Battery Monitoring for Electric Vehicle Battery Packs

June 5, 2016

Senior Project for

Matthew Hennes  
with advisor  
Jeffrey Gerfen

Contents

[Purpose 1](#_Toc452990010)

[System Requirements 1](#_Toc452990011)

[System Specifications 1](#_Toc452990012)

[System Architecture 2](#_Toc452990013)

[Hardware Overview 2](#_Toc452990014)

[Software Overview 3](#_Toc452990015)

[References 4](#_Toc452990016)

[Appendix 4](#_Toc452990017)

[Appendix A: Microcontroller Code 4](#_Toc452990018)

# Purpose

Battery packs containing multiple batteries in series must be kept balanced, with each individual battery at the same voltage in order to increase the longevity of the battery pack as a whole. If some batteries in the pack are a significantly higher voltage than others, those higher voltage batteries will charge more quickly, and will then be damaged by overcharging while the lower voltage batteries catch up. In lead acid batteries, overcharged batteries will gas, consuming electrolyte in the process, and will therefore need more frequent maintenance (watering) to be kept in working order. [1]

In order to keep battery packs balanced, individual batteries are must periodically be removed and charged individually, so that all batteries in the pack are at or near the same potential. To determine when this is necessary with a simple, “dumb” battery pack, a technician would typically open the battery pack and measure to potential across each individual battery by hand. This process could be greatly simplified if the measurement process happened automatically without the need to open the battery pack.

The purpose of this project is to develop a system capable of taking automated measurements of the voltages of the batteries in the battery pack for the Electric Vehicle Engineering Club’s electric van. The club was recently forced to replace two battery packs because they were destroyed by severely unbalanced batteries. This large cost for the club could have been avoided if a system like this was being used to keep track of individual battery voltages.

# System Requirements

This battery monitoring system will be used to measure the individual voltages of batteries in a battery pack. While it may also be capable of monitoring other designs, it will be specifically intended to monitor lead acid batteries wired in series to create a single, high voltage battery pack. The system will be capable of measuring the individual voltages of eighteen batteries. The maximum measurable voltage will be 250V, which makes the system ideal for measuring the voltages of standard 12V batteries such as the ones used in the automotive and marine industries. The system will be capable of measuring battery voltage to within 2% of the true value.

# System Specifications

|  |  |  |  |
| --- | --- | --- | --- |
| Specification | Minimum | Typical | Maximum |
| Input voltage |  | 216 V DC | 250 V DC |
| Number of individual batteries | 1 | 18 | 18 |
| Measurement accuracy | 360 mV | 240 mV | 61 mV |
| Measurement time (per battery) |  | 150 ms |  |
| Measurement frequency (18 batteries) |  | Once per minute | 20 times per minute |
| Auxiliary battery voltage | 6 V | 9 V | 12 V |
| System power consumption (from auxiliary battery) |  | 15 mA |  |
| Measurement current draw (from target battery pack, at 216 V) |  | 853 µA |  |
| Operating temperature | 0°C | 25°C | 70°C[[1]](#footnote-1) |
| Storage temperature | -65°C |  | 150°C |
| UART baud rate |  | 9600 |  |

# System Architecture

## Hardware Overview



Figure : Hardware Overview

The system is essentially comprised of a microcontroller which selects a battery to be measured, turns on an optocoupler to supply the battery’s voltage to a voltage divider. The voltage is then measured by an external analog to digital converter (ADC), which sends the measured value to the microcontroller over a serial peripheral interface (SPI) connection.

Figure 1 shows how a single battery would be connected in order to be measured. Because the microcontroller has limited general purpose input/output (GPIO) pins, 3:8 decoders are used to interface the microcontroller to the eighteen individual optocouplers (one per battery). Each of these decoders requires three GPIO pins to address, plus one GPIO pin to select which decoder is currently in use, for a total of four pins per decoder. Each decoder is connected to six optocouplers, so a total of three decoders are needed. These three decoders use a total of 12 GPIO pins, as opposed to the 18 pins that would be required to directly drive the optocouplers from the microcontroller. Further optimization could be achieved by using the same GPIO pins for the address signals to all three decoders, reducing the number of GPIO pins used to six, but this level of optimization is not necessary, so unique address pins are used for each decoder in order to improve code clarity.

The vast majority of microcontrollers and integrated circuits (such as ADCs) run on 3.3V or 5V, and will be destroyed by the types of voltages used in automotive battery packs. The full voltage of the battery pack (up to 250V) cannot be connected directly to the microcontroller or an external ADC, and therefore cannot be measured directly. To resolve this, a voltage divider is used, and voltage is measured across the smaller resistor. In this case we use a 249kΩ resistor and a 4.7kΩ resistor to form an equivalent resistance of 253.7kΩ. Voltage is measured across the 4.7kΩ resistor, which gives us a 53.98:1 gain. That is, each volt measured by the ADC is equivalent to 53.98 volts on the actual battery being measured. This is then compensated in software by multiplying the ADC’s output by 53.98 to calculate the real voltage of the battery.

Most microcontrollers have onboard ADCs, but these onboard ADCs are generally limited to 10-bit or lower resolution. Ten-bit resolution (1,024 steps) would give us a step size of 244.1mV step size when measuring a 250V battery. This step size defines the maximum (worst case) accuracy with which we can measure battery voltage. This 244.1mV step size equates to 2.034% of the nominal voltage of a 12V battery, which is just outside the required accuracy of 2%. To achieve higher accuracy, this system uses a discrete ADC with a 12-bit resolution (4,096 steps). With 12-bit resolution, our worst case accuracy is improved to 61.04mV, or 0.5086% of the nominal voltage of a 12V battery, well within the requirement of 2%.

## Software Overview



Figure : Software Flow

The microcontroller code for this system works in three primary steps: setup, measurement, and output. The setup step initializes all components as well as the microcontroller itself. It also starts up SPI (used to communicate with the ADC) and universal asynchronous receiver/transmitter (UART) (used for outputting results) on the microcontroller.

The measurement step runs in a loop until the voltage of each battery has been measured. First, it is ensured that all optocouplers are disabled (to ensure that a short between different batteries is not created). Then, the necessary signals are sent to the relevant decoder to turn on the optocoupler associated with the battery to be measured. After a short delay to allow transients to level off, the microcontroller then begins clocking the SPI clock signal, which initiates a measurement on the ADC. The ADC then returns the measured value to the microcontroller via SPI. The microcontroller stores this value and moves on to the next battery.

The output step iterates through all the recorded voltages for each battery, converting the raw ADC value to a voltage value, and sends it out via UART. Raw data comes from the ADC in the form of an unsigned integer between 0 and 4095. In order to obtain the voltage of each individual battery, the ADC’s output value for each battery is subtracted from the preceding value. So for example, the measurement for the fourth battery is the value measured on the fourth battery minus the value measured on the third battery. The raw value is kept for the first battery, as its negative terminal is connected to ground, which is also the signal used for the ADC’s negative input. The output of the ADC is an unsigned integer from 0 to 4,095, so once the potential difference for each battery has been calculated, it must be converted to a useful floating point representation. This is done by multiplying each battery’s measurement by a constant, floating point value which translates the ADC’s output into a meaningful voltage. See Appendix A: Microcontroller Code for the full source code.

# References

|  |  |
| --- | --- |
| [1] | C&D Technologies, "Charging Valve Regulated Lead Acid Batteries," C&D Technologies, Inc., Blue Bell, PA, 2012. |
| [2] | Battery University, "BU-410: Charging at High and Low Temperatures," Cadex Electronics Inc., 2 April 2016. [Online]. Available: http://batteryuniversity.com/learn/article/charging\_at\_high\_and\_low\_temperatures. [Accessed 6 June 2016]. |

# Appendix

## Appendix A: Microcontroller Code

// Top resistor: 248.4 k

// Bottom resistor: 4.68 k

// Req: 253.08 k

#include <SPI.h>

const int ADC\_VAL\_BUFF\_SIZE = 50;

const int NUM\_BATTERIES = 18;

const float V\_REF = 5.06;

const float R\_EQ = 253.08;

const float R\_MEAS = 4.68;

const float STEP\_SIZE = (2 \* V\_REF) / 8192;

const float R\_MULTIPLIER = R\_EQ / R\_MEAS;

const float V\_MULTIPLIER = R\_MULTIPLIER \* STEP\_SIZE;

const int ADC\_DISABLE = 6; //D7

const int DEMUX1\_ENABLE = 12; //D6

const int DEMUX1\_A = 8; //B4

const int DEMUX1\_B = 9; //B5

const int DEMUX1\_C = 10; //B6

const int DEMUX2\_ENABLE = 11; //B7

const int DEMUX2\_A = 19; //F6

const int DEMUX2\_B = 20; //F5

const int DEMUX2\_C = 21; //F4

const int DEMUX3\_ENABLE = 4; //D4

const int DEMUX3\_A = 1; //D3

const int DEMUX3\_B = 0; //D2

const int DEMUX3\_C = 2; //D1

// Setup the various components

void setup()

{

Serial.begin(9600);

// start the SPI library:

SPI.begin();

// Initialize ADC

pinMode(ADC\_DISABLE, OUTPUT);

digitalWrite(ADC\_DISABLE, HIGH);

// Initialize Demux 1

pinMode(DEMUX1\_ENABLE, OUTPUT);

digitalWrite(DEMUX1\_ENABLE, LOW);

pinMode(DEMUX1\_A, OUTPUT);

digitalWrite(DEMUX1\_A, LOW);

pinMode(DEMUX1\_B, OUTPUT);

digitalWrite(DEMUX1\_B, LOW);

pinMode(DEMUX1\_C, OUTPUT);

digitalWrite(DEMUX1\_B, LOW);

// Initialize Demux 2

pinMode(DEMUX2\_ENABLE, OUTPUT);

digitalWrite(DEMUX2\_ENABLE, LOW);

pinMode(DEMUX2\_A, OUTPUT);

digitalWrite(DEMUX2\_A, LOW);

pinMode(DEMUX2\_B, OUTPUT);

digitalWrite(DEMUX2\_B, LOW);

pinMode(DEMUX2\_C, OUTPUT);

digitalWrite(DEMUX2\_C, LOW);

// Initialize Demux 3

pinMode(DEMUX3\_ENABLE, OUTPUT);

digitalWrite(DEMUX3\_ENABLE, LOW);

pinMode(DEMUX3\_A, OUTPUT);

digitalWrite(DEMUX3\_A, LOW);

pinMode(DEMUX3\_B, OUTPUT);

digitalWrite(DEMUX3\_B, LOW);

pinMode(DEMUX3\_C, OUTPUT);

digitalWrite(DEMUX3\_C, LOW);

}

// Take measuremnts and output results

void loop()

{

unsigned int ADCResults[NUM\_BATTERIES];

char value[ADC\_VAL\_BUFF\_SIZE];

char battery[ADC\_VAL\_BUFF\_SIZE];

char raw[ADC\_VAL\_BUFF\_SIZE];

float voltage = 0;

// Take measurements

for (int i = 0; i < NUM\_BATTERIES; i++)

{

ADCResults[i] = measureBattery(i);

/\*

voltage = ADCResult \* V\_MULTIPLIER;

Serial.write("Voltage ");

String(i).toCharArray(battery, ADC\_VAL\_BUFF\_SIZE);

Serial.write(battery);

Serial.write(": ");

String(ADCResult).toCharArray(raw, ADC\_VAL\_BUFF\_SIZE);

Serial.write(raw);

Serial.write(" = ");

String(voltage).toCharArray(value, ADC\_VAL\_BUFF\_SIZE);

Serial.write(value);

Serial.write(" V\n");

\*/

delay(100);

}

// Output results

for (int i = 0; i < NUM\_BATTERIES; i++)

{

if (i == 0)

voltage = ADCResults[i] \* V\_MULTIPLIER;

else

voltage = (ADCResults[i] - ADCResults[i - 1]) \*

V\_MULTIPLIER;

Serial.write("Voltage ");

String(i).toCharArray(battery, ADC\_VAL\_BUFF\_SIZE);

Serial.write(battery);

Serial.write(": ");

String(voltage).toCharArray(value, ADC\_VAL\_BUFF\_SIZE);

Serial.write(value);

Serial.write(" V\n");

}

Serial.write("\n");

delay(1000);

}

// Take an ADC sample, and read it from SPI

unsigned int takeADCSample()

{

unsigned int result = 0;

SPI.beginTransaction(SPISettings(1700000, MSBFIRST,

SPI\_MODE0));

delay(100);

digitalWrite(ADC\_DISABLE, LOW);

delay(100);

result = SPI.transfer16(0);

delay(100);

SPI.endTransaction();

digitalWrite(ADC\_DISABLE, HIGH);

return result + 11; // compensate for offset

}

// Measure the specified battery's voltage

unsigned int measureBattery(int battery)

{

unsigned int ADC\_Value;

digitalWrite(DEMUX1\_ENABLE, LOW);

digitalWrite(DEMUX1\_A, LOW);

digitalWrite(DEMUX1\_B, LOW);

digitalWrite(DEMUX1\_C, LOW);

digitalWrite(DEMUX2\_ENABLE, LOW);

digitalWrite(DEMUX2\_A, LOW);

digitalWrite(DEMUX2\_B, LOW);

digitalWrite(DEMUX2\_C, LOW);

digitalWrite(DEMUX3\_ENABLE, LOW);

digitalWrite(DEMUX3\_A, LOW);

digitalWrite(DEMUX3\_B, LOW);

digitalWrite(DEMUX3\_C, LOW);

switch(battery)

{

case 0:

digitalWrite(DEMUX1\_ENABLE, HIGH);

//delay(10000);

break;

case 1:

digitalWrite(DEMUX1\_ENABLE, HIGH);

digitalWrite(DEMUX1\_A, HIGH);

//delay(10000);

break;

case 2:

digitalWrite(DEMUX1\_ENABLE, HIGH);

digitalWrite(DEMUX1\_B, HIGH);

//delay(10000);

break;

case 3:

digitalWrite(DEMUX1\_ENABLE, HIGH);

digitalWrite(DEMUX1\_B, HIGH);

digitalWrite(DEMUX1\_A, HIGH);

//delay(10000);

break;

case 4:

digitalWrite(DEMUX1\_ENABLE, HIGH);

digitalWrite(DEMUX1\_C, HIGH);

//delay(10000);

break;

case 5:

digitalWrite(DEMUX1\_ENABLE, HIGH);

digitalWrite(DEMUX1\_C, HIGH);

digitalWrite(DEMUX1\_A, HIGH);

//delay(10000);

break;

case 6:

digitalWrite(DEMUX2\_ENABLE, HIGH);

//delay(10000);

break;

case 7:

digitalWrite(DEMUX2\_ENABLE, HIGH);

digitalWrite(DEMUX2\_A, HIGH);

//delay(10000);

break;

case 8:

digitalWrite(DEMUX2\_ENABLE, HIGH);

digitalWrite(DEMUX2\_B, HIGH);

//delay(10000);

break;

case 9:

digitalWrite(DEMUX2\_ENABLE, HIGH);

digitalWrite(DEMUX2\_B, HIGH);

digitalWrite(DEMUX2\_A, HIGH);

//delay(10000);

break;

case 10:

digitalWrite(DEMUX2\_ENABLE, HIGH);

digitalWrite(DEMUX2\_C, HIGH);

//delay(10000);

break;

case 11:

digitalWrite(DEMUX2\_ENABLE, HIGH);

digitalWrite(DEMUX2\_C, HIGH);

digitalWrite(DEMUX2\_A, HIGH);

//delay(10000);

break;

case 12:

digitalWrite(DEMUX3\_ENABLE, HIGH);

//delay(10000);

break;

case 13:

digitalWrite(DEMUX3\_ENABLE, HIGH);

digitalWrite(DEMUX3\_A, HIGH);

//delay(10000);

break;

case 14:

digitalWrite(DEMUX3\_ENABLE, HIGH);

digitalWrite(DEMUX3\_B, HIGH);

//delay(10000);

break;

case 15:

digitalWrite(DEMUX3\_ENABLE, HIGH);

digitalWrite(DEMUX3\_B, HIGH);

digitalWrite(DEMUX3\_A, HIGH);

//delay(10000);

break;

case 16:

digitalWrite(DEMUX3\_ENABLE, HIGH);

digitalWrite(DEMUX3\_C, HIGH);

//delay(10000);

break;

case 17:

digitalWrite(DEMUX3\_ENABLE, HIGH);

digitalWrite(DEMUX3\_C, HIGH);

digitalWrite(DEMUX3\_A, HIGH);

//delay(10000);

break;

default:

digitalWrite(DEMUX1\_ENABLE, LOW);

digitalWrite(DEMUX1\_A, LOW);

digitalWrite(DEMUX1\_B, LOW);

digitalWrite(DEMUX1\_C, LOW);

digitalWrite(DEMUX2\_ENABLE, LOW);

digitalWrite(DEMUX2\_A, LOW);

digitalWrite(DEMUX2\_B, LOW);

digitalWrite(DEMUX2\_C, LOW);

digitalWrite(DEMUX3\_ENABLE, LOW);

digitalWrite(DEMUX3\_A, LOW);

digitalWrite(DEMUX3\_B, LOW);

digitalWrite(DEMUX3\_C, LOW);

break;

}

delay(100);

ADC\_Value = takeADCSample();

digitalWrite(DEMUX1\_ENABLE, LOW);

digitalWrite(DEMUX1\_A, LOW);

digitalWrite(DEMUX1\_B, LOW);

digitalWrite(DEMUX1\_C, LOW);

digitalWrite(DEMUX2\_ENABLE, LOW);

digitalWrite(DEMUX2\_A, LOW);

digitalWrite(DEMUX2\_B, LOW);

digitalWrite(DEMUX2\_C, LOW);

digitalWrite(DEMUX3\_ENABLE, LOW);

digitalWrite(DEMUX3\_A, LOW);

digitalWrite(DEMUX3\_B, LOW);

digitalWrite(DEMUX3\_C, LOW);

return ADC\_Value;

}

1. Note: It is not recommended to charge or discharge lead acid batteries above 50°C. [2] [↑](#footnote-ref-1)