An eGas Gauge for My Lectric XP eBike, Version 6.1

By R. G. Sparber

Introduction

I am the proud owner of a Lectric XP eBike. It has a range of about 20 to 40 miles, depending on many factors[2]. The ever-present question is – will the battery last until I get home? My eGas Gauge can't answer that, but it can provide a continuous readout of the battery's remaining energy. This information can help you manage your ride.

Conclusion

The eGas Gauge continuously displays[3] the remaining energy in the battery under all conditions.

If riding conditions do not change, the distance covered during the time the battery goes from Full to ¾ can be used to estimate the range: multiply this distance by 4.

The estimated cost of the electronics is under $10.

The built-in energy meter has a large error when the battery is more than 3/4 full or when there is a strain on the motor.

---

[1] This work is licensed under the Creative Commons Attribution 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by/4.0/ or send a letter to Creative Commons, PO Box 1866, Mountain View, CA 94042, USA. You are free to turn my proof-of-concept into a product. I ask nothing in return. You are not free to modify this article without my permission.

[2] Energy consumption is influenced by my speed, the wind, terrain, how much I pedal, my weight, tire pressure, road surface, plus the weight of any cargo.

[3] Here is a YouTube video of the display using a simulator that tricks the circuit into thinking the battery goes from full to empty in about 2 minutes.

Technical Summary

This system is a “Coulomb Counter.” It takes the integral of the current and subtracts it from the full battery’s capacity. The result is continuously conveyed to the user via two LEDs via their relative brightness.

There is a one-time[4] Calibration that enables the rider to define what they call Full and Empty. Each time the battery is fully charged, the user resets the count to Full.

Contents

[4] Over time, the battery will degrade. If severe enough, another calibration will be needed.

The Problem

Many factors influence how far I can ride on a charge.

The display, mounted on the handlebars, includes an “ENERGY BAR.” This is a bit of fiction because it displays battery voltage.

The battery voltage drops about 10% as I go from Full to Empty. Along the way, it rises and falls depending on how much current is flowing out of the battery. It is common to lose a few bars during a climb and reclaim most of them on the descent.

The motor controller automatically turns off when the battery voltage is too low. If my battery is almost fully discharged and I increase the current, the voltage can drop low enough to shut off the cruise control or the Controller.

My speed is a function of battery voltage and current. As the battery discharges, I go slower for the same current. So, to maintain the same speed, I must increase the current. Given a constant speed, I’ll be drawing more current at the end of my ride than at the start.

The battery’s capacity is rated in ampere-hours, not volts.

So what does this all mean? I know that my battery is fully charged when I unplug my charger, showing a green light. The battery is rated at 10.4 ampere-hours, so that is the most that I can take out.

Energy consumption is influenced by my speed, the wind, terrain, how much I pedal, my weight, tire pressure, road surface, plus the weight of any cargo. This is why it is so hard to predict range. However, we can monitor how many ampere-hours are still in the battery.

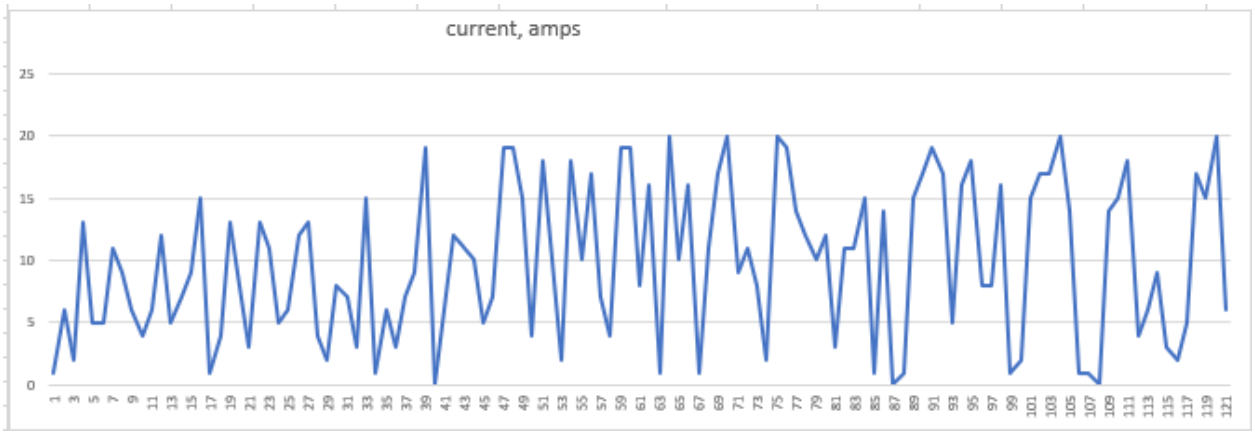
This gets us to the last piece of the puzzle: when is the battery Empty? I can define this point as when the cruise control shuts down or when the Controller shuts down. Or, I can define Empty as when there are only two ampere-hours left. It is a matter of personal preference. The user must be able to define what they mean by Empty.

In a field test, I was drawing about 12 amperes at low battery when my cruise control shut down. I define this as “Empty.” An inspection of the data collected by the eGas Gauge indicated that I had about 20% left in the battery. This is reasonable.

You will see how my design addresses all of these factors.

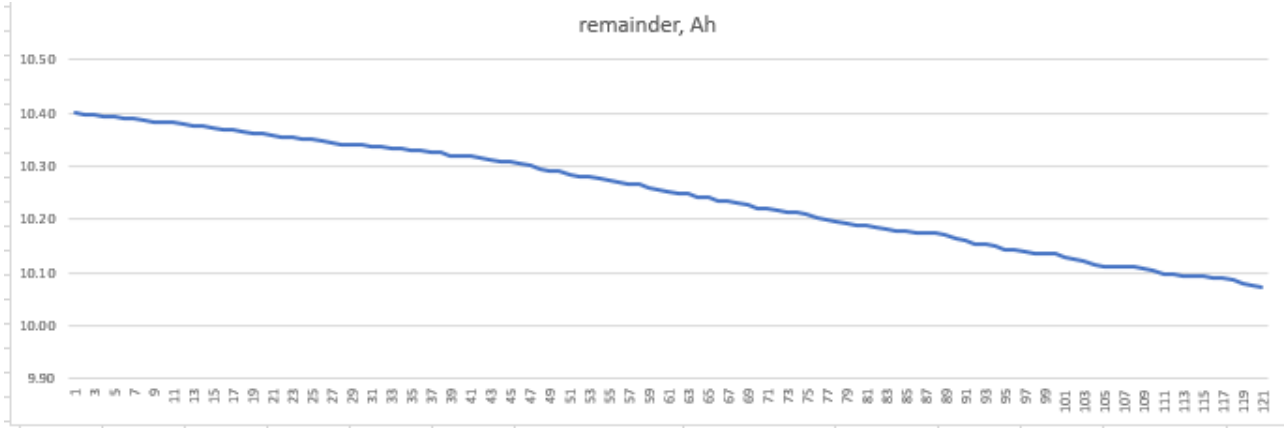
Ampere-hours

If I graph the current flowing out of my battery over time, it might look like this:



The current quickly changes from 0 to 20 amperes. Think of having a bucket of water and a cup. Every second you use the cup to remove some water. How much water remains depends on how much you take each second.

Looking at the remaining ampere-hours (Ah) in the battery, I would see



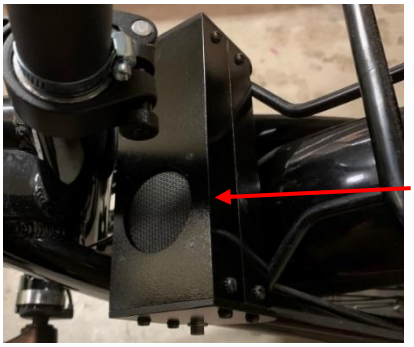
Starting with a full battery, we have 10.4 Ah. As I ride, current is flowing from the battery to the motor. How many ampere-hours remain in the battery depends on how much current is drawn each second[5]. In this example, I used about 10.4 – 10.1 = 0.3 Ah in 120 seconds. This is 0.15 Ah per minute. If I continued this pace, my battery would be depleted in $\frac{10.4\ Ah}{0.15\ Ah\ per\ minute} = 69\ minute.$

Since my current changes every second, it is not practical to perform this accounting manually. However, this is easy work for even the tiniest computer.

What does it look like?

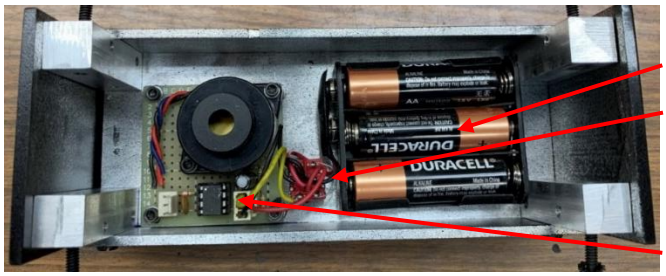
The eGas Gauge isn't much to look at. All of the complexity is in the software.

I knew the design would evolve, so I chose to build a large box. It is secured to the bike with the same bolts that hold the front of the back rack in place.



This hole with the screen on top was from a previous iteration of the design that involved sound.

[5] Theoretically, it is current drawn over time as a continuous function. However, from a practical standpoint, reading the current every second is good enough. I then assume that the current is at this value over the following second.



The box contains
three AA batteries in a holder,
a toggle switch, and

a circuit board.

That round thing on the circuit board is left over from the audio output iteration and is no longer there.

The circuit board contains a tiny computer system that costs only $1.50, four resistors, two capacitors, and two connectors.

The LED display mounts on the handlebars.

Features

- Although designed for the Lecric XP eBike, it should work on any eBike.
- The eGas Gauge measures the current flowing out of the bike's battery over time (ampere-hours) to estimate the remaining available ampere-hours. This is far more accurate than measuring voltage.
- It keeps the rider informed of the battery's state via a handlebar-mounted pair of red[6] LEDs.
- The user determines what they define as an Empty battery.
- The user's preference plus the remaining ampere-hours in the battery are stored in a memory that is not erased when power is removed.
- The eGas Gauge's internal batteries are tested each time the eGas Gauge is turned on.
- If the user forgets to turn off the eGas Gauge, they will see an LED flash every 2 minutes.
- After 10 minutes of the bike being turned off, the eGas Gauge will essentially[7] power down. Toggling the power switch off and then back to run will restore operation.
- Low electronics parts count: 4 resistors, two capacitors, one integrated circuit[8] (that costs $1.50), plus two LEDs. The current shunt is free.
- The internal batteries should last more than 200 hours.
- This design is freely offered to the public at no charge and includes details of the physical design, electrical design, and the software. If someone wants to transform this information into a product, they have my blessing. I do not expect, nor would I accept and money from them.

[6] I found that only red LEDs are visible thorugh my sunglasses.
[7] The current is less than 0.5 microamperes so will have minimal drain on the batteries that power the eGas Gauge.
[8] This is a system-on-a-chip with an impressive amount of computing power.

What is the eGas Gauge Doing?

You can skip to the User's Guide, page 15, if you don't care how it works.

Battery Theory

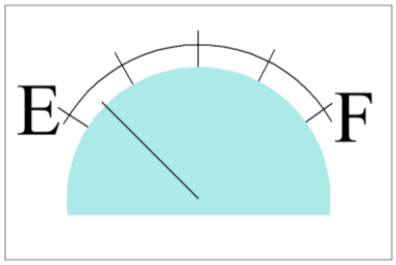
My battery is rated at 10.4 ampere-hours (Ah). Starting with a fully charged battery, if I draw a *constant* 10.4 amperes from the battery for $\frac{10.4\ AH}{10.4\ ampere} = 1$ hour, it will be empty. Draw a *constant* 5.2 amperes, and I can go for $\frac{10.4\ AH}{5.2\ ampere} =$ 2 hours. For this simple case,

$$run\ time = \frac{10.4\ ampere{-}hours}{constant\ current\ drawn\ in\ amperes}. \qquad (1)$$

Of course, I rarely draw the same current for more than a minute.
With the current continuously changing, it is not useful to predict the remaining run time. However, it is possible to calculate the remaining Ah:

$$remaining\ Ah = 10.4\ Ah - \ consumed\ Ah \qquad (2)$$

The Rider's View



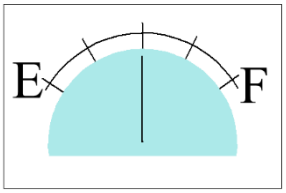
When riding, I don't care about ampere-hours. It is more useful to know how much is left in the battery, just like in my gas-powered car.

$$battery\ status = \frac{10.4\ Ah - consumed\ Ah}{10.4\ Ah} \qquad (3)$$

The battery status goes from 1 down to 0 as I ride.

Consider the simple case of drawing 5.2 amperes for 1 hour. My consumed Ah would be 5.2 $amperes \times 1\ hour = 5.2\ Ah$. Equation (3) predicts

$$battery\ status = \frac{10.4\ Ah - 5.2\ Ah}{10.4\ AH}.$$



$$battery\ status = \frac{1}{2}$$

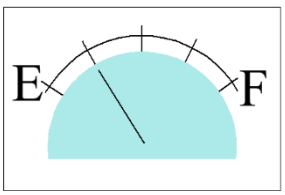
Say I drew 5.2 amperes for 1 hour and then 1.3 amperes for 2 hours. My consumed Ah would be

$$(5.2\ amperes \times 1\ hour) + (1.3\ amperes \times 2\ hours) = 7.8\ Ah$$

I can plug this consumed Ah into equation (3):

$$battery\ status = \frac{10.4\ Ah - consumed\ Ah}{10.4\ Ah} \qquad (3)$$

$$battery\ status = \frac{10.4\ Ah - 7.8\ Ah}{10.4\ Ah}.$$

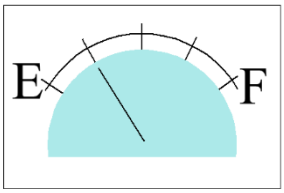


$$battery\ status = \frac{1}{4}$$

This hand calculation becomes unmanageable when dealing with the real world. I would need to record the current every second and then add it to the consumed Ah.

Then I would need to calculate the battery status. This bookkeeping is not compatible with staying upright on my eBike!

Key Point: When I fully charge the battery, I also reset the eGas Gauge's consumed ampere-hours to zero. Once every second, the eGas Gauge measures the current and adds it to the tally.



I drive two red LEDs that convey the remaining energy in the battery. Details are presented on page 15.

These LEDs also displays a warning when the AA batteries are low and when there is a memory failure.

I run Calibration that teaches the eGas Gauge the battery's capacity. This is repeated when there has been a noticeable degradation of the battery. See page 38 for details.

Measuring the Current From the Battery

The current, I, is measured by reading the voltage, V_{shunt}, across a shunt resistor, R_{shunt}.

$$V_{shunt} = R_{shunt} \times I \qquad (4)$$

R_{shunt} is the wire that runs between the battery and the electronics box inside my eBike. I'll explain this later.

Lithium-Ion Batteries and the eGas Gauge

I am not an expert on Lithium-Ion batteries. From a quick search of the web, it is clear that the available ampere-hours for a battery depend on many factors, including ambient temperature[9], load current variation, and the number of charge-discharge cycles. If there is a large change in ambient temperature, how you ride, or the passage of time, you may need to recalibrate the eGas Gauge. Fortunately, this only involves charging up your battery and riding your eBike until you say the battery is ready to be recharged.

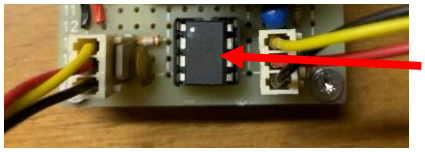
The Physical Design

I wanted to build an enclosure that was at least as rugged as my eBike. OK, maybe I overdid it.

My eGas Gauge is secured to my Lectric XP using the bolts used by the luggage rack's forward supports. The existing 10-32 Socket Head Cap Screws were replaced with longer ones.

[9] The circuit can measure ambient temperature but the error is so large that the resulting correction factor can make things worse.

The eGas Gauge is tucked away under my seat post, where it is out of the way.



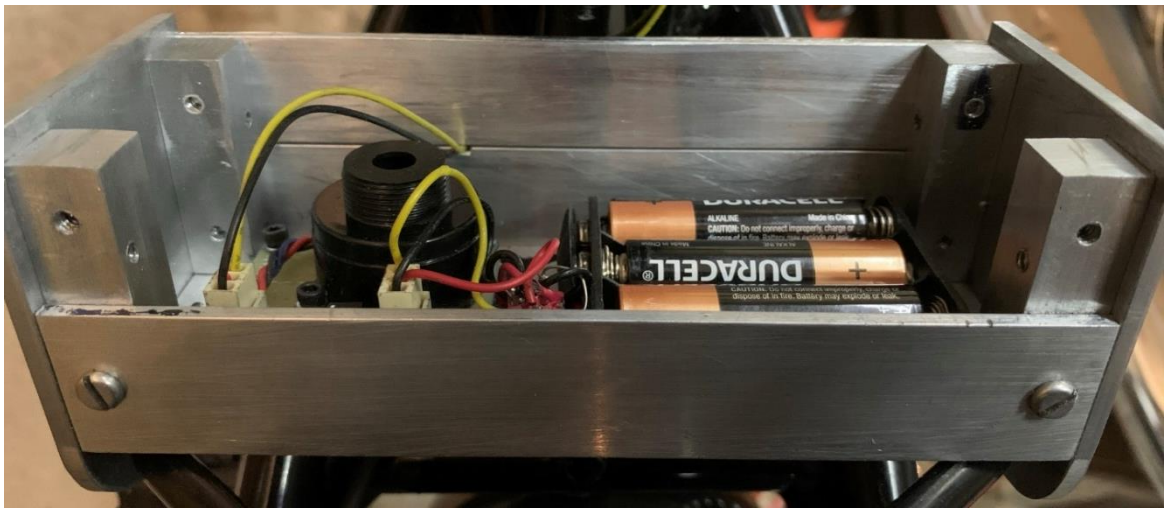
The calculations needed to track the remaining energy in the battery come from a delightfully small computer system called the ATtiny85. It costs about $1.50 in single lots.

The LEDs clamp to the handlebars. They are extremely rugged, compact, and low cost. This enclosure is made of aluminum.

Besides the hardware cost, you will need to load the ATtiny85 with my software. I use a USBtinyISP burner, which is driven by the Arduino IDE[10].

[10] Search the Web with keywords “arduino attiny85 download Sparkfun” for what I used.

I chose to use 1/8 inch thick aluminum extrusions to build my enclosure. The corners are secured with screws threaded into ½ inch square bars. This is one solid box.



The cover, not shown here, is removed when I need to replace the three AA batteries. I estimate that this must be done once a year.

It is well hidden, but a thin wire runs from inside the eBike and into the enclosure. It carries a voltage that is proportional to the current out of the eBike's battery.

Equally well hidden is a toggle switch that points down. Reach under the enclosure from the back side and lift up to turn on the eGas Gauge. Push down to start the Initialization process and the Calibration process.

Two LEDs flash out status information or provide a proportional indication with their relative brightness.

Error Indications

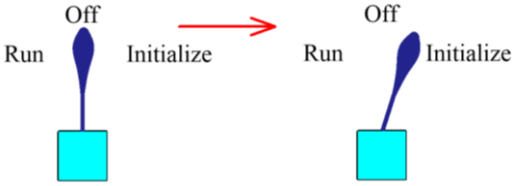
- the lower LED flashing short bursts[11] for twenty seconds - it means that the AA batteries powering the eGas Gauge need to be replaced soon[12]. The circuit will continue to work for a while, but the LEDs will be dim.

- The upper LED is on for 0.1 seconds, then the lower LED is on for 0.1 seconds. This pattern continuously repeats. – this is a hardware failure within the computer device.

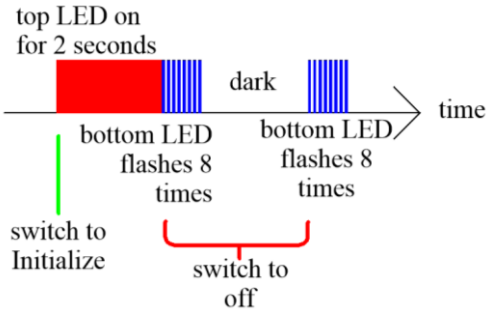
[11] 50 ms on and 50 ms off
[12] They should last over 200 hours of riding.

First Time Use

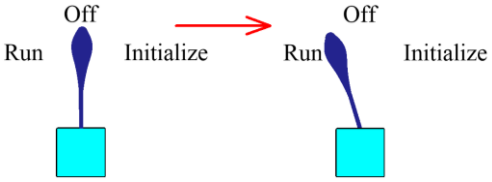
Before the eGas Gauge can tell you how much energy is left in your battery[13], it has to go to school. It will learn what you consider an empty battery plus calibrate its current sensing circuit. I call this Calibration.



Start with a fully charged battery and fresh AA batteries. Then move the toggle switch from **Off** to **Initialize**.



You will see the top LED light for 2 seconds, followed by the bottom LED flashing eight times[14] and then two seconds of darkness. This will be followed by another set of eight short flashes on the bottom. During the first set of short flashes or the dark interval, move the switch back to **Off.**



When you are ready to go on your Calibration ride, move the switch from **Off** to **Run**. Ignore the LED flashes.

Ride your bike until the battery, *in your opinion*, is *Empty*. It is common to consider 2 or 3 bars on the ENERGY BAR a comfortable definition of Empty. This is read when the bike has not moved for one minute. Then move the switch from **Run** to **Off**.

The draining of the battery to Empty does not have to be in one ride. Just remember to move the switch from **Off** to **Run** when you turn the bike on and return the toggle switch to **Off** when you turn the bike off.

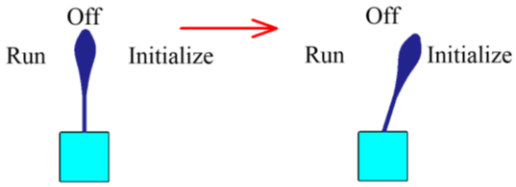
When the battery is Empty, move the switch from **Off** to **Initialize**. You will see the flash pattern described on page 16. Then the bottom LED will light for 2

[13] As the battery ages, its capacity will degrade. At some point, it may be necessary to recalibrate.
[14] Each short flash lasts 50 milliseconds and it is followed by 50 milliseconds of darkness.

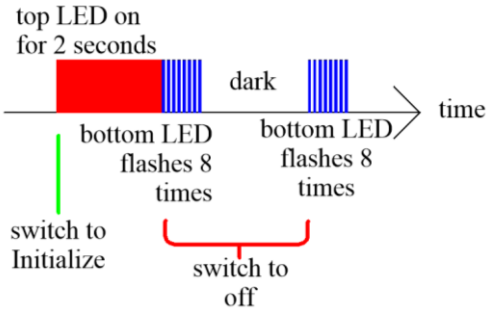
seconds which is followed by a continuous fast series of flashes[15] from the lower LED. This means Calibration has been completed. Then move the switch back to **Off**. The eGas Gauge is now ready for service.

If you later change your mind and decide that "Empty" is at a different level, just repeat the process, and this level will become the new bottom.

After Every Charging of Your eBike Battery



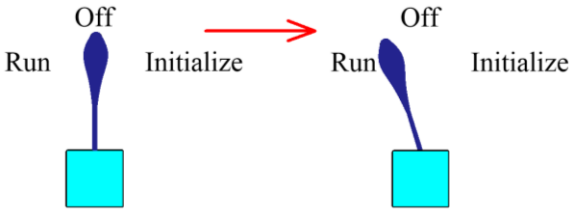
Every time you fully charge your eBike battery, move the toggle switch to **Initialize**.



You will see the two-second flash from the top LED followed by eight short flashes from the bottom LED, two seconds of darkness, and then another eight short flashes from the bottom LED. During the first set of short flashes or the dark interval, move the switch back to **Off**[16]**.**

The eGas Gauge now knows your battery is fully charged.

Ready to Ride

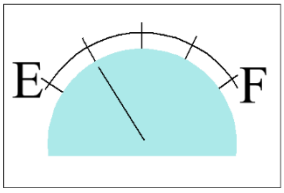


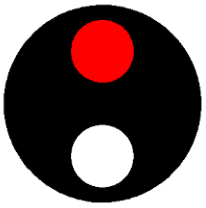
When you are ready to ride, move the toggle switch to **Run**.

You will immediately see the battery status.

[15] 50 ms on and 50 ms off

[16] If you see the lower LED come on for 2 seconds, you have entered the Calibration state and must perform the ***First Time Use procedure*** again. It isn't the end of the world but is best to avoid this hassle.





As energy level falls from full to ½, the LEDs smoothly transition from the top being on full brightness to the bottom being on full brightness. Then the cycle repeats as we go from ½ to empty. Here is a video of one cycle.

- When the battery is fully charged, the top LED is on full brightness, and the bottom LED is dark.
- At ¾ charge, both LEDs are on at the same brightness.
- At ½ charge, the top LED is dark, and the bottom LED is on full brightness.
- Then the cycle repeats.
- Just below ½ charge, the top LED goes on full brightness again.
- At ¼ charge, both LEDs are on at the same brightness.
- When I reach empty, the bottom LED is on full brightness, and the top LED is dark.

At the End of Your Ride

To save the AA batteries powering the eGas Gauge, switch to **Off** at the end of your ride.

When you ride again, move the toggle switch to **Run**. The circuit will remember how much eBike battery energy you have left.

If you forget to turn the eGas Gauge off, its batteries will run down, but the eBike's battery level is not lost.

Reminder Card

Until the procedure becomes second nature, you may find this card useful. I am assuming you have the LED option.

eGas Gauge

After Fully Charging battery:

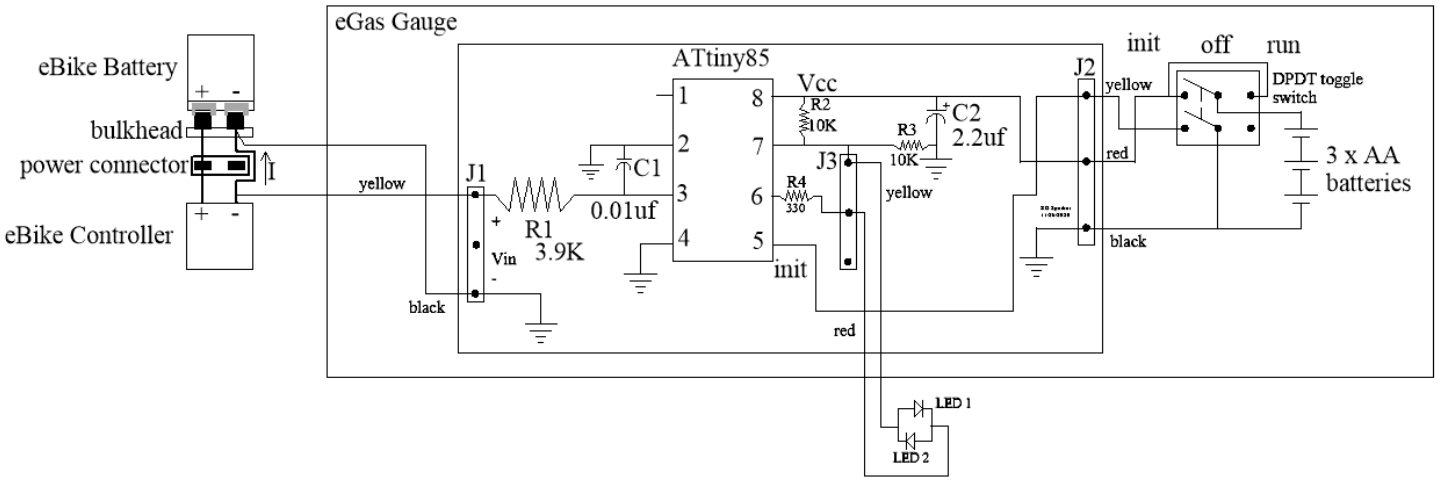
1. Off → Initalize
2. wait until 8 short flashes
3. Initalize → Off

Start of Ride: Off → Run
End of Ride: Run → Off

One Time Calibration:

1. battery fully charged
2. Off → Initalize
3. wait until first 8 short flashes
4. Initalize → Off → Run
5. Ride until battery empty
6. Run → Off → Initalize
7. wait until continuous short flashes
8. Initalize → Off

The Schematic and Bill Of Materials



Bill Of Materials

R1	3.9K, 0.1W, 10%
R2, R3	10K, 0.1W, 10%
R4	330, 0.1W, 10%*
LED 1 & 2	Red LEDs
C1	0.01uf
C2	2.2uf, 16V electrolytic
-	Center off DPDT toggle
J1 – J3	Keyed 3 pin connectors
-	3 AA Batteries
-	3 AA battery holder
-	ATtiny85
-	8 pin DIP socket

C2 will never see more than 5 volts. I chose one with a working voltage of 16 volts because I didn't have an electrolytic with a smaller working voltage.

If you can find a toggle switch that is spring-loaded on the init side, it would be better.

J1 – J3, and the DIP socket are optional. I use them because it makes it far easier to remove the circuit when I need to make changes.

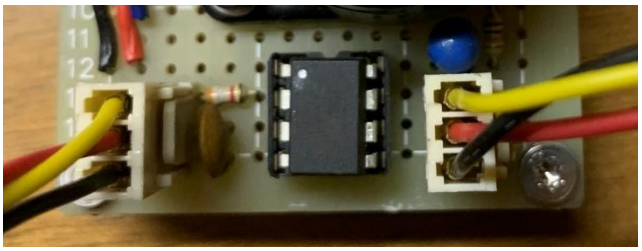
The current shunt resistor is not listed in the Bill Of Materials because there is nothing to buy.

Construction Hints

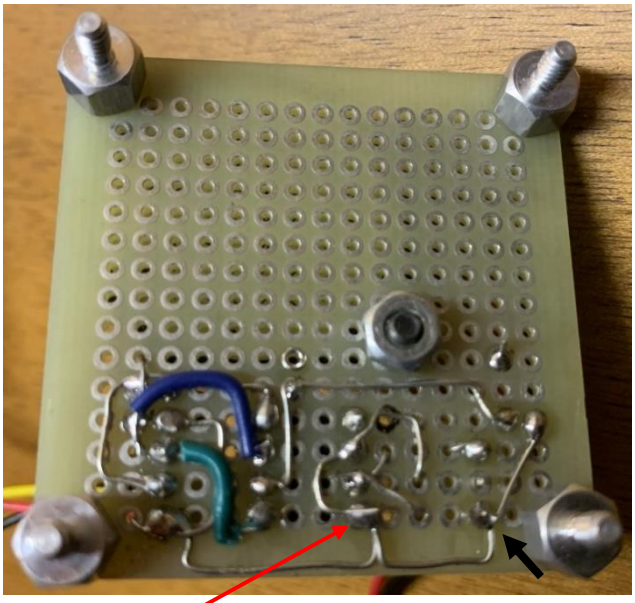
The Circuit Board

I used a 1¾ inch x 1¾ inch perf board but filled only about 1 x 1¾ of it.

The physical layout of the board closely matches the symbol placement of the schematic. The plugs were premade, so I assigned pins to match the wire color.



I use a dab of oil-based white paint to mark pin one on the ATtiny85. There is a recess there, and the paint is protected.



The board is supported on the corners by ¼ inch tall standoffs. Since there will be a lot of vibration, I used split washers with these standoffs.

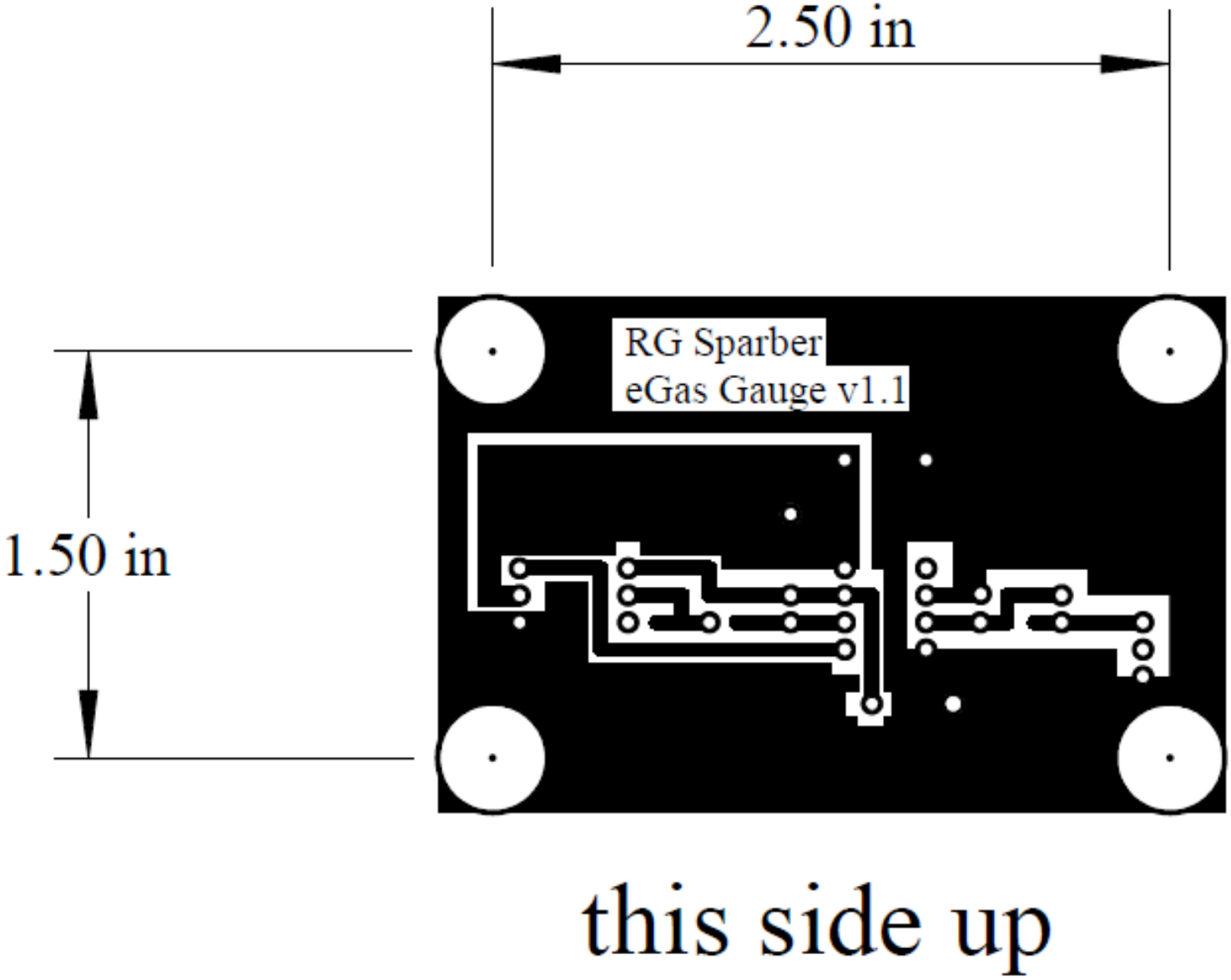
Each predrilled hole has a copper land around it. This made soldering in components easier than if I had used a blank board.

Bare wire is used for all connections. I did sleeve two wires using insulation from another wire. This is much easier than trying to strip the ends of tiny wires.

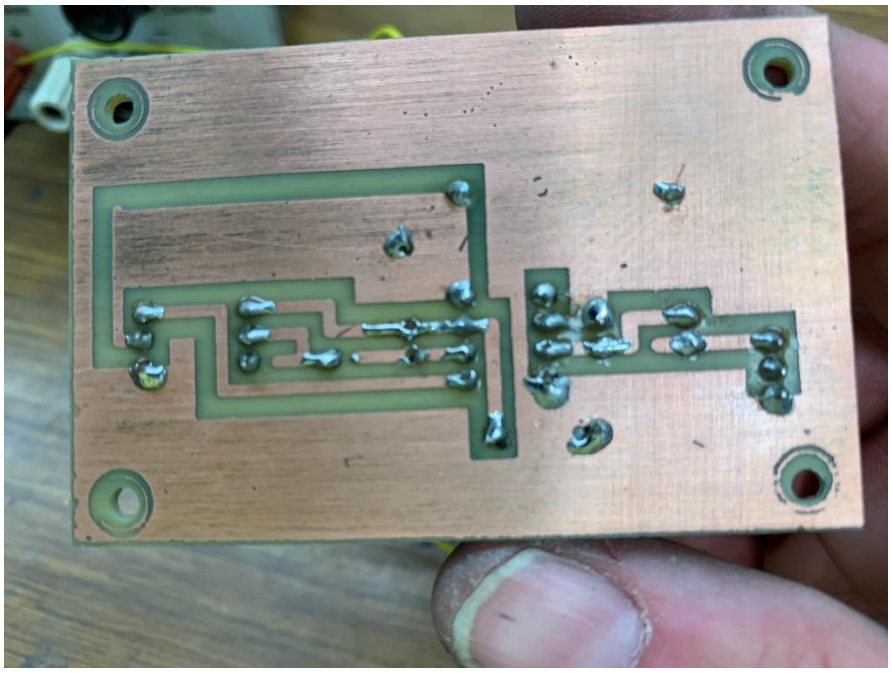
Pin 4 of the ATtiny85 is my single point ground.

This picture is a bit old. The nut you see above the wiring held a component that is no longer used.

If you prefer to etch your own board, here is single-sided artwork designed to tolerate a lot of over-etching. Of particular importance is that the standoffs on the corners must not contact the ground plane.



I remove as little copper as possible. This gives me the most ground plane to keep the circuit quiet plus uses as little etchant as possible.

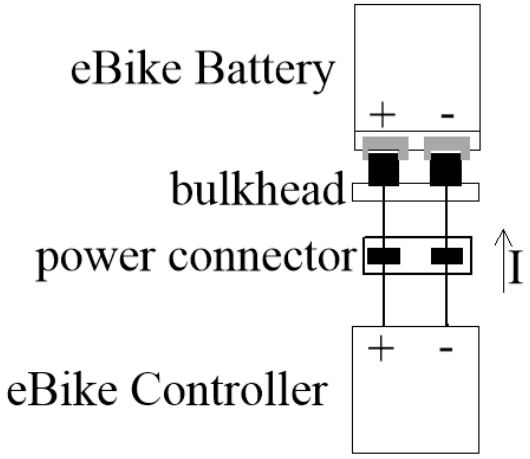


This is my first iteration before I realized that my stand-offs would short out to the ground plane. I also made the copper-free areas wider than necessary.

I made no attempt to minimize area.

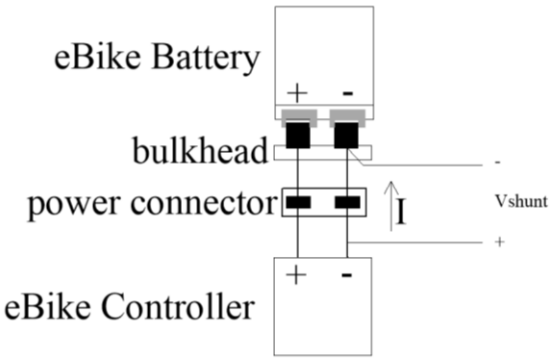
The ATtiny85 is running software version 2.3.

The Current Sensing Circuit: Electrical



Current flows out of the + terminal of the battery, through the bulkhead connector, and through the power connector. It then passes through the eBike Controller, back through the power connector and bulkhead before entering the – terminal of the battery.

I need to measure this current. A standard technique is to use a current shunt. It is a low resistance, high wattage resistor placed in series with the battery. The shunt transforms this current into a tiny voltage, which is fed into the circuit. This shunt can be any value between 3 and 6.4 milliohms[17].

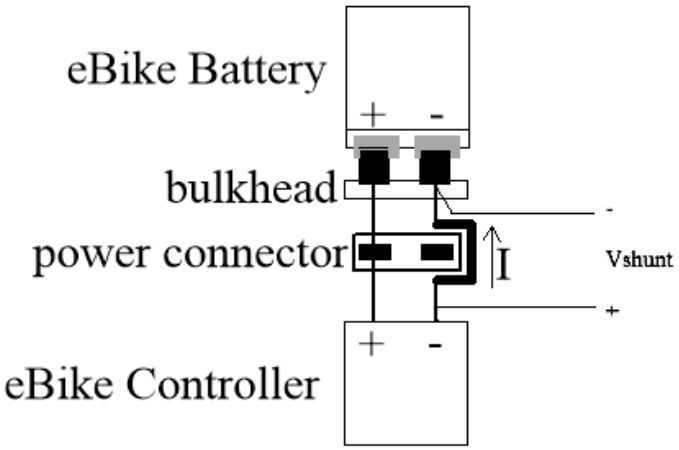


Being a frugal engineer, I didn't want to spend any money on a shunt. Besides, the shunt generates unwanted heat while wasting valuable battery power. Even at 3 milliohms, it dissipates as much as 1.2 watts.

My first idea was to measure the voltage drop between the Controller and the bulkhead. To my amazement, I generated a usable voltage as the current went from 0 to 20 amperes. But on closer inspection, I found that the relationship between voltage and current was not linear. At currents below about 10 amperes, the shunt's resistance was 3.7 milliohms. At 20 amperes is was more like 5 milliohms, but the longer I stayed at this current, the higher this resistance became. A rise of 1.3 milliohms may not sound like much, but it is a rise of 35%.

After some head-scratching, I realized that something was heating up and increasing the shunt's resistance. The obvious culprit was the power connector.

[17] See page 20 for details.



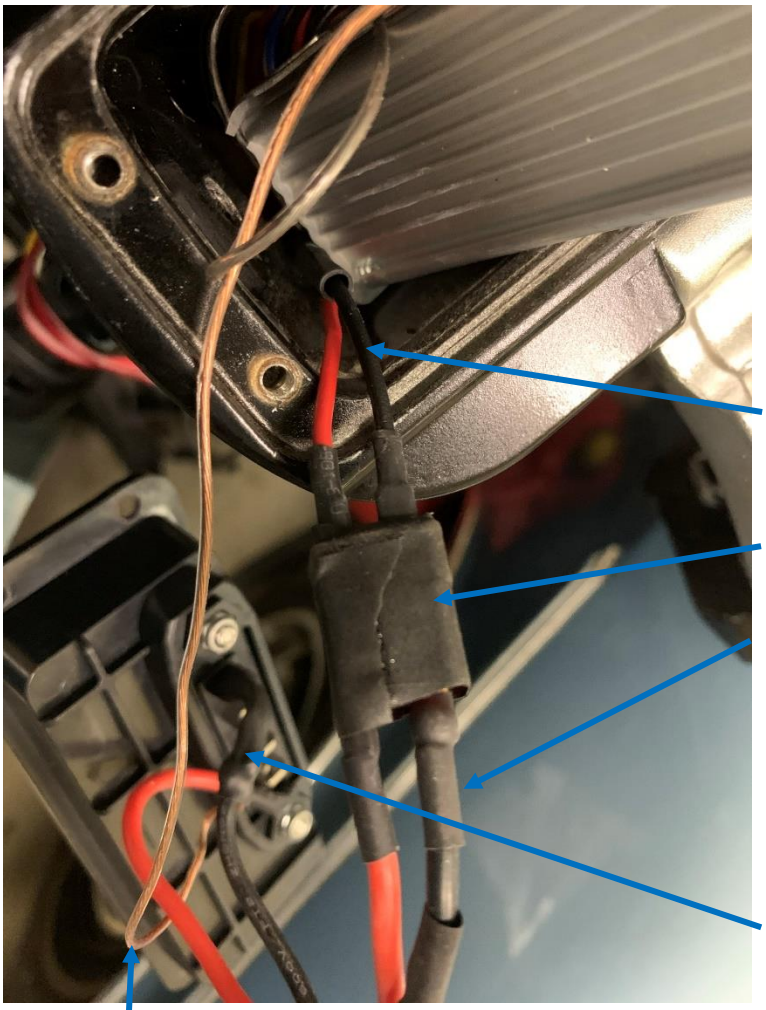
Using a few inches of heavy wire, I bypassed the power connector's negative side.

This solved the drifting shunt resistance problem. My resistance was a constant 3.0 milliohms.

The Current Sensing Circuit: Physical

Electrically, this is a simple solution. Physically, it was a little tricky and may be scary for some people. You do have to open up the eBike and mess with that wire.

Here is the bulkhead with the four mounting screws removed.



This bulkhead lifts out to reveal a maze of loose wires and the Controller.

The wire from the Controller connects to the

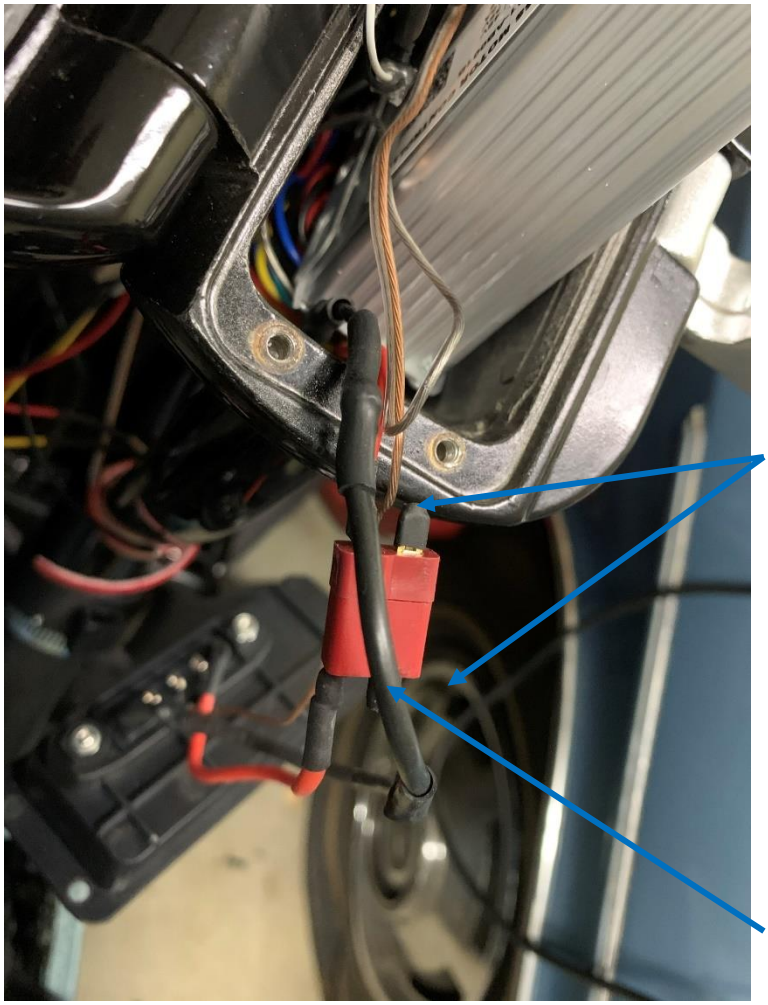
power connector

which connects to a heavier gauge wire that goes to the

bulkhead.

The power connector is wrapped in black 3M electrical tape to prevent it from falling apart due to vibration.

This thin, copper-colored wire is the sense wire going to the bulkhead.



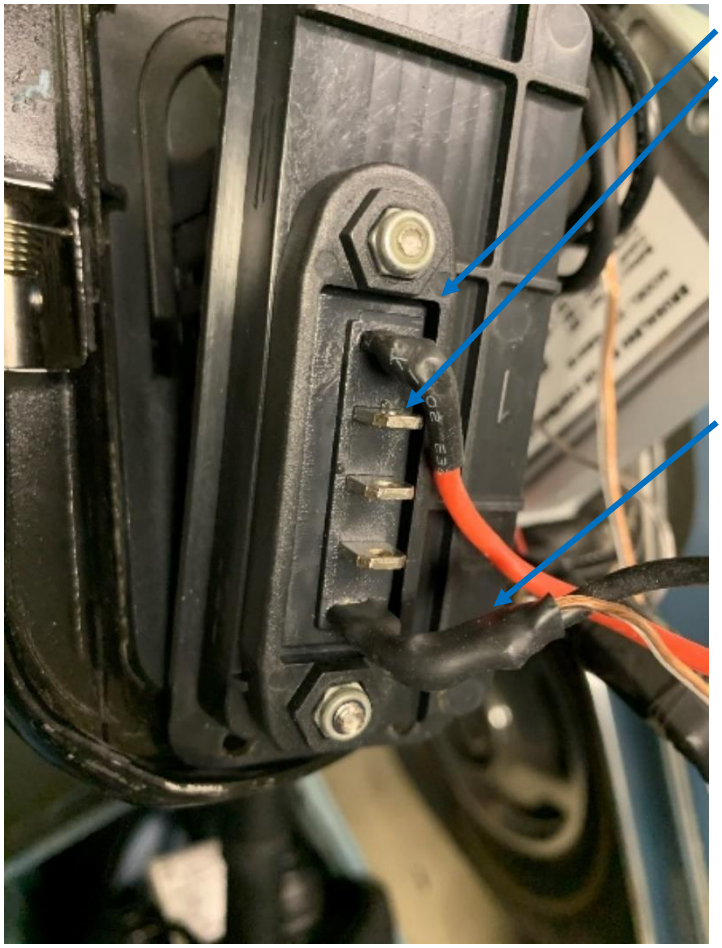
After cutting the black wire[18] on each side of the power connector, I soldered in my

Heavy gauge bypass wire.

This picture shows the tape removed from around the connector, but that turned out to be a minor mistake. Just leave the tape on.

Note that I used heat shrink over all connections. This is far more robust than electrical tape.

[18] Gasp! This is my $900 eBike.



The bulkhead consists of heavy metal fingers molded into a plastic frame. If you tried to solder directly to one of these fingers, there is a risk that the plastic would soften and the metal finger would move out of alignment. That would ruin the bulkhead.

I chose to cut the black wire about ½ inch back from the bulkhead. After sliding on the appropriate size piece of heat shrink tubing, I tinned both ends of the power wire plus my voltage sense wire.

Then I formed hooks and crimped the wires together. As I applied heat and more solder, I was able to safely make a low resistant, rugged, high current connection without significantly heating the metal finger.

Be careful where you point your heat gun as you shrink that tubing. You risk melting plastic parts that would be hard to replace.

Be generous with the length of the sense wires. You do not want to have to go back in here to attach a longer wire. Think about where you want to place the circuit's enclosure and run the wire there. Then add another foot.

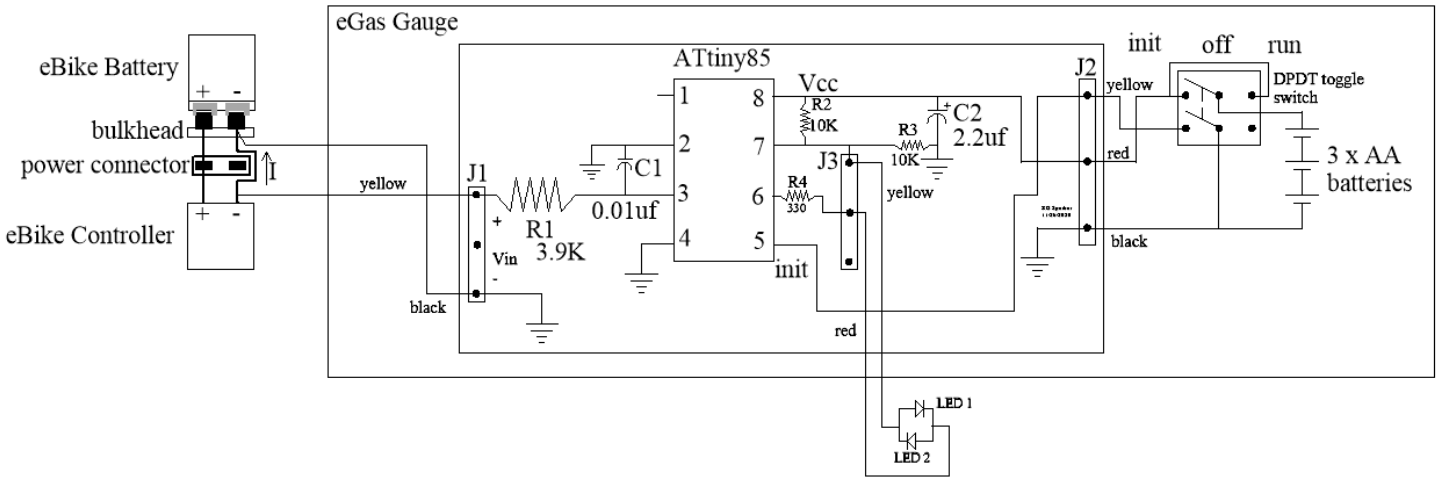
I didn't take a picture of the connection near the Controller. This wire is thinner, so it is easier to handle. I left about an inch between the body of the Controller and my connection.

With the wires out of the way, gently push the Controller down into position. Then stuff the many wires around it. The Controller must be down far enough to not press on the backside of the bulkhead.

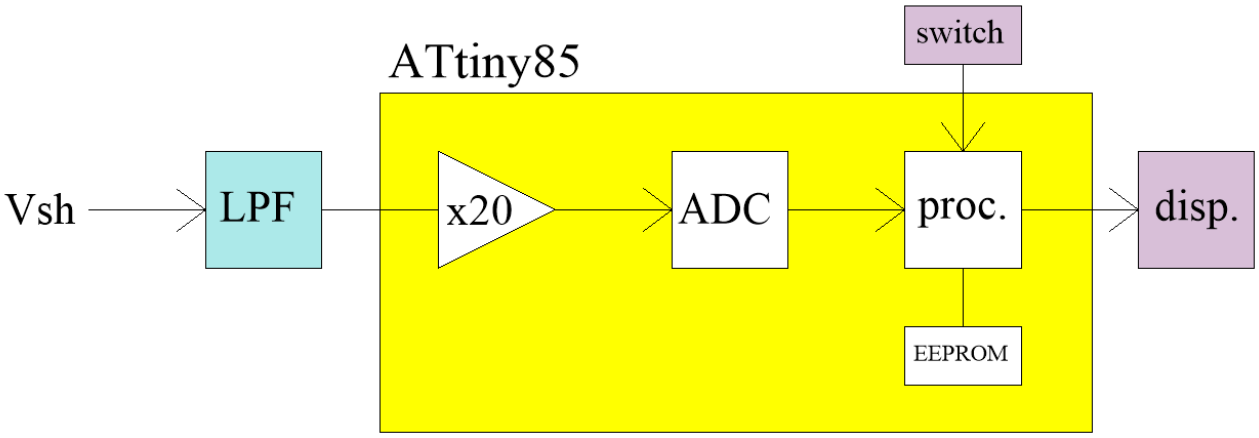
Replace the bulkhead and install the four screws. They should be snug. If you overtighten them, you run the risk of cracking the plastic.

I hope that wasn’t too scary.

Circuit Description



Overview



The eGas Gauge measures the current flowing out of the eBike's battery and drives a display. Almost all of the complexity is inside the ATtiny85 device. It is a system on a chip with both analog and digital functionality.

The current is converted to the shunt voltage, Vsh, by the shunt resistor described on page 25. It is fed into a Low Pass Filter (LPF) that removes frequencies that are too high to be processed. The resulting voltage is amplified by a factor of 20 before being fed into an Analog to Digital Converter (ADC). The processor reads the ADC when it needs to know the current flowing at that moment.

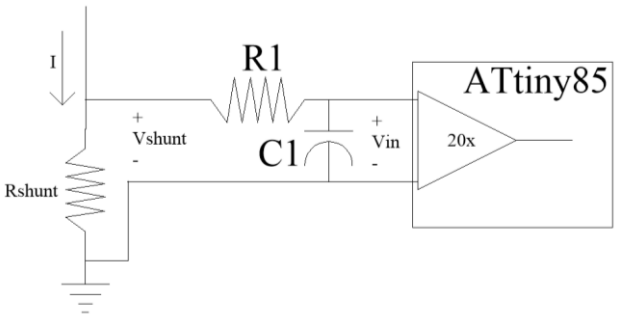
The processor reads the state of the switch to know when Initialization or Calibration has been requested. Data that must survive a loss of power is stored in the EEPROM. The processor outputs the remaining battery level to the display.

The Input

While riding, a current, **I**, flows from the eBike battery's + terminal and into the eBike Controller. This current returns on the – terminal. As this current flows from the Controller to the bulkhead, V_{in}, is developed. Given a shunt resistance of 3 milliohms, V_{in} will be a maximum of

$$3\ milliohms\ \times 20\ amperes\ = 60\ millivolts.$$

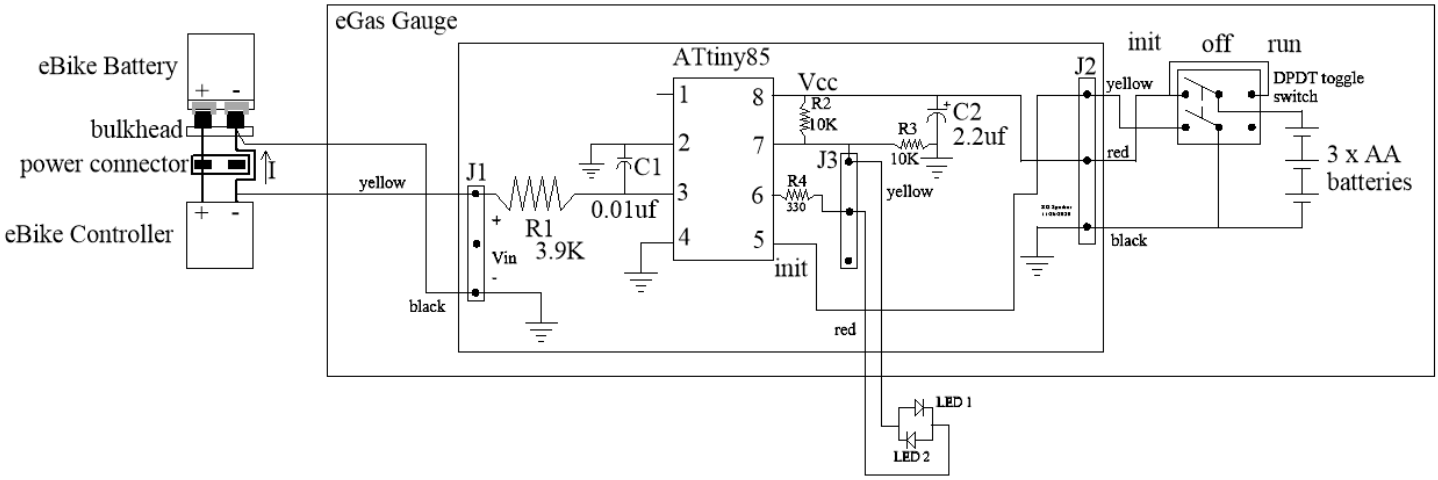
This may not seem like a lot, but the circuit cannot process an input voltage of more than 128 millivolts.



This voltage is applied to R1 and C1, which form a low pass filter with a corner frequency of about 4 kHz. It limits the noise into the ATtiny85 plus prevents aliasing in the Analog To Digital Converter. The voltage across C1 is applied to ATtiny85 pins 3 and 2. Pin 3 is the positive input of a differential amplifier. Pin 2 is the negative input. See the ATtiny85 spec sheet for details.

R2 and R3 divide the battery voltage by two, so it is within the range of the Digital Converter. It supports the AA battery check function.

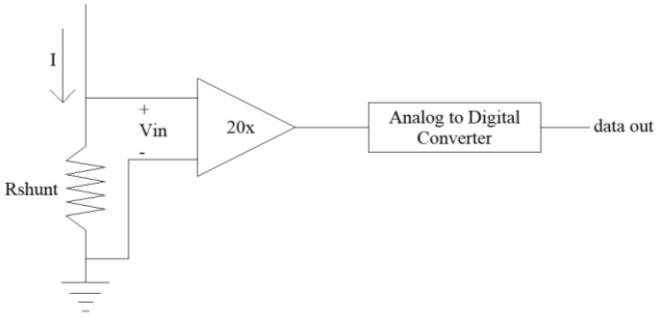
C2 attempts to keep the supply voltage quiet. Since I can't ride my eBike while monitoring with my oscilloscope, I have no idea how close I am to being in trouble. Field tests show that 2.2uf works reliably. Noise immunity is helped by the fact that the eGas Gauge is only grounded by the negative V_{in} connection. Changing to a power converter tied to the eBike's battery would increase noise.



Pin 1 is the ATtiny85's reset pin and has a weak internal pullup resistor. Do not connect to this pin.

The toggle switch is Double Pole Double Throw (DPDT) with center off. If you only have a Single Pole Single Throw switch, you can add a normally open single-pole pushbutton and make this work. Init would then mean holding down the button and moving the switch to on. For Run, just move the switch to on.

Don't let the size of the ATtiny85 fool you into thinking it is simple. The spec sheet runs to 226 pages. I will not attempt to duplicate this information here but will talk about the relevant elements at a high level. To fully understand my software, I suggest you have a copy of the spec sheet handy.



Inside the ATtiny85 is a differential voltage amplifier configured to have a gain of 20. It feeds into the ADC.

Although I don't know the exact value for R_{shunt} I do know its limits. The ADC uses a 2.56V reference, and this is the maximum voltage it can convert.

When the ADC sees 2.56V, it means that V_{in}, which is also V_{shunt}, must be $\frac{2.56V}{20} = 128\ millivolts.$

$$V_{shunt} = R_{shunt} \times I \qquad (4)$$

or

$$R_{shunt} = \frac{V_{in}}{I} \qquad (5)$$

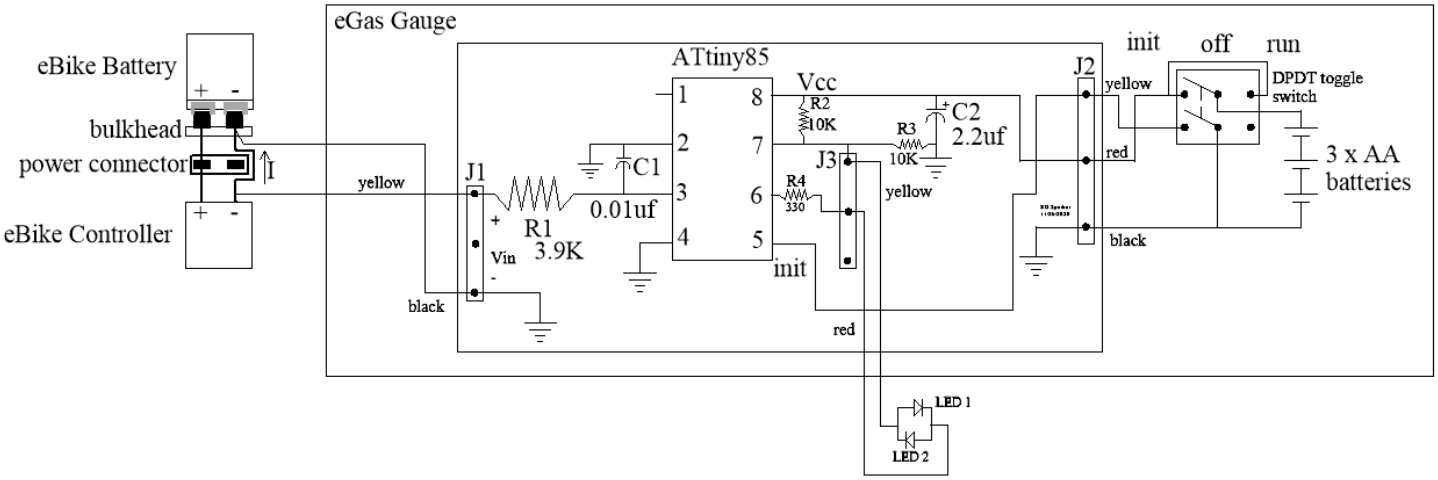
The maximum current is 20 amperes. Since I know the maximum V_{shunt} and the maximum current, I can calculate the maximum shunt resistance is

$$maximum\ R_{shunt} = \frac{128\ millivolts}{20\ amperes}$$

$$maximum\ R_{shunt} = 6.4\ milliohms$$

This is a damn small number! However, on my eBike, it is 3 milliohms. Your eBike should be in the same neighborhood, assuming you follow the construction hints. Since I did not test a shunt resistance less than 3 milliohms, I'll call this the minimum R_{shunt}.

The circuit draws about 6 mA, except for the brief moments when the LEDs are on. I'm using 1500 mAH AA batteries. This means I should be able to run for about $\frac{1500\ mAH}{6\ mA} = 250\ hours$. My typical ride is 1.5 hours, so I can expect to replace the batteries every $\frac{250\ hours}{1.5\ hours\ per\ ride} =$ 167 rides. If I rode three times a week, the batteries should last $\frac{167\ rides}{3\ rides\ per\ week} = 56\ weeks$. That's more than a year.



After I move the toggle switch from Off to Run, the circuit powers up, and the software detects that pin 5 is high[19]. This causes the code to read the AA battery's voltage. To do this, ATtiny85's pins 6 and 7 are configured as inputs, which effectively disconnects the LEDs from pin 7. The software then reads the voltage on pin 7. This voltage equals approximately half of the battery voltage, Vcc.

When the toggle switch is moved from Off to Initialization, the circuit powers up, and the software detects that pin 5 is grounded. That begins the Initialization sequence. If the user then moves the switch back to Off, the EEPROM is updated to show that the eBike's battery is full. If the user leaves the switch in the Initialization state, the software reads the voltage on pin 7 and saves it as the value when fresh AA batteries have been installed. This value is stored in the EEPROM along with the eBike battery's capacity, as will be described on page 38.

[19] There is an internal pull-up resistor.

The Output

After the battery check, pins 6 and 7 are configured as outputs. When pin 7 is high, and pin 6 is low,

LED1 is on and
LED2 is off. When pin 6 is high, and pin 7 is low, LED1 is off, and LED2 is on.

By alternating the states of pin 6 and 7 fast enough, I change the relative brightness of the LEDs. Note that the total brightness is constant.

The High-Level Software View

Overview

The software takes up half of the available program store and 10% of the data memory. I do many direct writes to hardware registers, so I had to fully understand the ATtiny85. I give credit to Atmel. Their spec sheet is well organized, clear, and accurate.

Key Design Elements

I wasted a lot of effort trying to accurately measure R_{shunt}. I also wrestled with not knowing what each user defines as a battery in need of recharging. Then, one night, the solution popped into my head – let the software do it! This is the Calibration function described in the User's Guide.

Mark Twain once said, "I didn't have time to write you a short letter, so I wrote a long one." In other words, simplicity takes effort. I spent most of my design effort simplifying the User interface. I would build display hardware, write the associated code, field test it, be dissatisfied, and start over.

My current design has two key elements: the energy scale is divided into twelfths, and the display consists of two LEDs.

Why twelfths? I found that I could count four quick flashes of light at a single glance but no more: four flashes – quarters of the scale. This was too coarse, so I added a second LED. At first, I found ways to indicate quarters of the quarter. All schemes failed my field test. They were too confusing or distracting.

A point of annoyance was that half within a quarter was not centered. Eventually, I realized that I must divide the quarter into an odd number of parts if I want to indicate the center. Having five parts was too confusing, so I settled on three parts. This let me indicate high, middle, and low. Given the two LEDs, the top one indicates high, the bottom LED indicates low, and when both are on, that is the middle. If I've done my job right, you will see this as glaringly obvious.

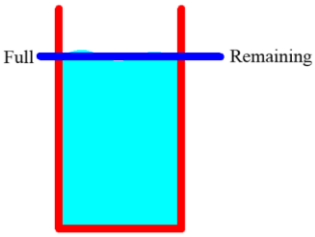
Given quarters and thirds of quarters, we end up with 4 x 3 = 12 parts. Hense, I divided the scale by twelfths.

Calibration Theory

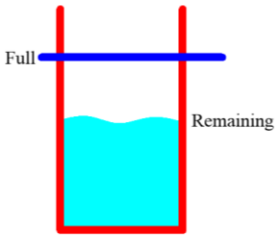
When is the battery "Empty?" It is, nominally, a 10.4 ampere-hour battery, so one answer is to say it is Empty when I have drawn 10.4 ampere-hours from it. Most people would be uncomfortable draining the battery this much. Think of how often you fill the gas tank on your car. If you don't let the needle go below ¼, then the effective definition of Empty is ¼. "Empty" is in the eye of the beholder.

Since I have not figured out how to write code that can read your mind, the next best thing is to ask. This is what Calibration does.

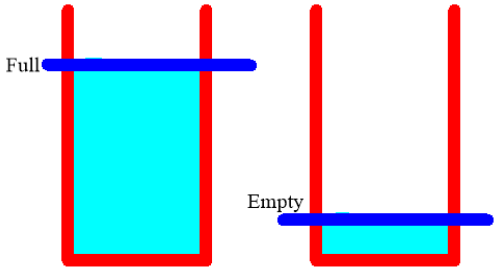
Two variables are involved in this process, FullChargeCount, and RemainingEnergyCount. I will use a bucket of water to explain.



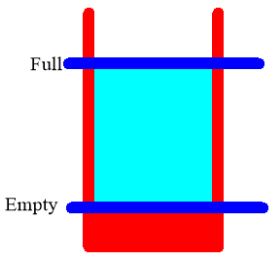
When the bucket is full, I put a mark on the bucket and call it "Full." I also have a piece of tape with a mark on it, pointing to this same level. Call this the "Remaining."



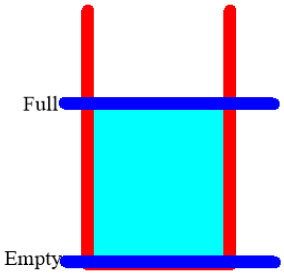
Then I randomly take water out of the bucket. Each time, I record how much I took plus move the tape so it again points to the current level of water. It doesn't matter what units I use on my measuring cup.



At some arbitrary point, I decide the bucket is empty enough, and I want to refill it. I mark this water level as "Empty." My records show the total amount of water I scooped out of the bucket. This is the *change* in water level.

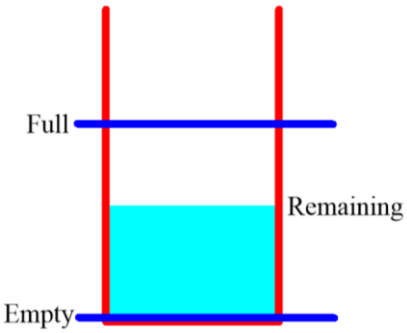


Notice that the remaining water effectively doesn't exist.



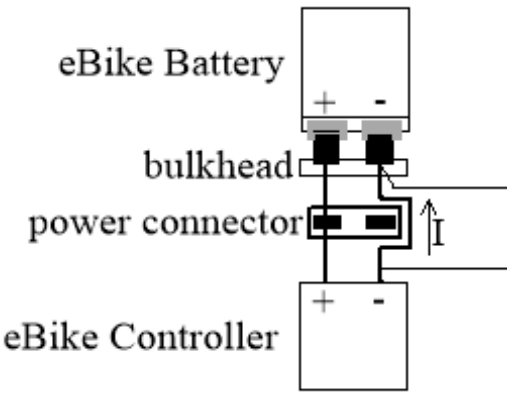
The user would be none the wiser if I moved my Full and Empty lines down the side of the bucket. Empty now equals zero. Full equals the change in water level. My bucket is "calibrated."

I refill the bucket to the Full line and set Remaining equal to the Full value. This time I do not look into the bucket but do keep track of what I take out using the same measuring cup used during Calibration.



Each time I take some water, I subtract that amount from Remaining. Without looking into the bucket, I know how much water is left.

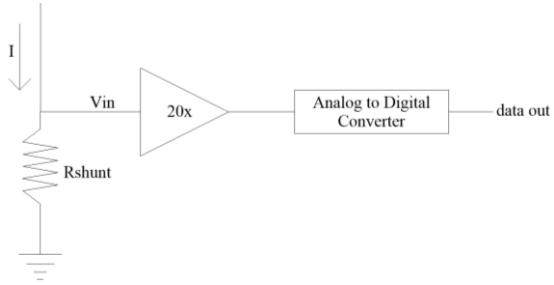
Notice that the capacity of the bucket is not important. I only need to know how much water I took out before I felt it is was time to refill. It is also not important to know the units on the side of the measuring cup. Each time I refill the bucket, I initialize Remaining to Full.



Let's return to the eBike's battery. My scheme does not need to know the capacity of the battery. I just need to know how many ampere-hours I took out before I felt it was time to recharge. I also do not need to know the relationship between the current out of the battery and the corresponding number generated by the circuit. When I recharge the battery, I initialize Remaining to Full.

The Logic Behind the Software

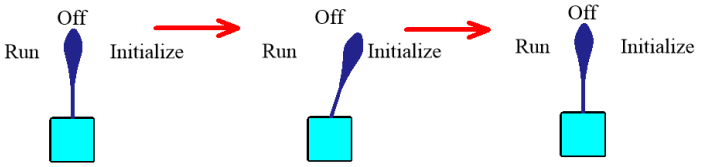
The software does not bother with ampere-hours. It deals in step-seconds.



The ADC is configured to be differential 10 bit[20], which means it outputs a number between -512 and +511. The maximum value that can be subtracted from the Remaining Energy Count each second is therefore 511. This corresponds to the maximum possible current from the battery, which is 20 amperes. Given that the battery is rated at 10.4 ampere-hours, we can sustain this current flow for $\frac{10.4\ ampere{-}hours}{20\ amperes} = 0.52\ hours$, which is 1872 seconds. The Remaining Energy Count would drop by $511\ counts\ per\ second\ \times 1872\ seconds = 956{,}592$ counts. By defining Remaining Energy Count as a long, it can easily contain this number. Furthermore, if the Remaining Energy Count was initialized to 0, we could still handle the maximum *change* in count. The Remaining Energy Count would contain a negative number. The calibration logic handles this case because it is only looking at the change in count.

Right after installing the software into the ATtiny85, the EEPROM that holds the FullChargeCount may be corrupted. This condition is detected, and FullChargeCount is set to 448,402. This number comes from assuming a shunt resistance of 3 milliohms, a battery capacity of 10.4 Ah, and an idea ADC conversion.

RemainingEnergyCount is set to zero. This is the same as an empty bucket with Full being marked above the top. Don't worry, this will be soon corrected. The user must first fully charge the eBike's battery. Then they go from **Off** to **Initialize** and back to **Off**. This will set RemainingEnergyCount to 448,402. This value should be close to the value set by the user so the eGas Gauge should be usable even without Calibration.

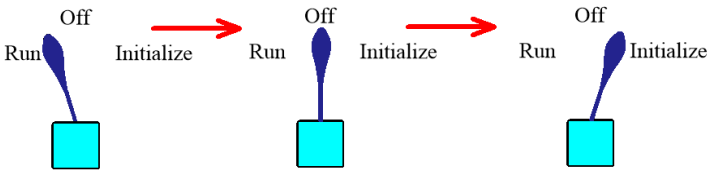


[20] Due to limiations in the ATtiny85, I can only get the 20X amplifier if I also accept the ADC configured to be differential. This is not all bad because it enables me to cancel the DC offset in this amplifier and the ADC.

The user then goes from **Off** to **Run** and rides their eBike until…

the battery level, as indicated by the ENERGY BAR[21], is considered, by them, to be ready to recharge (i.e., Empty). It can be with any number of bar segments.

During this time, RemainingEnergyCount is decremented by the count equivalent of the current each second.



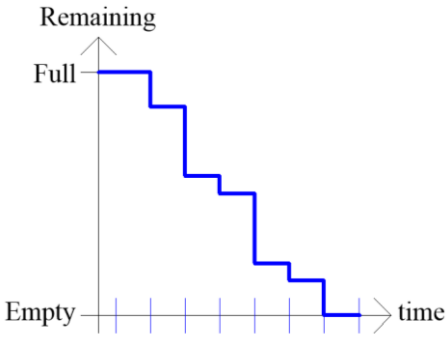
With the battery at "Empty," the user goes from **Run** to **Off** and then to **Initialize,** but this time they leave it in this position until they see a continuous stream of short flashes. This means Calibration was performed. Then they go back to **Off**.

During Calibration, the ***change*** in RemainingEnergyCount is calculated, and the result is saved as the FullChargeCount.

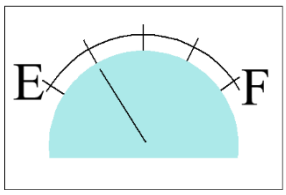
Recap: during the Calibration ride, the battery goes from Full to Empty while we recorded the number of ampere-hours drained from it. This number is the ***effective*** capacity of the battery. By storing this value as the FullChargeCount, we have characterized the battery as seen through the circuit.

Charge the battery back up and go do an **Initialization,** so the software knows the battery is fully charged.

[21] As read when the bike is not moving since this is a better indication of the available energy level.



When it is time for a ride, go from **Off** to **Run**. The RemainingEnergyCount starts out at the FullChargeCount, and will decrement as ampere-hours are used.



System Status Flash Sequences

Flash sequences are used to indicate when the battery powering the circuit is low, when we are done with Initialization, and when we are done with Calibration. They do not look anything like the eGas Gauge flashes. This, hopefully, minimizes confusion.

Initialization and Calibration only take a few milliseconds but, to give the user plenty of time to react, I stretch this out to many seconds. I do not want the user to accidentally do a Calibration since it will wipe out their hard-earned data related to the battery's capacity.

The Power Up Strategy

When the toggle switch is moved from **Off** to **Run**, a circuit battery test is performed. If the AA batteries are weak, you will see twenty seconds of short flashes. Then normal operation will be attempted. If these batteries are weak, the LEDs will be dim.

After the battery test, the eGas Gauge will run while the user rides.

The Power Down Strategy

The bike is drawing near zero current when the eGas Gauge is powered up. The start-up software records this current as a count and adds 2. The resulting number is the power-down threshold. When the measured current falls below this threshold for more than 10 minutes, the ATTiny-85 is put into low-power mode, and both LEDs are turned off. The resulting current drain is about 0.5 microamps. The batteries that power the eGas Gauge will likely leak before this amount of current runs them down.

EEPROM Strategy

I would like to save the remaining energy count to EEPROM just ***before*** the toggle switch is set to **Off** so the data will be available the next time the circuit is turned on. That would be one save to EEPROM per eBike trip. I don't know how to predict when power is about to be lost without adding a lot more hardware[22].

At the other extreme, I could save to EEPROM every second. This is a bad idea because the EEPROM will wear out after 100,000 writes. In 27 hours, I would have to replace the ATtiny85.

My compromise is to save data to EEPROM often enough that it last at least 5 years. This is $\frac{100{,}000\ writes}{5\ years}$ = 20,000 writes per year or 384 writes per week. Say I ride 3 times a week. This is $\frac{384\ writes\ per\ week}{3\ rides\ per\ week}$ = 128 writes per ride. Assuming each ride is 1.5 hours, this is $\frac{128 \text{ writes per ride}}{1.5\ hours\ per\ ride} = 85$ writes per hour or $\frac{85\ writes}{1\ hour} \times \frac{1\ hour}{60\ minutes} = 1.4\ writes\ per\ minutes$. I will do one write per minute, so the EEPROM should last more than 7 years.

When I turn the circuit off, I might lose one minute of data. On average, it will be ½ minute of data. Since the bike will be stopped, most of this time will be with no current flowing, so this data loss will have a minimal effect on the total count.

When I do write to EEPROM, I first read what is stored. If what I am about to write is the same as is stored, I do not do the write. This further saves EEPROM cycles.

I could write a more elaborate EEPROM function that does a write-read check. If the byte read back does not equal the byte written, it could move to a previously unused address. I guess that in 7 years, I will want a new eBike with an accurate energy level display.

[22] This could be done by having the user request power down and then have the software pull the plug after the EEPROM saves.

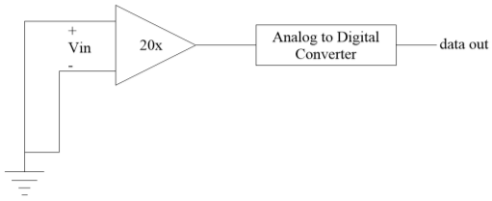
The Medium-Level Software View

A word about my programming style: I write so people can understand. This means I am generous with my comments plus with the naming of things. All constants and variables end with their data type. I have found that this heads off many stupid bugs related to mixing data types. For example, I have RemainingEnergyCountLong. This variable means Remaining Energy Count, and it is a long. I hope you say "duh." IMHO, that is a damn sight better than calling it "stuff1." My subroutine names are equally descriptive.

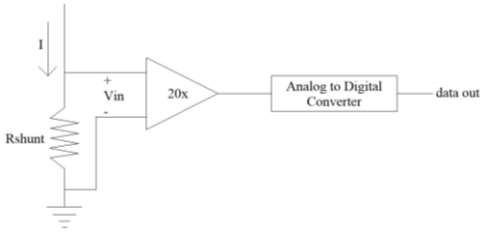
Arduino code is organized into the setup( ) subroutine that only runs at power-up and the loop( ) subroutine that runs repeatedly.

The subroutine setup() is where I look to see if **Initialize** is active at power-up. If it is, I initialize the RemainingEnergyCountLong plus its EEPROM copy to FullChargeCountLong.

If **Initialize** is not active, I read EEPROM and use it to initialize RemainingEnergyCountLong.



Here is where things get a bit tricky. By writing to hardware registers, I can measure the DC offset error of the amplifier. The input is set to 0, and the ADC tells me what it sees. This is the offset that is stored in ADC_OffsetLong. When I'm measuring the eBike's current, I subtract this offset. This removes the offset from my readings.



Before I leave setup(), I configure the amplifier for normal operation.

When setup() completes, I start loop(). As the name implies, we execute from the top of loop() to the bottom and then repeat. How long this takes depends on the included code.

On every cycle of loop(), I call Time(). This is where I keep all timers used by other subroutines. Communications are via flags. Each timer sets a flag to indicate its time is up and then starts timing all over again. The subroutine that uses the flag clears it.

When the Time To Sample flag is true, which happens every second, we

1. Read Current As Count
2. Update Energy Count
3. Translate Remaining Energy Count To the number of flashes related to quarters and which LEDs to drive to indicate bottom-middle-top.
4. Set the Time To Sample flag to false

Time() will set the Time To Sample flag true one second since the last time it set it true.

Every minute, we update the EEPROM with the remaining energy count.

The Low-Level Software View

I refer you to the code for this level of understanding. You should find the design familiar if you understand the high and medium-level software views.

I have included an EEPROM error reporting function. When we write to the EEPROM, the data is always read back. A mismatch generates an error that is reported by toggling the LEDs 100 ms on each.

If you find a comment confusing or wrong, please let me know so I may correct it. You can find my code at Rick.Sparber.org/eBikeGauge.ino.

Configure the IDE for an ATtiny85 with an internal 8 MHz clock.

This code has only been thoroughly tested on an ATtiny85. I know that it will not run on the Pro Micro because the hardware registers are laid out differently, plus it is not possible to access all needed inputs.

At the time of this writing, my program uses 56% of the available program store, 14% of global variable memory, and 3% of the data EEPROM. There is plenty of room for new features.

The First Field Testing

I rode until the ENERGY BAR read one bar with the bike stopped for 1 minute. The display showed 42.2 to 42.3 volts. I'll call this Empty.

This data was taken with an LED interface that flashed out the state of charge. It had a resolution of 16^{th} of full scale.

Calibration

After moving the toggle switch to Calibration and back to Run, I rode home. The eGas Gauge now knows how many counts have to be decremented as the battery goes from Full to (user-defined) Empty.

I again fully charged the bike's battery and did an Initialization.

Test Ride

My goal was to run the battery down and record all status updates. My ride was over a variety of terrains and at a variety of speeds.

I drained the battery over three rides. The tier numbers are related to an old user interface.

First Set of Data

Time minutes	Total minutes	First tier	Second tier	Comments
0	0	4	0	Full
15	15	4	0	at 7/8
17	17	3	4	at 7/8; zone change
30	30	3	2	at ¾; end of ride
0	30	3	2	at ¾; New ride
14	44	2	4	at 5/8; 3.8 miles
30	60	2	2	at ½; 7.8 miles; 46.9V; ENERGY BAR at ½
45	75	2	0	at 3/8; 11.6 miles; 46.1V
60	90	1	3	at 5/16; 14.1 miles; 45.7V; end of ride

Analysis of the First Set of Data

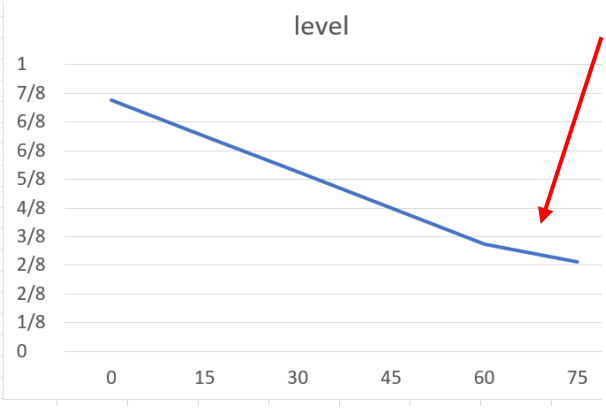
Focusing on the data highlighted in green, I can calculate an esimated range at each point in my ride. This is done by taking the distance ridden and dividing it by the change in remaining Ah.

Remaining Ah	Miles ridden	Estimated range, miles
3/4	0	-
5/8	3.8	30
1/2	7.8	31
3/8	11.6	31
5/16	14.1	32

The estimated range at the start of the ride was 6% lower than the estimte at the end of the ride. This says more about the consistency of the conditions during the ride. Still, this is nice but surprising result.

When the eGas Gauge signaled we were at ½, the ENERGY BAR showed the same thing after the bike sat for 2 minutes. This is confirmation that the two systems agree. So if you are willing to stop your bike for a few minutes before reading the ENERGY BAR, it is as accurate as the eGas Gauge. Of course, the benefit of the eGas Gauge is that you do not have to stop.

Graphing the remaining ampere-hours over time, I see:



From 60 to 75 minutes, I was going downhill so I used fewer ampere-hours. This is why the rate of consumption leveled off a little.

Between 30 and 90 minutes, I rode 14.1 miles and used 7/16th of the capacity. If the terrain was representative of the entire ride, I could expect to go $14.1 \times \frac{16}{7} = 32$ miles on a charge. I could also expect to ride for $(90 - 30) \times \frac{16}{7} = 137$ minutes on a charge. This means I should be able to ride for another 47 minutes. Time for another ride!

Second Set of Data

Time minutes	Total minutes	First tier	Second tier	Comments
0	90	1	3	at 5/16; New ride; 46.4V
15	105	1	1	3.5 miles; 45.1V
25	115	0		5.1 miles; 44.6V
31	121	0		44.6V
40	130			7.5 miles; 43.8V shutdown but recovered to 44.0V
50	140	0		9.1 miles; 42.7V
55	145	0		9.7 miles; 41.4V; toggled and heard Empty

Analysis of the Second Set of Data

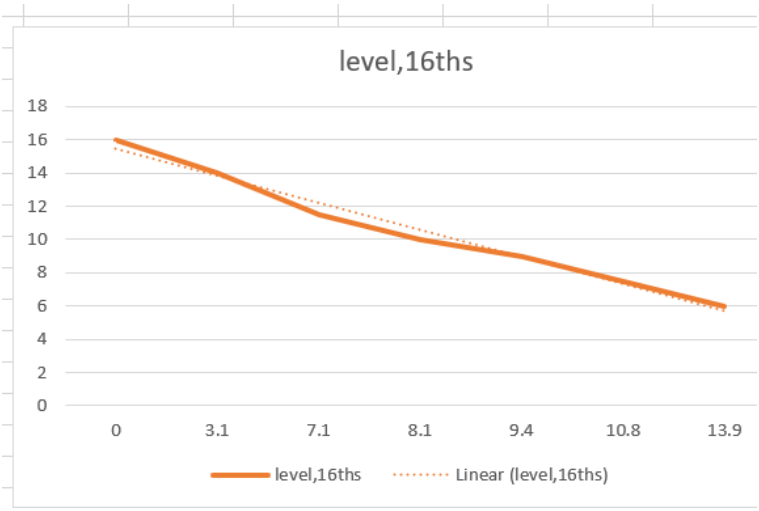
When the eGas Gauge indicated Empty, the display showed 41.4 to 42.7 volts. 40.6V corresponds to 10%, and 41.3V corresponds to 15%, yet it clearly is close to 0. Also, note that the Controller shut down at 44.0V, which corresponds to 35%. During Calibration, I defined Empty as 42.2 to 42.3 volts.

The Second Field Testing

This time, I ran the battery down until the cruise control dropped out while drawing about 12 amperes. Due to improvements in my instrumentation, I was able to directly read the EEPROM. I had drawn 362,353 count-seconds. My estimate of the capacity of the battery was 448,402 count-seconds. This means that my definition of Empty is when the battery has about 20% left in it. For the best life out of the battery, this is a good place to be.

I also read the half AA battery voltage. When compared to directly measuring this voltage, I found that the ATtiny85 reading was only 2.9% low. This is within the ±10% tolerance of the voltage divider.

The Third Field Test



This time, I was using the two LED interface. I recorded the miles driven each time I saw a change in the LED state. The terrain had many hills, some wind, and I was running PAS3.

The dashed line is the best fit. For the first 3 miles, I used slightly more energy than the best fit. This was mostly uphill. Then from about 3 to 9 miles, I was using slightly less energy. This was mostly level ground. I turned around on the trail at 7 miles. From 9 to 14 miles, I was mostly going downhill.

I found it interesting that the energy consumption per mile was almost constant as I rode a closed loop. This implies that terrain and wind are secondary factors. Speed is the primary factor that determines energy consumption.

Going from 16/16th down to 6/16th is a change of 10/16th. This got me 13.9 miles Extrapolating to 0, this says I have a range of $\frac{13.9\ miles}{\left(\frac{10}{16}\right)} = 22$ miles with PAS3.

A few days later I ran the same course but at PAS2. Using the same process, I estimate that the range is 41.2 miles.

Conclusion

The eGas Gauge did a better job of predicting when the battery was Empty than the voltage. During Calibration, I defined Empty when the voltage was around 42.2V, yet, with some resting, the voltage was around 41.4V to 42.7V. I think this reflects the variability in the voltage readings and not error in the eGas Gauge. When I saw 0 on the display, there wasn't much left in the battery.

The first set of field data demonstrated the value of using the change in Ah to predict the range. When the rider gets to the middle of ¾, multiply the trip odometer reading by 4 to get the estimated range. Accuracy depends on having a consistent set of riding conditions. If speed, wind, terrain, or energy contribution from the rider changes, this estimate will be off.

Final Comments

The ATtiny85 is not an easy place to do debugging. Instead, I first ran the code on a Pro Micro and used print statements and TeraTerm to see what was going on. When all looked right, I changed the code to run on the ATtiny85. This involved changing most of the writes and reads to hardware registers. Although the Pro Micro is a far more capable system, it cannot take differential readings or perform offset cancelation. Therefore, there was some painful debugging necessary on the ATtiny85.

Far too late in the debugging phase, I brought out my SkinnyPrint hardware and software. It let me debug by using single byte print statements. Although cryptic, this did make the work much easier.

I strive for simple hardware designs. As I look at the schematic, I am happy to see so few components and such a low price. The complexity of the software doesn't bother me at all. My goal is to have a design with an acceptable User interface with a readout accurate enough for the application.

Acknowledgment

Thanks to Eduardo for suggesting the pointer to programming the ATtiny85.

I welcome your comments and questions.

If you want me to contact you each time I publish an article, email me with "Subscribe" in the subject line. In the body of the email, please tell me if you are interested in metalworking, software plus electronics, kayaking, and/or the Lectric XP eBike so I can put you on the right distribution list.

If you are on a list and have had enough, email me "Unsubscribe" in the subject line. No hard feelings.

Rick Sparber
Rgsparber.ha@gmail.com
Rick.Sparber.org