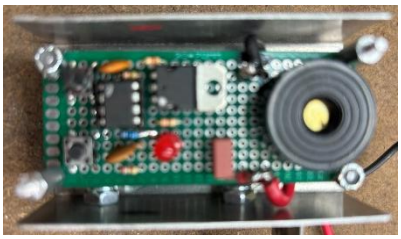
A Chevy Bolt EV 12V Battery Monitor

By R.G. Sparber

Scope



This device continuously monitors the 12-volt battery in a Chevy Bolt EV (electric car) and triggers an audible and visual alarm when the voltage drops below a user-defined threshold. This threshold is set just above the point where the car's computers start to fail randomly.

The cost of materials is under $10. The fun of designing and building it was priceless.

Background

When the 12V battery in an internal combustion engine car starts to fail, you can tell because it becomes difficult to start.

My battery-powered electric car, a 2023 Chevy Bolt EV, has a 12V battery that powers all the car's computers, plus a 400V battery that powers the traction motor. The 400V battery also powers a converter, which charges the 12V battery.

I can only speak to my experience with this car – when its 12V battery is nearing end-of-life, I get rare error messages and temporary functionality issues.

When I took my car to the dealer, their first question was the age of the 12V battery. Sure enough, the battery failed the load test.

Chevy did an amazing job of designing the monitoring and caring of the 400-volt traction battery in my car. I have no idea why they totally ignored the 12V battery.

I needed to add a system to my car that monitored for a weak 12V battery.

Contents

Know Versus Educated Guess

I am confident that a low battery voltage will cause sub-system computers to generate error messages. This condition is commonly called a brownout.

Reading about my particular battery, an AGM[1] type, I believe that 12V is a reasonable threshold for performing a load test. It should be low enough to avoid false alarms yet high enough to tell me before computers start to fail.

What I don't know is how the low-voltage condition behaves over time. I assume that very short dips in voltage are filtered out by each computer's power supply, but at some point, the computer will experience random errors.

I now have a new AGM battery in my car with my monitor set up to alarm when the voltage drops below 12.00 volts. I have no simple way to simulate a bad battery, so I will have to wait until the battery reaches end-of-life to know if my circuit properly warns me before I see random computer errors.

Details of the Problem

My car will run on a battery that fails the load test, but intermittent computer failures are likely. Eventually, the battery is so bad that the car stops running. Experience has shown that testing the battery with a load tester every 6 months is insufficient for a 3-year-old battery here in Phoenix[2].

The key parameter is the minimum battery voltage. If it dips below 12.0V[3], we are about to have computer problems. It is sufficient to alarm if this condition persists for more than a few milliseconds. I would then run a load test. If there were too many false alarms, I would raise the voltage threshold.

A secondary concern is the circuit's stability over temperature. It operates in an uncontrolled environment over the full temperature range of the electronics: -40°C to +125°C.

[1] https://lifelinebatteries.com/agm-batteries/
[2] Our high temperatures greatly shorten the life of a lead acid battery.
[3] See https://lifelinebatteries.com/agm-batteries/ and https://www.renogy.com/blogs/general-solar/agm-battery-voltage?srsltid=AfmBOoqL_YV46k2luOsPwY0faNFDDtutEGSxSbZh6mJ9URswvNTE9ePP

Existing Solutions

You can connect a voltmeter to your battery, but it will only tell you what is going on now. The voltage may have dipped low enough to cause a computer to falter in the past, but now it is high enough not to cause a problem.

You can buy a product called an Ancel BM200 Car Battery Monitor. According to their website[4], the device records the voltage every 2 minutes when the car is not being started and every 100 milliseconds during cranking.

Since my car doesn't have a starter, I assume the device records every 2 minutes. This sample rate may not be often enough to catch a momentary brownout that crashes one of the computers. I have no data on this failure mode.

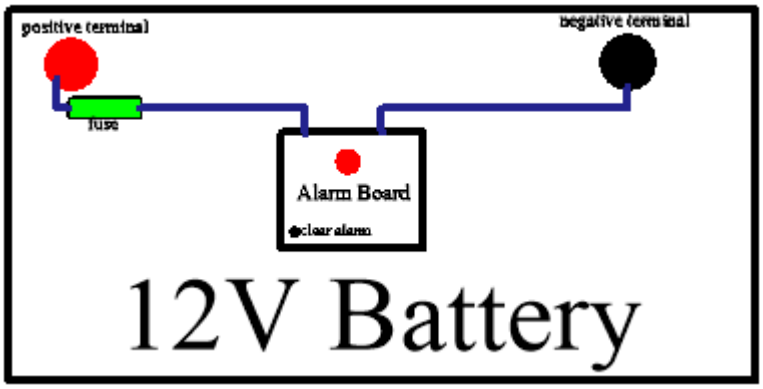
My design checks continuously, then runs a 50-millisecond scan to minimize false alarms. Here, I assume the car's computers can tolerate a single dip in voltage, but a sustained brownout would cause problems.

I'm also concerned about the dependence of any commercial device on an app. All too often, such devices become useless when the company stops supporting the app.

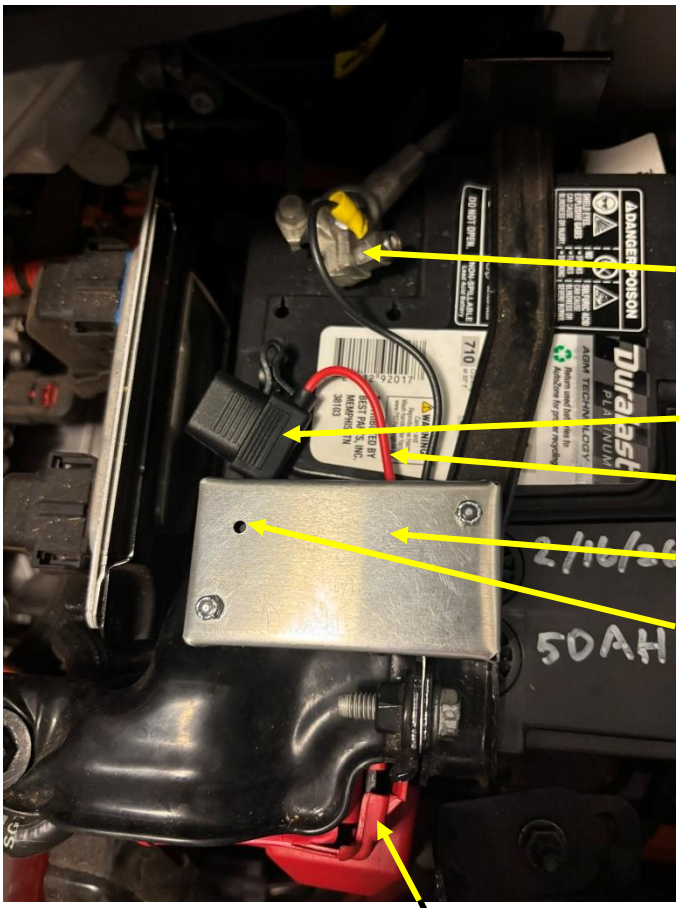
Of course, my biggest problem with any store-bought device is that I learn nothing from buying it. The monitor I designed and built for about 1/3 the price of the BM200 was challenging, fun, educational, and matched exactly what I wanted. Even if it costs more than the store-bought device, I would still make my own monitor.

[4] https://www.ancel.com/products/ancel-bm200

My Solution



I have designed, built, tested, and installed a circuit in my EV that continuously monitors the minimum terminal voltage of the 12V battery. If the voltage drops below 12.0 volts, we measure it 5 more times at 10 millisecond intervals. If the majority of these readings is below this threshold, an LED flashes and the beeper sounds until the circuit is reset by either pressing the RESET button or removing power.



Negative terminal

In-line 3 amp fast blow fuse

Positive terminal wire

Monitoring computer inside metal box

Reset button hole

Positive terminal under the red plastic cover

High-Level Design

Power and signal come directly from the 12V battery. Although the circuit draws no more than 16 milliamperes, we must use a fuse as close to the positive battery terminal as possible to avoid a fire in the event of a fault.

I use an ATTiny85 processor running a program that uses 2312 bytes of program storage and 13 bytes of global variable space. It might seem like overkill to use a processor for such a simple task, but keep in mind that the ATTiny85 costs around $3 and is in an 8-pin package. In quantity and smaller packages, the cost is below $2.

The circuit has three sources of error: resistors, ADC offset, and the voltage reference. The majority of the deviation from nominal is evident during circuit assembly. They are also sensitive to temperature. I use a two-step strategy to minimize these sources of error: calibration and temperature-compensated threshold.

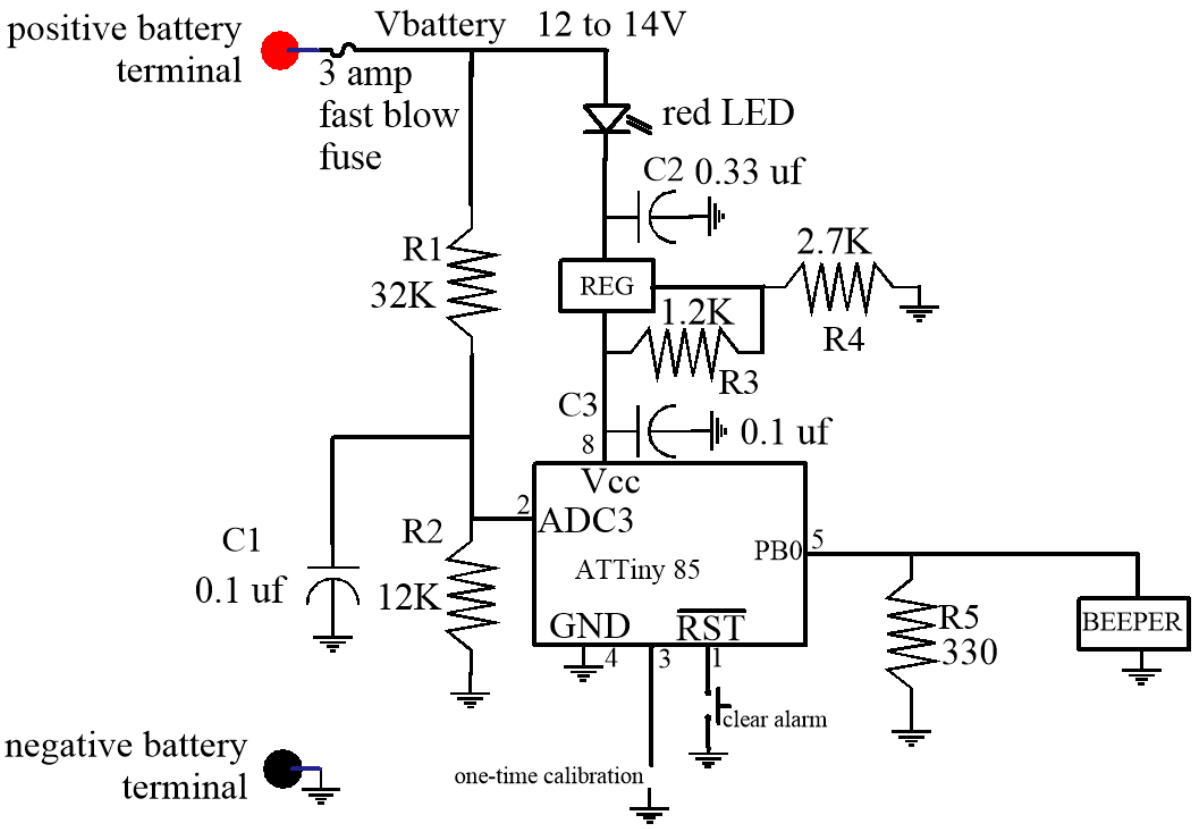
After I assembled the circuit, I powered it from 12.000 ± 0.006 volts[5]. Then I pushed the calibration button hidden inside the enclosure. Calibration caused the system to measure the applied battery voltage and record it as the battery threshold voltage in non-volatile memory (EEPROM). This action nulled out all errors at the current temperature. Given the accuracy of my calibration voltage source and the ADC's limitations[6], the threshold is 12.00 ± 0.01 volts, which is ten times more accurate than needed in this application.

The system then recorded the ambient temperature in the EEPROM for later use in creating a temperature-compensated voltage threshold.

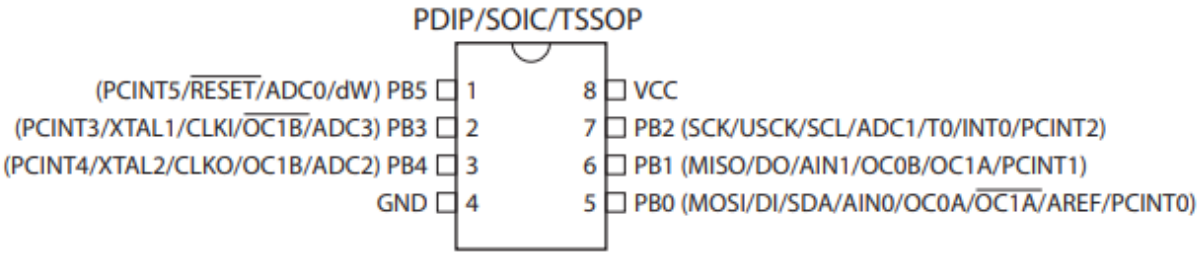
During normal operation, the circuit continuously measures the applied battery and the ambient temperature. It uses these parameters to determine if the battery voltage is below the temperature-compensated voltage threshold.

[5] I used a digital voltmeter with a stated accuracy of +/- 0.05%. When measuring 12 volts, this is an uncertainty of +/- 6 mV.
[6] This is a 10 bit ADC using a 4 volt reference. Nominally, this means an uncertainty of ± ½ count which is ± 2 mV at the ADC. At the battery, this is an error of ± 7 mV.

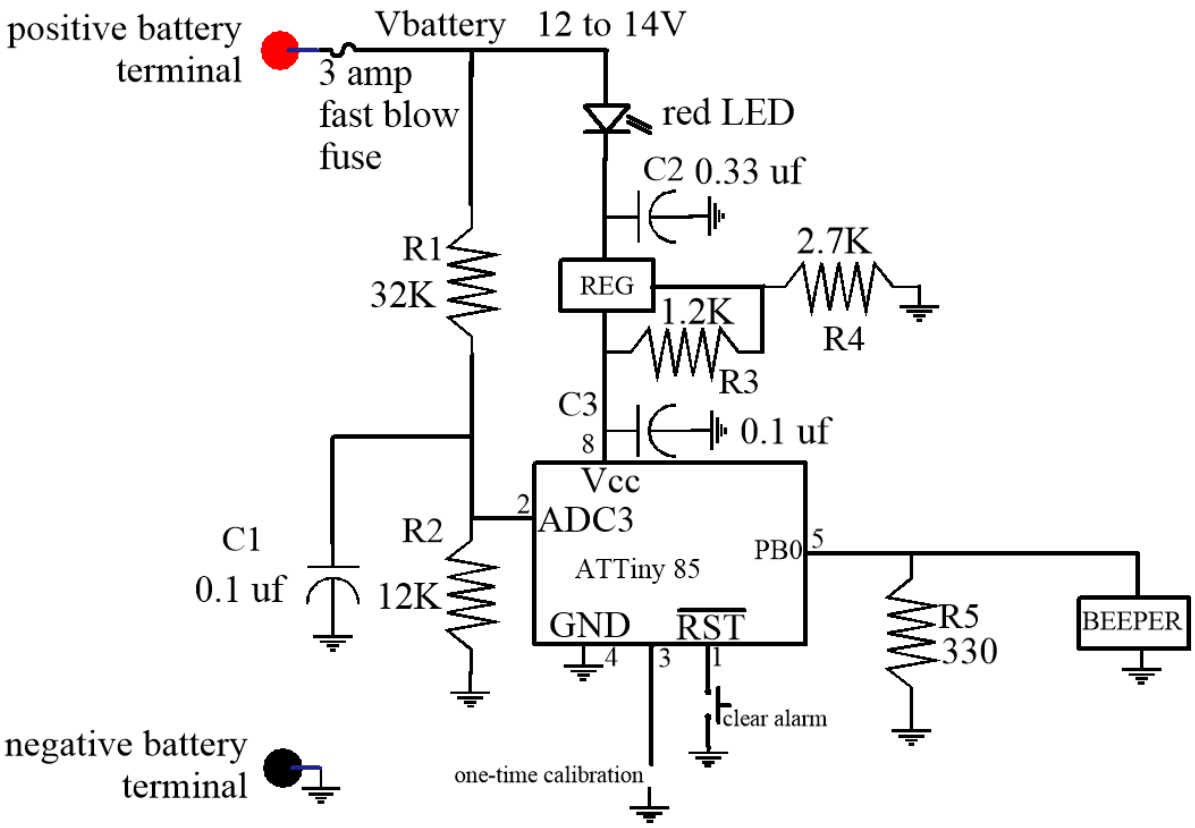
Hardware Design



The heart of the monitor is the ATTiny-85, in an 8-pin Dual In-line Package (DIP).



The ATTiny85 can be powered from a V_{cc} of 4V, and Vref can be set to V_{cc} under software control. This gives me a nominal supply current drain of under 4 mA and an Analog-to-Digital Converter (ADC) with a max input of 4V.



Alarm Strategy

My visual output is a red LED in series with the voltage regulator. It glows as the ATTiny's typical 5 milliamperes flows through it. When pin 5 goes to logic 1, an additional $(\frac{3.5V}{330\ \Omega} =)11$ milliamperes flows, making the LED brighter. The contrast between these two brightness levels is evident. Dim means the processor is powered, and bright means there is an alarm.

An alarm indicates it is time to perform a load test on the battery, as it may be near the end of its life. My experience has been that this condition does not constitute an emergency, just a warning. I check my various fluids under the hood each month, so checking for the flashing LED would be a small addition to the task.

For those who do not check their fluids monthly, the beeper would be more useful than the flashing LED. However, the beeper is large and costly.

The final choice of output device is to be determined.

The Temperature Compensation Strategy

I used a hot-air gun to raise the entire circuit's temperature, and a noncontact thermometer to estimate this temperature. At room temperature, I calibrated the circuit using a 12.000 volt source. Then I raised the circuit's temperature by about 40°F and observed a 14 mV increase in the threshold.

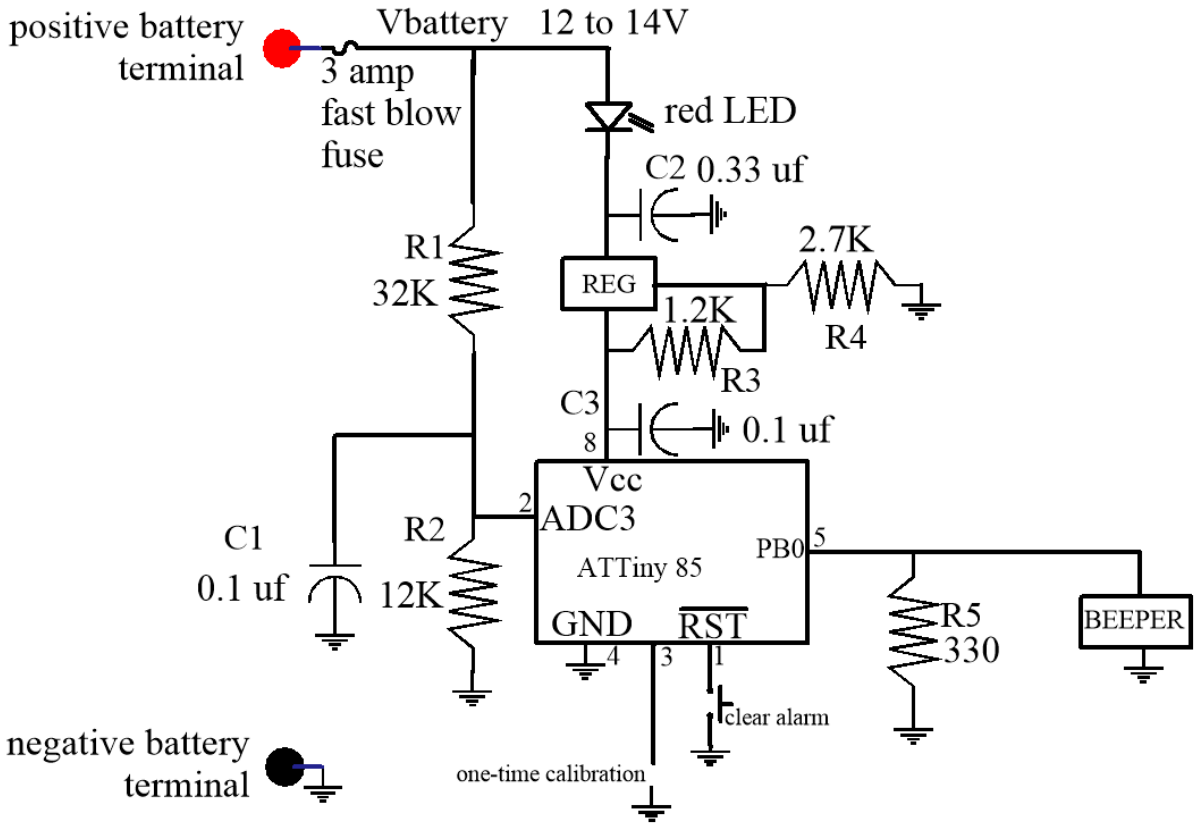
The ATTiny85 includes a temperature sensor. For absolute accuracy, the user must calibrate it. However, I'm only interested in relative changes in temperature. Furthermore, only a small correction to the threshold is needed, so even a large error in this compensation would have a small overall effect.

I empirically found that the change in temperature count, measured by the ATTiny85, divided by -2.6 provided a correction voltage that adds to the threshold to provide temperature compensation.

With this code in place and the circuit at room temperature, I verified that my threshold was at 12.00 volts. Then I used my heat gun to raise the temperature to where it was too hot to touch. The temperature-compensated threshold did not change.

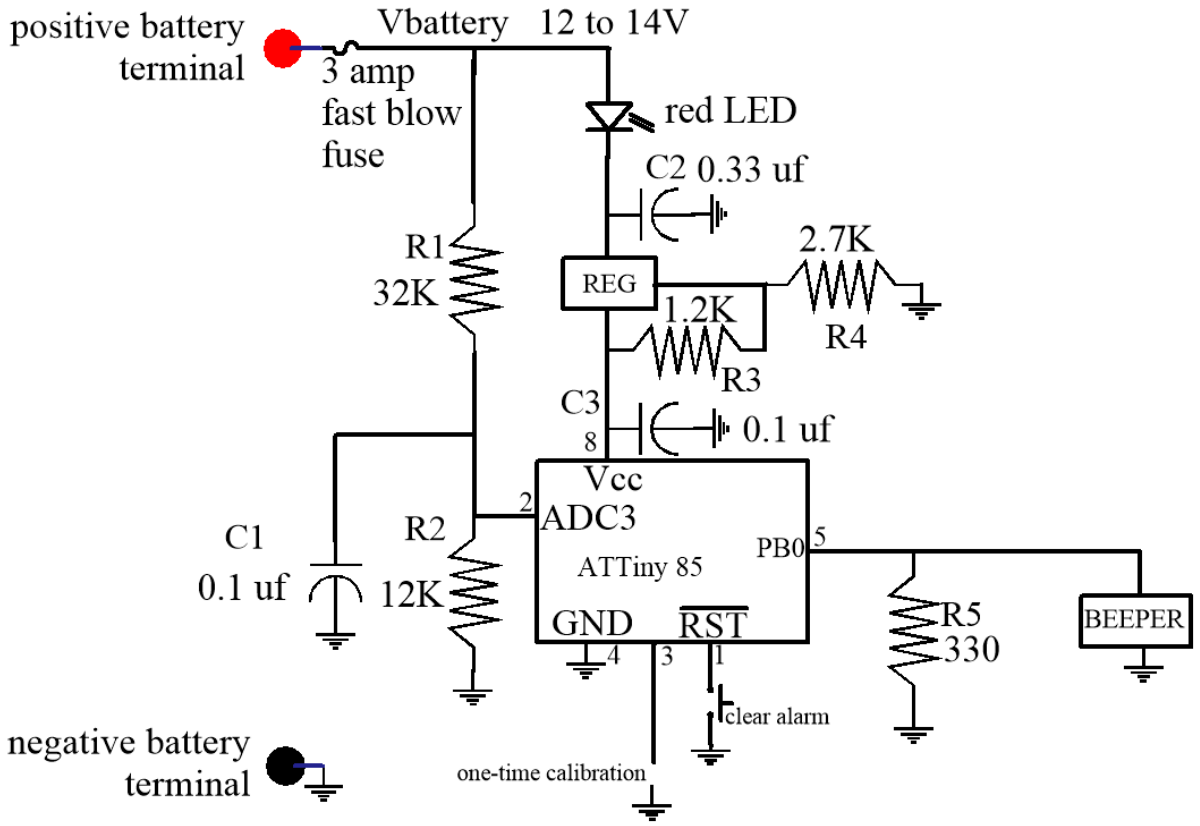
Circuit Design Choice

I am using a 3-terminal adjustable voltage regulator set up to output a nominal 4 volts. This voltage is connected to Vcc and serves as the ADC's reference voltage. My choice was based on having a large quantity of these devices, not on them being the lowest-cost solution. I could have used a 3-terminal 5-volt regulator, which would have saved two resistors and some space.



While monitoring the battery, the circuit typically draws 5.3 mA continuously. I have not measured the parasitic current drain on the 12V battery. AI claims it is 30-50 mA, but I cannot find any supporting documentation. This seems low given that I see a small spark when the battery is disconnected.

High-Level Circuit Description



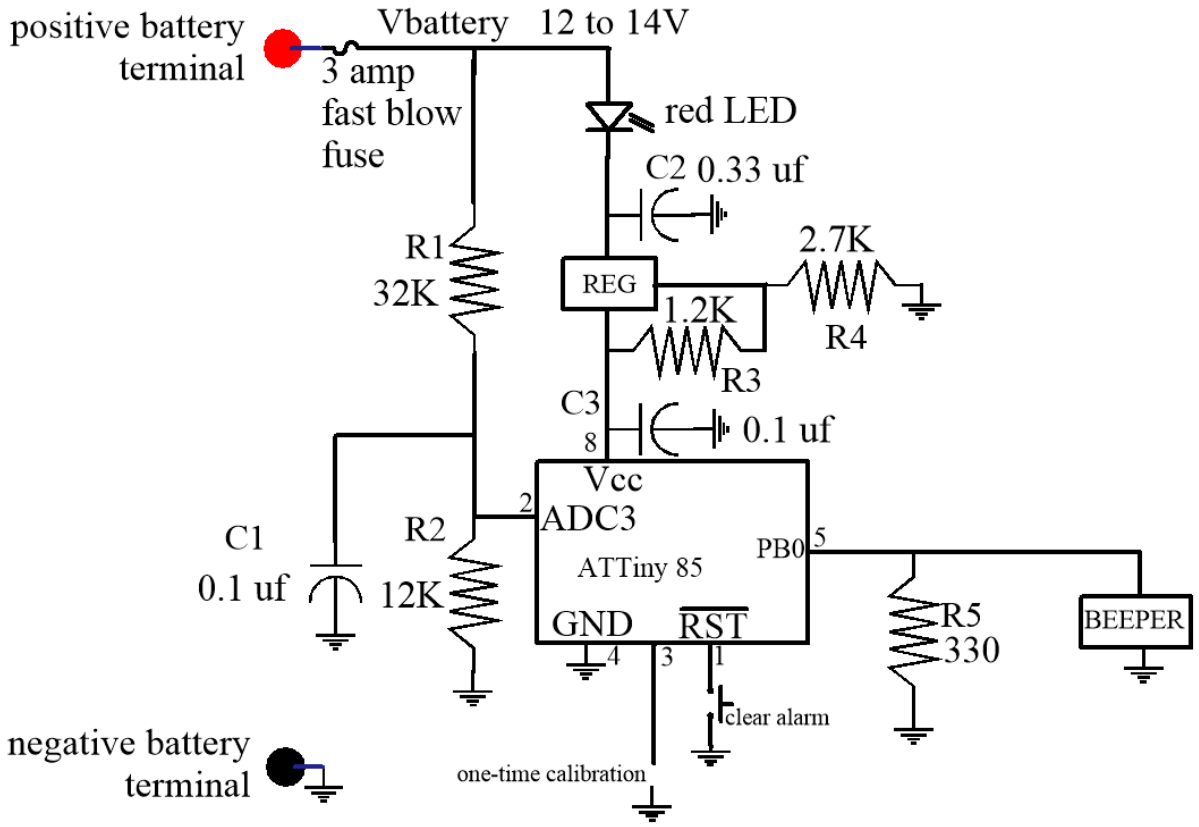
$V_{battery}$ is divided by R1 and R2 and fed into pin 2, which is the input to the Analog-to-Digital Converter.

When $V_{battery}$ is set to the threshold voltage[7], I momentarily connect pin 3 to ground. This action tells the software we are in calibration mode. The voltage on pin 2 is converted into a count and stored as our threshold. Note that this cancels the error in R1, R2, the ADC, and its reference voltage. The ATTiny contains a temperature sensor, which we use to record the ambient temperature during calibration.

During normal operation, while $V_{battery}$ is above the threshold, pin 5 is low. When $V_{battery}$ falls below the threshold, even briefly, the software pulses pin 5 up and down until the computer receives a reset.

[7] This is a one-time calibration, performed during fabrication.

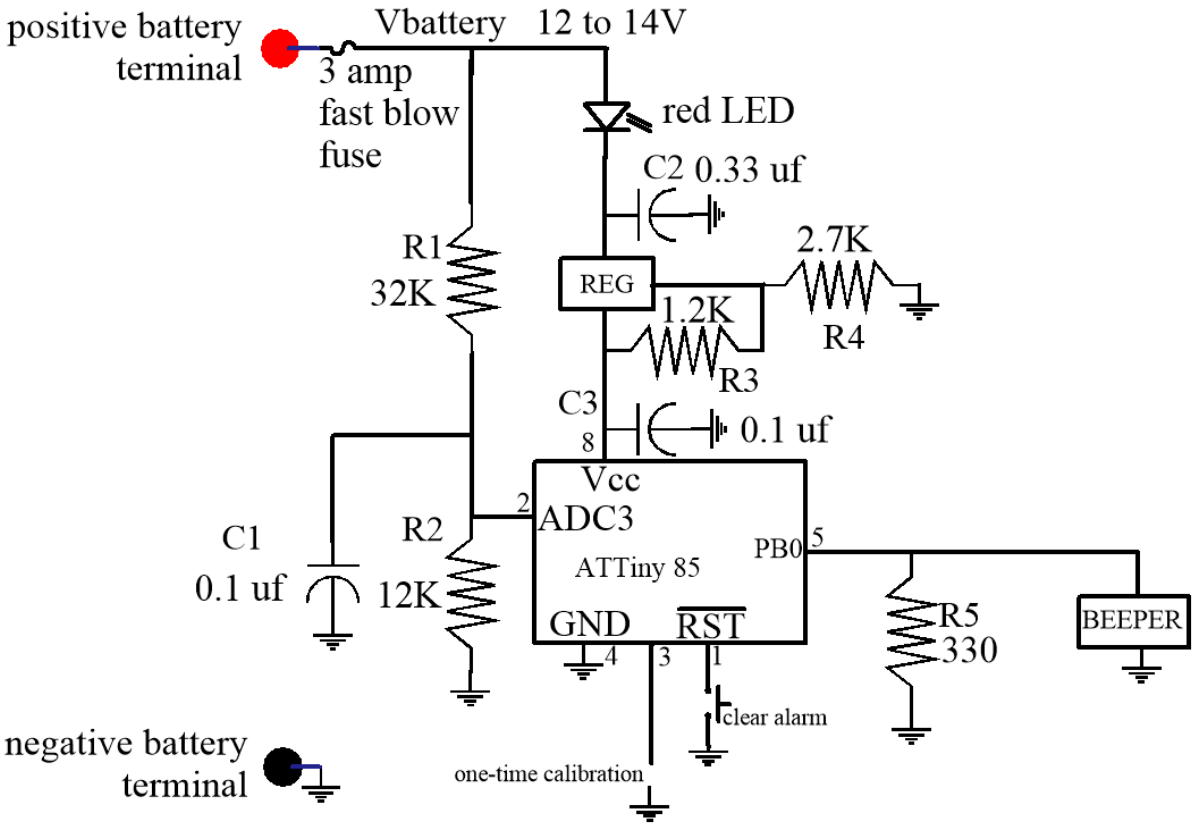
Nominally, the ATTiny draws 5 mA. This current flows from the battery, through the red LED, into the 3-terminal regulator, and into pin 8. It exists via pin 4 and returns via the negative battery terminal.



When pin 5 goes high, 11 mA flows through R5. This current is pulled through the regulator and therefore passes through the LED. This change in current from 5 to 16 mA is easily discerned as the red LED goes from dimmer to brighter.

When pin 5 goes high, the beeper also sounds.

Detailed Circuit Analysis



In general,

$$V_2 = \frac{R_2}{R_2+R_1} \times Vbatt \qquad (1)$$

$$V_2 = \frac{12K}{12K + 32K} \times Vbatt$$

$$V_2 = 0.272727 \times Vbatt$$

The ADC converts the input voltage to a count based on

$$count = \frac{V_2}{V_{ref}} \times 1024 \qquad (2)$$

Nominally, VREF is 4V. When Vbattery is at 12.000V, the nominal voltage on pin 2 is ($0.2727 \times 12.000V =$) 3.2727V

$$count = \frac{3.2727V}{4.0000} \times 1024$$

$$count = 837.8 \; so \; call \; it \; 838$$

The ADC rounds to the nearest count, which means an error of ±½ count, or ±2 mV at the ADC, which translates to an error of ($\frac{\pm 2 \; mV}{0.2727}$ =) +/- 7 mV at the battery.

Ideally,

$$V_2 = \frac{R_2}{R_2+R_1} \times Vbatt \qquad (1)$$

$$count = \frac{V_2}{V_{ref}} \times 1024 \qquad (2)$$

$$count = \frac{\{\frac{R_2}{R_2+R_1} \times Vbatt]}{4.0000} \times 1024 \;\; (1 \text{ into } 2)$$

$$count = \frac{\{0.272727 \times Vbatt]}{4.0000} \times 1024$$

$$count = 69.82 \times Vbatt \qquad (3)$$

$$where \; 69.82 \; is \; in \; counts \; per \; volt$$

For example, a count of 838 ideally is the result of applying 12.002V. The count goes down to 837 when we apply 11.988V.

The resistors and V_{ref} will not be ideal, but I assume they will not drift significantly from their values during calibration, assuming no change in temperature. This means that the count will change, but that doesn't matter. We are only interested in alarming when the applied voltage falls below the calibration voltage.

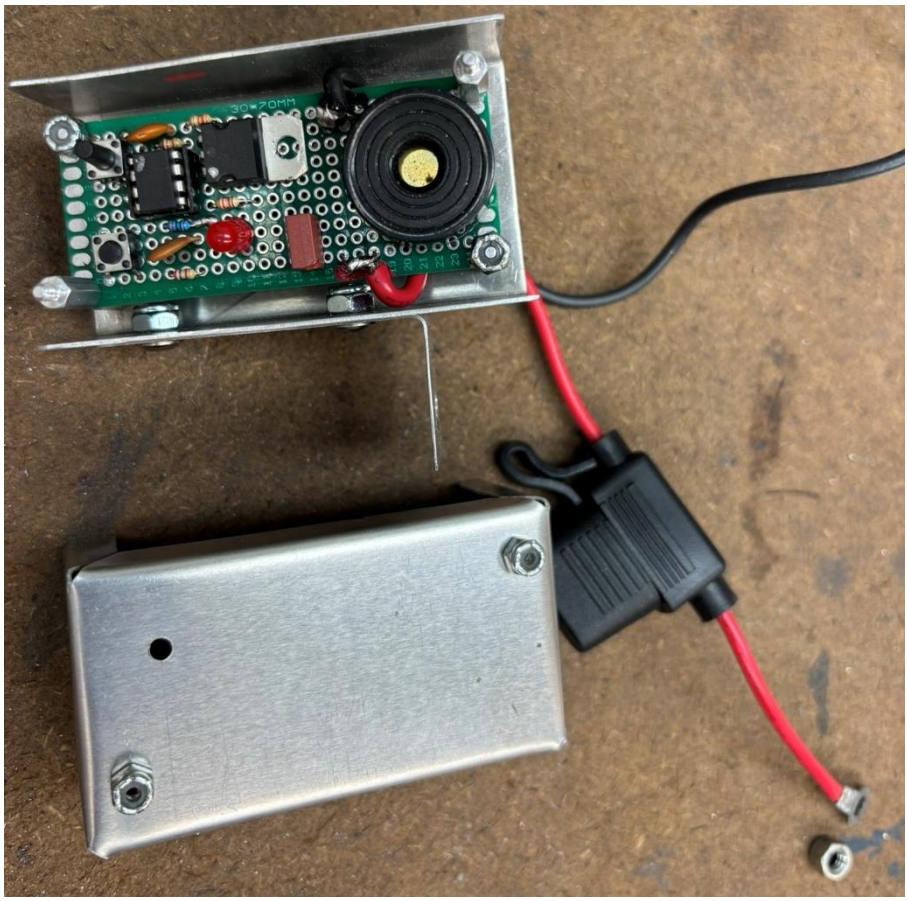
And finally, I checked the safe voltage limit. In the absolute worst case, with Vbattery = 14V and V2 = 4.1V, which is above the ADC's range but below the safe input range for the ATTiny[8].

[8] The maximum input voltage is 0.5V above V_{cc}. V_{cc} minimum is 4 – 5% = 3.80V so 0.5V above this absolute worst case is 4.4.3V. This is 0.2V above the maximum input voltage so we are safe.

Parts List

- ATTiny85 processor
- Red LED
- 32K, 12K, 1.2K, 2.7K, and 330 ohm 5% 0.1 watt
- (2) 0.1 uf ceramic
- 0.33 uf ceramic
- Pushbutton
- Piezoelectric beeper (optional)
- Circuit board
- Enclosure, rubber grommet for wires, mounting hardware[9]
- 3 amp fast blow automotive fuse in holder

Place the fuse as close to the positive battery terminal as possible to minimize the length of the unfused wire.



[9] The hardware will experience vibration so use nylon lined lock nuts where possible and Loctite in other places.

Software Overview

The code is written in C++ and compiled in the Arduino Integrated Development Environment (IDE). I set low-level ATTiny states using direct writes to hardware registers.

The code operates in two modes:

1. Calibration
2. Monitor battery voltage

Calibration is a one-time action performed during fabrication. A precision 12.000 volt bench supply powers the hardware. Then pin 3 is momentarily grounded. This starts the calibration software. The ADC count is stored in non-volatile memory along with a count representing the ambient temperature. Then the LED flashes once.

After briefly pressing the reset button, the software enters battery monitor mode. It performs these tasks continuously.

1. Read the ADC and store the count
2. Use the ADC to read the ambient temperature.
3. Calculate the temperature-compensated threshold using the stored threshold count, the stored calibration ambient temperature, and the current ambient temperature.
4. Compare the ADC reading to the temperature-compensated threshold.
    a. If the ADC reading is below the threshold, scan 5 more times and vote to see if the majority of readings are below the threshold.
        i. If the majority of readings are below the threshold, go into the alarm state.
        ii. If the majority of readings are above the threshold, restart the procedure.
    b. If the ADC reading exceeds the threshold, restart the process.

In the alarm state, we flash the LED between dim and bright, changing every ¼ second. Each time the LED becomes brighter, the beeper sounds. We leave this state when the user pushes the reset button.

Hardware Registers

Some standard Arduino functions work as expected in the ATTiny. Other Arduino functions may exist, but I never found them, so I went back to the spec sheet and worked directly with its hardware registers. This approach requires a careful read of the spec sheet, available at https://www.microchip.com/en-us/product/ATtiny85, which is 234 pages long. I didn't have to read all of it, but I did have to understand enough to know what I needed to read. In the following, you can use the register's name as a search parameter to find more information in this document.

ADMUX

This register controls aspects of the ADC and a multiplexer that selects which input the ADC will measure.

When measuring the battery voltage, I set the following bits:

- Use V_{cc} instead of V_{ref} set bit 7 = 0, bit 6 = 0, bit 4 = 0
- To have the ADC output be right-adjusted, set bit 5 = 0
- The right 4 bits are MUX[3:0] = 0011, which ties our ADC input to ADC3 and makes it a single-ended input. This works in conjunction with analogRead(ADCport_Byte). ADC3 is physical pin 2, which is our divided-down battery voltage.

All of these bits are assigned to the constant I call `ADMUX_OperationalConfigurationByte`.

When measuring the temperature, the MUX is set to a port that does not drive a pin. This port ties to the on-chip temperature sensor.

- Use 1.1V as Vref (bit 7 = 1, bit 6 = 0, bit 4 = 0)
- To have the ADC output to be right-adjusted set bit 5 = 0
- The right 4 bits are MUX[3:0] = 1111: ADC4 is the temperature sensor input.

I assigned these bits to the constant `ADMUX_TemperatureConfigurationByte`.

CLKPR

This register controls the system clock's prescaler. By experience, I have found that the prescaler sometimes changes from x1 to x8, which slows system timing by a factor of 8. As a defensive measure, I set the prescaller to x1 in setup().

- Bit 7 is 1, and all other bits are 0
- Then I set bit 7 to 0, and that locks in the remaining bits.

EEPROM

The ATTiny contains a small quantity of non-volatile memory, EEPROM. My program uses four bytes.

When reading and writing to the EEPROM, we use these hardware registers:

- EECR is the EEPROM Control Register. I use it to tell the hardware that I want to write to the EEPROM, and it tells me whether the EEPROM is ready for this operation.
- EEPE is a status bit that says when you can access the EEPROM. It is located in the EECR
- EEARL is set to the LSB of the address. In my case, it is the address.
- EEARH is set to the MSB of the address. In my case, it is always 0.
- EEDR is where the read data is placed.

Fortunately, the spec sheet includes the code, in C++ needed to read and write to the EEPROM. Most of it makes sense.

Software Middle Level View

In this section, I assume the reader has minimal understanding of Arduino but some basic understanding of C++.

setup()

All code in setup() executes at power-up or reset. This is where I set up the environment, plus retrieve the data stored in the EEPROM.

loop()

While setup() only runs once, loop() executes all code within it until power is removed or the reset pin is pulled low.

The first function in `loop()` is `calibrationQ()`. The "Q" stands in place of "?" because "?" is an illegal character in a function name. It checks whether the calibration signal is active (pin 3 momentarily grounded). If this pin is low, a calibration is performed. This would be done only as the final step in the board's fabrication. Results are stored in the EEPROM.

The second function in loop() is `checkThreshold()`. It reads the battery voltage and acts on the results.

Software Low-Level View

See GitHub for the code: https://github.com/rgsparber/Chevy-Bolt-EV-electric-car-12-Volt-Battery-Monitor

What Next?

This project is protected under Creative Commons. You are free to copy, modify, and sell it with no obligation to me other than not removing my name from my design. If I were going into production, I would switch to surface-mount devices, look into ATtiny devices with less program storage, switch to a fixed-voltage regulator, and use a 3D-printed enclosure made of PETG. A double-sided board can easily support the circuit.