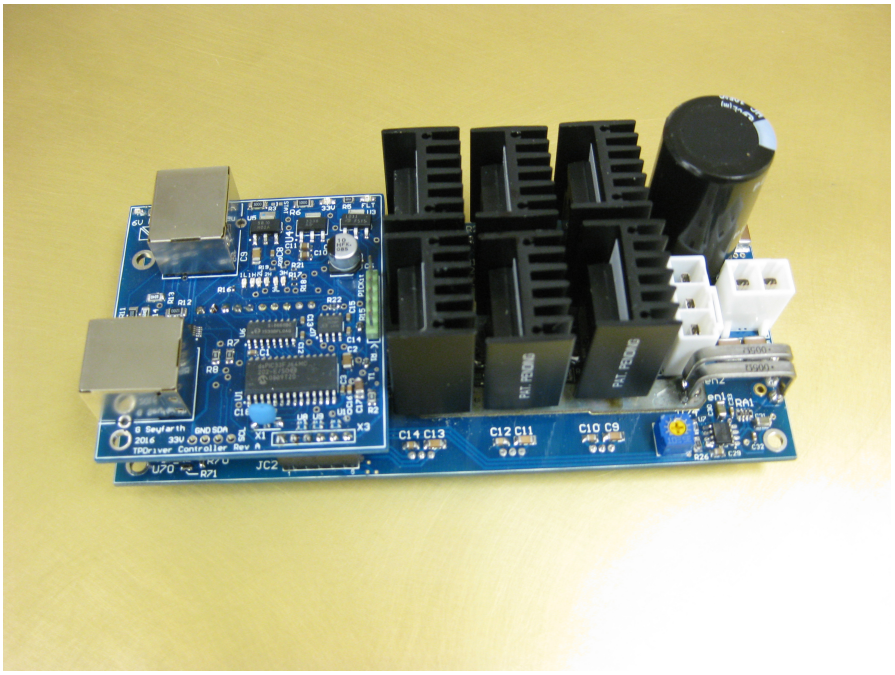


FIGURE 2.7: Photo of the completed Rev 3A vector driver. The control board is stacked on top of the power board.

The following two subsections give a more in-depth overview of the control board and power board. Rev 3A consists of two boards, a control board and a power board, which fit together with a 6-pin header and a 10-pin header. A photo of the completed board can be seen in Fig. 2.7.

2.2.1.1 Control Board

A block diagram of the control board can be seen in Fig. 2.8, and a photo of the board can be seen in Fig. 2.9.

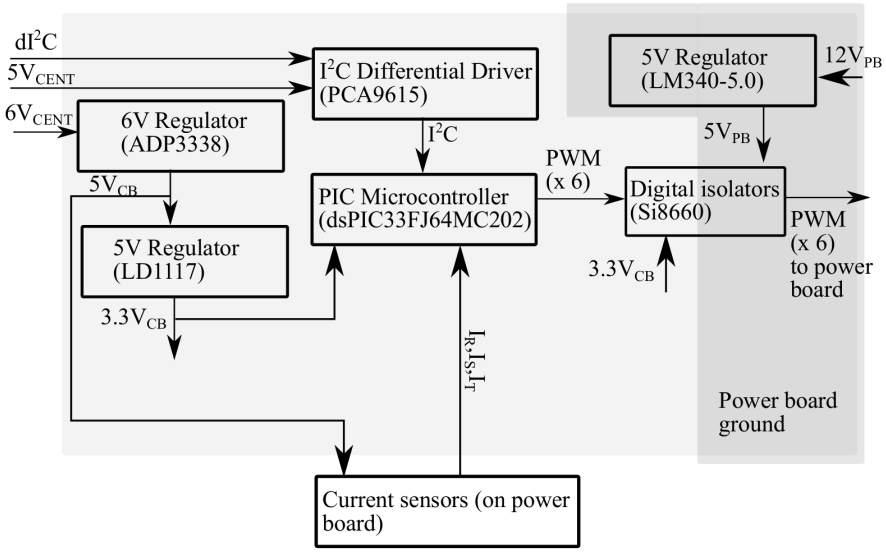


FIGURE 2.8: A block diagram of the control board.

The board is powered from an external 6 V supply, labeled $6V_{CENT}$ in the block diagram, delivered via the RJ45 connectors from the central I^2C distribution board. An external 5 V signal, labeled $5V_{CENT}$, is used only for the differential I^2C signal. The 6 V signal is converted to 5 V, labeled $5V_{CB}$, which exits the board through the lower 6 pin header and powers the three Hall effect sensors on the power board. The feedback signal from the Hall effect sensors is dropped to 3.3 V via a voltage divider before connecting to the ADC on the PIC microcontroller. The $5V_{CB}$ is also converted to 3.3 V, labeled $3.3V_{CB}$, which powers the microcontroller and the various indicator LEDs. A differential I^2C signal, labeled dI^2C, enters the board via the RJ45 connectors, is converted to single-ended I^2C via a differential driver (NXP PCA9615), and finally travels to the microcontroller. The board has pads for differential biasing resistors (R9-R14) which should only be populated if the particular control board is at the end of a differential I^2C run. They help to eliminate signal reflections on the bus. See the NXP PCA9615 datasheet for additional information.

The microcontroller outputs six PWM signals, made up of three complementary pairs, into a digital isolator (Silicon Labs Si8660) which electrically separates the control logic from the high power board. These PWM signals then travel through the 10 pin header, down to the power board, and into the gate driver.

The 10 pin header also provides a 12 V supply, labeled $12V_{PB}$, from the power board. This 12 V supply is converted to 5 V, labeled $5V_{PB}$, on the control board and is then used to supply the power board side of the digital isolators as well as six LEDs indicating the rotor flux angle. Note that both of these signals physically exist on the control board but are electrically isolated from the control logic via the digital isolators. The LEDs are driven by a hex inverter in order to avoid affecting the PWM signals - it was observed that the presence of the LEDs resulted in a strange quad-state PWM signal (instead of a binary on-off signal) and using the hex inverter protects the PWM signals from the LEDs. The microcontroller can be reprogrammed with a PicKIT via the 6 pin header labeled 'PICKIT' along the right edge of the board.

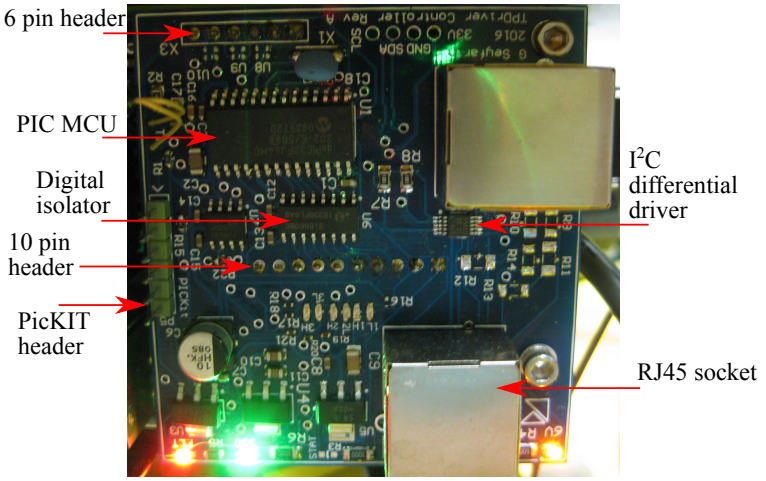


FIGURE 2.9: Photo of the control board.

The control board has 10 LEDs in total. Six are the previously mentioned rotor flux indicator LEDs. There is also an orange LED indicating the presence of the 6 V supply, a green LED indicating the presence of the 5 V supply on the control board side, a blue LED which is connected to a free pin on the microcontroller and which can be used for debugging, and a red LED which indicates a gate driver fault. If the power board is not receiving 48 V, then the fault LED will be on.

2.2.1.2 Power board

A block diagram of the power board can be seen in Fig. 2.10. The board receives an input of 48 V (but should be able to handle 96 V as well) which gets converted to 12 V by a switching DC-DC converter. The 12 V supply powers the gate driver and the overcurrent amplifier. The gate driver receives the six (inverted) PWM signals via the 10-pin header to the control board. The signals are inverted by the gate driver and sent to the MOSFET gates. The high side signals operate with a charge pump to bring them to the input voltage + 12 V in order to turn on the high side MOSFETs.

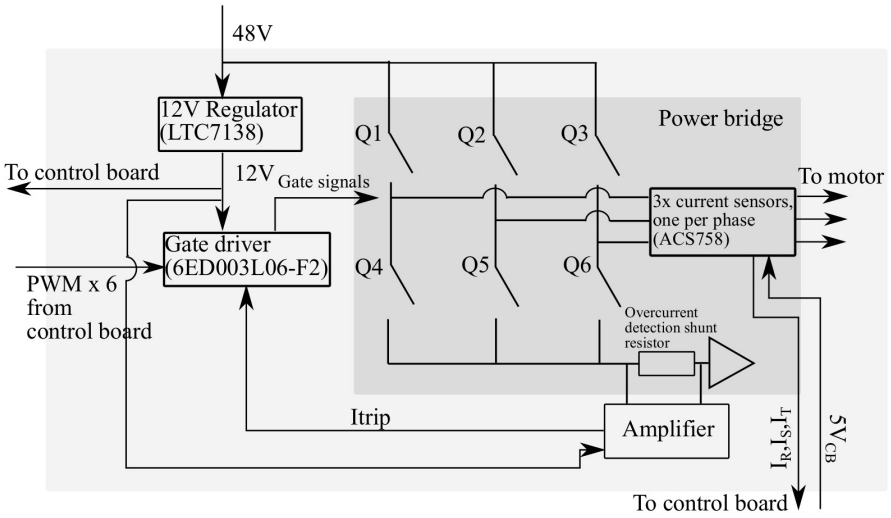


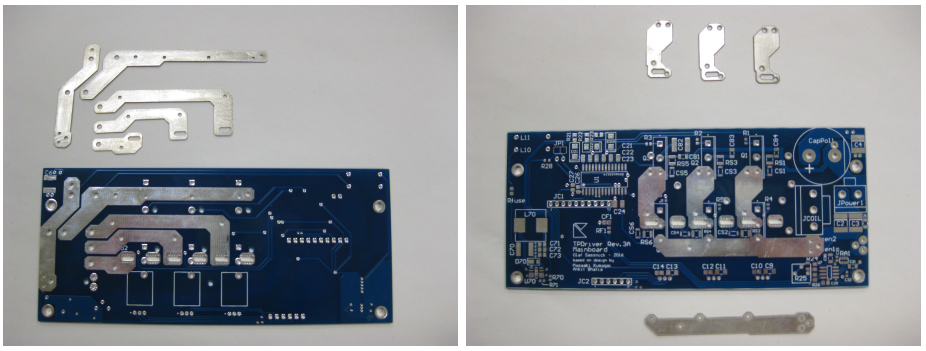
FIGURE 2.10: A block diagram of the power board.

The six MOSFETs are wired in a standard three-phase power bridge configuration, with the midpoint of each pair of MOSFETs going out to the motor windings. Each MOSFET has an RC snubber across its drain and source pins to suppress EMI and avoid overshoot when they are switched on. The current through each of the three phases is measured by a Hall effect sensor. The measurement pins of the sensor are isolated from the power, ground, and output pins. The sensors are powered by the control board and also send their measurements back to the control board via the 6 pin header.

The MOSFETs used in this revision of the power board have roughly one half of the 'on' resistance of the MOSFETs used in version 1. This drastically reduces the power lost to heat and, combined with larger heatsinks and the copper plating described below, helps to eliminate overheating problems.

2.2.1.3 Copper plating

A main feature of the power board is the copper plating used on the bridge. Manufacturing circuit boards from heavy copper (more than 2 oz) is expensive, and it is also extremely difficult to get heavy copper boards manufactured with plated through holes and/or fine pitch pads and traces. Only the bridge traces need to support large currents, but there are a number of supporting components with very finely spaced pins or pads; the Hall effect sensors, for example, have three finely spaced pins (Vdd, Gnd, and Vout), and the sensors must be placed physically near the bridge.



(a) (b)

FIGURE 2.11: On the left: Power board Rev 3A, back, with matching copper plates On the right: Power board Rev 3A, front, with matching copper plates

To allow for high currents on the bridge, plated through holes, and fine pitch components all on a single circuit board, it was decided to have 20 mil (0.020 inches) thick multi-purpose 110 copper sheets lasercut into small plates that match the traces on the bridge. The lasercutting was done by FedTech at a price of $302.40 for 20 sets of plates. The matching traces on the bridge were left exposed with no solder mask covering them. Additionally, small holes for rivets were placed in the traces and matching holes were cut into the copper plates.

Fastening the plates to the boards consisted of 1) applying solder paste to the exposed bridge traces on the circuit board, 2) fastening the copper plates to the circuit board with rivets, and 3) baking the board in a reflow oven to heat all of the solder paste uniformly. As there are plates on both sides of the circuit board, the rivets are necessary to hold the plates on the bottom of the board during the reflow process. Attempting to solder the plates to the board by hand would be impractical as the plates are very large and difficult to heat with a single iron. See Fig. 2.12 for a photo demonstrating the method of attaching the plates to the board. Note that the nylon spacers in the photo are only used to hold the plate up to expose for the solder paste for the purposes of the picture.

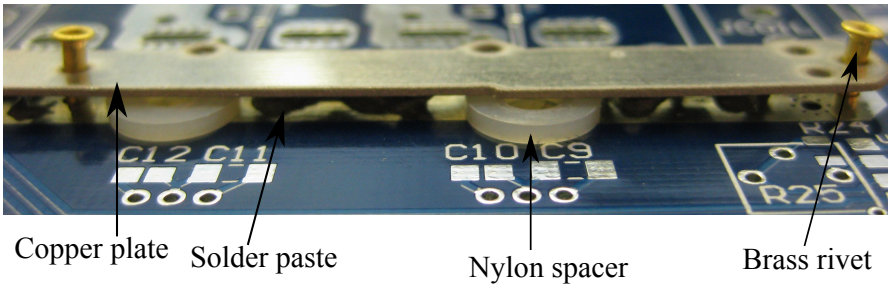


FIGURE 2.12: Photo of the method used to fasten the plates to the board. The nylon spacers are not involved in the fastening process and are only used to hold the plate up in order to show the solder paste.

To improve solderability and to prevent the copper from corroding over time, the copper plates were bathed in a liquid tin solution (MG Chemicals Liquid Tin, solution 421) to plate them in tin. A picture of the tin-plated copper plates can be seen in Figs. 2.11(a) and 2.11(b).

2.3 Chapter 2 Conclusion

A more technical overview of the spherical induction motor (SIM) was given. The construction of the stators, windings, and frame was discussed. The angular velocity of the SIM is sensed using a set of three optical gaming mouse sensors. Torque control is obtained by approximating each stator as applying a force parallel to the stator's long dimension and deriving a linear mapping from the set of six stator forces to the torque applied on the ball. The design of a new set of vector drivers was presented; the new vector drivers split the high-power and control sections of the board onto two physically different circuit boards for isolation purposes. The high-power board has custom copper plates fastened to the power bridge with brass rivets and solder for facilitating high currents.

Chapter 3

SIMbot

3.1 SIMbot System Overview

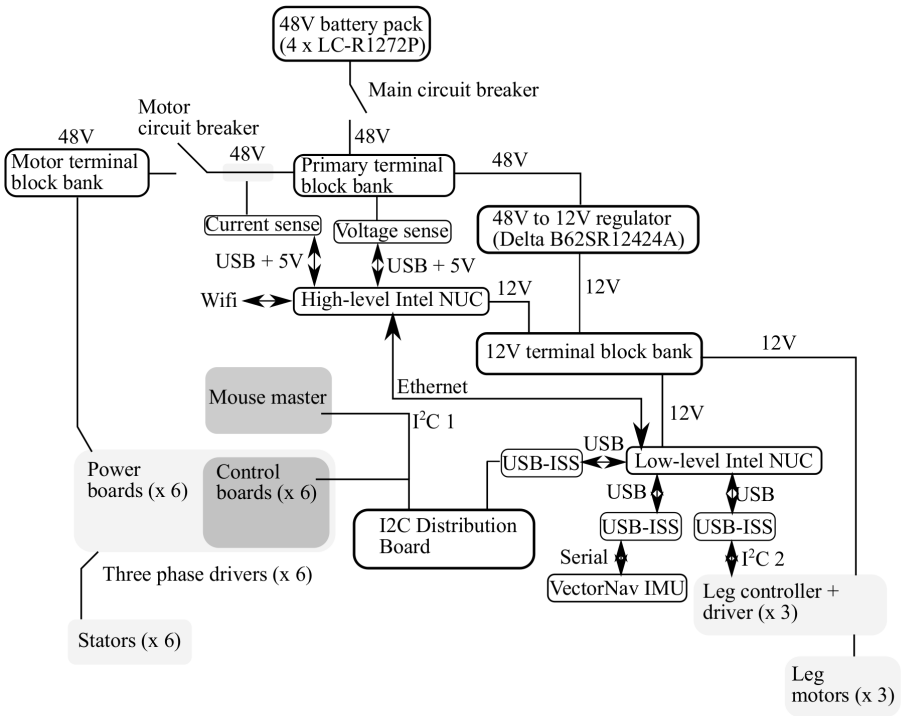


FIGURE 3.1: Block diagram of entire SIMbot system.

The previously described SIM was flipped over and bolted to the bottom of a ballbot frame, resulting in SIMbot. A picture of SIMbot can be seen in Fig. 3.2. A general

block diagram overview of the entire SIMbot system can be seen in Fig. 3.1. Portions of this chapter come from a previously published paper [4].

3.1.1 Sensing

SIMbot uses a VectorNavTM VN-100 inertial measurement unit (IMU) for sensing body angles and body angular rates. The VN-100 has an onboard Kalman filter which provides estimates of the sensor attitude.

SIMbot relies on the previously described set of three AvagoTM ADNS-9800 optical gaming mouse sensors, shown in Fig. 2.1, positioned at 120° increments along the equator of the ball to sense the ball's angular velocity. The mouse sensors report the number of counts of displacement at 134 counts per millimeter by tracking the surface texture of the ball's surface.

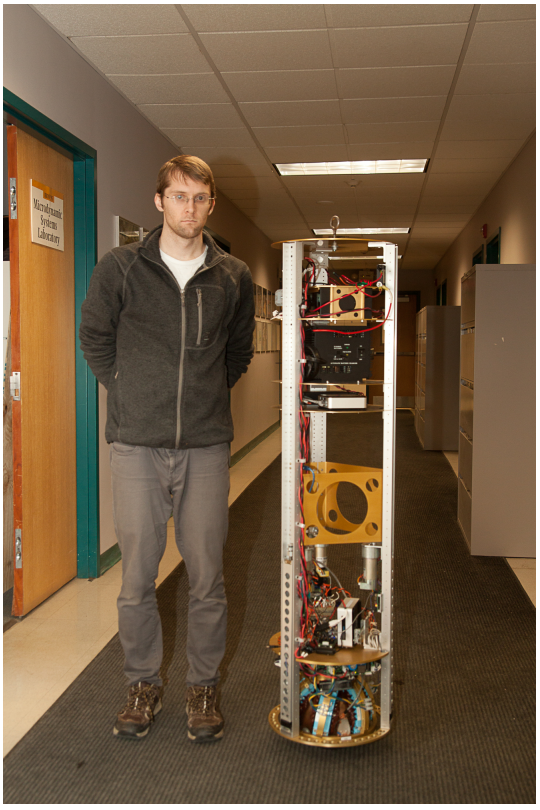


FIGURE 3.2: Photo of SIMbot next to the author.

3.1.2 Communication and processing

SIMbot uses an Intel^{TM} NUC 5i5MYHE running the QNX real-time operating system to perform all balancing calculations. This real-time computer communicates with the VectorNavTM VN-100 IMU via serial, the leg motor drivers via an I2C bus, and the SIM vector drivers and mouse interface via a second I2C bus. The mouse interface is an intermediate PIC microcontroller which reads the three individual mouse sensors via an SPI bus. We convert the real-time computers native USB to I2C and serial with Devantech USB-ISS modules.

The real-time computer closes the balance controller loop at 200 Hz, currently limited by overhead introduced by converting from USB to I2C. It is expected that the control loop rate can be improved by switching to native I2C communication. An Intel^{TM} NUC D34010WYK performs trajectory generation. This computer will be used for other

high-level computations such as navigation and perception in the future. It runs Robot Operating System (ROS) on Ubuntu Linux, giving access to many existing packages for navigation and mapping.

3.1.3 Legs

SIMbot has three legs which are conceptually identical to those used on the IMBdrive ballbot. The legs are used only for stability when SIMbot is powered down. Each leg is driven by a linear drive screw powered by a dc motor with an attached quadrature encoder. The end of each leg has a spring-loaded limit switch with a ball caster. The limit switch detects contact with the floor and with the body when the legs are going down and up, respectively. Each leg is run by a separate motor driver with an integrated PID controller for velocity control of each individual leg.

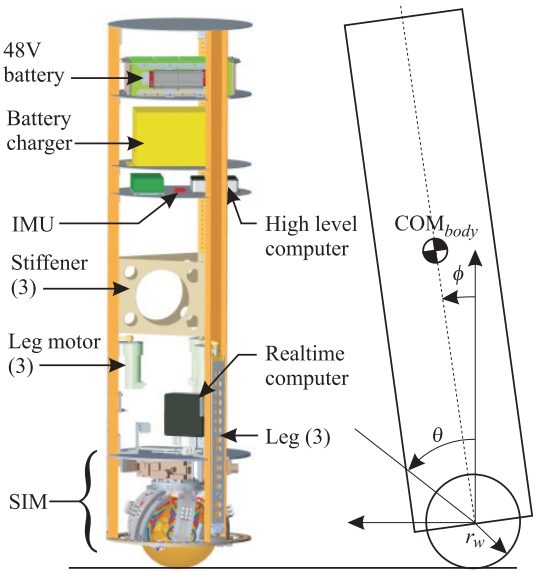


FIGURE 3.3: CAD model of SIMbot and a depiction of the planar model.

3.1.4 Power distribution

The main power is supplied by four 12.7
V 7.2 A-h lead acid batteries (Panasonic LC-R127R2P1) wired in series. These batteries are connected to the six motor controllers to supply them with 48 V as well as an isolated DC-DC converter (Delta Electronics B62SR12424A) which accepts as input 18-106 V and outputs 12.4 V. The 12.4 V supplies the three leg motors and their controllers, both Intel NUCs, and another dc-dc converter which outputs 6V. The 6 V supply is distributed to the six control boards, and is also converted down to 3.3 V to supply the ADNS9800 controller. The power distribution scheme can be seen in the general block diagram for the entire system in Fig. 3.1.

3.1.5 Custom electronics

In addition to the six motor controllers, a number of other custom circuit boards were developed for SIMbot.

3.1.5.1 Current monitoring

To facilitate power efficiency tests, a custom current monitoring board was developed. The two standard methods of measuring current are 1) measuring the voltage drop across a small resistor, called a shunt resistor, placed in series with the load and 2) using the Hall effect. It was decided to use a Hall effect sensor as it has minimal impact on the circuit under measurement and also allows for isolated measurement; since the Intel NUC, which receives the current readings via USB, is on the other side of the batteries relative to the isolated 12 V dc dc converter, an isolated measurement is necessary.

The board uses an ACS758 Hall effect sensor - the same used on the motor controllers - which is read by an analog-to-digital converter (ADC). Both the sensor and the ADC are powered by a precision voltage reference (REF196). The ADC outputs its readings via SPI through a USB-SPI bridge (Silicon Labs CP2130) which allows the Intel NUC to read the current measurements and publish them to ROS. The ADC is read at 700 Hz and low pass filtered with a cutoff frequency of 300 Hz. A block diagram of the board can be seen in Fig. 3.4.

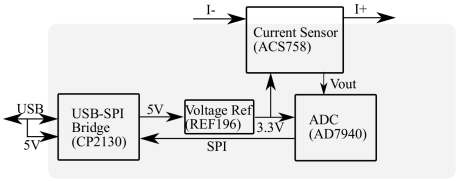


FIGURE 3.4: Block diagram of the current sensing board.

3.1.5.2 Voltage monitoring

The voltage supplied to the motor controllers is monitored by an ADC (Linear Technology LTC2471) which outputs its readings over I^2C. A voltage divider drops the motor voltage to the ADC's input range of 0 to 1.25 V. The readings are passed through a digital isolator in order to allow the readings to be read by the high level Intel NUC. The I^2C signal is converted to USB by an FT232H breakout board from Adafruit on the Intel NUC side of the isolation barrier. The voltage readings are then published to ROS by the high level NUC at 100 Hz.

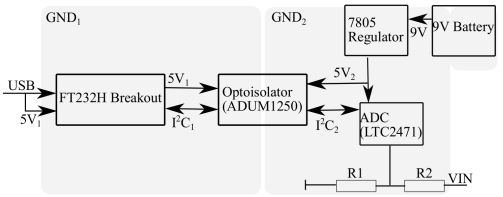


FIGURE 3.5: Block diagram of the voltage sensing board. VIN is the voltage to be measured, and R1 and R2 form a voltage divider.

The high-voltage side of the isolation barrier is powered by a 9 V battery, which is input into a 7805 dc-dc regulator in order to provide 5 V to the ADC and the digital isolator.

This board was built by hand to avoid the turnaround time necessary for custom circuit boards.

A block diagram of the board can be seen in Fig. 3.5.

3.1.5.3 I^2C Distribution Board

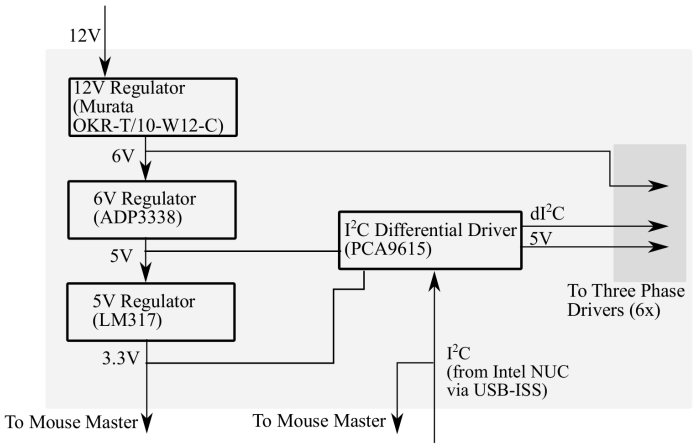


FIGURE 3.6: Block diagram of distribution board

The I^2C bus which runs between the six motor controllers and the mouse master is distributed via a custom distribution board. A block diagram of the board can be seen in Fig. 3.6. The board takes as input a 12 V supply and a single-ended I^2C bus. It outputs 6 V, 5 V, 3.3 V, a differential I^2C signal, and the single-ended I^2C signal. The 6 V rail powers the six control boards. The 5 V rail powers the differential side of the I^2C bus. The 3.3 V rail powers the mouse master board.

3.2 SIMbot Control and Modeling

3.2.1 Dynamics

1) Modeling assumptions: Following the approach of the IMB-based ballbot (see [1], the dynamics of SIMbot are simplified to two linearized planar models, one in the median sagittal plane and one in the median coronal plane. The model is based on several assumptions:

- The motion in the median and sagittal plane is decoupled
- The motion in each of these two planes is identical

- The ball rolls without slipping
- The floor is flat and level

These assumptions are reasonable as the coupling between sagittal and coronal planes is through products of body angular rates and body lean angles, both of which should be very small. As a result, the linearized 3D model is, in fact, decoupled in the sagittal and coronal plane. Using the two planar models, we design two identical stabilizing PID controllers.

3.2.2 Planar model

The SIMbot in each orthogonal plane is abstracted to a rigid rectangle on top of a rigid disk. As shown in Fig. 3.3, state variables θ and ϕ define the ball angle and body lean angle respectively. The equations of motion for the planar ballbot are as follows:

$$M(q)\ddot{q} + C(q,\dot{q}) + G(q) + D(q,\dot{q}) = \begin{bmatrix} \tau \\ 0 \end{bmatrix}, \tag{3.1}$$

where $q = [\theta, \phi]^T$ is the generalized coordinate vector, $M(q)$ is the inertia matrix, $C(q,\dot{q})$ is the Coriolis matrix, $G(q)$ defines the gravitational forces, $D(q,\dot{q})$ are an approximation of the frictional forces acting on the ball and τ is the applied torque between body and ball. These terms have been worked out in detail in numerous previous works; refer to [1]. It is interesting to note that although the SIM itself is overactuated, SIMbot is an underactuated system, having no actuator which directly controls its body lean angle.

3.2.3 Control Architecture

The SIMbot control scheme is based on three nested controllers. The controller at the lowest level is encapsulated by the balancing controller and finally the outer loop controller. A nested yaw controller is responsible for achieving the desired yaw angles. Fig. 3.7 shows the balancing controller nested inside the outer loop controller and Fig. 3.8 details the yaw controller.

3.2.3.1 Vector Controller

The lowest level controller is a field-oriented vector control scheme: one for each of the six stators in the SIM. This was discussed in more detail previously in Section 2.1.4.

3.2.3.2 Balancing Controller

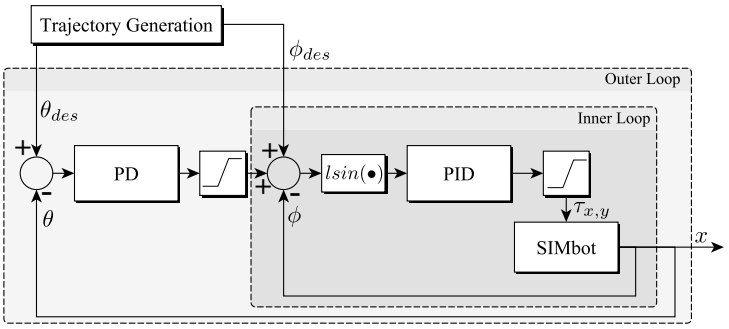


FIGURE 3.7: Balancing and outer loop controller structure

The balancing controller maintains a given lean angle setpoint. This is a conventional hand tuned PID controller identical to the one used on the IMB-drive ballbot [1]. Two instances of this controller handle balancing in each of the orthogonal directions. The balancing controller determines the torque τ to be generated by the SIM in order to maintain the lean angle setpoint ϕ_{des}. A saturation on output torque prevents the controller from requesting more than 8 Nm from the SIM to avoid overstressing a vector driver. After the installation of the new vector drivers (see Sec. 2.2), this limit was raised to 14 Nm. With the new vector drivers, the primary limitation was found to be the battery voltage - see Sec. 4.2.1. A block diagram of the balancing controller can be seen in Fig. 3.7.

3.2.3.3 Outer Loop Controller

The outer loop controller tracks the desired ball position by outputting desired body lean angles. The error between desired ball position θ_{des} and the observed ball position θ is driven to zero using the PD controller. The outer loop controller can act as a station-keeping controller which maintains ball position or as a trajectory tracking controller for point-to-point motions. A block diagram of the outer loop controller can be seen in Fig. 3.7.

3.2.3.4 Yaw Controller

In order to command a desired yaw azimuth of the robot, a nested PID controller commands yaw torques through the actuator matrix. The controller (see Fig. 3.8) consists of an inner loop PI controller that closes the loop on yaw angular velocity and an outer loop PD controller feeding back yaw angle (integrated from yaw angular velocity reported by the IMU) and velocity. The output of both controllers is saturated to prevent high yaw torques.

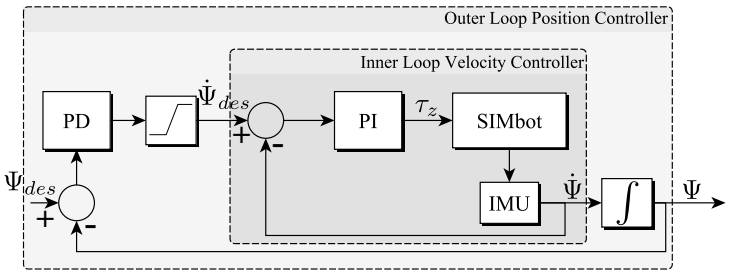


FIGURE 3.8: Yaw control block diagram

3.3 Chapter 3 Conclusion

An overview of the entire SIMbot system was given. SIMbot's main sensor is a VectorNav VN-100, which outputs filtered attitude estimates. SIMbot has a high-level and a low-level computer; the high-level computer runs ROS and handles navigation and planning, and the low-level computer handles the main control loop. Power is provided by four 12V lead-acid batteries wired in series. Custom electronics were developed to measure and report via ROS the current drawn by the motor and the voltage across the motor. SIMbot relies on a nested control scheme. At the lowest level, the six vector drivers run a force controller. The balancing controller, one level above the vector drivers, outputs desired torques in the x-z and y-z plane according to a PID control law. The outer loop controller outputs desired lean angles based on ball position or velocity, which get fed to the balancing controller for tracking.

Chapter 4

Experimental Results

This chapter reports results which 1) demonstrate the feasibility of SIMbot, 2) demonstrate the performance of the new vector drivers, and 3) compare SIMbot to the IMB-based ballbot directly on metrics such as top speed, peak acceleration, and power efficiency. These results can be found in Sections 4.1, 4.2, and 4.3, respectively.

4.1 Feasibility results

This section details a set of experiments published in [4] which demonstrate the feasibility of the SIMbot. Note that all of these experiments were performed prior to the development of the new driver boards and as such the torque on the ball was limited to roughly 8 Nm. Four experiments were performed: balancing while maintaining a zero lean angle, stationkeeping, a short point-to-point motion, and recovery from an initial lean angle. For all experiments, position data was obtained from the mouse sensors on the SIM and orientation data was obtained using the IMU on SIMbot. All experiments were performed on 9.5 mm thick soft foam tiles to protect the soft copper layer of the ball. The tiles add additional rolling resistance which makes it easier for the robot to balance, since the tiles damp the motion of the ball.

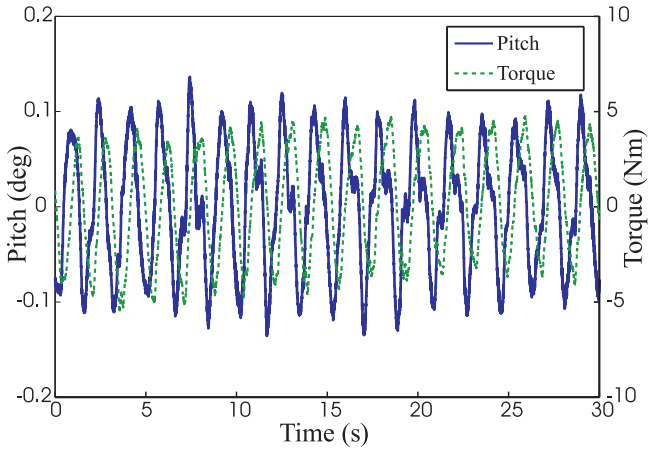


FIGURE 4.1: Balancing performance for a single axis.

4.1.1 Maintaining zero lean angle

See Fig. 4.1 for the pitch angle and torque used to control the pitch angle during thirty seconds of SIMbot attempting to maintain a zero lean angle. This experiment was free from disturbances. SIMbot is able to maintain a zero degree lean angle to within 0.15. Fig. 4.2 shows SIMbots position over the same 30 s. Despite not explicitly controlling for position, SIMbot stays within 15 mm of its starting position during the entire balancing operation.

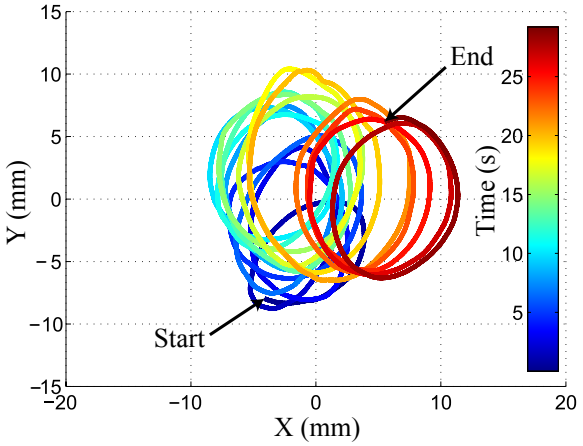
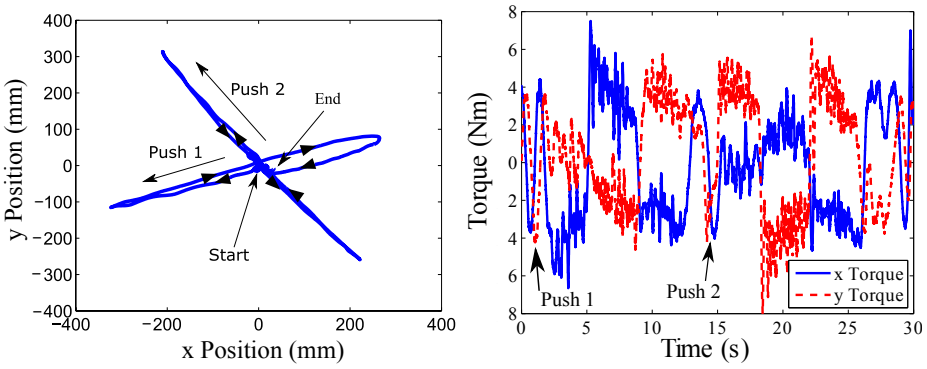


FIGURE 4.2: Position on the floor during 30 seconds of balancing only.

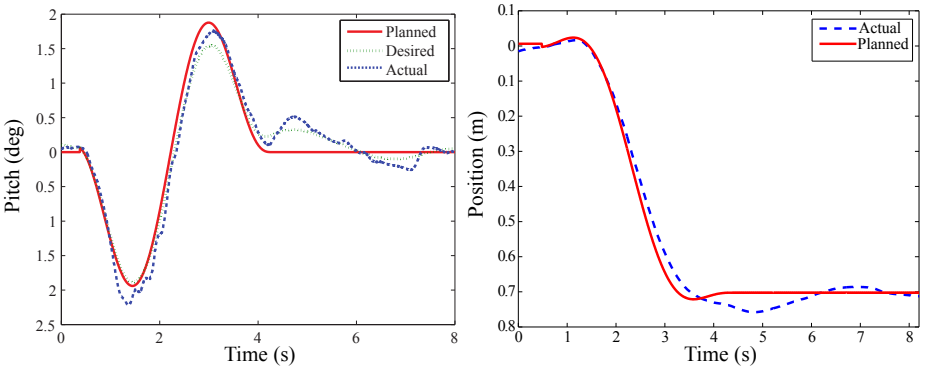
4.1.2 Stationkeeping

Roughly 10 N of force was applied to the robot for about 2 s, once in each of two orthogonal directions, while running the stationkeeping controller. After undergoing initial displacements of up to 0.4 m, SIMbot is able to return to its initial position. See Fig. 4.3(a) and 4.3(b) for a plot of SIMbots position and the torques applied to the ball during the experiment.



(a) Figure A (b) Figure B

FIGURE 4.3: On the left: SIMbot's position on the floor while stationkeeping. On the right: Control input while stationkeeping.



(a) (b)

FIGURE 4.4: On the left: SIMbot's lean angle tracking during a point-to-point motion. Desired is the feedforward lean angle plus a feedback control, planned is the feedforward lean angle only, and actual is SIMbot's lean angle. On the right: SIMbot's position tracking during the trajectory.

4.1.3 Point to point motion

A 0.7 m point-to-point motion was demonstrated with SIMbot using the outer loop controller described in Section III-B to generate and track desired ball trajectories. See Fig. 4.4(a) for body angle tracking results and Fig. 4.4(b) for position tracking results.

4.1.4 Initial lean angle recovery

We tested SIMbot's ability to return to a zero degree lean angle when starting from an initial non-zero lean angle. One experimenter held the robot stationary while another commanded the desired initial lean angle to SIMbot's outer loop controller. SIMbot was then released and a recovery trajectory was planned and executed. The recovery trajectory is designed to take SIMbot from its initial state to a final resting state with

zero lean angle and zero ball velocity. As we are primarily concerned with returning the lean angle to zero degrees in this experiment, we did not include the ball position feedback term in the outer loop controller. SIMbot was able to recover from initial lean angles of up to 3° as shown in Fig. 4.5.

4.1.5 Discussion of feasibility results

When attempting to maintain a zero lean angle, SIMbot's performance is comparable to the IMB-based ballbot in both zero-point motion (the amount of position drift while not explicitly attempting to maintain position) and lean angle error about zero degrees. SIMbot's performance in behaviors requiring more torque such as more aggressive point-to-point motions and recovery from larger lean angles is limited by the vector drivers, which can produce torques of up to approximately 8 Nm. It is encouraging that the pure balancing motion, stationkeeping, and the short-trajectory with a maximum lean angle of less than 2.5° all feasible with the 8 Nm limit. For comparison, the IMB-based ballbot operates in a nominal body angle range of ±5° and demonstrated a 12° body lean angle in [3].

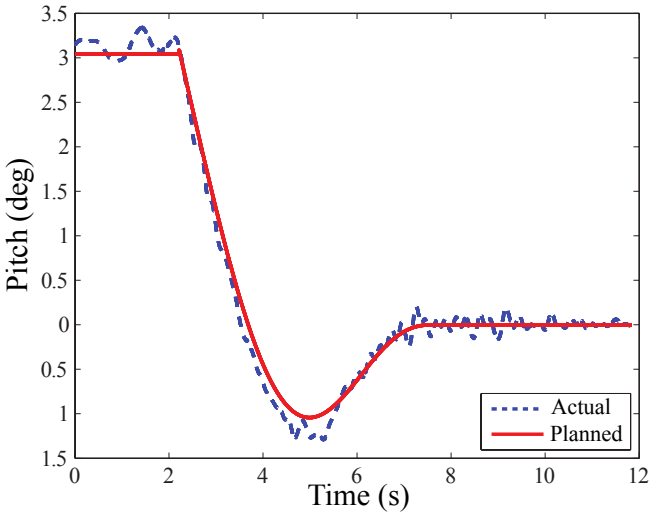


FIGURE 4.5: Recovering from an initial lean angle of three degrees.

4.2 Vector driver performance

Upon completion of the Rev. 3A three phase driver board, a single board was connected to a single stator and powered with 48 V. The rotor was attached to an IMADA single axis force gauge, capable of measuring up to 500 N in a single axis, using one of the aluminum plates which originally clamped the two copper hemispheres together during the construction of the rotor. A picture of the experimental setup can be seen in Fig. 4.7. An Extech clamp meter was attached to the power line to provide current measurements. Force measurements were taken 3-4 seconds after a particular torque was commanded. A plot of forces obtained vs the current used can be seen in Fig. 4.6.

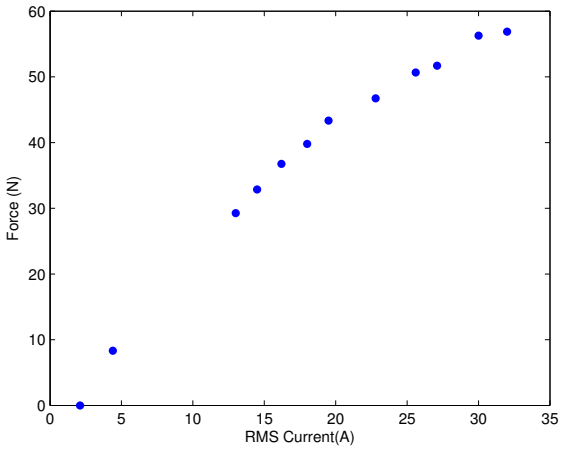


FIGURE 4.6: Amount of force produced on the ball with a moment arm of 0.125 m.

4.2.1 Six stator torque measurements

The same experiment was repeated with all six stators connected. With all six stators, the torque on the ball was calculated, using the moment arm of 0.125 m. The same experiment was again repeated, but with an additional 12 V battery connected in series with the original 48 V to produce 60 V. The results for this experiment can be seen in Fig. 4.8.

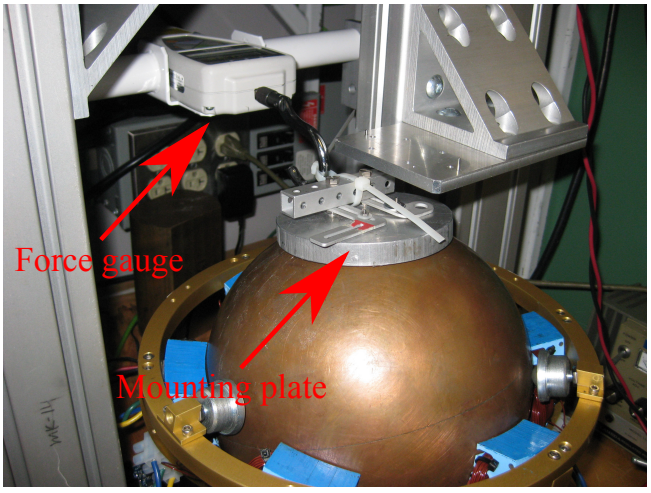


FIGURE 4.7: Experimental setup for force and torque measurements.

4.2.2 Three phase driver board performance discussion

With regard to the single board results, it is likely that the driver board can handle upwards of 32 A (the maximum measured in this experiment) but this would not be feasible with all six stators connected; peak torque with six stators would require, in the case of a requested torque vector lying directly in between two stators, four stators

at maximum force. The power distribution components on SIMbot (terminal blocks, battery terminal connectors, etc) were in general chosen to handle only up to 100 A, and four stators operating at over 30 A each could possibly be unsafe as the battery draw would exceed the rated current values for the distribution components.

With regard to the results for all six stators, it can be seen in Fig. 4.8 that a large amount of current is required to produce any torque at all. This is due to a combination of the magnetizing current and the friction in the drive mechanism. As the magnetizing current produces zero torque, it simply results in a current offset when looking at a plot of torque vs current.

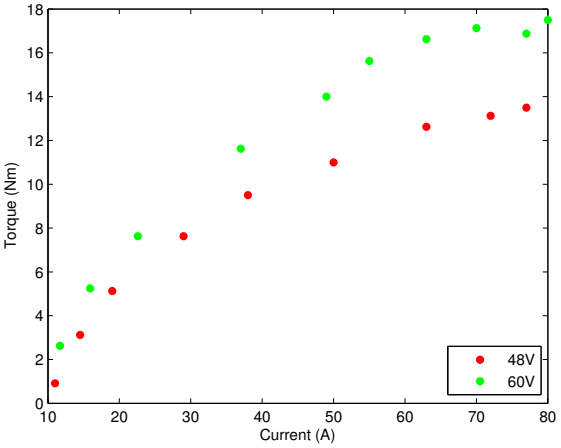


FIGURE 4.8: Amount of torque produced on the ball by six stators at 48 V and at 60 V.

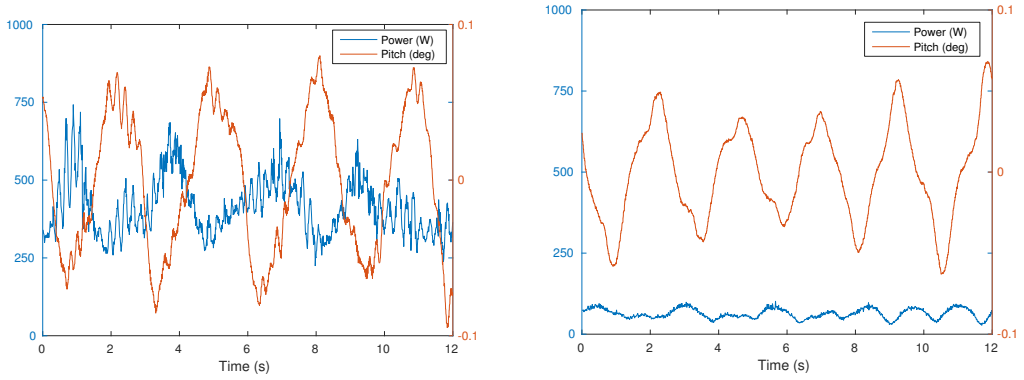
Figure 4.8 reveals a curve in the torque-current plot. Note that this shape can also be seen to a lesser extent in the results for a single stator in Fig. 4.6. One possible explanation for this is that the power drawn by the motor approaches the maximum power deliverable by the batteries - a simple model for power delivered to the motor would be

$$P = (V - 4\mathrm{R}_b I)I \tag{4.1}$$

where R_b is the battery resistance, I is the dc current, V is the nominal voltage, and the factor of four comes from having four batteries in series. This would give a similar quadratic shape to the one observed in the data and is also consistent with the increase in torque observed upon adding an additional battery.

4.3 SIM vs IMB power efficiency and performance results

The final set of experiments performed in this work were designed to compare to the new SIM-based ballbot directly against the original IMB-based ballbot. These experiments were originally presented in Sec. 1.2.



(a) (b)

FIGURE 4.9: On the left: Power drawn by and pitch of SIMbot while maintaining zero lean angle. On the right: Power drawn by and pitch of the IMB-based ballbot while maintaining zero lean angle.

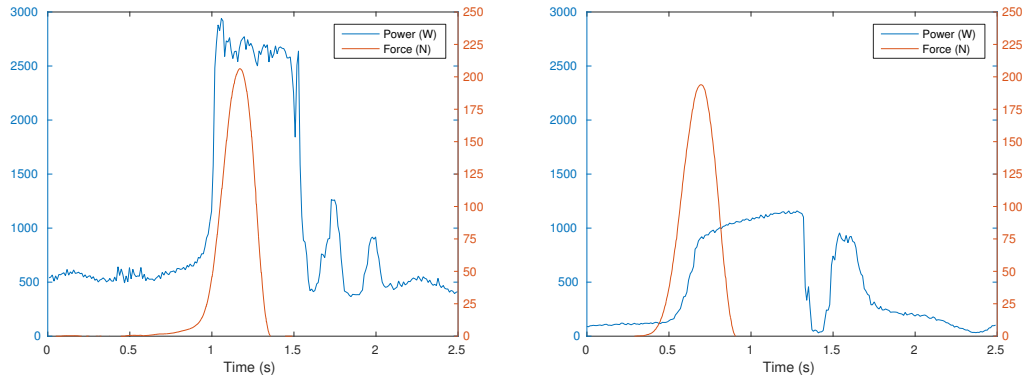
4.3.1 Experimental setup

All experiments were performed on soft foam tile. All pose data (lean angles, speeds, etc) were collected using each robot's on-board sensors. The IMB-based ballbot uses the same IMU (VectorNav VN-100) as SIMbot, but uses encoders attached to its stainless steel rollers to compute its velocity. Voltage and current was sensed on each robot using the voltage and current sensing electronics described in Sec. 3.1.5. The current sensor was connected so as to measure only the current which goes to the motor.

As SIMbot weighs less than the IMB-based ballbot, additional weight was added to SIMbot in an effort to match the inertial properties of the two robots. Two 6.35 kg lead bricks and a single 8.84 kg solid brass brick were fixed to an extra deck on SIMbot which was mounted at 0.865 m above the center of the ball. This brings the center of mass of SIMbot from an unweighted 0.68 m to approximately 0.745 m above the center of the ball. The bricks bring SIMbot's body mass to 63.54 kg. The IMB-based ballbot body weighs 65.65 kg and has a center of mass 73 cm above the center of the ball. It should be noted that the weight of the ball used by SIMbot is 7.46 kg while the weight of the ball used by the IMB-based ballbot is 2.44 kg; the balls were left alone as it would be nearly impossible to match their inertial properties as well.

4.3.2 Power efficiency while balancing

Each robot was commanded to maintain a zero lean angle while balancing on top of the soft foam tile. The lean angles and power drawn while balancing for SIMbot can be seen



(a) (b)

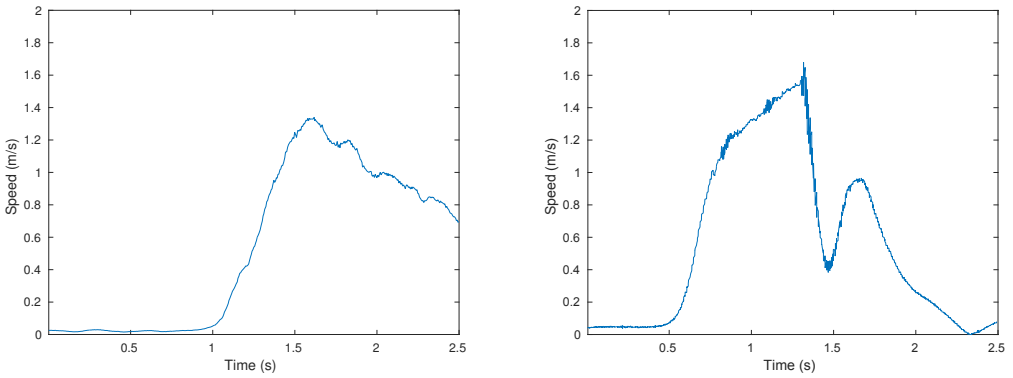
FIGURE 4.10: On the left: Power drawn by and force applied to SIMbot. On the right: Power drawn by and force applied to the IMB-based ballbot.

in Fig. 4.9(a). The lean angles and power drawn while balancing for the IMB-based ballbot can be seen in Fig. 4.9(b).

4.3.3 Peak acceleration and impulse response

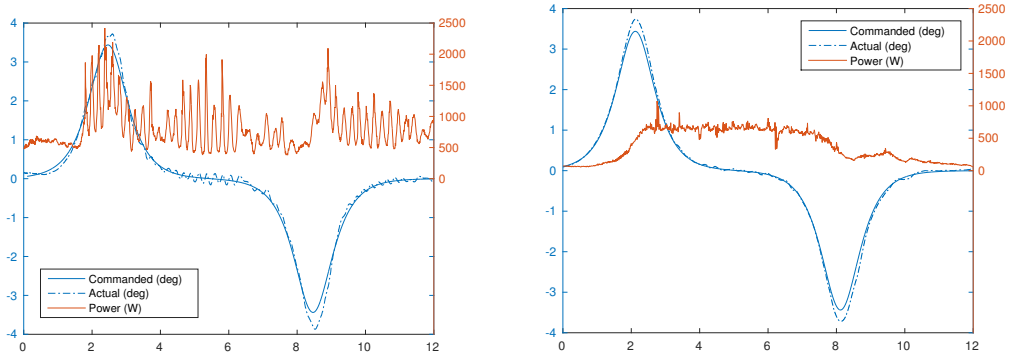
SIMbot was fixed to a frame with a slack line and carabiner for safety and then pushed by an experimenter with a force gauge. SIMbot was allowed to reach a steady state lean angle and was then decelerated by a second experimenter to prevent SIMbot from reaching the end of the line. The experiment was repeated with the IMB-based ballbot. SIMbot was able to maintain balance for peak forces up to 204 N and a net impulse of 47.01 Ns. SIMbot was not able to recover from a similarly shaped impulse with a peak force of 214 N and a net impulse of 54.19 Ns. During the 47.01 Ns impulse, SIMbot accelerated at approximately 3.0 m/s^2. The force applied to SIMbot and the power drawn during this push can be seen in Fig. 4.10(a). SIMbot's speed during the push can be seen in Fig. 4.11(a). Note that an experimenter caught SIMbot to slow it down at approximately 1.6 seconds.

The experiment was repeated for the IMB-based ballbot. The IMB-based ballbot began to fail at roughly the same peak force and impulse as SIMbot. The IMB-based ballbot withstood an impulse of 46.15 Ns and failed on an impulse of 50.44 Ns. The force applied and the power drawn by the IMB-based ballbot during the largest successful impulse can be seen in Fig. 4.10(b), and speed can be seen in Fig. 4.11(b). During this impulse, the IMB-based ballbot experienced a peak acceleration of roughly 5.3 m/s^2. An experimenter intervened to slow the IMB-based ballbot down at approximately 1.4 seconds.



(a) (b)

FIGURE 4.11: On the left: Speed of SIMbot during impulse. On the right: Speed of IMB-based ballbot during impulse.



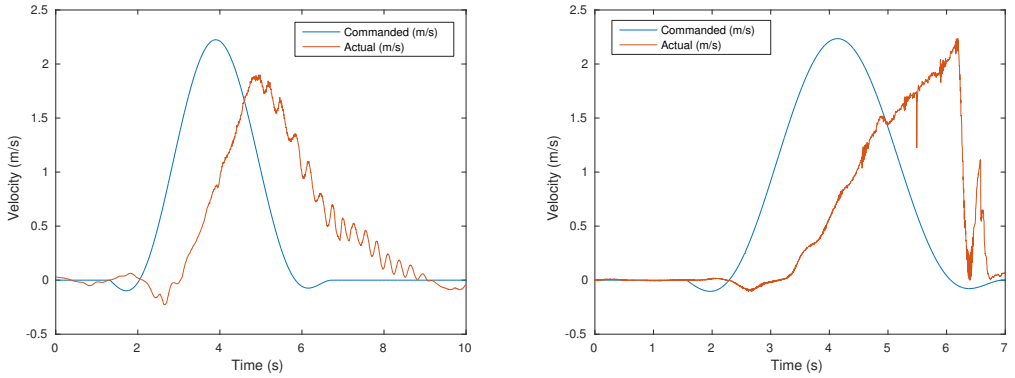
(a) (b)

FIGURE 4.12: On the left: Lean angle tracking and power drawn for SIMbot during an aggressive point-to-point motion. On the right: Lean angle tracking and power drawn for the IMB-based ballbot during an aggressive point-to-point motion.

4.3.4 Power efficiency during an aggressive point-to-point motion

Both SIMbot and the IMB-based ballbot were commanded to track a hyperbolic secant trajectory, as described in Sec. 1.2 and shown in Fig. 1.5. These trajectories were commanded in an open-loop fashion with no feedback based on ball position. The parameters used for this trajectory were $k = 6$ and a final time $t_f = 12.0$ s. The peak lean angle, α, was set to 3.43°. Typical ballbot lean angles range from about $-5°$ to about 5°, so this would be considered a fairly aggressive trajectory.

It is important to note that ballbots are underactuated and as such are subject to a set of dynamic constraints. Ballbots are not able to track arbitrary lean angle trajectories as they may not satisfy these dynamic constraints. The hyperbolic secant trajectory used in this experiment was used primarily because it ensures that both robots follow



(a) (b)

FIGURE 4.13: On the left: Velocity tracking during a top speed manuever for SIMbot. On the right: Velocity tracking for the IMB-based ballbot during a top speed manuever. The IMB-based ballbot fell at approximately 6.1 seconds.

exactly the same lean angle trajectory and because it is very easy to adjust its shape and peak lean angle. Thus this experiment should be used to evaluate power draw only and not control performance.

The lean angle tracking and power drawn during this 12 second trajectory for SIMbot can be seen in Fig. 4.12(a). SIMbot drew a total of 2.69 Wh during the 12 second trajectory. The results for the IMB-based ballbot can be seen in Fig. 4.12(b). The IMB-based ballbot drew 1.29 Wh during the 12 second trajectory.

4.3.5 Maximum speed

In order to investigate the maximum speed of both the IMB-based ballbot and SIMbot, feasible velocity trajectories were generated using the differentially flat planner described in [14] and tracked using the outer loop controller described in Sec. 3.2.3.3. The outer loop controller used here was a velocity controller which output desired body lean angles based on the velocity error of the ball. The robots were attached to a frame with a slack line during this experiment to mitigate failures.

The highest speed that SIMbot was able to achieve was 1.9 m/s over 4 m. The velocity during this trajectory can be seen in Fig. 4.13(a). The highest speed that the IMB-based ballbot was able to achieve was 2.25 m/s in the middle of a trajectory. The velocity during this trajectory can be seen in Fig. 4.13(b). The IMB-based ballbot fell during this trajectory but was caught by the frame.

The velocity tracking this experiment is very poor; there was no feedforward term in the velocity controller, so some lag would be expected. Furthermore, recall that the

intention of this experiment was simply to have the robots go faster and faster in a controlled manner until failing, not to demonstrate excellent velocity tracking.

4.3.6 Power efficiency and performance comparison discussion

While maintaining zero lean angle, the IMB-based ballbot is approximately 4-5 times more efficient than SIMbot, despite having similar lean angle performance. During the point-to-point trajectory, the IMB-based ballbot proved to be roughly twice as efficient as SIMbot. During the impulse response experiment, SIMbot drew a peak power of approximately 2.5 kW, while for a nearly identical impulse, the IMB-based ballbot drew only approximately 1 kW. In general, the IMB appears to be 2-3 times more efficient than the SIM when used as the drive mechanism for a ballbot.

With regard to the impulse response experiment, SIMbot did surprisingly well, considering that its peak torque output (14 Nm) is much lower than the IMB-based ballbot's peak torque output (40 Nm), although some amount of torque is lost in the IMB due to friction. While the SIM has friction as well, it is significantly lower than that of the IMB at 0.4 Nm of Coulomb friction [3] to 3.8 Nm in the IMB [1]. In the impulse response experiment, it is possible that, although the dc motors on the IMB can create higher peak torques than the SIM, they are not able to generate enough torque to adequately accelerate the ball at higher speeds. In Fig. 4.11(b), for instance, the acceleration begins to decrease after the robot reaches about 1 m/s; this may be due to the rollers slipping against the ball.

Despite concerns such as stator end effects, a relatively large torque rise time, and a lower peak torque, SIMbot's performance was very comparable to the IMB-based ballbot. Ballbot dynamics in general are fairly slow, which may be the reason why the torque rise time did not prevent SIMbot from being feasible. Judging from the performance of the IMB-based ballbot in the impulse response experiment, being able to generate large enough torques at moderate speeds may be more important than the absolute peak torque output of the motor.

With respect to efficiency, it appears that the SIM itself is what causes SIMbot to be much less efficient than the IMB-based ballbot. This is very consistent with the findings of [15], which cites the efficiency of their SIM as 2%; this is much lower than conventional motors. There are many tradeoffs which must be evaluated in the construction of the SIM and which affect the efficiency of the resulting motor; for instance, thicker wire in the windings would result in less energy lost to heat but fewer turns of wire in the same space, resulting in weaker magnetic fields and lower performance. The laminations used in the construction of the stators, built from electrical steel in the SIM in this work,

could be made from a material with a higher magnetic permeability, but at a higher cost. The thickness of the copper hemisphere could be optimized; thicker copper results in lower I^2R losses from eddy currents, but increases the air gap between the windings and the steel core which would reduce performance.

4.4 Chapter 4 Conclusion

Experimental results for the feasibility of the SIMbot design, the new vector drivers, and for power efficiency and performance between SIMbot and the IMB-based ballbot were presented. Prior to the installation of the new vector drivers, SIMbot was able to hold a zero degree lean angle to within about 0.15° under no disturbances, which is comparable to the IMB-based ballbot. It was also able to recover from a 3° lean angle, as well as execute a short point-to-point motion. The new vector drivers were demonstrated to be capable of at least 32 A per stator and to improve the torque output of the SIM from about 8 Nm to about 14 Nm when running at 48 V. After the installation of the new vector drivers, SIMbot was shown to have a top speed (at least 1.9 m/s) and peak acceleration (at least 3.0 m/s^2) comparable to the top speed and peak acceleration of the IMB-based ballbot (at least 2.25 m/s and at least 5.3 m/s^2, respectively). However, it was found experimentally that SIMbot is generally 2-3 times less efficient than the IMB-based ballbot.

4.5 Future Work

With data demonstrating the inefficiency of SIMbot, presumably due to the SIM itself, it remains to be determined exactly why the SIM is inefficient and what the biggest sources of energy loss are. A number of speculations were made in Sec. 4.3.6, including the gauge of the wire used for the stator windings, the thickness of the copper shell, and the type of laminations used in the stators, but doing a real experimental quantification of the effect of each of these parameters on the efficiency of the resulting motor, backed by theory and modeling, would be a first step in understanding how to build a more efficient SIM.

From a scientific perspective, it would be interesting to study the diffusion of the induced magnetic fields in the rotor as well as the result of an already magnetized portion of the rotor rolling underneath a stator in operation. The insight gained from such an investigation could lead to more efficient or higher performance controllers for spherical induction motors.

One current shortcoming of SIMbot is its inability to operate on hard surfaces due to the soft copper shell. A solution [16] to this would be to use an additional ball, an interposer ball, on top of which the copper ball could sit. SIMbot would then be driven in reverse, with only the interposer ball touching the ground. The interposer ball could be made from rugged materials as it does not have to support magnetic fields or eddy currents. This would also allow for greater coverage of the copper ball by stators. This setup is shown in Fig. 4.14.

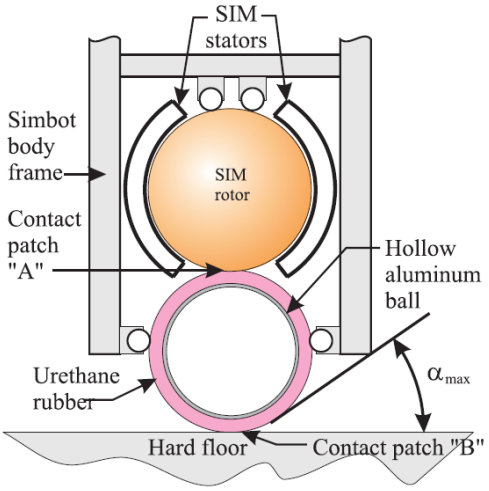


FIGURE 4.14: Interposer ball. Figure courtesy of Ralph Hollis.

Finally, the ball transfers which support the weight of SIMbot could be replaced with an air bearing [16]. This would reduce the friction in the drive mechanism to near zero and result in a ballbot which comes even closer to the approximation consisting of a cylinder on top of a rigid sphere used for modeling purposes.

Appendix A

Control Board Rev 3A Schematics

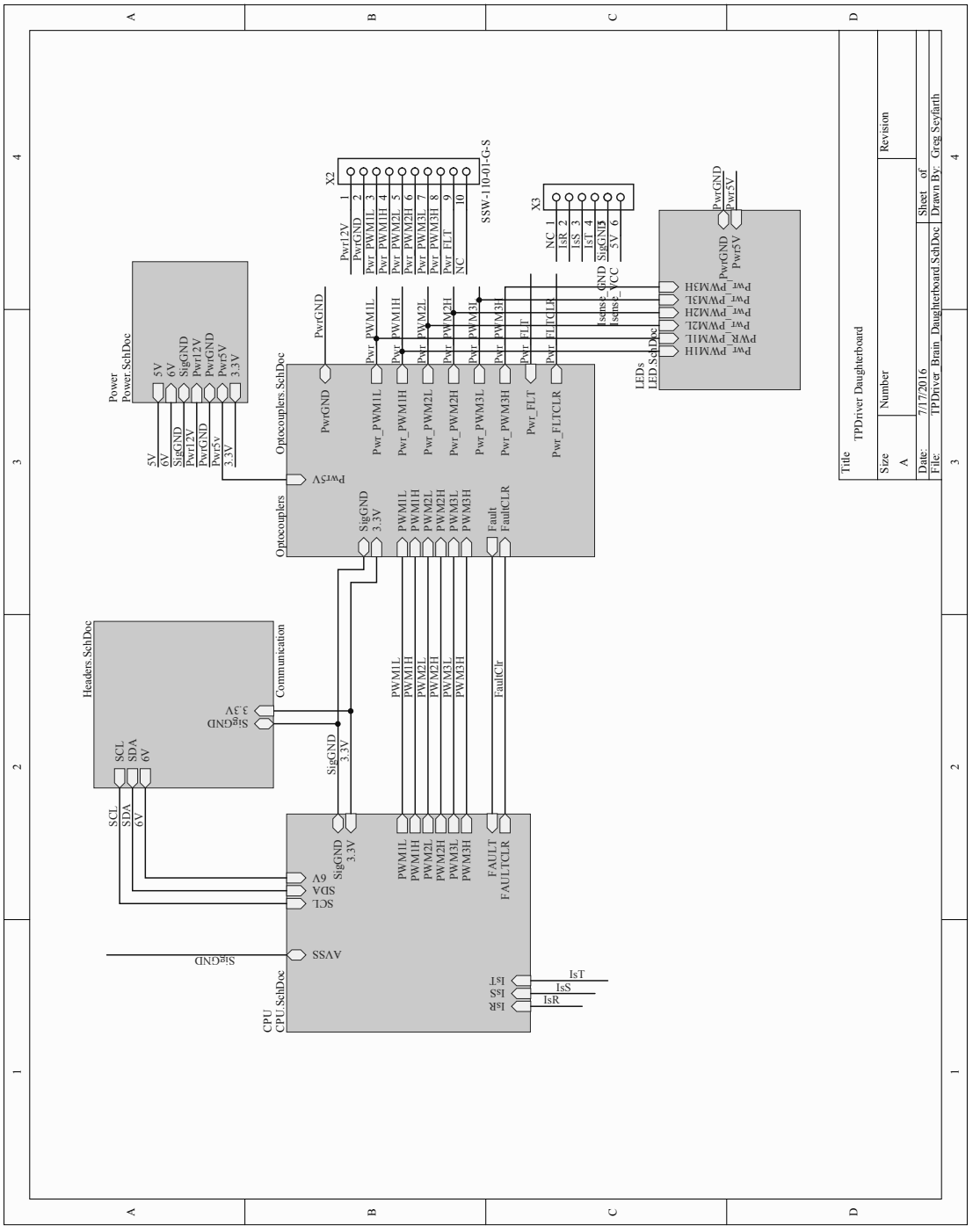


FIGURE A.1: Top level schematic for the control board

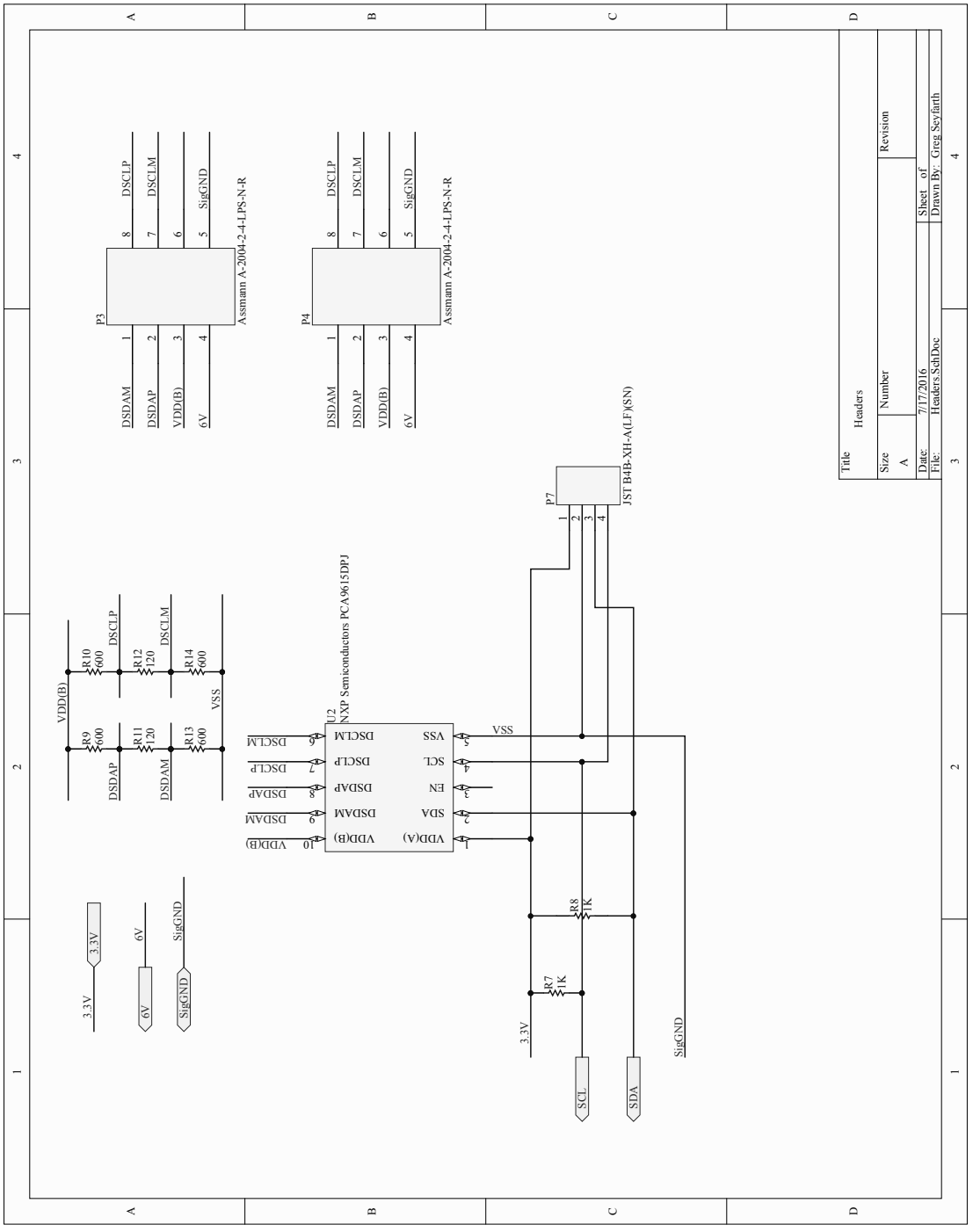


FIGURE A.2: Schematic for I^2C headers and subsystem.

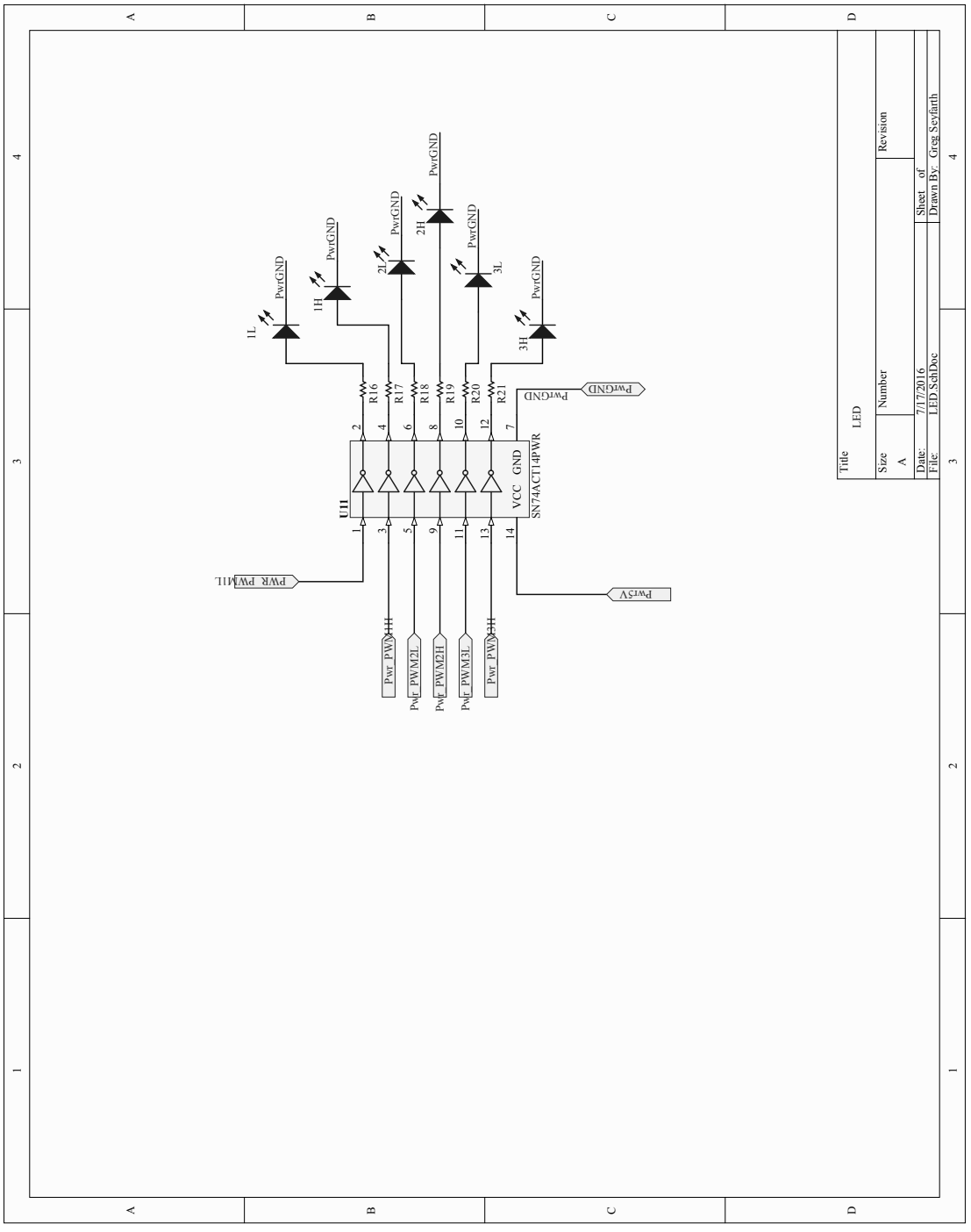


Figure A.3: Schematic for the flux angle LEDs and hex inverter

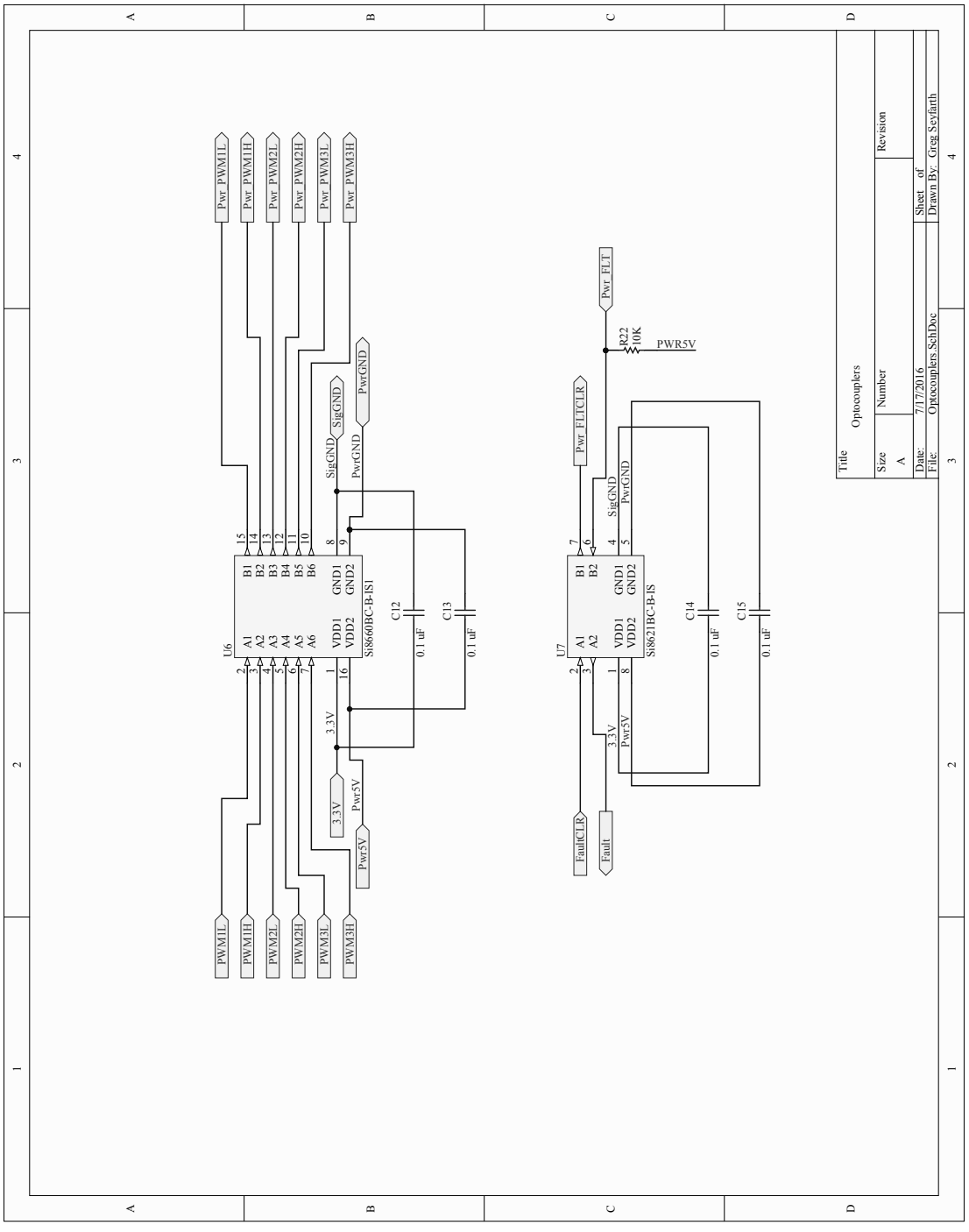


FIGURE A.4: Schematic for the isolators

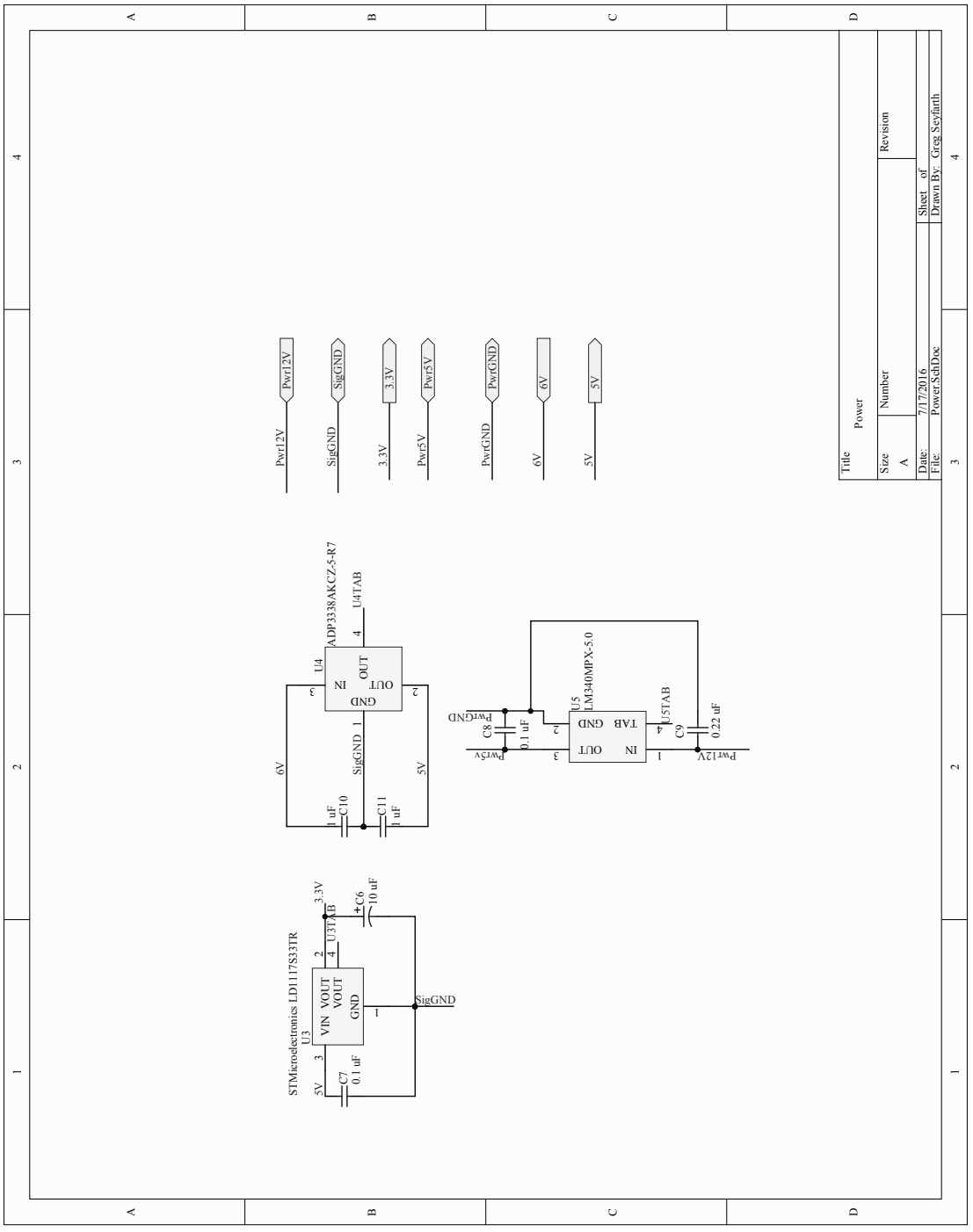


FIGURE A.5: Schematic for power distribution

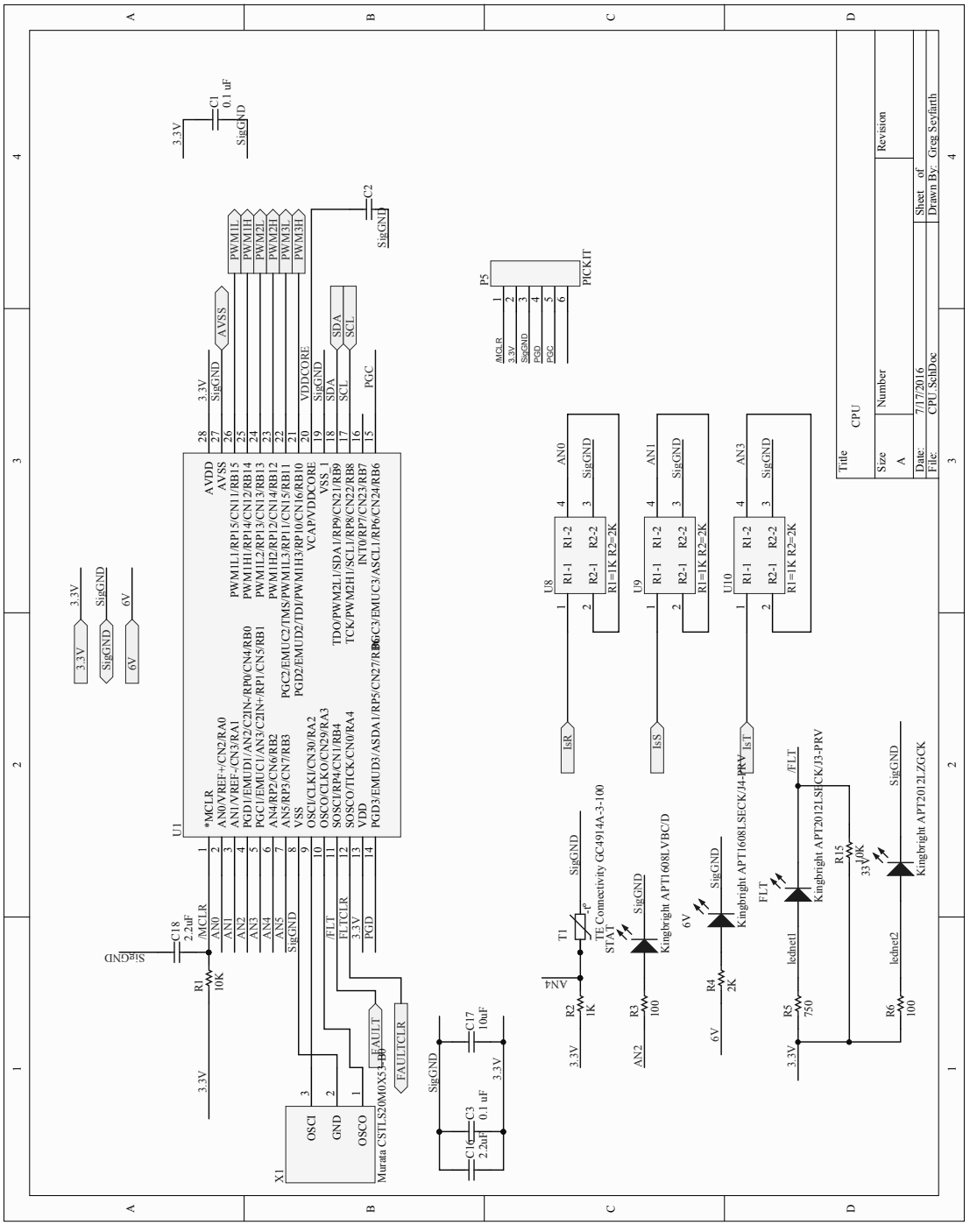


Figure A.6: Schematic for the microcontroller

Appendix B

Power Board Rev 3A Schematics

A few notes:

- L10 and L11 should be shorted with wires. They affect the gate driver signal to the low side MOSFETs. They should be removed for future revisions.
- The overcurrent detection circuit (U7A/U7B) never worked reliably but is still included on the board for future debugging. Fuses were used for overcurrent protection. To disable overcurrent protection via the gate driver, ground the Itrip pin on the gate driver. This was done for the six drivers on SIMbot as of July 2016.

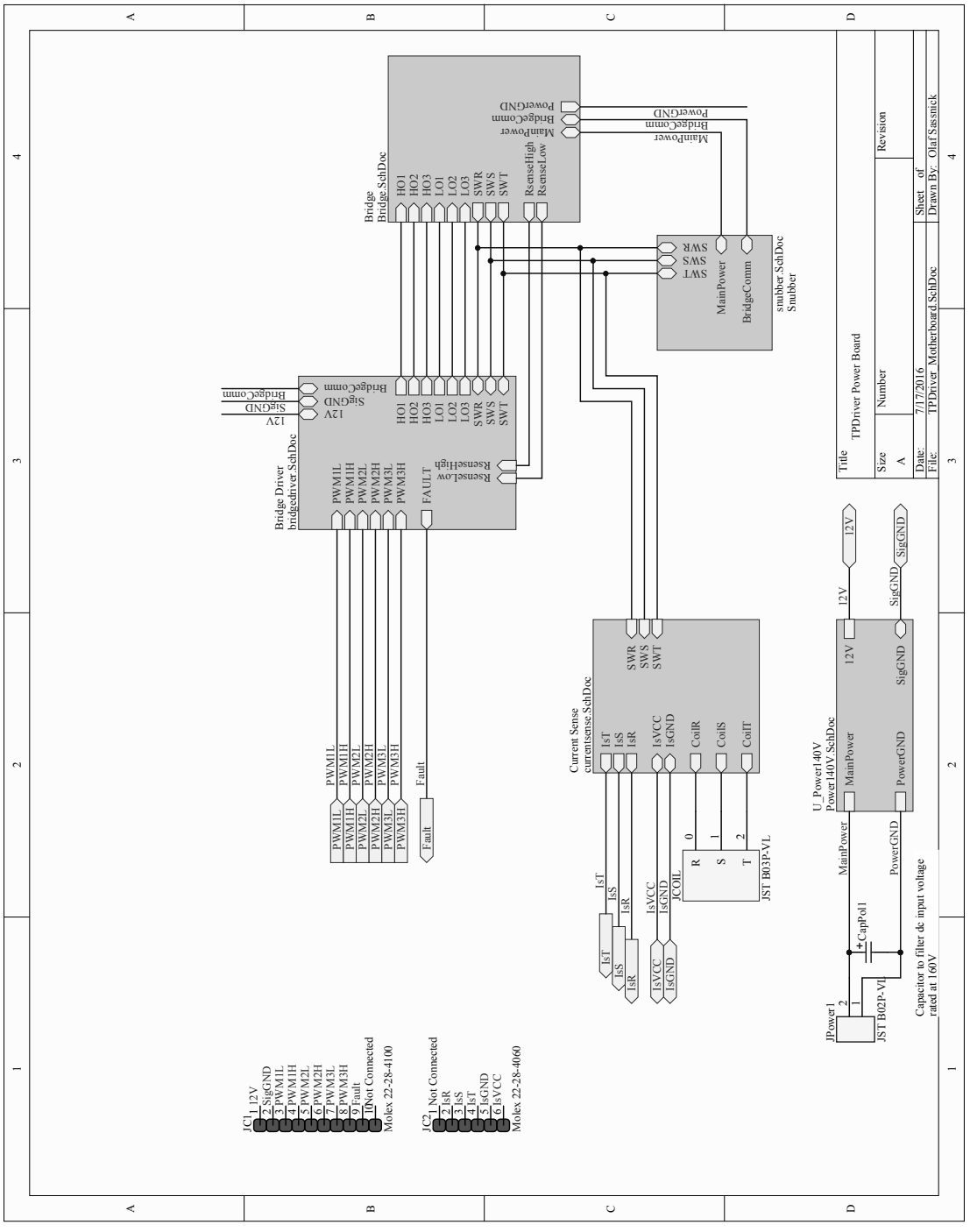


FIGURE B.1: Top level schematic for the power board

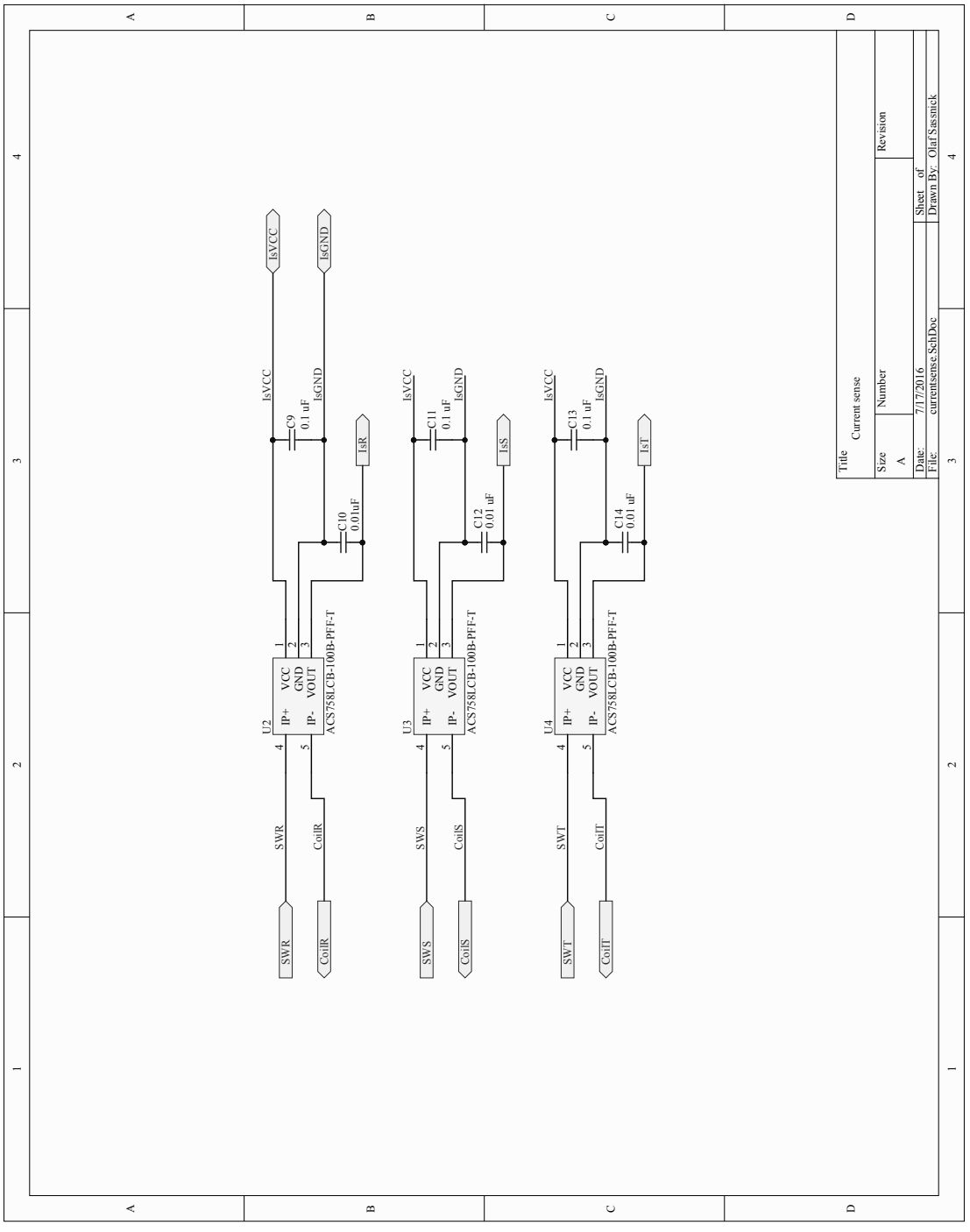


FIGURE B.2: Schematic for current sensors

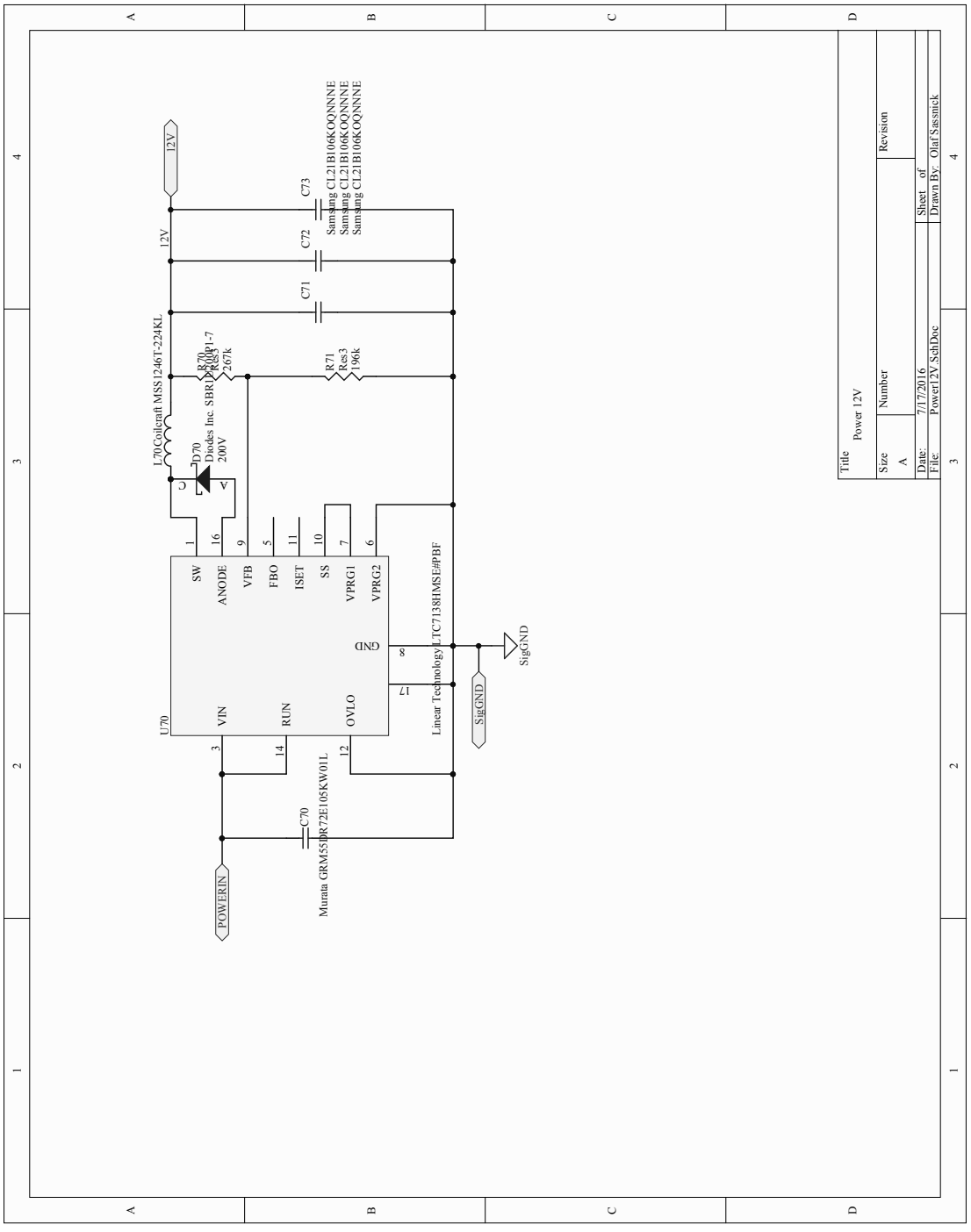


FIGURE B.3: Schematic for switching 12V regulator

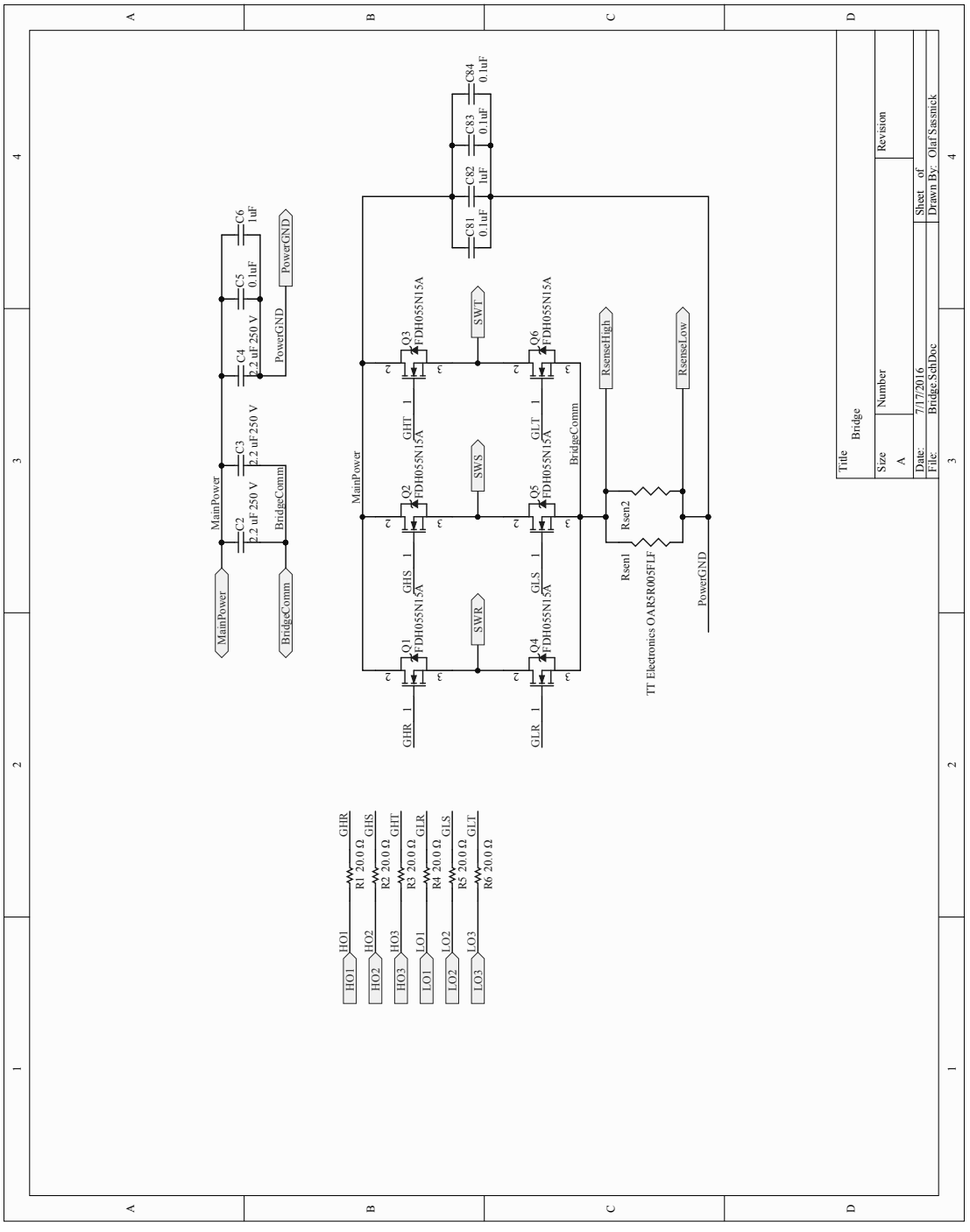


FIGURE B.4: Schematic for the power bridge

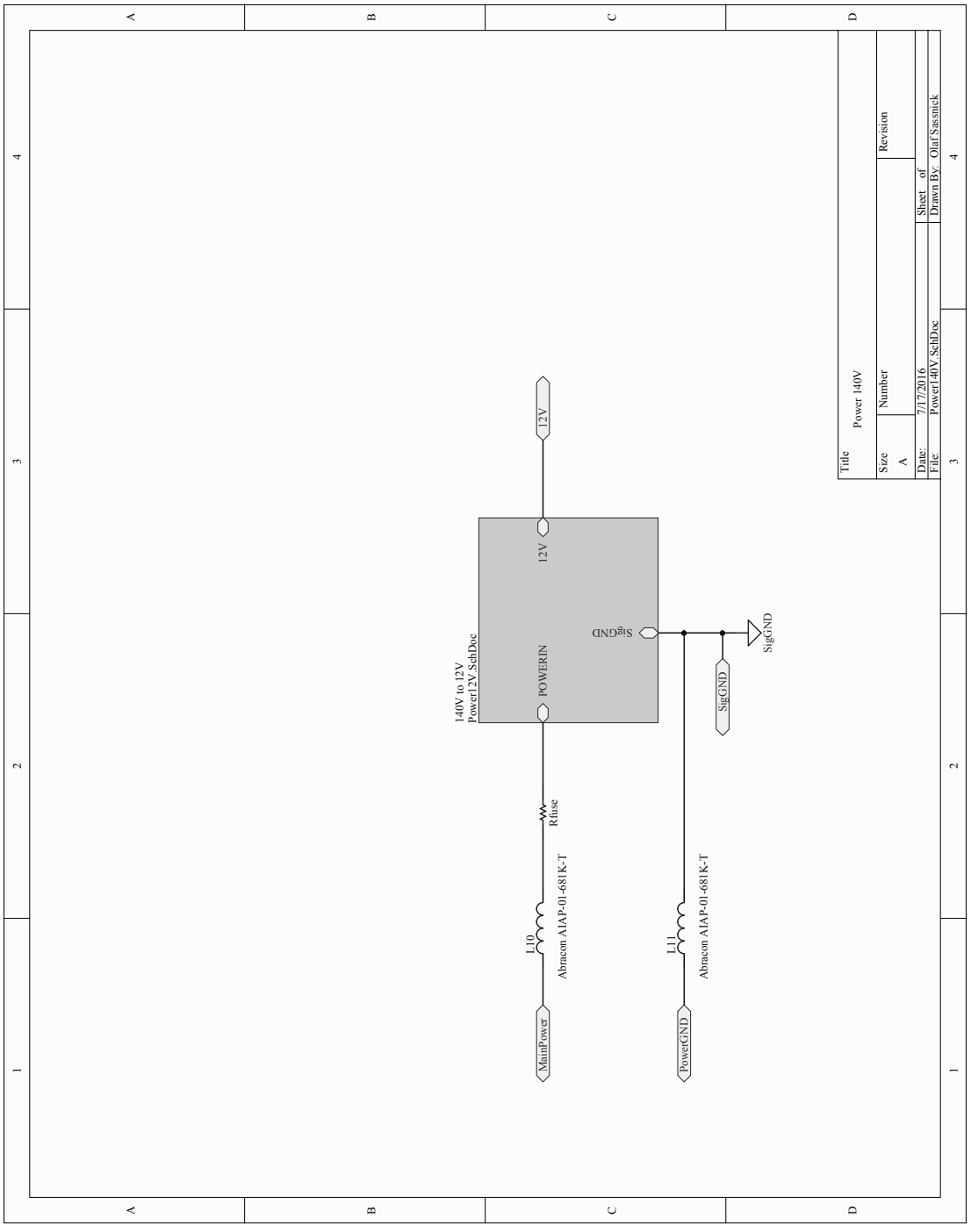


FIGURE B.5: High-level schematic for voltage converter

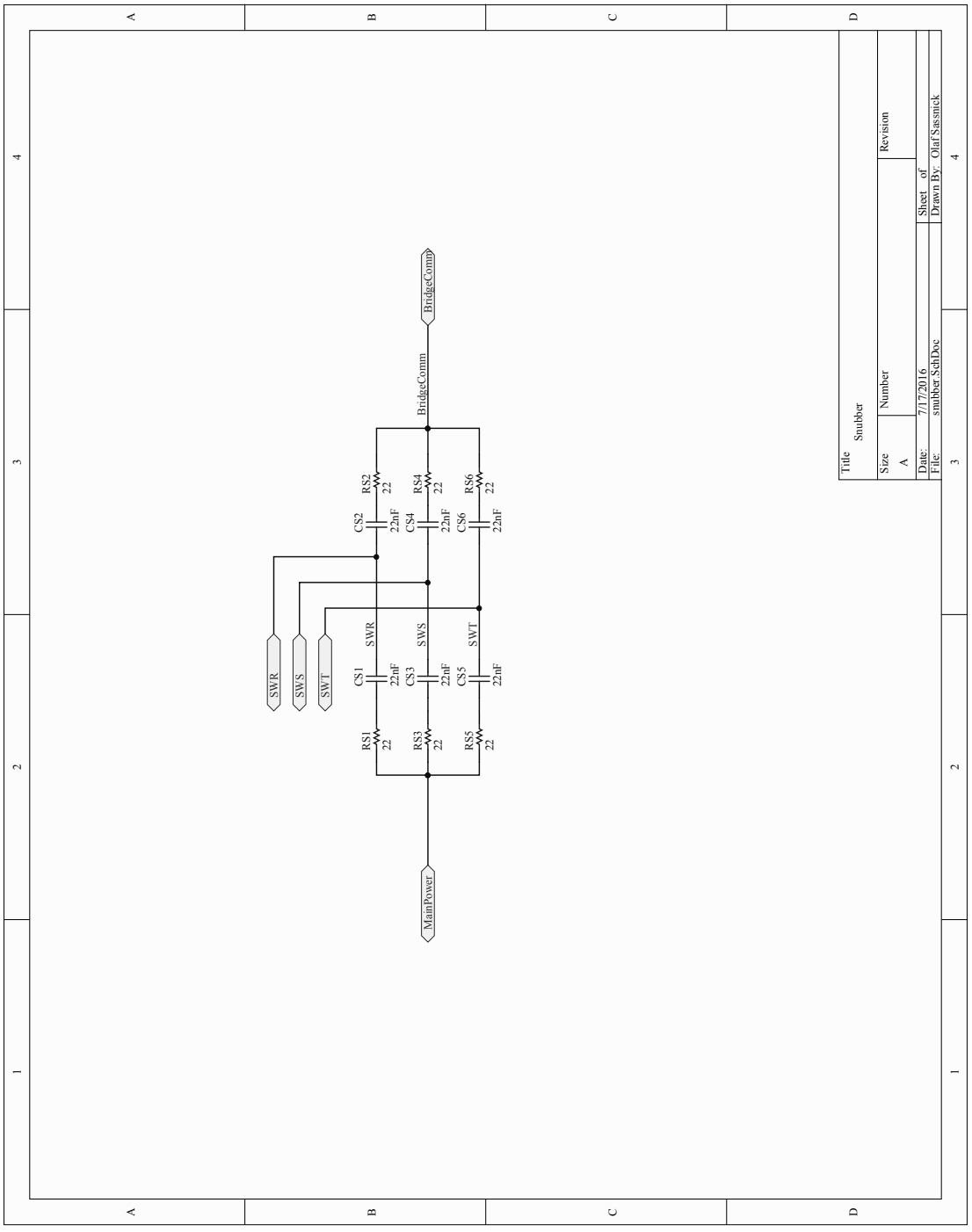


FIGURE B.6: Schematic for RC snubbers

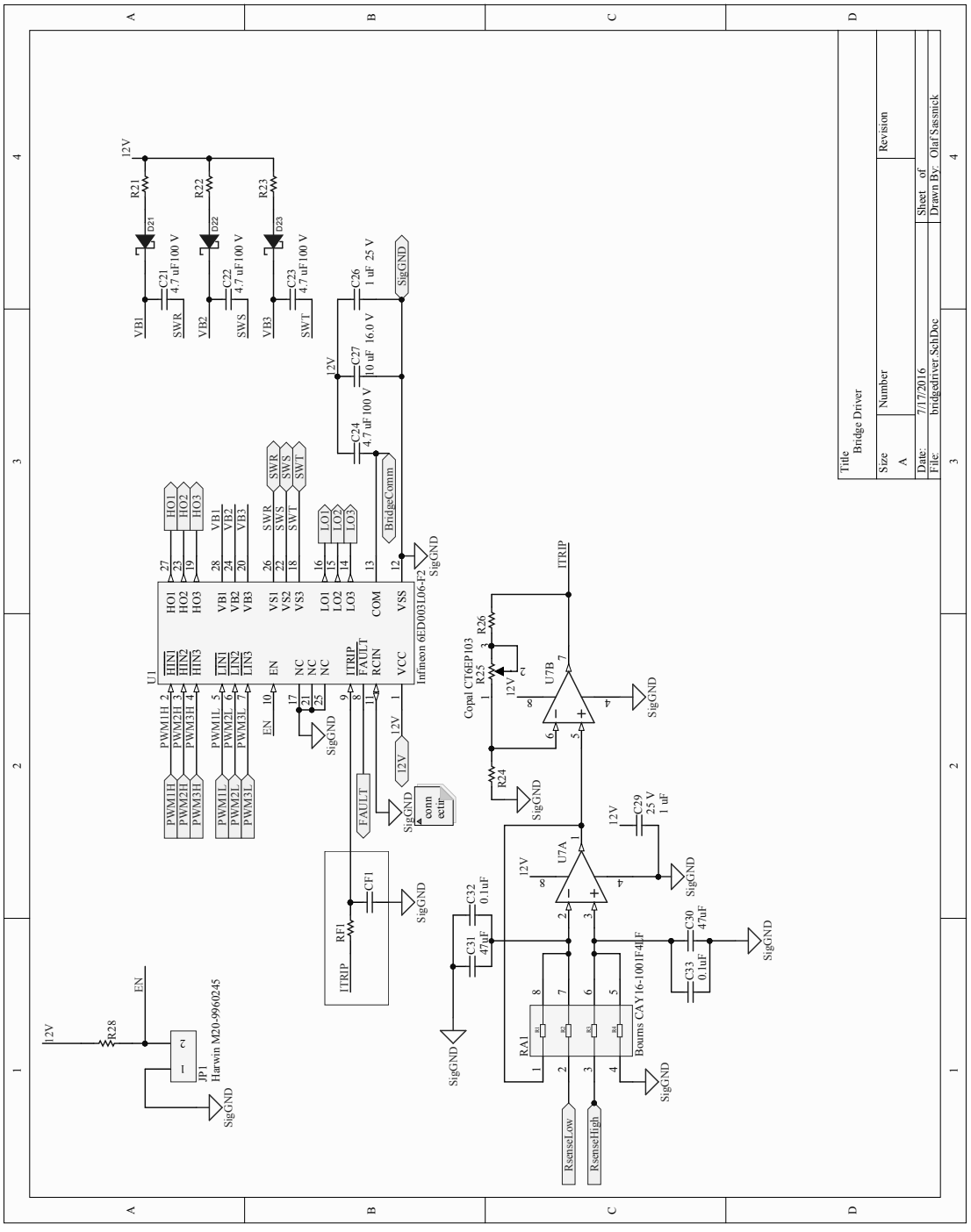


FIGURE B.7: Schematic for the bridge driver

Appendix C

Current Sense Schematic

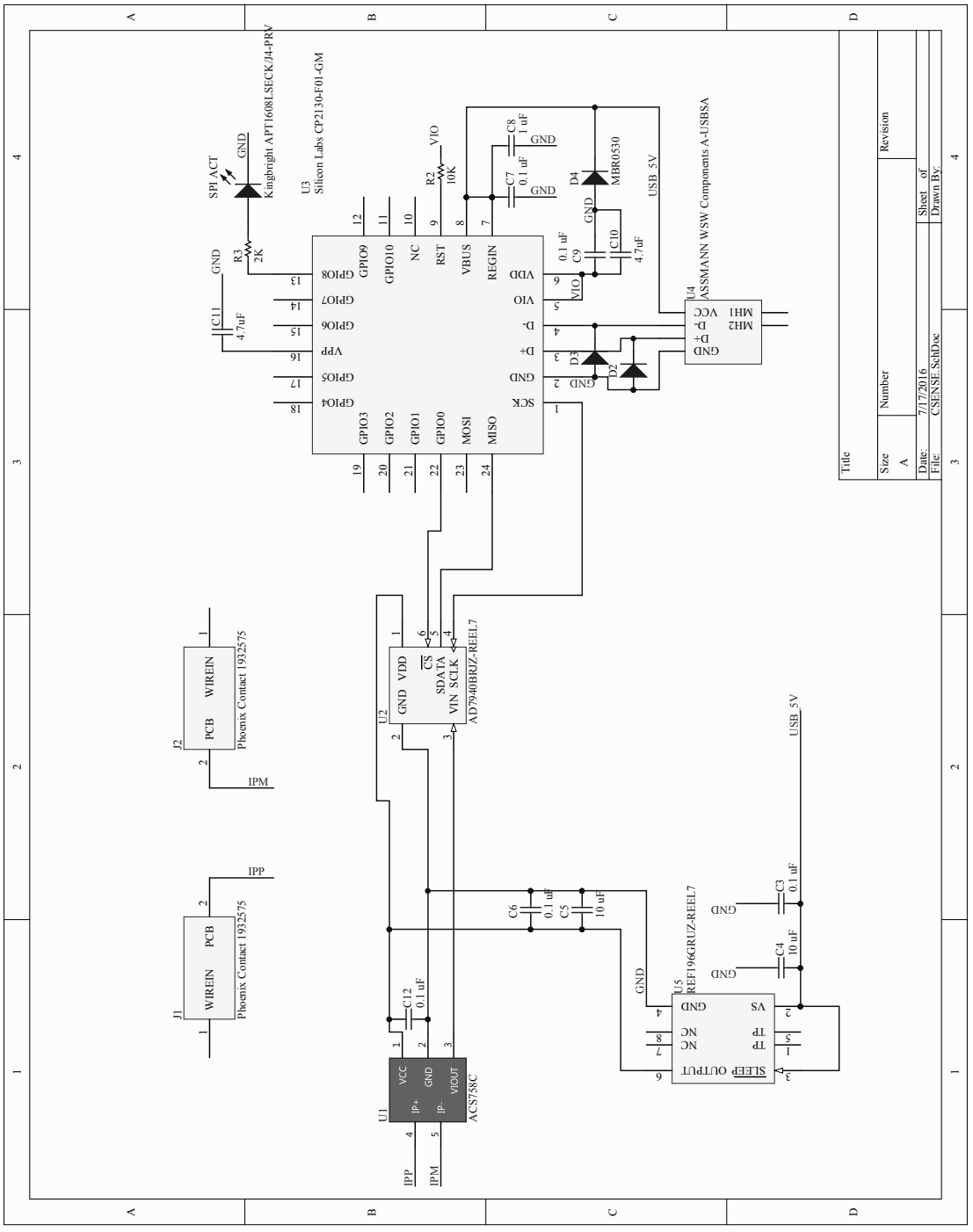


FIGURE C.1: Current sense schematic

Appendix D

Vector Driver Code

This appendix has the code that runs the sensorless field oriented controller on the PIC MCU. It was originally written by Masaaki Kumagai. It was maintained by Ankit Bhatia and Greg Seyfarth. It can also be found at http://clarinet.msl.ri.cmu.edu:9999/wumple/sim_PIC.git.

File: TPHardware.c

```
#include "TPHardware.h"
#include "main.h"

volatile long PWMsysclock=0;
volatile unsigned int phase=0;
volatile unsigned int phasedelta=0;
volatile int AD[4];
volatile int ADB[4];

void OnPWMCycle(void)
{
        AD[0]=ADCbufferA[0];
        AD[1]=ADCbufferA[1];
        AD[2]=ADCbufferA[2];
        AD[3]=ADCbufferA[3];
        ADB[0] = ADCbufferB[0];
        ADB[1] = ADCbufferB[1];
        ADB[2] = ADCbufferB[2];
        ADB[3] = ADCbufferB[3];


        PWMsysclock++;
        phase+=phasedelta;
}
void ClockUp(void)
{
        // from dsPIC manual ex39-1
        CLKDIVbits.PLLPRE=2;  // N1=4  /4  20MHz->5MHz
        PLLFBD=30;            // M=32  x32 5MHz -> 160MHz
        CLKDIVbits.PLLPOST=0; // N2=2  /2  160MHz -> 80MHz
```

```
        __builtin_write_OSCCONH(0x03);
        __builtin_write_OSCCONL(OSCCONL|0x01);

        LATB=0x8000;
        // wait for switch
        while(OSCCONbits.COSC != 0b011)
                ;
        LATB=0xc000;
        // wait for PLL stable
        while(OSCCONbits.LOCK!=1)
                ;
//      LATB=0;
}

//not needed.
void IORemapping(void)
{
        // setting for UART port
        __builtin_write_OSCCONL(OSCCONL&(~0x40));  // unlock RP
        __builtin_write_OSCCONL(OSCCONL|0x40);  // lock RP
}
/* copied from lib src */
void SetMCPWM1DeadTimeAssignment(unsigned int config)
{
    P1DTCON2 = config ;
}

void SetupMCPWM(void)
{
        unsigned int period;
        // PWM cycle, Fcy -> prescaler -> count
        // 40M/20k=2000, centered PMW /2
        period=999;
        // interrupt setting
        ConfigIntMCPWM1(PWM1_INT_EN & PWM1_INT_PR2 &
                    PWM1_FLTA_DIS_INT & PWM1_FLTA_INT_PR7);

        /* dead time setting activate->A innactivate->B */
        SetMCPWM1DeadTimeAssignment(PWM1_DTS1A_UA & PWM1_DTS1I_UB &
                                PWM1_DTS2A_UA & PWM1_DTS2I_UB &
                                PWM1_DTS3A_UA & PWM1_DTS3I_UB);
    /* deadtime : Fcy tick * DTxS * DTx */
        /* see: pwm12.h */
        SetMCPWM1DeadTimeGeneration(PWM1_DTAPS1 & PWM1_DTA40 &
                                PWM1_DTBPS1 & PWM1_DTB40);
        /* fault sensitive */
        /* only A, disabled , if need, remap io required */
        SetMCPWM1FaultA(PWM1_FLTA1_DIS & PWM1_FLTA2_DIS & PWM1_FLTA3_DIS & PWM1_FLTA1_DIS);

        OpenMCPWM1(period,
                0,  /* special event trigger for ADC ->sec14 MC 70187, 14-15 */

                PWM1_EN & PWM1_IDLE_STOP &
                PWM1_OP_SCALE1 & /* postscaler for interrupt */
                PWM1_IPCLK_SCALE1 & PWM1_MOD_UPDN,  /* prescaler 1:1, updown */
```

```
                PWM1_MOD3_COMP & PWM1_MOD2_COMP & PWM1_MOD1_COMP & /* compli ope */
                PWM1_PEN3H & PWM1_PEN2H & PWM1_PEN1H &
                PWM1_PEN3L & PWM1_PEN2L & PWM1_PEN1L,

                PWM1_SEVOPS1 & /* special, ADC */
                PWM1_OSYNC_TCY & /* output override timming */
                PWM1_UEN);  /* PWM duty set eneble */
}


void __attribute__((__interrupt__, no_auto_psv)) _MPWM1Interrupt(void)
{
        IFS3bits.PWM1IF = 0;
        OnPWMCycle();
}

void SetDutyCycle(unsigned int ph1,unsigned int ph2,unsigned int ph3)
{
        SetDCMCPWM1(1,ph1,0);
        SetDCMCPWM1(2,ph2,0);
        SetDCMCPWM1(3,ph3,0);
}

void SetManualOutput(unsigned int ph1,unsigned int ph2,unsigned int ph3)
{
        #define SOL 950
        PDC1 = (ph1)/32;
        PDC2 = (ph2)/32;
        PDC3 = (ph3)/32;
}

void SetOutput(int ph1,int ph2,int ph3)
{

        #define SOL 950
        if(ph1<-SOL) ph1=-SOL;
        if(ph1> SOL) ph1= SOL;
        if(ph2<-SOL) ph2=-SOL;
        if(ph2> SOL) ph2= SOL;
        if(ph3<-SOL) ph3=-SOL;
        if(ph3> SOL) ph3= SOL;
        write_16(D4,ph1);
        write_16(D6,ph2);
        write_16(D8,ph3);

        SetDCMCPWM1(1,(1000-ph1)&0x7ff,0);
        SetDCMCPWM1(2,(1000-ph2)&0x7ff,0);
        SetDCMCPWM1(3,(1000-ph3)&0x7ff,0);
        LATBbits.LATB7=(ph1<0)?1:0;
}
void DisablePWM(void)
{
        OverrideMCPWM1(PWM1_POUT_1H & PWM1_POUT_2H &PWM1_POUT_3H &
                    PWM1_POUT_1L & PWM1_POUT_2L &PWM1_POUT_3L &
```

```
                    PWM1_POUT1H_INACT & PWM1_POUT2H_INACT & PWM1_POUT3H_INACT &
                    PWM1_POUT1L_INACT & PWM1_POUT2L_INACT & PWM1_POUT3L_INACT);
}

void EnablePWM(void)
{
        OverrideMCPWM1(PWM1_GEN_1H & PWM1_GEN_2H &PWM1_GEN_3H &
                    PWM1_GEN_1L & PWM1_GEN_2L &PWM1_GEN_3L &
                    PWM1_POUT1H_INACT & PWM1_POUT2H_INACT & PWM1_POUT3H_INACT &
                    PWM1_POUT1L_INACT & PWM1_POUT2L_INACT & PWM1_POUT3L_INACT);
}

unsigned int __attribute__((space(dma))) ADCbufferA[17],ADCbufferB[17];
// buffer[16]=phase

// ==========================
//  ADConversion
int SetupADC(void)
{
        // setup DMA
        DMA0CONbits.AMODE = 0; // Configure DMA for Peripheral indirect mode
        DMA0CONbits.MODE = 2; // Configure DMA for Continuous Ping-Pong mode
        DMA0PAD = 0x0300; // Point DMA to ADC1BUF0
        DMA0CNT = 3; // 4 DMA request
        DMA0REQ = 13; // Select ADC1 as DMA Request source
        DMA0STA = __builtin_dmaoffset(&ADCbufferA);
        DMA0STB = __builtin_dmaoffset(&ADCbufferB); // not need
        IFS0bits.DMA0IF = 0; //Clear the DMA interrupt flag bit
        IEC0bits.DMA0IE = 0; //Stop the DMA interrupt enable bit
        DMA0CONbits.CHEN=1; // Enable DMA

        // setup AD
        OpenADC1(ADC_MODULE_ON & ADC_IDLE_CONTINUE & ADC_ADDMABM_ORDER &
             ADC_AD12B_10BIT & ADC_FORMAT_INTG &
                         ADC_CLK_MPWM & ADC_AUTO_SAMPLING_ON &// trigger source -> MCPWM
             ADC_SIMULTANEOUS & ADC_SAMP_ON ,

             ADC_VREF_AVDD_AVSS & ADC_SCAN_ON &
             ADC_SELECT_CHAN_0123 & // ADC_CONVERT_CH0123 &
             ADC_DMA_ADD_INC_4 & ADC_ALT_BUF_OFF &// ########## need to check
             ADC_ALT_INPUT_ON ,
                       // note: ADC_DMA_ADD_INC_ (i.e. SMPI bits) decide number of blocks,
                       /at least number of scan
                       // where as section DMA said they should be cleared(=1)
                       // or, simply use register indirect mode in DMA.AMODE

             ADC_CONV_CLK_SYSTEM & ADC_CONV_CLK_3Tcy & // 40MHz=25ns -> 75ns
             ADC_SAMPLE_TIME_16 ,

             ADC_DMA_BUF_LOC_4 ,

             ENABLE_AN0_ANA & ENABLE_AN1_ANA & ENABLE_AN2_ANA &
             ENABLE_AN3_ANA & ENABLE_AN4_ANA & ENABLE_AN5_ANA ,
             ENABLE_ALL_DIG_16_31 ,
```

```
            0x0, 0x8  // scan setting for ch3,4
            );
        // set PWM driven timing
        P1SECMPbits.SEVTDIR = 1;  // when count down
        PWM1CON2bits.SEVOPS = 0b0000;
        P1SECMPbits.SEVTCMP = 900;  // 0-999

//line 1: ch123= normal input AN012
//line 2: for setting B, not used
//line 3: (ch0 autoselect), default ch3
//line 4: for setting B
        SetChanADC1(ADC_CH123_NEG_SAMPLEA_VREFN & ADC_CH123_POS_SAMPLEA_0_1_2 &
                ADC_CH123_NEG_SAMPLEB_VREFN & ADC_CH123_POS_SAMPLEB_3_4_5,
                ADC_CH0_POS_SAMPLEA_AN3 & ADC_CH0_NEG_SAMPLEA_VREFN &
                ADC_CH0_POS_SAMPLEB_AN3 & ADC_CH0_NEG_SAMPLEB_VREFN);

        return 0;
}

// volatile unsigned int DMAbufferF = 0;
void __attribute__((__interrupt__,no_auto_psv)) _DMA0Interrupt(void)
{
        IFS0bits.DMA0IF = 0; //Clear the DMA0 Interrupt Flag
/*      if(DMAbufferF == 0)
        { // proc buffer A
                ADCbufferA[16]=phase;
        }
        else
        { // proc buffer B
                ADCbufferB[16]=phase;
        }
        DMAbufferF ^= 1;
        LATBbits.LATB4=(DMAbufferF)?1:0;*/
}


// input 16bit output 16bit
// cos=1-(79/64)x^2+(16/64)x^4-(1/64)
int Cosine(int x)
{
//      return x;
        int neg=0;
        long a,a2,r;
        if(x==0x8000) return -32767;
        if(x<0) x=-x;
        if(x==0x4000) return 0;
        if(x>0x4000)
        {
                neg=1; /* rev */
                x=0x8000-x;
        }
        a=x;   // a=x, 14Fp
        a = (a*a +0x1000)>>13; // x^2 ,15FP
        a2= (a*a +0x4000)>>15; // x^4 ,15FP
        r = (a*a2+0x4000)>>15; // x^6 ,15FP
```

```
        r+= 79*a;  r-=(a2<<4);
        r=(r+0x20)>>6;    // /64, 15fix
        if(r&0x8000) r=0x7fff;
        if(r<1) r=1;
        x=r;

        if(neg) return -(0x7fff-x);
        else    return   0x7fff-x;
}
```

File: i2c.c

```
#include "i2c.h"

volatile char i2cUpdate;
unsigned char RAMBuffer[256];
unsigned char *RAMPtr;

struct FlagType Flag;

void i2c1_init(void){
  I2C1CON=0x8200;

  I2C1ADD = addr;

  IFS1 = 0;
  int i;
  for (i = 0; i < 256; i++) {
    RAMBuffer[i]=0;
  }

  RAMPtr = &RAMBuffer[0];
  _SI2C1IF=0;
  _SI2C1IP=6;
  Flag.AddrFlag = 0;
  Flag.DataFlag = 0;

  _SI2C1IE = 1;
  i2cUpdate = 0;
}

void __attribute__((interrupt,no_auto_psv)) _SI2C1Interrupt(void)
{
  unsigned char Temp; //used for dummy read

  if((I2C1STATbits.R_W == 0)&&(I2C1STATbits.D_A == 0))  //Address matched
  {
    Temp = I2C1RCV;   //dummy read
    Flag.AddrFlag = 1;  //next byte will be address
  }
  //Write request
  else if((I2C1STATbits.R_W == 0)&&(I2C1STATbits.D_A == 1)) //check for data
  {
    if(Flag.AddrFlag)
    {
```

```
      Flag.AddrFlag = 0;
      Flag.DataFlag = 1;  //next byte is data
      RAMPtr = &RAMBuffer[0] + I2C1RCV;
    }
    else if(Flag.DataFlag)
    {
        *RAMPtr = (unsigned char)I2C1RCV;// store data into RAM
        /* Flag.DataFlag = 0; */
        Flag.AddrFlag = 0;//end of tx
        /* RAMPtr = &RAMBuffer[0]; //reset the RAM pointer */
        RAMPtr++;
        i2cUpdate = 1;
    }
  }
  //Read Request
  else if (((I2C1STATbits.R_W == 1))&&((I2C1STATbits.D_A == 0)||(!I2C1STATbits.P)))
  {
    Temp = I2C1RCV;
    I2C1TRN = *RAMPtr;  //Read data from RAM & send data to I2C master device
    I2C1CONbits.SCLREL = 1; //Release SCL1 line
    while(I2C1STATbits.TBF);//Wait till all
    /* RAMPtr=&RAMBuffer[0]; //reset the RAM pointer */
    RAMPtr++;
  }
  _SI2C1IF = 0; //clear I2C1 Slave interrupt flag
}

inline unsigned int read_8(unsigned char buf)
{
  return RAMBuffer[buf];
}
inline unsigned int read_16(unsigned char buf)
{
  return (RAMBuffer[buf] + (RAMBuffer[buf+1]<<8));
}
inline unsigned long read_32(unsigned char buf)
{
  unsigned long ch1 = (unsigned long)read_16(buf);
  unsigned long ch2 = (unsigned long)read_16(buf+2);
  return ((unsigned long)ch2<<16)|ch1;
}
inline void write_8(unsigned char buf, unsigned char val)
{
  RAMBuffer[buf] = val;
}
inline void write_16(unsigned char buf, unsigned int val)
{
  RAMBuffer[buf++] = val&0xff;
  RAMBuffer[buf] = val>>8&0xff;
}
inline void write_32(unsigned char buf, unsigned long val)
{
  RAMBuffer[buf++] = val&0xff;
  RAMBuffer[buf++] = val>>8&0xff;
  RAMBuffer[buf++] = val>>16&0xff;
```

```
  RAMBuffer[buf] = val>>24&0xff;
}
```

File: main.c

```
#include <main.h>

// Three phase driver test

// option bits, -> .h
_FBS(RBS_NO_RAM & BSS_NO_BOOT_CODE & BWRP_WRPROTECT_OFF); // no change
_FSS(RSS_NO_RAM & SSS_NO_FLASH & SWRP_WRPROTECT_OFF);     // no change
_FGS(GSS_OFF & GCP_OFF & GWRP_OFF); // probably no change
_FOSCSEL(FNOSC_PRI & IESO_ON);      // clock option 1
_FOSC(FCKSM_CSECMD & IOL1WAY_OFF & OSCIOFNC_ON & POSCMD_HS); // clk opt main
_FWDT(FWDTEN_OFF & WINDIS_OFF & WDTPRE_PR128 & WDTPOST_PS32768); // WDT
_FPOR(PWMPIN_ON & HPOL_OFF & LPOL_OFF   // PWM init state, should be condidered
      & ALTI2C_OFF & FPWRT_PWR64); // not significant
_FICD(JTAGEN_OFF & ICS_PGD3);  // ICS should be set to using PG port

_FUID0(0x19); _FUID1(0x74); _FUID2(0x03); _FUID3(0x31);  // user id

//Global Variables



int main(void)
{
        int i,j,repc;
        long ns;
        long lastclock;
        long cyclerest;
        int enabled, turnon;   // 1 if enabled with life, 1 when just enabled
        int vce; // vector control enable
        unsigned int mode=0;
//      unsigned int buff[16];
  phase = 0;

        // InitializeHardware -> TPHardware.c
        LATB=0x0;
        TRISB=0x037f;  // RB10-15 out(PWM), RB7=out(test point, debug)
        TRISA=0x000f;  // RA4 out(Fault clear)
        PORTAbits.RA4=1;
        ClockUp();
        SetupMCPWM();
        DisablePWM();
        SetupADC();
  i2c1_init();
        for(i=0;i<16;i++) ADCbufferA[i]=i*0x11;
        i=0;
        while(PWMsysclock<2000) ;
        PORTAbits.RA4=1;  // fault clear
        EnablePWM();
        // /INIT
        write_8(Omega,50);
```

```
      write_8(Amplitude,100);
      write_8(Mode, 1);
      write_16(Lifetime, LIFETIME_INIT);
      // vector control param
      write_16(LR_Param,500);
      write_16(Daxis_Igain,5000); write_16(Daxis_Pgain,4);
      write_16(Qaxis_Igain,5000); write_16(Qaxis_Pgain,4);
ResetVector();


      InitSD();
      ns=PWMsysclock;

      enabled=0; turnon=0; repc=0;
EnablePWM();
      while(1)
      {
              // lock to 10kHz, 20kHz 2cycle.
              ns+=2;
              cyclerest=ns-PWMsysclock;   // it should be >=0
              //anytime, but single -1 should be fine
              while(ns>PWMsysclock)
                      ;
              i+=2;
  /* enabled=1; */
              write_16(Cycle_Count,read_16(Cycle_Count)+1);
              //LATB=(LATB&0x03ff)|(i&0xfc00);
              LATBbits.LATB7=(phase &0x8000)?1:0;

              // life count
              turnon=0;

              PORTAbits.RA4=1;
              if(read_16(Lifetime)>0)
              {
                      if(!enabled) { turnon=1; EnablePWM(); phase=0; InitSD();
                      PORTAbits.RA4=0; }
                      enabled=1;
                      write_16(Lifetime,read_16(Lifetime)-1);
               }
               else
               {
                      DisablePWM();
                      SetDutyCycle(0,0,0);
                      if(enabled) write_8(Mode,(read_8(Mode)&0xff00)|CM_Idle);
                      // ->change mode to Idle when timeout */
                      enabled=0;
                      ResetVector();
               }

              mode=read_16(Mode);
              /* mode=2; */
              vce=0;
              switch(mode&0xff)
              {
```

```
                case CM_Idle:    // idling
                        SetDutyCycle(0,0,0);
      phasedelta=0;
                        write_16(Cos_Th,Cosine(read_16(Theta)));  // cos func check
                        break;
                case CM_Manual:  // manual output set
                        phasedelta=read_16(Omega);
      unsigned long amp = read_16(Amplitude);
      SetOutput(SDR16((unsigned long)Cosine(phase-0)*amp),SDR16((unsigned long)
      Cosine(phase-21845)*amp),SDR16((unsigned long)Cosine(phase-43691)*amp));
      /* SetManualOutput(((float)amp/255)*(unsigned int)(Cosine(phase)+0x8000),
      ((float)amp/255)*(unsigned int)(Cosine(phase-21845)+0x8000),
      ((float)amp/255)*(unsigned int)(Cosine(phase-43691)+0x8000)); */
                        break;
                case CM_Vector: // vector control
                        vce=1;
                        break;
                }
                j=VectorControl(vce);
                j=CalcSD();
    if (j)
      InitSD();
    write_vector_i2c();
                if(i2cUpdate) {
                        write_16(Lifetime, LIFETIME_INIT);
                        i2cUpdate = 0;
                }
        }
}
```

File: vector.c

```
#include "vector.h"
#include "main.h"


// ================================================
// vector detection

int Vectorth;
long Vid,Viq,Vomega;
int Er,Es,Et;
long Ed=30,Eq=30;
long IEd=0,IEq=0;
long Emax=100;
void ResetVector(void)
{ Vectorth=0; IEd=0; IEq=0; Emax=100; };

void write_vector_i2c(){
  write_32(Vector_TH,Vectorth);
  write_32(V_Id, Vid);
  write_32(V_Iq, Viq);
  write_32(V_Omega, Vomega);
  write_32(E_r, Er);
  write_32(E_s, Es);
```

```
  write_32(E_t,Et);
  write_32(IE_d,IEd) ;
  write_32(IE_q,IEq) ;
  write_32(E_Max,Emax);
  write_16(AD_0,AD[0]);
        write_16(AD_1,AD[1]);
        write_16(AD_2,AD[2]);
        write_16(AD_3,AD[3]);
        write_16(AD_4,ADB[2]);
}



void VectorDetect(int iR,int iS,int iT, int outf)
{
  int id_c;//, iq_c;
  long iq_c;
  int ir,is,it;
        long ia,ib,Ea,Eb,e;
        int c,s;
  id_c = read_16(Comm_Id);
  iq_c = (long) read_32(Comm_Iq);
      int omega = read_16(Omega);

        // Inverse Clarke transform
        // ia=(2*iR-iS-iT)/3
        // ib=(iS-iT)/sqrt3
        ia=SDR10((long)(2*iR-iS-iT)*21845);  // 21845/65536 << 6
        ib=SDR10((long)(iS-iT)*37837);      // 37837/65536 << 6

        // vector angle detect
        //vectorth=phase;
  //Inverse Park Transform
  //Vid = vector id
  //Viq = vector iq
        c=Cosine(Vectorth); s=Cosine(16384-Vectorth);
        Vid=SDR15( c*ia+s*ib);
        Viq=SDR15(-s*ia+c*ib);
        //if(Vid<0) { Vid=-Vid; Viq=-Viq; Vectorth+=0x8000; }
        if(Vid==0) Vid=1;
        if(Vid<0)
        {
    if(Viq>0) {
      Vid=-Vid;
      Viq= 10*Vid;
    }
    else {
      Vid=-Vid;
      Viq=-10*Vid;
    }
                Emax-=20;
        }
        if(Viq> Vid*10){
    Viq= Vid*10;
    Emax-=10;
```

```
  }
        if(Viq<-Vid*10) {
    Viq=-Vid*10;
    Emax-=10;
  }
        Vomega=(Viq<<10)/Vid;
        Vomega=SDR10(Vomega*(long)read_16(LR_Param));
        Vectorth+=Vomega-omega;

        Emax++;

        // 22V, 100/500
        if(Emax<40) Emax=40;
        if(Emax>EMAX_MAX) Emax=EMAX_MAX;
        if(!outf)  return;

        // vector detection
        // --------------------------------
        // control output


#define INTMAX  1280001 // 500*256

        // vector PI feedback
        // Id current on d-axis
        e=((long)(id_c)-Vid);
        IEd+=SDR12((long)(read_16(Daxis_Igain))*e);
        if(IEd> INTMAX) IEd= INTMAX;
        if(IEd<-INTMAX) IEd=-INTMAX;
        Ed=SDR8(IEd+e*(long)(read_16(Daxis_Pgain)));
        if(Ed<-Emax) Ed=-Emax;
        if(Ed> Emax) Ed= Emax;

        // Iq current on q-axis
        e=((long)(iq_c)-Viq);
        write_32(COMMTEST,e);
        //write_32(COMMTEST,iq_c);
        IEq+=SDR16((long)(read_16(Qaxis_Igain))*e);
        if(IEq> INTMAX) IEq= INTMAX;
        if(IEq<-INTMAX) IEq=-INTMAX;
        Eq=SDR8(IEq+e*(long)(read_16(Qaxis_Pgain)));
        if(Eq<-Emax) Eq=-Emax;
        if(Eq> Emax) Eq= Emax;

        // vector output transform
  //Park Transform
        Ea=SDR15( c*Ed-s*Eq);
        Eb=SDR15( s*Ed+c*Eq);
  //Clark Transform
        Er=Ea;
        Es=(-(Ea<<7)+Eb*222)>>8;
        Et=(-(Ea<<7)-Eb*222)>>8;
        SetOutput(-Er,-Es,-Et);   // sometimes need '-' otherwise not
}
```

```
long ViqS,VidS,VomegaS;
int VSc;
int VectorControl(int outf)
{

        int iR,iS,iT;

        iR=(AD[1]-515);
        iS=(AD[2]-515);
        iT=(AD[0]-515);


        VectorDetect(iR,iS,iT,outf);
        if(VSc>=4096)
        {
                VSc=0;
    write_8(D1,0x01);
                VidS=ViqS=VomegaS=0;
        }
        VidS+=Vid; ViqS+=Viq; VomegaS+=Vomega;
        VSc++;
        if(VSc==4096) return 1;
        return 0;
}

// synchro detection
int SDcycle;
// (65536/8)/2   65536=base cycle, 8=base freq /2clock

long SDCS[SDNUM], SDSS[SDNUM];

void InitSD(void)
{
        int i;
        SDcycle=0;
        for(i=0;i<SDNUM;i++)
        {  SDCS[i]=SDSS[i]=0; }
}

int CalcSD(void)
{
        int i;
        unsigned int ph=phase;
        long c,s;


        c=Cosine(ph);       // cosine
  s=Cosine(16384-ph); // sine
  for(i=0;i<SDNUM;i++)
        {
                SDCS[i]+=SDR10(c*AD[i]);
                SDSS[i]+=SDR10(s*AD[i]);
        }

        SDcycle++;
```

```
        if(SDcycle==SDSN) return 1; // finish
        else return 0;
}
```

File: TPHardware.h

```
#ifndef _TPHARDWARE_H_
#define _TPHARDWARE_H_


#include <p33Fxxxx.h>
#include <uart.h>
#include <pwm12.h>
//#include "slnode11dspic.c"
/* #include "slnode.h" */
#include "adc.h"
// sub source for TPDriver.c
//
//Variables

extern volatile long PWMsysclock;
extern volatile unsigned int phase;
extern volatile unsigned int phasedelta;
extern volatile int AD[4];
extern volatile int ADB[4];

//Functions
void IORemapping(void);
void SetMCPWM1DeadTimeAssignment(unsigned int config);
void SetupMCPWM(void);
void __attribute__((__interrupt__, no_auto_psv)) _MPWM1Interrupt(void);
void SetDutyCycle(unsigned int ph1,unsigned int ph2,unsigned int ph3);
void SetOutput(int ph1,int ph2,int ph3);
void SetManualOutput(unsigned int ph1,unsigned int ph2,unsigned int ph3);
void DisablePWM(void);
void EnablePWM(void);
extern unsigned int __attribute__((space(dma))) ADCbufferA[17],ADCbufferB[17];
int SetupADC(void);
void __attribute__((__interrupt__,no_auto_psv)) _DMA0Interrupt(void);
void ClockUp(void);
int Cosine(int x);

void OnPWMCycle(void);
#endif
```

File: i2c.h

```
#ifndef _I2C_H
#define _I2C_H

#include <stdint.h>
#include <p33Fxxxx.h>
```

```
extern unsigned char RAMBuffer[256];

struct FlagType{
  unsigned char AddrFlag:1;
  unsigned char DataFlag:1;
};

extern struct FlagType Flag;



void i2c1_init(void);
void __attribute__((interrupt,no_auto_psv)) _SI2C1Interrupt(void);
void write_8(unsigned char buf, unsigned char val);
void write_16(unsigned char buf, unsigned int val);
void write_32(unsigned char buf, unsigned long val);
unsigned long read_32(unsigned char buf);
unsigned int read_16(unsigned char buf);
unsigned int read_8(unsigned char buf);

#endif
```

File: main.h

```
#ifndef _MAIN_H_
#define _MAIN_H_

//Includes:
#include <p33Fxxxx.h>
#include <uart.h>
#include <pwm12.h>
#include "adc.h"
#include "vector.h"
#include "i2c.h"

#include "TPHardware.h"
//Defines:
extern volatile char i2cUpdate;
#define SDR8(a)  (((a)+  0x7f)>> 8)
#define SDR9(a)  (((a)+  0xff)>> 9)
#define SDR10(a) (((a)+ 0x1ff)>>10)
#define SDR11(a) (((a)+ 0x3ff)>>11)
#define SDR12(a) (((a)+ 0x7ff)>>12)
#define SDR13(a) (((a)+ 0xfff)>>13)
#define SDR14(a) (((a)+0x1fff)>>14)
#define SDR15(a) (((a)+0x3fff)>>15)
#define SDR16(a) (((a)+0x7fff)>>16)

//I2C Register Map
//16 bit fields
#define Cycle_Count 0
#define Lifetime 2
#define LR_Param 4
#define Daxis_Igain 6
#define Daxis_Pgain 8
```

```
#define Qaxis_Igain 10
#define Qaxis_Pgain 12
#define Mode 14
#define Theta 16
#define Cos_Th 18
#define Omega 20
#define Amplitude 22
#define Comm_Id 24
//#define Comm_Iq 26
#define AD_0 28
#define AD_1 30
#define AD_2 32
#define AD_3 34
#define Vector_TH 36
#define AD_4 38 //this should be the thermistor

//32 bit fields

#define COMMTEST 42
#define Comm_Iq 46
#define V_Id 50
#define V_Iq 54
#define V_Omega 58
#define E_r 62
#define E_s 66
#define E_t 70
#define E_d 74
#define E_q 78
#define IE_d 82
#define IE_q 86
#define E_Max 90
#define SDCS_0 94
#define SDCS_1 98
#define SDCS_2 102
#define SDCS_3 106

//Debug Flags
#define D1 128
#define D2 129
#define D3 130
#define D4 131
#define D5 132
#define D6 133
#define D7 134
#define D8 135


#define LIFETIME_INIT 9000 // should result in 0.9 s timeout

#endif

//  RegFileL[0]= Cycle count
//  RegFileL[1]= Lifetime (in 10kHz, 0.1ms)
//  RegFileL[3]= LR parameter
//  RegFileL[4]= D axis I gain
```

```
//  RegFileL[5]= D axis P gain
//  RegFileL[6]= Q axis I gain
//  RegFileL[7]= Q axis P gain

//  RegFileS[1]= Mode { report mode:16bit  control mode:16bit }
//      control mode 0:idle, 1:manual omega/amp, 2:vector id/iq
//  RegFileS[2]=
//  RegFileS[3]= Cosine(RegFileS[2]);  in Idle state for test
//  RegFileS[4]= Omega      in Manual state
//  RegFileS[5]= Ampletude  in Manual state

//  RegFileS[4]= OmegaForward
//  RegFileS[6]= Command Id
//  RegFileS[7]= Command Iq
```

File: vector.h

```
#ifndef _VECTOR_H_
#define _VECTOR_H_

#include <p33Fxxxx.h>
#include <uart.h>
#include <pwm12.h>
#include "adc.h"
#include "TPHardware.h"
#include "main.h"

#define SDNUM 4
#define SDSN  4096
#define CM_Idle   0
#define CM_Manual 1
#define CM_Vector 2
#define EMAX_MAX 2200//#Emax acts as a floating torque limit...
//increase this to increase max torque


extern int Vectorth;
extern long Vid,Viq,Vomega;
extern int Er,Es,Et;
extern long Ed,Eq;
extern long IEd,IEq;
extern long ViqS,VidS,VomegaS;
extern int VSc;


extern int SDcycle;
extern long SDCS[SDNUM], SDSS[SDNUM];

void ResetVector(void);
void VectorDetect(int iR,int iS,int iT, int outf);
int VectorControl(int outf);
void InitSD(void);
int CalcSD(void);
#endif
```

Bibliography

[1] Umashankar Nagarajan, George Kantor, and Ralph Hollis. The ballbot: An omnidirectional balancing mobile robot. *The International Journal of Robotics Research*, 33(6):917–930, May 2014.

[2] Masaaki Kumagai and Takaya Ochiai. Development of a robot balancing on a ball. In *Control, Automation and Systems, 2008. ICCAS 2008. International Conference on*, pages 433–438. IEEE, 2008.

[3] Ankit Bhatia, Masaaki Kumagai, and Ralph Hollis. Six-stator spherical induction motor for balancing mobile robots. In *Proceedings 2015 IEEE International Conference on Robotics and Automation*, pages 226 – 231. IEEE, 2015.

[4] Greg Seyfarth, Ankit Bhatia, Olaf Sassnick, Mike Shomin, Masaaki Kumagai, and Ralph Hollis. Initial results for a ballbot driven with a spherical induction motor. In *Robotics and Automation (ICRA), 2016 IEEE International Conference on*, pages 3771–3776. IEEE, 2016.

[5] Tom Lauwers, George Kantor, and Ralph Hollis. One is enough! In *Proc. Intl. Symp. for Robotics Research*, pages 12–15, 2005.

[6] S Doessegger, P Fankhauser, C Gwerder, J Huessy, J Kaeser, T Kammermann, L Limacher, and M Neunert. Rezero. *Focus Project Report, Autonomous Systems Lab., ETH Zurich, Switzerland*, 2010.

[7] Masaaki Kumagai and Ralph L Hollis. Development and control of a three dof spherical induction motor. In *Robotics and Automation (ICRA), 2013 IEEE International Conference on*, pages 1528–1533. IEEE, 2013.

[8] Masaaki Kumagai and Ralph L Hollis. Development of a three-dimensional ball rotation sensing system using optical mouse sensors. In *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, pages 5038–5043. IEEE, 2011.

[9] Masaaki Kumagai and Ralph L Hollis. Development and control of a three dof planar induction motor. In *Robotics and Automation (ICRA), 2012 IEEE International Conference on*, pages 3757–3762. IEEE, 2012.

[10] M Kumagai. Development of a linear induction motor and a vector control driver. In *SICE Tohoku chapter workshop material*, pages 262–9, 2010.

[11] Shegeki Toyama, Shigeru Sugitani, Zhang Guoqiang, Yasutaro Miyatani, and Kentaro Nakamura. Multi degree of freedom spherical ultrasonic motor. In *Robotics and Automation, 1995. Proceedings., 1995 IEEE International Conference on*, volume 3, pages 2935–2940. IEEE, 1995.

[12] Kok-Meng Lee, Hungsun Son, Jeffry Joni, et al. Concept development and design of a spherical wheel motor (SWM). In *IEEE International Conference on Robotics and Automation*, volume 4, page 3652. IEEE; 1999, 2005.

[13] K Kaneko, I Yamada, and K Itao. A spherical dc servo motor with three degrees of freedom. *Journal of dynamic systems, measurement, and control*, 111(3):398–402, 1989.

[14] Michael Shomin and Ralph Hollis. Differentially flat trajectory generation for a dynamically stable mobile robot. In *Robotics and Automation (ICRA), 2013 IEEE International Conference on*, pages 4467–4472. IEEE, 2013.

[15] Masaaki Kumagai. Torque evaluation method of spherical motors using six-axis force/torque sensor. In *2016 IEEE International Conference on Robotics and Automation (ICRA)*, pages 2416–2421. IEEE, 2016.

[16] Ralph Hollis. personal communication, 2015.