eCVT electronics

2024-25 / Katie Partee and Cole Abbott
Contact info: katherinepartee2027@u.northwestern.edu /

kgpartee@gmail.com / 708-315-3767 /

coleabbott2026@u.northwestern.edu

Table of Contents

What is an ecvt?

I went into a lot of depth on the basics of a cvt and some basics of electronic principles in my design doc last year, so if you have literally no idea where to start, I recommend reading it (link here) to get a basic background. Honestly I recommend you read it regardless. Not to toot my own horn here, but it's a pretty fun read. Then you should read this whole thing! Unfortunately, I can almost guarantee that this one will be significantly less organized, but hopefully it will be even more informative and fun!

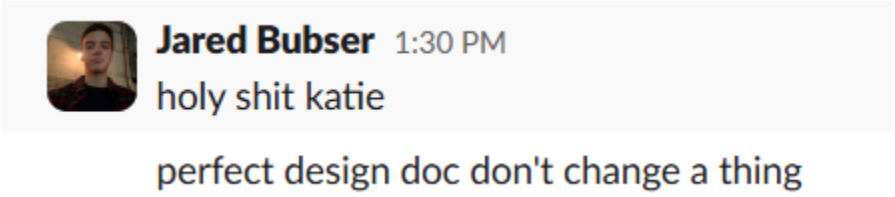

Figure 1: Endorsement from Jared Bubser regarding you reading my design doc(s)

A fair amount of the design specifics have since changed from the initial design doc, so I will give a functional overview here and go into a lot more depth on each part, what it does, how we implemented and tested it, and why we picked it later.

Basically, there are two sets of sheaves (conical metal pieces) connected by a cogged v-belt that slide together or apart to create different gear ratios. In our ecvt, this motion is created by a motor that spins a lead screw. A lead screw translates rotational motion into linear axial motion, and here, one sheave (the primary) is actuated along the axis of the engine shaft while the other (the secondary) remains fixed. The secondary sheave responds to the changing pressure with a spring that allows it to expand or contract (do the opposite motion of the primary). We have a linear bearing and post for stability and to keep things sliding smoothly. We use a linear potentiometer and read the voltage upon startup to get an initial position, then use the encoder built into the motor to read the position throughout operation.

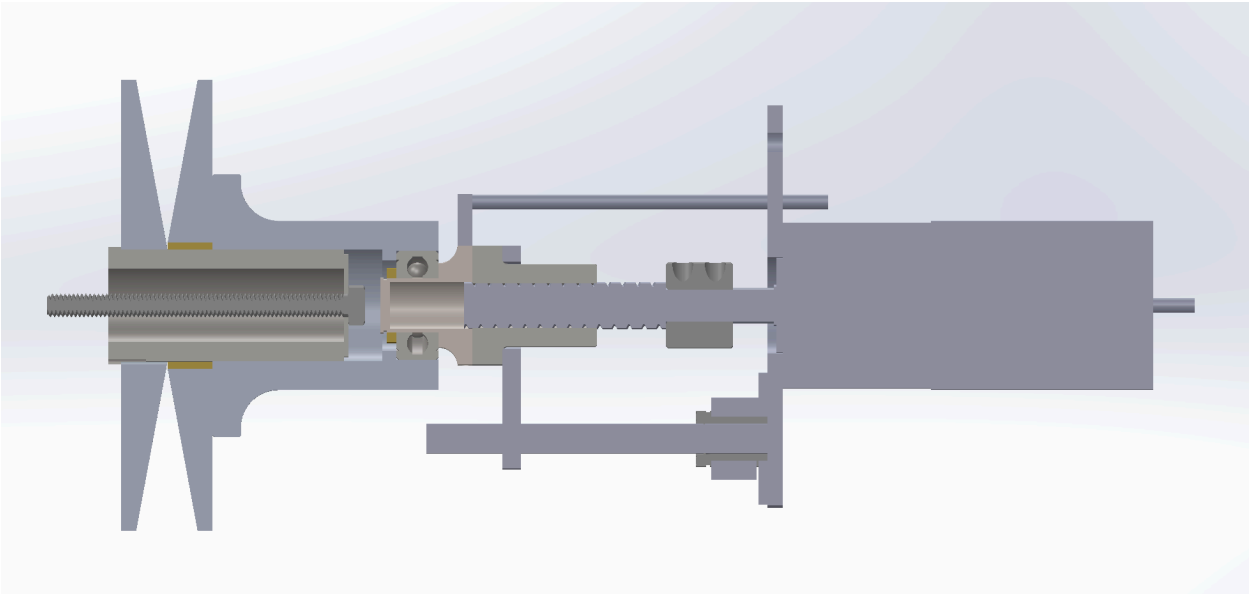

Figure 2: Cross section of primary

We read the RPM of the engine using magnets embedded in the back of the stationary primary sheave and then send both the rpm and the position of the moving primary sheave into the code, which moves the motor (and thus the sheave) to the desired position. Most of the things are mounted on a plate welded to a tube that can bolt to the frame. We have a board box where the rest of the electronics are housed that is mounted behind one of the wings above the splash guard. The batteries were located between the pedals and the hall sensor is mounted against the back of the CVT cover against the engine.

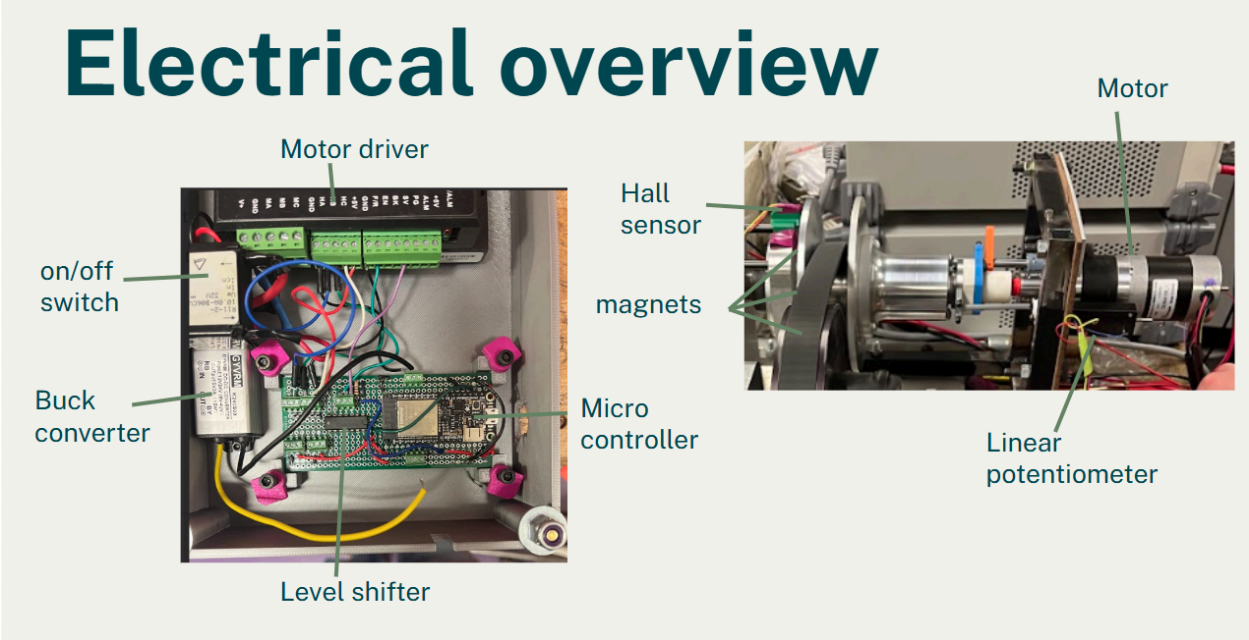

Figure 3: Electrical overview

Part by part breakdown

Motor

In my opinion, the motor is the most important part of your system and thus the thing to pick first and spec everything else off of it. You also need the torque calculations from the mechanical side of things in order to pick a motor that actually provides sufficient side force to hold the belt in place without slipping. The side force should be the minimum torque you look for the motor to provide.

When looking at motors, we first started looking at planetary geared brushless DC motors. This is a very common type of motor, and we needed a geared motor to achieve the amount of torque to create the necessary side force on the belt. As we continued to look into these motors, we found that it was difficult to create the necessary torque (3Nm, but looking to be conservative, motors rated for 5-6Nm) while also having the necessary speed to change position quickly enough (300 rpm, the speed necessary to move the sheaves from fully open to fully closed in 1 second). The motors that have this combination of traits were either very heavy, very expensive, or required a very high current draw.

Because of the drawbacks we discovered with geared brushless DC motors, we began to explore other options. Stepper motors are a great option to get the necessary torque while maintaining a higher speed, but they are quite inefficient (like 10% efficiency). However, hybrid steppers fit our needs perfectly; they were lighter, required less current, and were closer to our budget. By moving forward with a hybrid stepper, we would need to reconfigure our electrical

system, which is one con, so I kept looking at alternatives in the background of my hybrid stepper search (spoiler alert we would not end up going with a hybrid stepper).

As I began to look into stepper motors more closely, I began to realize that the ones that were 24V and hit the torque/speed we needed tended to be around 7-8 lbs, which is a bit too heavy. They also were not at all conservative with torque or rpm requirements, and gave very little wiggle room around our desired values. I was pretty stuck on this type of motor because the coding side seemed really intuitive to me as mentioned above, and as someone who doesn't really know how to code, this was really important to me. However, the closeness to the needed torque/speed and weight meant that this was not a very viable option in the end.

Another solution we looked into was increasing the battery size to 48V, as this would be a much lighter option. Since ebikes run on 48V, it wouldn't be too hard to find a lightweight battery module, which is another benefit over the 24V. However, this also increased the motor price and made it harder to find compatible parts. It also meant we would need to have a two-stage step down process to get to a voltage level compatible with the microcontroller, since it would dissipate a lot of heat energy to step directly from 48V to 3.3V.

After being pretty much set on a stepper, I continued to do a *lot* of googling in search of one that could work at 24V and instead found a geared brushless DC motor that was in our price range. This is the motor that we ended up using. This motor matched all of our criteria: reasonable weight, reasonable current draw, runs on 24V, reasonable price, over 3 Nm of torque, and over 150 RPM. If we had more money, I would have gotten a motor from here: https://www.maxongroup.com/maxon/view/product/370687 (this is what calpoly uses in their ecvt and what my buddy on their team recommended to me). Alas their motors that produced the torque/rpm we needed were like $500.

We were originally planning to use an H-bridge to drive the motor, but this does not work for a BLDC motor since they run off of a three-input system rather than just a power and ground. Instead, we would need to get a specialized motor driver, which is essentially a plug-and-play board of circuitry that translates the microcontroller's outputs into signals that make the motor go. This helps us drive the motor directly from the microcontroller's PWM signal, rather than needing to build our own separate board or doing a bunch of complex circuitry (which would have been a waste of time when we just need a system that works). It also makes the code a lot simpler since it runs essentially off of digital signals to change speed/direction.

Battery

Initially, we were planning to use a 12V power supply, since this was the norm for motors I have worked with before and seemed to be a common number. However, after looking at motors, we realized that there was no way in hell that was happening with our torque/speed requirements. So, instead, we upped our power source to 24V, which is the next most common battery size (if you are already at 12V, for a vehicular application, generally your next option is 24V).

Based on the rules (B.10.1.1.3 - Minimum Voltage and Capacity), we need to have a power source that has sufficient capacity to last the entirety of the endurance race. The battery was the last thing I found since I needed the current draw from all the components to calculate the necessary capacity, and I will walk through the math and other things to consider.

For each electrical part, the current draw should be listed somewhere on the spec. The motor is the only thing that really matters, since everything else in comparison will have a negligible current draw, but for our motor, we have a rated current draw of 7A, and the rest of the components together were around 0.2A. Just to be safe, I rounded up to 10A and for a four hour endurance race, it needs a total capacity of 10A * 4 hrs = 40 Ah. Also because I could not find one 24V battery that met our weight, price, and capacity requirements, I just got two 12V and connected them in series. This worked just fine.

There's also only certain types of permitted batteries (B.10.1.1.5 - Chemistry), and since we didn't want an incredibly heavy battery, I opted for LiFePO4. The batteries also need to be in a hard waterproof case that won't break easily. Most car batteries already come like this so that isn't too much of a concern. I will say the batteries we chose this year broke *very* easily. The case on two of them cracked for reasons unknown to me, so if you want to get the same ones next year, be very gentle in storage and handling. Otherwise, if you can find some that are equally cheap and also fit all the other requirements, I would perhaps purchase those instead.

This power source was intended to power both the motor and the logic. We would use a buck converter to get the voltage down from 12V to the 5V needed by the ESP and hall sensors in the motor. We chose a buck converter over any other sort of voltage regulator because switching converters are much more efficient than any other sort, and a buck converter minimized power loss while switching. It also prevents overheating, which can cause other issues in the system.

Position sensing

There were a few position sensing options that we considered. The first was a series of limit switches. This idea was quickly thrown out because it would be difficult to implement and very imprecise. There were very few benefits to this approach. The second option was an encoder, and the third option was a linear potentiometer. We ended up choosing the secret fourth option which was both a linear potentiometer and an encoder. The encoder was built into the motor, so there wasn't really any work to choose that; there was thought put into choosing the potentiometer. For our application specifically, I wanted something that was both high precision and low noise.

The best types of potentiometers for this are conductive plastic element, polymer film, or cermet type. We ended up with a conductive plastic element since cermet potentiometers tend to give a scratchier reading. In order to pick a specific one, I needed to figure out the required stroke length, which was a little over the maximum travel of the sheaves.

My initial plan was to fully rely on the potentiometer for position sensing, but when we got it on the test setup, we found that no matter how good of a potentiometer it was, sticking right on a vibrating mount is going to introduce a fair amount of noise. So, we used it to sense the position upon startup then read the encoder for position reading in normal operation (see the coding section for more on that).

Hall sensor

My initial idea for the hall sensor function was for it to trigger an interrupt in the code and increment a counter every time a magnet passed by it, meaning it would output high when a magnet passed and low every other time. However, as I began testing the hall sensor that we stole from formula, I realized that what it was actually doing was outputting a steady analog voltage that increased as the magnet got closer. This was when I discovered that there were different kinds of hall sensors.

The kind we wanted was a Latching Hall Sensor (specifically this one from DigiKey), which clamps high when a magnet passes with one polarity and goes back low when a magnet passes with the opposite polarity. There was still a bit of noise there and the high and low were still relative to the distance of the magnet from the sensor, so the interrupt idea was still inviable, but this was a lot easier to work with. I'm sure we could have put in some circuitry to make it able to trigger an interrupt, and that's prob something to look into for next year (on a pcb?!?) but it worked just fine to implement a software threshold.

We then milled out divots in the back of the stationary sheave and press fit/epoxied the magnets in. Something to note here– be very careful with the polarities when installing the magnets!! For our application, we needed them to alternate polarities and we did not do that the first time. Removing the magnets was a very tedious and annoying process, so I recommend you quadruple check the polarities you need and the polarities you are actually installing.

If you do find yourself needing to remove the magnets, we took a small drill bit and drilled until it cracked. If you can remove the cracked pieces at this step, do so. That is the ideal scenario. Instead what happened to us was that most of it just turned to magnet dust which was really annoying to deal with. We ended up just pulverizing the magnet until it completely turned to dust with a dremel and removing the dust with paper towels/other magnets. We scraped out the dust/smaller chunks with a screwdriver once it was broken enough.

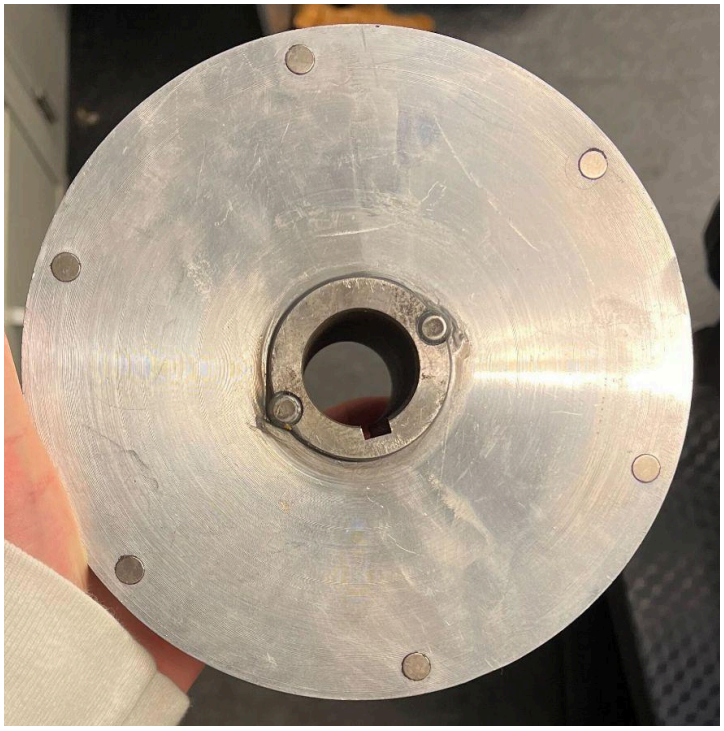

Figure 4: Primary sheave with magnets embedded

Microcontroller

We are using an ESP32 because I had some on hand from EE202. Specifically, it is the NodeMCU dev board. The functionality shouldn't differ between devboards but some of the pinouts do, and it's a lot easier if you just keep it consistent.

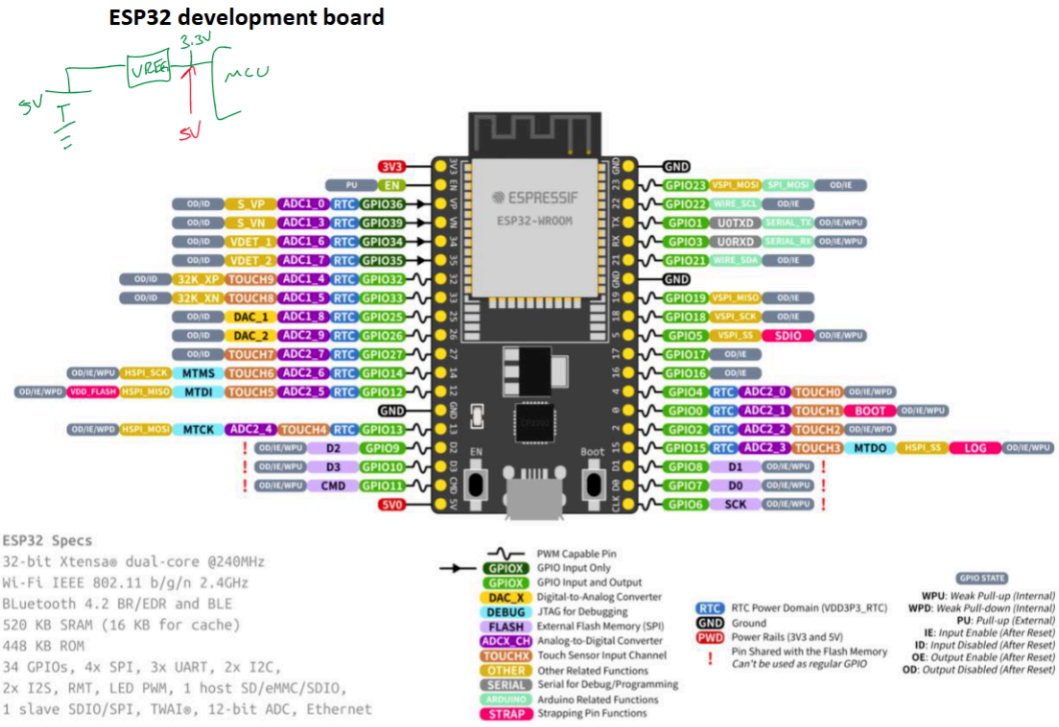

Figure 5: Pinout of our microcontroller.

One thing to be extra careful of– some pins output a signal when code is uploading. We broke things multiple times because of this, so double and triple check everything a pin does before you plug anything into it.

Board

Instead of a PCB, we just soldered all of the relevant components by hand to a proto board. This worked pretty well but it was tedious and rather time consuming. It was also a bit of a mess if we made mistakes in placement/spacing. However, it did enable us to keep testing and modifying on a breadboard for longer which was very necessary to get it functioning as well as it did before going on the car. My biggest advice here would be to lay out everything and mark it/take a picture before you start soldering. I would also recommend soldering all the chips on before you do any of the wires. Another thing is to make sure you keep in mind the orientation of the board in its mounting and where wires are going to be coming from. It will make your life so much easier if you minimize how much wires will overlap when you are plugging things in.

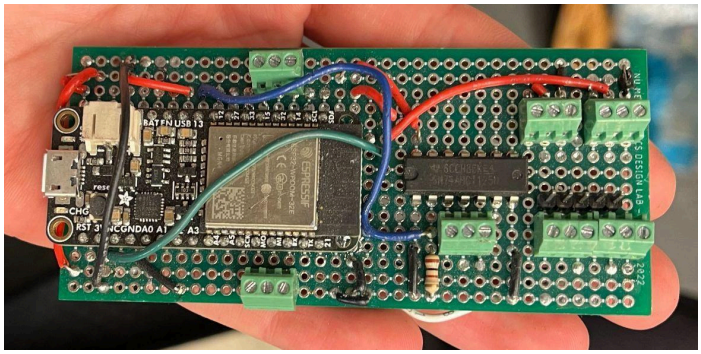

Figure 6: One version of our board, the final one is neater i promise

We used screw terminal connections for any wires that needed to plug into the board. These worked pretty well, and I would recommend using them again. The most difficult part was finding a screwdriver small enough to open/close them. I believe we stole one from the mechatronics lab, but if/when you find one, guard it with your life. We should probably invest in some electronics screwdrivers for baja but oh well that is what theft is for.

Code outline and diagram

To be completely honest, my first attempt at the code was a massive mess. I had never really written in C++ before, I didn't know what a header file was, I used only global variables, everything was in interrupts, and I wrote it all over the summer before I actually had any of the components to test with. I Do Not Recommend doing this. Please for the love of god spare yourself and do not do this. Thankfully, I don't think you should have to start any code from scratch again, and the code that exists should be well-documented (if i have not done so by the time you are reading this pls bug me i really should have done that as i was going which i would recommend you do as well).

I also did not set up github properly to integrate with vs code at first, which you can probably see by the txt files probably floating around in there still. I would highly highly recommend doing that before you start coding anything. Here is the github: GitHub - kgpartee/ecvt. DM me on slack to add you as a collaborator if I have not figured out how to set up an organization by the time you are reading this.

Here is how you integrate github with vs code: Working with GitHub in VS Code Using Git source control in VS Code

Make sure you always pull manually before you start working in vs code because it does not always update to the latest version.

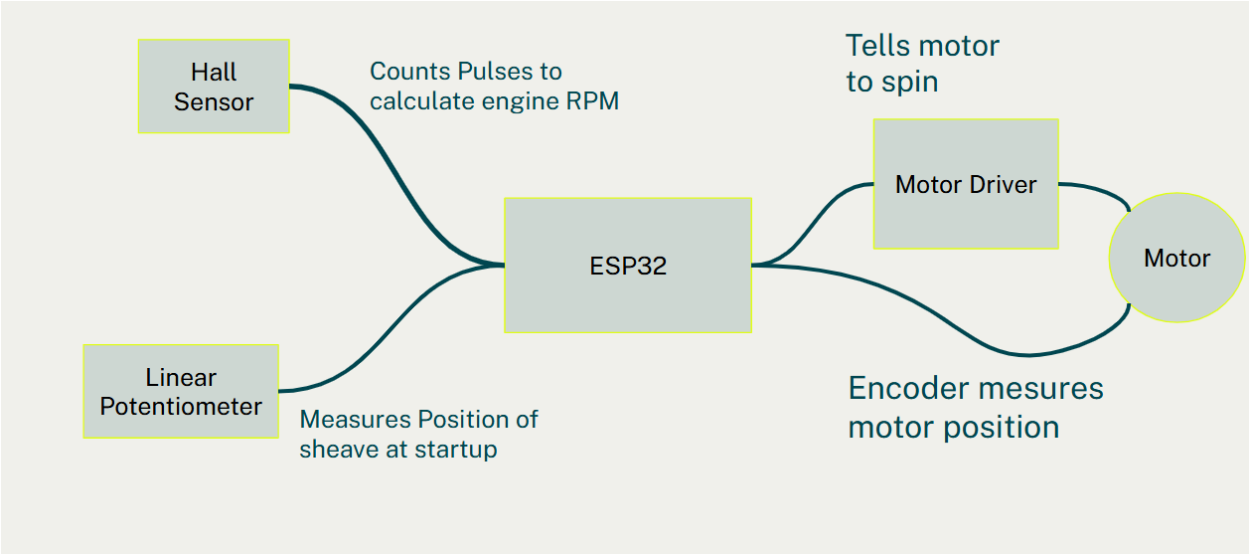

Figure 7: Broad overview of code function

Operational modes

The first mode is idle, which is where the engine is spinning but nothing is being sent to the wheels, so the car is still stationary. This is required to be between 1600-1800 by the rules; on Clank it is currently set to 1750. This means that the sheaves are disengaged, or separated enough that the belt is not clamped until the engine rpm reading gets above 1900 rpm, since we include extra leeway to make sure we are not engaging because of noise or slight changes in the idle speed.

The second phase is low gear, where the car begins slowly accelerating, and the sheaves are engaged but the engine has not yet reached 3000 rpm for normal operation. In our initial implementation, we did not include a low gear. This made acceleration no better than a regular cvt. We then used a clamping function on our pid loop output that introduced a lower bound that linearly scaled with the engine RPM. This forced the sheaves to engage slowly and led to a much smoother acceleration without the delay and jolt that come with a regular cvt.

Next is normal operation. Once the engine reaches 3000 rpm, the cvt will move in and out normally to try and keep it there while adjusting the wheel speeds accordingly. It takes the engine rpm and current position as inputs into a pid loop and outputs the direction and speed the motor needs to spin to get it to the desired position in response to the engine's deviation from 3000 and the sheaves' deviation from the position required to achieve it.

Last is overdrive. Basically, once the sheave reaches its highest position (set in the code based on the potentiometer reading right before the sheaves crash), the sheaves will stay there and the rpm will simply increase linearly. It's like a fixed gear ratio in a regular transmission.

Reading an encoder

The purpose of the encoder is to keep track of the rotational position of the motor. There are different types of encoders (magnetic/hall, optical) but functionally they work the same, by generating 2 sets of pulses (A and B) that pulse as the motor spins. They are offset slightly so you can tell which direction the motor is moving in, see image below.

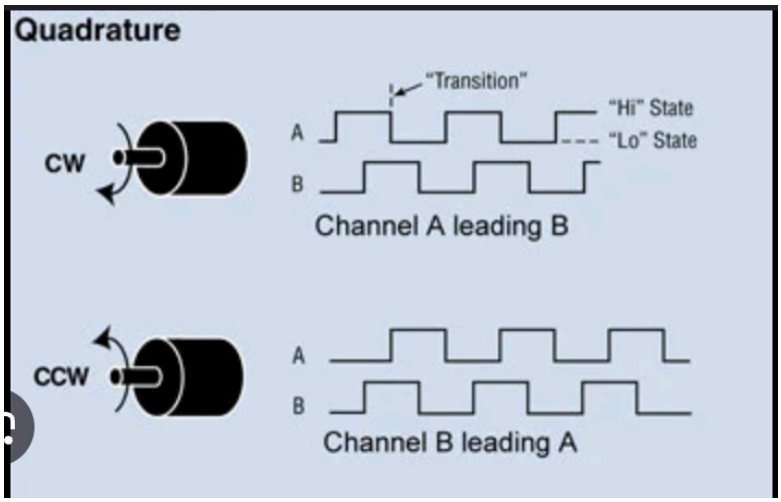

Figure 8: Encoder Quadrature diagram

To keep track of the motor positions, you need to count the number of pulses, and also keep track of if A or B pulse first, and combine these to know direction and distance. IE if A pulses before B, add 1 to the rotation count, and if B pulses before A, subtract 1.

There are a few ways to accomplish this, one could simply poll the gpio pins very fast, but this is very inefficient and requires a lot of CPU time and annoying logic. You could have an interrupt on the pulses, and poll the other channel increment or decrement a counter in the ISR accordingly, but at high motor speeds this would also take a lot of CPU time. Thankfully the ESP provides a peripheral that can do this automatically without the CPU, the Pulse Counter (PCNT).

The pulse counter is a pretty simple peripheral, it just increments or decrements a counter on rising or falling edges. Additionally, there is a “level” signal that can control the action taken on a rising or falling edge. By using one signal as the edge signal and the other as the level signal, a PCNT unit can be used to decode the quadrature signal.

Thankfully there is a library to take care of the setup for us, ESP32Encoder. All we need to do is tell it what pins the A and B signals are connected and it takes care of the rest. You can set a mode, half or full quad, full gets more resolution, so that is what you should use (for some reason we use half this year, I think I just got confused which one was the higher resolution one). The code for this is extremely simple so does not require a new file, The encoder is initialized in pid.cpp in the setup_pid_task function.

Our motor has 5 wires on the encoder header. 2 are power and ground, and 3 are encoder signals, the 3rd is just another signal to get more resolution, it is probably possibly to use all 3 with the ESP32, it would just require some more logic and to write our own encoder library. I don't think it would be worth it but if someone wants to, they should go for it. It would be a good learning experience, and you could make a nice library and post it to github for others to use (or maybe we could just find someone else who has already done it) We just use 2 of the 3, it does not matter which 2, just be consistent because different combos will change the “positive direction” and switching them can cause a runaway motor. Note that all 3 signals must be also connected to the motor driver in the correct order or it will not run.

Other note: We also use the pulse counter for the hall sensor. The 2 libraries we use do not correctly communicate which channel they are using, and trying to use the same one, not good. Solution for this year was to hard code which channel the pulse counter uses into the libraries code (which gets downloaded locally by platformio) however sometimes platformio decides to redownload the library, and delete this modification, not good. Fix this next year.

Reading the hall sensor

Reading the hall sensor is very similar to reading the encoder (the encoder is actually just 2 hall sensors) We also use the pulse counter peripheral, this time with a different library: ESP32PulseCounter which simply uses the pulse counter to count pulses (no using level signal to increment / decrement) This library requires a bit more setup compared to the encoder library, which can be found in pulse_counter.cpp. This code sets up the pulse counter to increment the counter on any pulse from the hall sensor. Note this is a 16 bit counter so it will overflow every ~3.5 minutes at 3000rpm, currently we just ignore this, should prob fix in the future but does not cause any major problems at the moment.

To calculate the engine rpm, we remember the time since the last call to get_engine_rpm, and the amount of pulses since then, and can calculate the rpm from that (basically # pulses/time since last call. Pretty simple calculation). We call this every few milliseconds. Is this the best way of doing it? Probably not. Do I know of a better way? No I do not. Maybe do some research and reevaluate.

PID loops / Motor control

Most electric motors take 2 input signals: a binary direction signal, and an “analog” / PWM signal. For a brushed motor, the PWM is proportional to the voltage, which is somewhat proportional to the torque (torque also depends on the motor speed). For a brushless motor (what we have) the motor driver has a lot more control over what is happening (the motor controller is actually another microcontroller (MCU) with some fancy high current circuits). Regardless, the brushless motor driver takes a PWM signal that is roughly proportional to the motor torque.

So, you want to spin your motor to a certain position, or track some trajectory. How do you do that if the only way to control the motor is with a direction and power signal? In an ideal world, we could calculate we need to have *x*% PWM for *y* seconds to move to position *z*. However there are so many unknown factors that this will not work very well in the real world. Instead we will control our motor based on the difference between the desired position (Setpoint) and the actual position (error).

We use the error to make some response, which is a combination of PWM and direction (PWM is absolute value of response, direction is sign)

The simplest way of doing this is to make the response proportional to the error

$$\text{Result} = K_P * \text{Error}$$

where K_P is a parameter that can be tuned to make the controller behave well. This controller is basically a spring mass system, the “force” (PWM signal) is proportional to the “position” (error is distance to the desired position; the motor / rest of the system is the mass). As you know, if you have just a mass and a spring, the system will oscillate. However, since the system will have some sort of damping it. Sometimes a P controller like this will work fine. To tune a P controller, simply increase the K_p term (stiffen the spring) until the system starts to oscillate/ overshoot the setpoint, then back off the term a bit. If your motor can track the desired trajectory well, then you can call it a day. If the motor is slow or laggy, then you need a stronger response, but you can’t just increase the K_p term because of the overshoot. We need more damping to cancel out that overshoot.

A damper provides a force proportional to the velocity. We can add this with a term proportional to the derivative of the error

$$\text{Result} = K_P * \text{Error} + K_d * \frac{d}{dt}\text{Error}$$

This damping term can help get rid of overshoot, start with it at zero, and once the K_p term starts causing overshoot, start to add K_d until it goes away.

One drawback of an Error based controller is that there always must be some error to get the motor to move, if your controller is well tuned, this might not be an issue. However, if there is some constant force / torque on the motor, then there always must be some error to generate some response to cancel that out. This is called steady state error. To get rid of this, we can add and Integral term

$$\text{Result} = K_P * \text{Error} + K_d * \frac{d}{dt}\text{Error} + K_i \int \text{Error}$$

As the error accumulates, the K_I term will cancel out the steady state error. However, adding a K_i term can introduce bad oscillations and chaotic behavior if not tuned well. For example, if you put a force on the motor that the motor cannot overcome even at 100% power, then the integral term will grow very large. When this force is released, the integral term will dominate the response and the system will act uncontrollably until the integral is canceled out. To prevent this integral windup, a I_MAX term can be used to prevent the integral term from accumulating.

However in our controller we left out the integral term because it always seemed to hurt more than it helped

In addition to using PID control for the motor position, we also use it to control the engine RPM with an input of the RPM error and output of the sheave position. However, I don't think that this worked that well and want to reevaluate for next year. I think the ideal control mechanism is to measure the secondary speed as well, then we can calculate the ideal gear ratio between the primary and secondary that will put the engine RPM at the ideal value. Then we can just move the sheave to the correct position to achieve that ratio. The ideal RPM value will depend on the car's speed. At low speeds a lower engine RPM is ok, and at high speeds a higher RPM is ok, but at our mid range speeds where we are most of the time we want the ideal 3000RPM.

How to use teleplot

Teleplot is a serial plotter extension for VScode. It allows you to plot multiple variables at a time, as well as 2d and 3d plots. All you have to do is print out your variables (with Serial.print) in the form ">name: value\n" this is very useful for PID tuning. Print your setpoint and actual value, drag them onto the same graph, and you can clearly see overshoot, oscillations, and steady state error.

Debugging advice

If you ever spend more than ~1 hour trying to figure something out, write down exactly what the issue was, and if you fix it, what the solution was or what deep hole of some web forum you found the answer in. Trust me on this, it is for your own sanity. If the solution is deleting it completely and rewriting, that is still valuable information. We kept having the same error where the controller would randomly reboot (boot code 0x013 or something like that) and we never wrote down the solution and had to refigure it out every time because we never remembered how to fix it. I still don't remember. Whoops. It's also just generally good design practice to have information to back up your design decisions going forward.

NScope / Oscilloscope

When your code that involves input/output pins doesn't work, (which will inevitably happen a lot) it can be hard to debug because it's hard to tell what is actually happening at the pin. Ie, is my motor not spinning b/c the motor driver is set up wrong, or it is just because the esp32 is not generating a PWM signal.

Using an NScope or Oscilloscope can be a very useful tool to see what is actually happening in your circuit. The NScope is a portable oscilloscope with 4 channels and a function generator. It plugs into your computer and will graph voltage over time.

ESP32 Weird pins

The ESP32 can behave quite strangely sometimes, there are some pins that you should avoid using because they are used for the boot sequence, uploading code, or to communicate with the built in SPI flash memory chip. See this website for a somewhat comprehensive list: esp32-pinout-reference-gpios, however sometimes these pins vary on different dev boards I think. Avoid connecting the motor PWM to a pin that outputs PWM signal at boot, because that will cause your motor to spin uncontrollably and break everything, ask us how we know.

ESP32 Crashing

Sometimes the ESP32 will get stuck in a cycle of crashing, rebooting and crashing again. Each time it will output some register dumps and boot codes, and sometimes an error message. Sometimes these are useful, sometimes not. I think the 2 most common causes of this type of error are incorrectly configuring a peripheral, and using a pin that should not be used. However sometimes these problems can be hard to debug. I usually start by commenting out parts of the code until the problem stops, which will narrow down the problem. Then you can inspect the code to see what the problem is.

In Circuit debugging

Not sure if this is supported on the ESP32, and if it is it may require some extra hardware such as a JTAG programmer. In circuit debugging allows you to add breakpoints to your code, step through, and inspect variables. It can be very useful.

Other debugging tools include the good old fashioned print statement, but be careful using this in an interrupt, because it is slow and can cause the WDT to get angry. You can also use toggling a GPIO pin and looking at it with the NScope, which is helpful sometimes.

Mounting

Board/Box

We mounted the protoboard, buck converter, circuit breaker, and motor driver in a 3D printed box. The box had holes for the screw mounts on the buck converter and driver and usb port on the microcontroller. We secured the microcontroller with corner pieces that fit on top of the board and held it in place. There was a slot for the wires to leave and for the breaker to snap in. This was kind of a slapped-together box, and while it kept things in place, it was definitely not the best use of space.

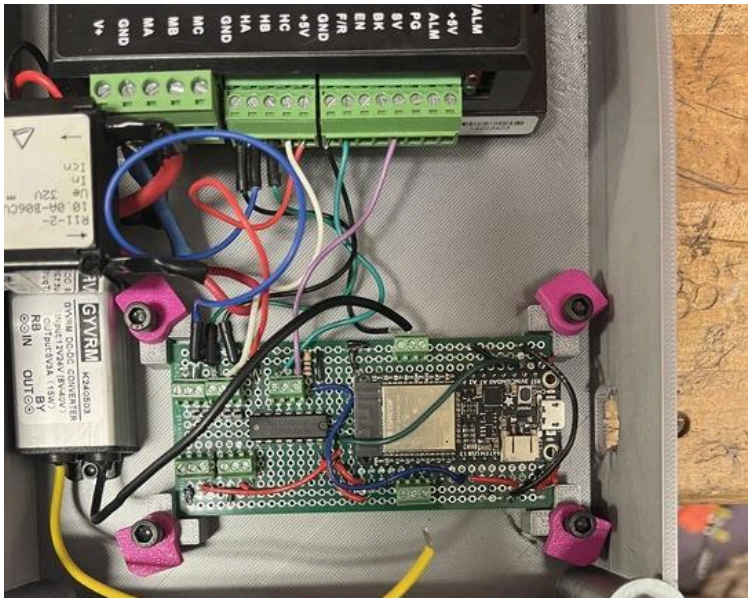

Figure 9: Interior of electronics box

We initially had a lid that bolted into the box and the bolts went through the same tabs that the board was mounted onto. This worked well to keep everything secure, but it was annoying to access anything in the box because we had to unmount it in order to take the lid off. Our second iteration had a snap fit lid, but this also was not ideal since it did not fit and wouldn't snap in properly.

We attached the box to the car with some random extra chassis tabs that we found in some corner of the autobay. We lined them up with the chassis behind the wing panels next to the fuel tank. This worked very well, but in the future I would like their location to be a bit more thought out and not just wherever it fits. It would also be nice to be able to access the box without taking off the wing panel every time we needed to check a wire.

Hall sensor mounting

To mount the hall sensor to the CVT cover, we waterjetted a sheet metal (16 gauge 1010 steel) mount (Fig. 10). The hall sensor itself was placed peeking out of a small rectangular hole at the center of the mount. To hold the hall sensor in place, we essentially sandwiched it between two layers of metal after wrapping the pins and wires in heat shrink tubing.

The mount was attached to the CVT cover via a pair of slots on either side, which were spaced to match the slots attaching the cover to the car. This way, it could be held in place between the existing cover and washers. At the time, the CVT cover was already manufactured, so we wanted some sort of a drop-in solution that wouldn't require modifying it. The slots were convenient in that they (in theory) would allow us to adjust the positioning of the mount to get the best hall sensor reading.

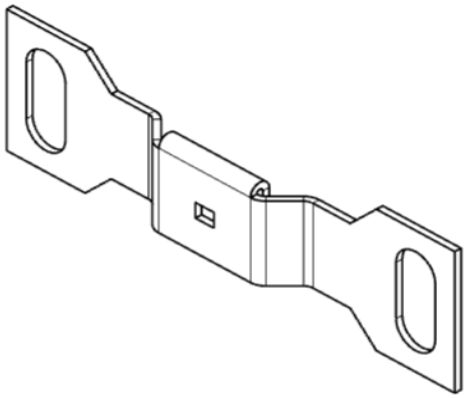

Figure 10: A drawing of the hall sensor mount

Design was pretty rushed, so manufacturing was imprecise. Because the sheet metal we chose was too thick and because we wanted to leave room for the pins without causing damage, we milled out a shallow slot behind the sensor hole. When folding the mount, though, the slot sort of collapsed in on itself, which was unfortunate. In general, folding the mount did not go well, since the side tabs got in the way when we tried bending the top tab to sandwich the sensor in place.

We also found that the mounting slots were not long enough for the hall sensor to be positioned directly underneath the magnets, which was highly problematic. We ended up needing to extend the slots by welding an extra piece of sheet metal to each tab. In the future, note that it is generally a good idea to measure things before waterjetting.

One last problem with the mounting system is that we didn't really have a good way to route the hall sensor wires out of the CVT cover. We ended up passing them through the gap between two of the cover's adjacent surfaces, but even then, they were somewhat loose. It is also generally a good idea to keep loose wires away from rapidly spinning objects, so consider that next time too.

Battery mounting

The rule regarding battery mounting is as follows:
"B.10.1.1.1 - Mounting
All batteries shall be mounted with sound engineering practices and not come loose during normal operation, a collision, or rollover. Battery terminals shall be insulated and protected against an electrical short."

I would say we failed spectacularly here. This was very last minute and thrown together rather poorly. We found a strip of sheet metal and bent it to roughly fit over the top of the batteries and bolt them down to the footplate. We decided to put them between the pedals, because that was where there was room on the car to attach them without putting something heavy too high up. It

worked well enough at first, but during testing, the battery came loose and shorted the system right before we were supposed to test it at Squires. That was not a fun night.

Figure 11: Battery mounting bar, not attached to anything

We iterated on it by adding extra seat foam between the top of the bracket and the batteries for a tighter compressive fit and covered everything in electrical tape to insulate it from future shorts. This worked to have a more secure fit, but while the batteries were installed, the case on one of them cracked, so clearly it was still not the greatest. I think we broke every single subsection of that rule, so perhaps it was a good thing we did not run the ecvt at comp this year, since the battery mounting would not have passed tech.

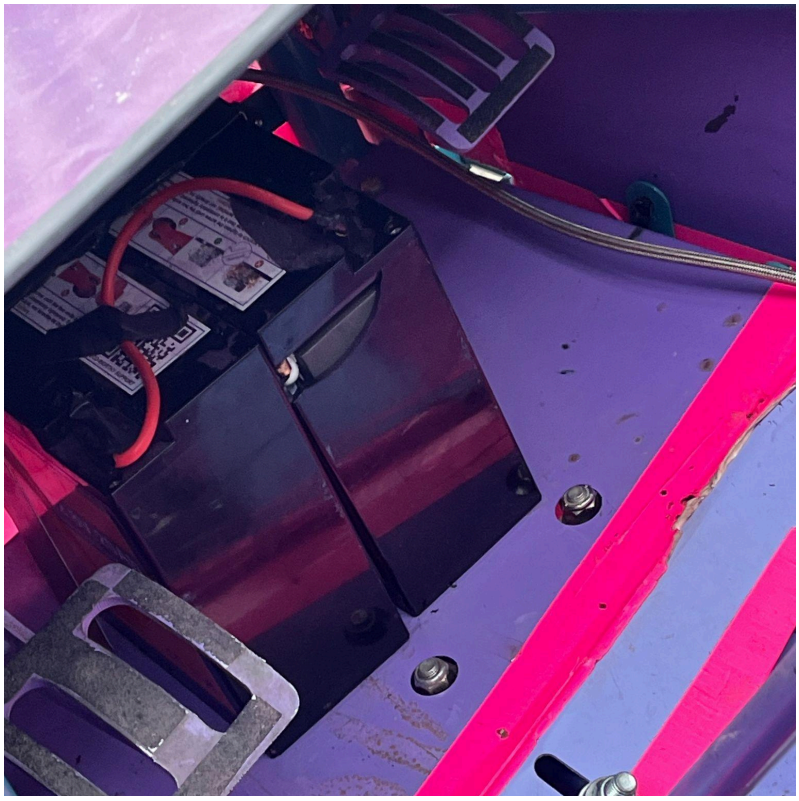

Figure 12: Batteries mounted in the car between the pedals, one is cracked open.

Major takeaways from this are to put thought into it when actually designing the car, make it more robust, and insulate everything. George suggested more of a tray situation where it cups the bottom of the batteries and then has a strap over the top. I think anything would be better than the current sitch.

If you learn nothing else from this section, I hope you learn this: TAPE YOU R DAMN TERMINALS FOR THE LOVE OF GOD TAPE EVERYTHING!!!! Electrical tape is your best friend, and any bit of exposed metal should be thoroughly covered in it. Trust me, this will save you many hours, tears, and headaches if you just insulate everything (you don't even have to insulate it properly– I would argue electrical tape is not ideal insulation) from the get-go.

Testing

This is basically going to be a how-to-validate-electronics section for people who have never had to do that before. I know a lot of this is going to be very basic/common sense but for someone who had never had to break down such a huge project and test in little chunks, I did not realize how little the chunks really should have been. I am hoping this can be a helpful guide, even if it is a bit too basic, on how to start with a really little thing and then build up into a whole project.

Testing set up

Our testing setup was a repurposed cvt tuning stand that I think was a 399 project from way back in the day. I know George had something to do with it at least. Either way, it rotted in north garage for 5 years because we never really tuned our cvt since it worked well enough, until Kevin revived it and we started using it to tune an ecvt. It has two very chunky motors with output shafts that fit the primary and secondary, placed at the same spacing as they would be on the car. There is a metal frame on the primary sheaves where we mounted our motors and sensors on a piece of wood.

Figure 13: Bare test setup

We powered the motor off of a power source that we stole from the mechatronics lab. They are usually chained together, but if you get a star bit from the shop and unscrew one side of the handle on the side of the power source, you can get it off and bring it to the autobay or wherever you are working. Just make sure you remember to bring it back so they don't get mad at us. We ran the "engine" motor off of the ecvt batteries since it required a much higher current draw than the power supplies could give. We did not power the secondary motor so it could act as a load on the system simulating the rest of the powertrain. We found some power sources when we were cleaning out the old daq cabinet though that might work better so it is worth trying those too.

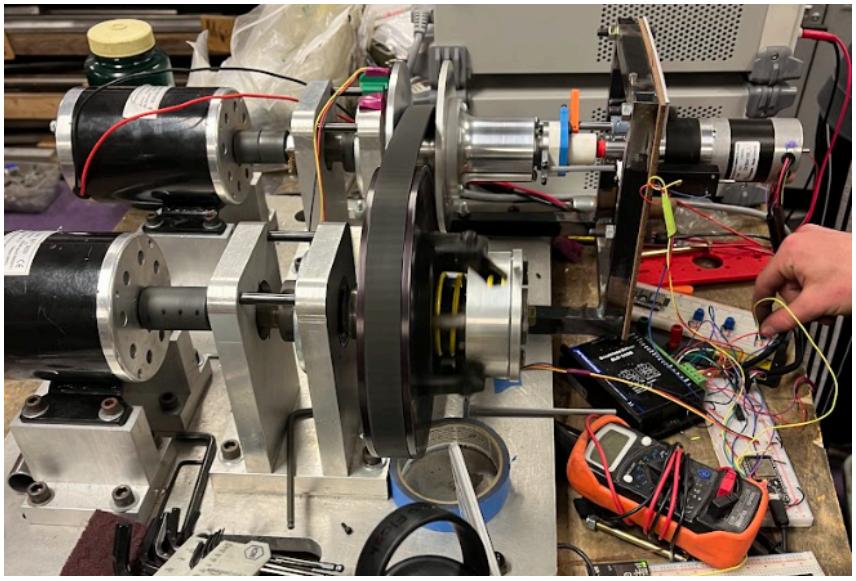

Figure 14: Test setup with everything attached, power sources from the mech lab in the background

Motor testing

The first thing we did was make sure the motor spun. Since it is a BLDC motor, you can't just plug the power and ground into a power supply and make it spin, so we also needed the motor driver to make it work. After we plugged the motor power wires into the driver, we discovered that the encoder wires also needed to be plugged in to make it go. Then it spun! Yippee!

We also used this to figure out how the motor driver works. It needs to have wires in every spot in the bottom two connector segments to function. In the top segment, it needs to be grounded and have a direction (F/R) and PWM (SG) signal. There are potentiometers on the side to tune motor max speed and current protection. We left both of these alone but it's worth trying to tune the motor speed for next year. We also discovered that the top +5V port is an output port not an input port and as such doesn't need to be connected to anything. If something is going wrong with the driver, it will have a red LED. In normal operation, it will have a green LED.

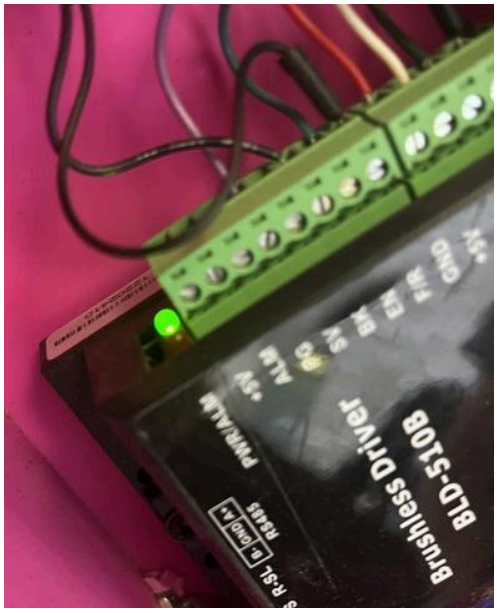

Figure 15: Debugging LEDS on motor driver (currently green)

The next thing to test with the motor was if it could take a PWM signal and change speeds and directions as expected. We ran a sine wave through it, and it did in fact speed up, slow down, and change directions.

Hall sensor

We hooked the hall sensor up to the microcontroller and then plotted the reading on Teleplot. We waved various magnets by it to see how it reacted. Based on the graphs, we determined a threshold value to set as “magnet passed” and set this as a constant in our code.

Once we established this, we tested that the hall sensor would actually increment a counter so we could calculate RPM. We put our code in the loop and had it output the counter and hall sensor analog output to make sure the reading was what we were expecting and the counter was counting up properly. Once we got this working, we implemented the RPM calculation code, and tested this on the test setup.

We tested spacing from the sheave in a similar way; we stuck the hall sensor to the test setup and looked at the output as it ran. We slid it closer/farther from the magnets until it had a minimally noisy result but also wasn’t concerningly close to the spinny bits. We used the spinning setup to test the rpm. First, we spun it by hand so we could actually estimate the rpm that it should be, then we tested it by running the motor. We didn’t have a secondary way of validating that the rpm reading was correct besides it looking right, so it might be worth double checking the reading with a tach but it worked well enough.

Potentiometer

We connected the potentiometer to the microcontroller and from power to ground. We measured the reading when the sliding part was all the way to either end, and found the points where it was at the maximum sheave travel. This testing was more a sanity check than anything else. We used the voltage readings to map to an actual position.

PID tuning

This is gonna be a hell of a section so buckle up. This took actual months. Hopefully you will just need to tweak the tuning we already did but in the event you ever need to tune a PID loop for yourself, here is how we did it:

Our PID controller governs the position of the sheaves; we intake the current position and rpm and output a direction and speed (setpoint) for the motor to turn to get it to the optimal position to keep the rpm at ~3000 rpm, since this is where the engine outputs peak power.

Before you start testing, make sure you include an emergency shutoff in your system. Whether this is a button, a wire that is easy to unplug, or keeping your finger on the power source itself, always be ready to cut power to the motor at a moment's notice.

The basic process is to start by increasing your P term and running your system, then repeating until the output becomes unstable (meaning it does not converge to your setpoint value, it continues increasing or decreasing towards infinity). After that, you want to set it to roughly half the value where it became unstable. The P term helps with delay in the response. Next, you want to start increasing your I term until it becomes unstable again. The I term helps correct steady state error. After that, you want to set it to the highest value where it was still stable. Last, you want to start increasing the D until the oscillations reduce. The D term acts as a damping term.

Once you have done this, you should have a passable system response. However, there is still a lot of tweaking to do most likely. Based on what you are prioritizing (minimal delay, minimal oscillations, minimal steady state error), you can slightly increase the P, I, or D based on what effect you want and see how it responds. It comes to a point where there may be tradeoffs between certain performance traits and you have to make a decision which is the most common use case for the car, or which is more important to the car performing well. Use your own best judgement. If you are interested in more of the math behind this or to get a deeper understanding of feedback systems, I Highly Highly recommend taking EE360. I loved that class and it was mostly meches. Also Randy Freeman is the goat I love that man.

We applied this basic process to a bunch of different test cases, building up to the actual car. First, we tested it without a load attached or spinning the sheaves. We just set the setpoint to a sine wave function and got the motor to trace it. We ignored the rpm input for most of our tuning, since the only thing that affected was the setpoint we fed it, not what it did with the setpoint.

Once we got it working with no load, we fed power to the engine motor and started spinning things. We tuned it against the same sine wave. After that, we tried it with a few different clamped sine waves with different frequencies and amplitudes.

Once we got it working with the repeated signals, we attached a potentiometer to an input pin on the microcontroller and converted the reading from this to our setpoint. This way, we could manually simulate acceleration/deceleration and give it a more random pattern to follow. After this, we also implemented the rpm input by feeding it through a P controller and clamping it at a maximum and minimum sheave position based on the rpm. We determined the idle rpm based on the engine manual and set an idle position whenever we are at or below this rpm.

Finally, we put everything in the car! We tested it first up on the jacks, and one person pulled the throttle while the other watched the graphs and physical shifting. This step required surprisingly minimal tweaking to the constants, which was nice validation for our testing stand. After that, we drove the car with the ecvt. Cole sprinted next to the car watching things shift while I drove, and Konrad watched how the car behaved to acceleration/deceleration. We modified a few things, specifically the low gear, and tuned the rpm P controller, but she was basically ready to go.

Actual observed benefits:

After reading all of this (assuming you actually read this far), you are probably wondering why put in so much work for an ecvt when we already have a mechanical cvt that works just fine? Well, I am so glad you asked, dear reader! We don't have a lot of numbers to back this up, but there were several observed benefits to the ecvt:

Tunability

The only way to customize a regular cvt is by switching out the springs and weights, and then checking the response by driving the car around. This, as I am sure you can imagine, takes forever and is not very precise. With an ecvt, we can beep boop a few numbers then upload it and visually see the response get plotted, which is much more convenient. It is also infinitely more customizable; there are only a few types of springs or weights that we can install, and we can change the pid constants to be whatever we want them to be.

Cheaper

A mechanical cvt costs somewhere in the realm of $2500. That's kind of insane. Our eCVT costs ~$300 for the electronic parts by my incredibly rough estimate. Even with the stock for the mechanical parts and the mechanical secondary, that's a lot cheaper!

Acceleration

With the implementation of a low gear, we were able to have much smoother acceleration. When I was driving it, I noticed much less of a lurch when the car first started, meaning the

sheaves were waiting less time to engage and were engaging at a slower rate. The smoother acceleration means more accurate tuning on lower gear ratios, and is easier to drive.

In order to get the low gear, we used a linearly increasing lower bound on the widest sheave setting that is proportional to the engine rpm when the rpm is below a certain value. This means the car will actually move before it reaches the rpm at which peak power is produced, and the driver doesn't have to accelerate the engine for so long before the cvt actually engages.

More future development

While we put together a really cool project and I am beyond proud of the ecvt, there is so much still to be done to improve it and so many different directions to take it. These are my top priorities for things to implement next year, and some possible ideas for things to do after we get those initial things working.

RPM Control Method

Right now we have that PID loop to control engine RPM based on sheave position, and have some weird logic to clamp to a minimum sheave position based on engine RPM. I don't think this is the ideal solution for a few reasons. It has a lot of latency, is hard and arbitrary to tune, does not respond to hard braking / downshifting well, and it doesn't actually do a great job of keeping us at the ideal engine RPM.

Controlling engine RPM is hard, controlling the sheave position is easier (although I do think we need to do a better job at this, better motor driver might help, also maybe a different motor). Sheave position is directly proportional to the gear ratio, and knowing the angle of incline of the sheaves, we can do some basic trigonometry to calculate the radius where the belt is hitting and get the gear ratio from there. This does not include when the cvt is in overdrive/idle.

Say we know the RPM of the secondary at some moment in time, Then we could calculate the gear ratio that would put the engine at the ideal RPM, then we could move the sheave to achieve that gear ratio. This is different from the current control method because the input to our pid loop would be the gear ratio and our setpoint would change according to the graph below rather than staying static at 3000.

So how do we determine the ideal RPM? This is where we can tune, we can draw a graph, like the one below, that maps wheel speed (proportional to secondary speed) to engine RPM, and just lookup the ideal RPM for some wheel speed

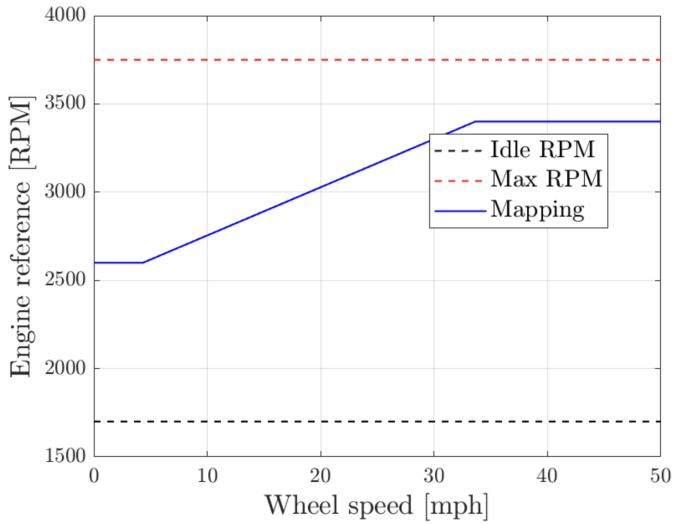

Figure 16: graph from bruin racing github

Manual Mode

While the cvt is rather nifty, it doesn't always know what the car needs in the same way as the driver. For example, there are times where we want to prioritize torque over power, like hill climb or sled pull. Manual control allows the driver to tune the cvt to different functionalities on the fly and as necessary.

The first step in creating a functional manual mode is getting it to shift to a few distinct gear ratios and getting it to follow the torque curve instead of the power curve. My initial idea is to have a low gear setting, an overdrive setting, and a torque max setting. It would be easy to just have a button that triggers each of these that is push on/push off. There would need to be some software that only one mode can be enabled at a time, but that shouldn't be too hard to implement.

The next step is to have complete driver control. What this means is that the driver can move/slide/twist something and the sheaves will move as they do it. Turner suggested something that mimics a regular shifter because it is very important for the driver to have some sort of physical response from the car. Buttons are a lot easier to implement but it's hard for the driver to know whether or not something is actually happening. You could look into a massive potentiometer for the shifter, but that would likely be very noisy and not offer the best control, even if you shove the output through a lot of filtering. There also needs to be something that shifts into driver controlled mode (similar to the section above) that would ideally display on the dash which mode it is in.

Waterproofing

We had basically no waterproofing at all this year, so having it at all is a big step. I would emphasize selecting waterproof commercial off the shelf parts where possible (like the buck

converter, circuit breaker switch, etc) since that's one less thing you need to worry about. This also goes into things like the box that holds everything and where things are mounted to prevent a water/dust/mud intrusion into the electronics system.

Another thing to look into is getting more robust connectors than the shitty ones we stole from the mechatronics lab. Tej (baja alum) works at a company that apparently sells military grade electrical connectors, so we should reach out to them. Alternatively, apparently formula just emails companies and asks for samples of connectors and gets all their shit for free that way, so it's worth reaching out to Anton (their pm for 25-26) about how he does that. Apparently he asked for so many that he had to make a kitten start reaching out on his behalf with a fake name. Lol.

Pedal Pressure Sensors

This year, we had difficulties with downshifting. When the driver slams on the brakes, the PID loop takes too long to respond and move the sheaves apart, and during this delay, the engine stalls. In order to remedy this, I want to have some sort of pressure sensing behind the pedals so when they are depressed above a certain threshold rate, we just slam the sheaves back and override the pid controls. I think just smacking some linear potentiometers behind the pedals would be easiest to implement and would also produce a satisfactory result.

Mechanical secondary tuning

It is possible to tune a mechanical cvt; it is just a pain in the ass so we do not usually do it. However, it is a much smaller pain in the ass to only tune a mechanical secondary. This just involves swapping out different springs and driving the car to see which gets the best response.

Data logging

Currently, when we tune things, we just look at output graphs on the laptop as the system is running. It would be instrumental to be able to log the data as the car is driving then look at it later. I think either switching to microcontrollers with SD card slots or adding some sort of SD card breakout to our current board would be a pretty easy fix.

The hard part here is getting the data to store in a humanly parsable way and/or creating some sort of decoder that can read the data once we have stored it. This is a lot more code heavy than anything, and there are a lot of frameworks that exist out there already. Formula has a very complex way of doing it but they also do things at maximum efficiency, so if we are slightly less concerned about efficiency and also have each board log its own data, I think we could do it with less difficulty.

Very future improvements

Apologies i really cannot even rn so you are getting bullet points in this section. It's not that important anyways compared to the rest of this doc.

- Double actuation - also actuate the secondary sheaves electronically
- Model the system and simulate - make some sort of matlab simulation to tune the system without running it
- Technically solar panels are allowed eyes looking to the side emoji

Appendices

My vision for Arjun/Adam

To next year's ecvt electronics team, this is my brain vomit of what I want you to do as of 5/22/25:

The first and most important thing to implement is pressure sensing behind the pedals and in code differentiating this so when they are depressed at a certain rate the motor slams back (also get a nicer motor that can spin at higher speeds) and we fix the downshifting issue that way bc that is what needs to happen for it to be driveable.

Other things to implement is manual modes, first have a button for certain gear ratios and a high torque mode then once that is working make some sort of physical shifter where the driver can have control of the literal shifting , my initial thought is get a massive potentiometer but that would likely be pretty noisy so that would be something for you to figure out how to implement.

Another thing that needs to happen is better waterproofing and mounting bc that's a mess rn. That was originally a separate project but if its you and cole working on this i will prob just fold it in and have the mechanical person get more involved in that and make it all more cohesive. (katie from the future: it is once again a separate project. It will probably still end up being partly/mostly your job). i also want to tune the secondary, so test it with different springs and get some sensors in there to visualize how the system is responding as a whole. possibly do some groundwork for secondary actuation but that would not go on the car this year. you also would be working with powertrain daq people to see how the cvt affects overall efficiency and maybe tuning things around that. i can't really predict what kind of data is gonna come out of ptrain daq so idk what that looks like but you would be in charge of integrating with those subsystems.

Code

I was going to include All Of The Code in case that was useful to have the code preserved in its entirety at this point in time. Then I remembered that's literally the point of the github and also four files in, I had already added several pages to the design doc. Instead, I recommend you just look at the github. I will be going through and adding more comments to the code so if a part doesn't make sense just check back for more context. Or ask me or Cole.

Thoughts and takeaways and advice that are less design related

This is going to be a bit of a ramble a la “everything Laura knows” so feel free to skip this part. The point of this section is to give advice and explain my process outside of purely design (things like people headaches, organizational methods that worked for me, ways to keep myself motivated etc) because the way you approach an electronics project is very different than most other projects on baja. The structure and timeline and just general experience is simply not the same. There’s no CAD, FEA, or manufacturing, and the skills you gain doing anything else on the team do not translate in the same way they do across other subteams. However, there is a way to do this type of project so that you still fit within the culture of Baja in a way that I think I did not, at least at the beginning. This is most of my advice on making sure you are still connected and integrated with the team even though most of them will have literally no idea what you are doing.

Make a timeline early and stick to it. There will be things that you have to do that you probably don't even realize exist as steps of this process. Get “done” as early as possible then validate. I have been talking a lot with the formula people because there is not much electronics project experience on Baja, and also from my own experience the biggest way to make sure things work is to test many times in many ways and test repeatedly. A lot of times issues go away if you let it sit overnight or add a delay or change your spacing or do something else small and inconsequential. However in order to design for robustness (which is the whole point of baja) you need to validate that your project will work every time. Having a timeline that you stick to helps you get things done on time and also helps you outline everything that needs to be done. My point in saying this is also to make sure that you build in a lot of time for testing. Most projects get designed, manufactured, and then tested, but you should be testing and validating each component and each section of code that you write as you go along. Additionally, make sure you stay up to date with major subteam/whole care deadlines. Ask team leadership, go hunting in the drive, check the notion, whatever you need to do to make sure your personal time expectations mesh with the rest of the team’s.

Communication is a two-way street. If team leadership has no idea what your project does, they will kinda leave you to it. However “leaving you to it” for me at least involved learning about major deadlines a week beforehand, not being involved in broader design decisions that affected my project, and generally being isolated from the rest of the team. Even though it is not technically your job to do this, things got a lot better when I started setting up check ins with team leads, sending them updates with the project, and reading the glovebox notes weekly. It is a lot to juggle, and technically your team lead’s responsibility, but sometimes they just don't know what support you need and thus don't offer any. You can be proactive without being a bother and you deserve the same support as all the mechanical projects.

Going along with that, **don't be afraid to make your needs clear.** I met with Jared and Konrad in January (ish?) to talk about baja more broadly, but obviously the ecvt and what I felt was going poorly with it came up. Konrad immediately came up with a solution for my feeling isolated and cleaned off/moved an electronics area into the autobay on the spot. Jared started checking in with the project more and made sure that he was the one to tell me when major design decisions were made (such as not running ecvt at comp). All that is to say the team has your

back. They are there for you and if things are not going well, or even just not the absolute best they could be, people will fix things if and only if they know there is something to fix. That's what Baja is all about: caring. And people do care, I promise.

The most important skill i learned is to just **figure it the fuck out**. There was basically zero electronics support this year, which I am hoping to fix for following years, and there was no infrastructure to build off of. The ecvt people built everything completely from the ground up and that involved a lot of late nights, many hours on google, a little bit of github copilot, and a lot of throwing shit at the wall and seeing what stuck. However, as my rugby coach always says, don't shovel shit, and what he means by that is if you get a pass that was not thrown well, don't bobble it off for the next person to deal with. Take a breath, readjust your grip, and get tackled if necessary. I am afraid that I am kind of shoveling shit down to you, but keeping with the metaphor, all the best plants come from manure. It functions! And it functions rather well! But a lot of the things we did were not the most elegant, and a lot of the decisions we made were because it works like this and not the other way and we're not really sure why. So in that sense, that we're passing down a lot of function over fashion, I am shoveling shit. But this came from a very small team led by essentially a freshman creating a functional ecvt with no support in a year and a half, when it took a team like UCLA 8 years to successfully implement. Just to give you some perspective. And that was possible because we learned to just FITFO. While that scrappy mentality is something that i believe to be very central to Northwestern baja and what allows us to have a driving car every year (unlike certain other teams), i am hoping you will be able to have a bit more certainty in your design decisions, a deeper understanding of why and when things don't work, and more elegance in what you produce.

Stay organized. Spreadsheets are your friend. Or maybe not. Just do something to keep everything written down. Use the notion, use your gcal, cover your desk and phone and laptop in color coded post its with daily and weekly todo lists like a mad scientist (personally a big fan of this strat). Whatever it is, you need some sort of System. It doesn't need to make sense to anyone else as long as you understand it and communicate it effectively using the team methods of staying organized. Use your System consistently and constantly. You should be checking in with your tasks ideally at least once a day, at minimum once before each meeting.

It will be very difficult to **stay motivated**. At least for me, I pushed myself way too hard to start (it did not help that i did not have a clear timeline for myself) and got burnt out by the end of the summer. Having a bad timeline coupled with working two jobs meant I was stressed out of my mind kind of constantly. I spent my lunch breaks at work doing baja and every moment that i wasn't eating, sleeping, or working out when i wasn't at work was spent doing baja. I would not recommend this if you can at all avoid it. This was mostly self-inflicted (the whole working two jobs and also doing baja thing) but still, don't inflict it on yourself. It takes a certain amount of ambition to take on an electronics project on a mechanical team so it's important to recognize when you are pushing yourself to a point of destructive stress. A little bit of stress is good as motivation, I think, but there is a point where it is unhealthy. On the motivation front, it is very easy to lose sight of how cool what you are doing is. We have never really had a successful electronics project (until now) and now that is expanding into a whole team! That's so sick!! And

you, dear reader, are now a part of that journey!! Lean on your teammates when you get burnt out. I would have personally shed a lot more tears and gotten a lot less done without Cole, Rafa, and Nick.