Battery Monitoring System for Vehicle Leakage Current

Personal Project:

[#1]

Adam Michelin

Gothenburg, 2024

Abstract

The Battery Monitoring System (BMS) developed for monitoring vehicle leakage current presents a promising solution to the challenges of unexpected battery discharging. This report outlines the development process, including the rationale, methodology, components used, and measurement principles. Results from initial tests, along with discussions on the system's effectiveness and comparisons with other diagnostics tools, are provided. The report concludes with recommendations for future improvements and the significance of this project as a prototype for troubleshooting automotive electrical issues. It is worth noting that this project is a proof of concept undertaken for personal interest, aimed at enhancing understanding in electrical engineering and broadening personal project experiences.

1. Introduction

Upon encountering repeated instances of the starter battery in a family member's car being depleted after just 2 days of non-use, the need to investigate and rectify the underlying cause became paramount. Despite seeking assistance from a reputable service center, the persistent issue of unexplained and continuous battery discharge remained unresolved.

The starter battery, with a capacity of approximately 70Ah, should ideally hold a charge for at least some weeks under normal conditions. However, its complete depletion within 48 hours indicates an average draw of about 1.5 Amperes, meaning a persistent and unexplained drain on the system is occurring when the car is left unused.

Facing these challenges and the limitations of help from the service center, a proactive fix was developed: a Battery Monitoring System (BMS). This electrical device, designed to be compact, does a few key things. It keeps an eye on how much current the battery is using both the charging and discharging cycles, stores that data, and then shows it visually for analysis. The idea is to track the power usage over time in different situations and environments, hoping to find out why the battery keeps draining. This data-driven approach not only helps the troubleshooting but also gives solid evidence to explore new strategies if the usual fixes don't do the trick.

2. Material & Method

2.1 Vehicle and Battery Information

The vehicle under study is a "Volvo V60 D6 Plug-in Hybrid 2013" equipped with a 70Ah starter battery.

2.2 Components Used

The measurement device comprises the following components:

- ESP32 (WEMOS LOLIN D32, Espressif).
- DS3231 module (Real Time Clock).
- SD card reader module (SPI communication).
- Adafruit ADS1115 (16bit, Analog-Digital-Converter module).
- Shunt Resistor (75mV = 50A).
- Adafruit_SSD1306 (128x64 OLED display)

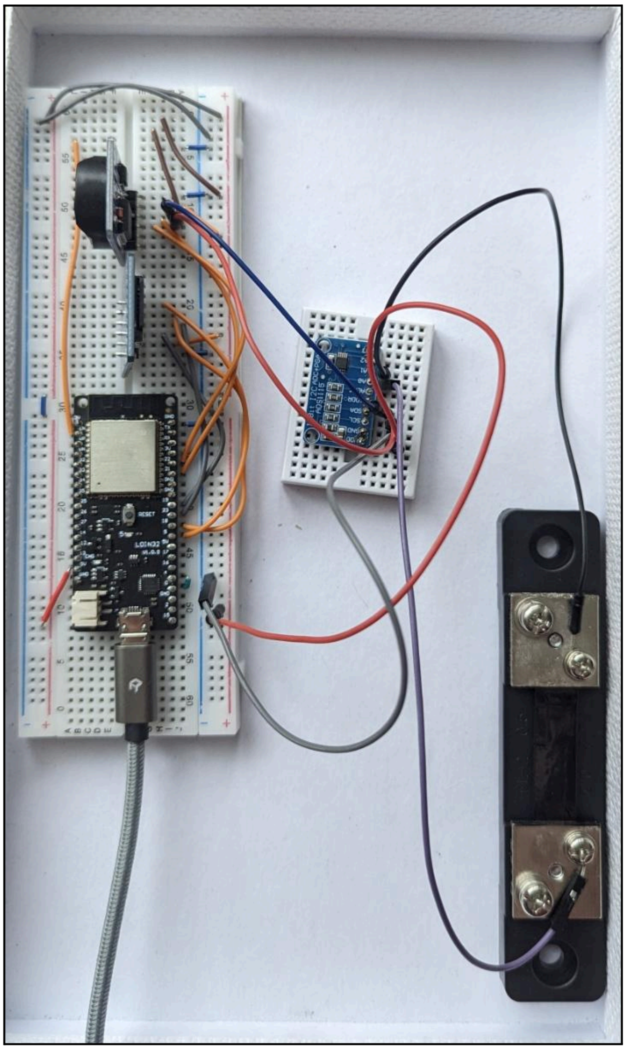

Figure 1, prototype on breadboard

2.3 Libraries and Programming:

The necessary libraries for ESP32 programming include:

- Arduino.h
- Wire.h
- Adafruit_GFX
- Adafruit_SSD1306.h
- image_data.h (custom library)
- SPI.h
- SdFAT.h
- RTClib.h
- string.h
- Adafruit_ADS1X15.h

The ESP32 was programmed using PlatformIO in Microsoft's Visual Studio Code (VS Code), enabling board configuration, programming, and debugging for various microcontrollers.

2.4 Electrical Setup:

The electrical circuit was initially prototyped on a breadboard for ease of debugging and testing flexibility with other modules. The final product remains on a breadboard but may transition to a custom PCB or other form based on the owner's preferences.

2.5 Measurement Principle:

The measurement device primarily monitors the voltage drop across the shunt resistor, which is strategically low in resistance to minimize its impact on the car's battery circuit and current flow. This voltage drop is measured and amplified through a 16-bit ADC for enhanced resolution. Without the 16-bit ADC, the voltage would be converted via the built in 12-bit ADC on the ESP32. The current resolution would then be too low, measuring steps of only 12mA, changing to a 16-bit ADC gives us the resolution of 0.7mA which is subjectively chosen to measure even the smallest current changes to better understand the underlying problem to the discharging.

2.6 Data Collection and Analysis:

The ESP32, with the assistance of the RTC and SD card reader, stores voltage readings every second and converts them into actual current values based on the calculated resolution. This data is then organized into minute-based arrays and stored on the SD card for further analysis. A Python script is used for visualizing this data in graph format, aiding in the interpretation of battery current over time.

3. Results

3.1 First Graph of Monitored Battery System

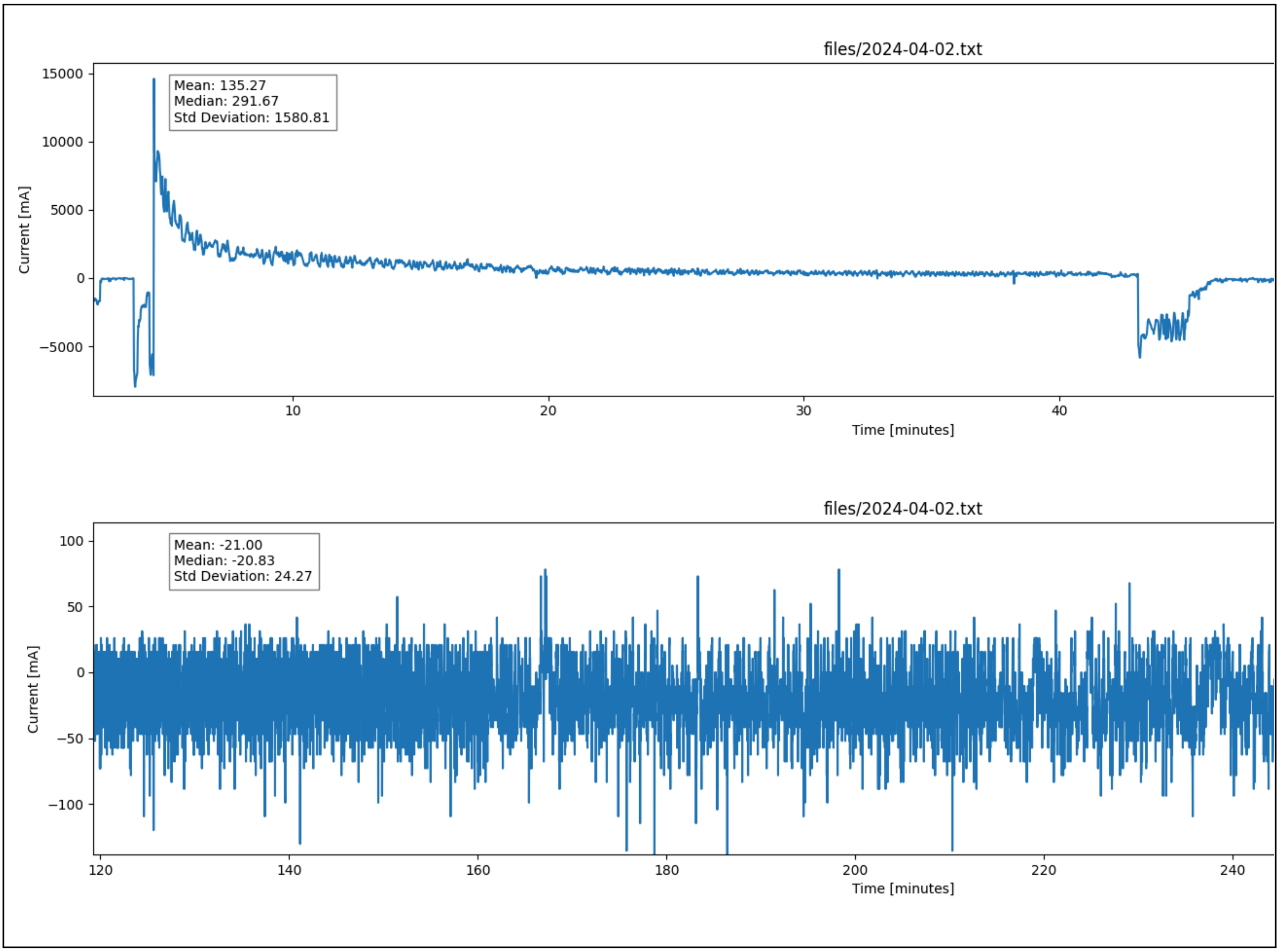

Figure 2, first test run

3.2 Simplified Circuit

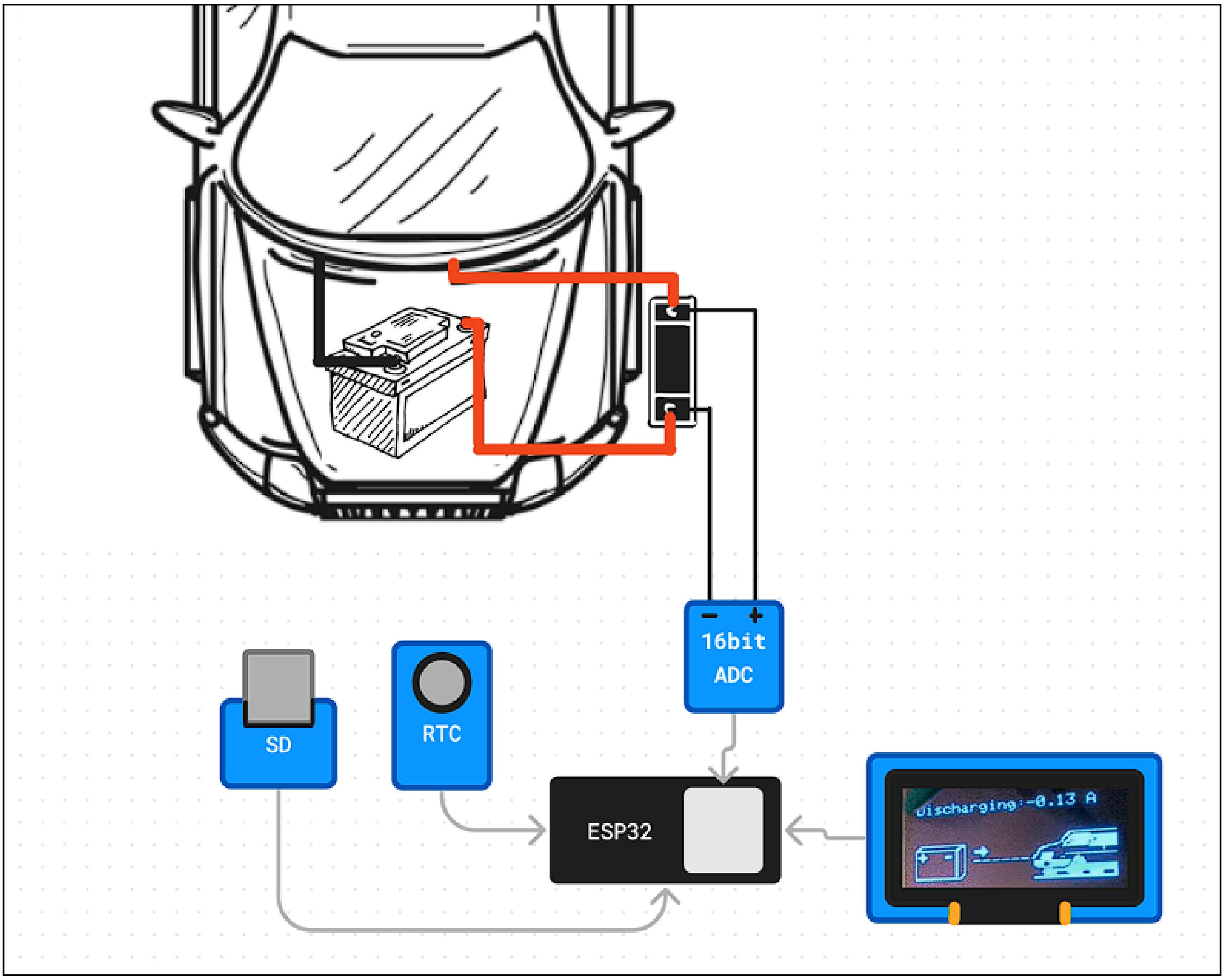

Figure 3, simplified circuit of the finished device

4. Discussion

4.1 Battery Discharge Analysis

Since the device only ran a single test lasting a few hours before the making of this report, we haven't yet identified a solution to the issue of unwanted discharging. As we conduct more experiments and analyze the data, we will update this report with our findings.

Regarding the device's performance, it met our expectations. The graphs generated are clear, showing the precise measured current at each timestamp.

4.2 Effectiveness of Monitoring System

The Battery Monitoring System (BMS) has proven highly effective due to its user-friendly design, clear display, and intuitive operation, which have laid a solid foundation for troubleshooting current leakage in the car.

The device's functionality includes the ability to measure both positive and negative voltages across the shunt resistor, thanks to the 16-bit ADC. Initially, the project involved integrating an amplifier and offset circuit to handle negative voltages shown in figure 4 & 5. This would firstly offset the maximum negative voltage (-75mV) to 0V, then amplify the signal to 5V as maximum positive voltage (the ESP32's maximum input voltage). As a result, the signal is converted to a range of 0-2.5V, where negative voltages represent current flow from the battery (drawn current) and positive voltages represent current charging the battery. This approach would enhance the device's usability and makes interpreting the data more intuitive for users if not an ADC with negative input voltage would have been used.

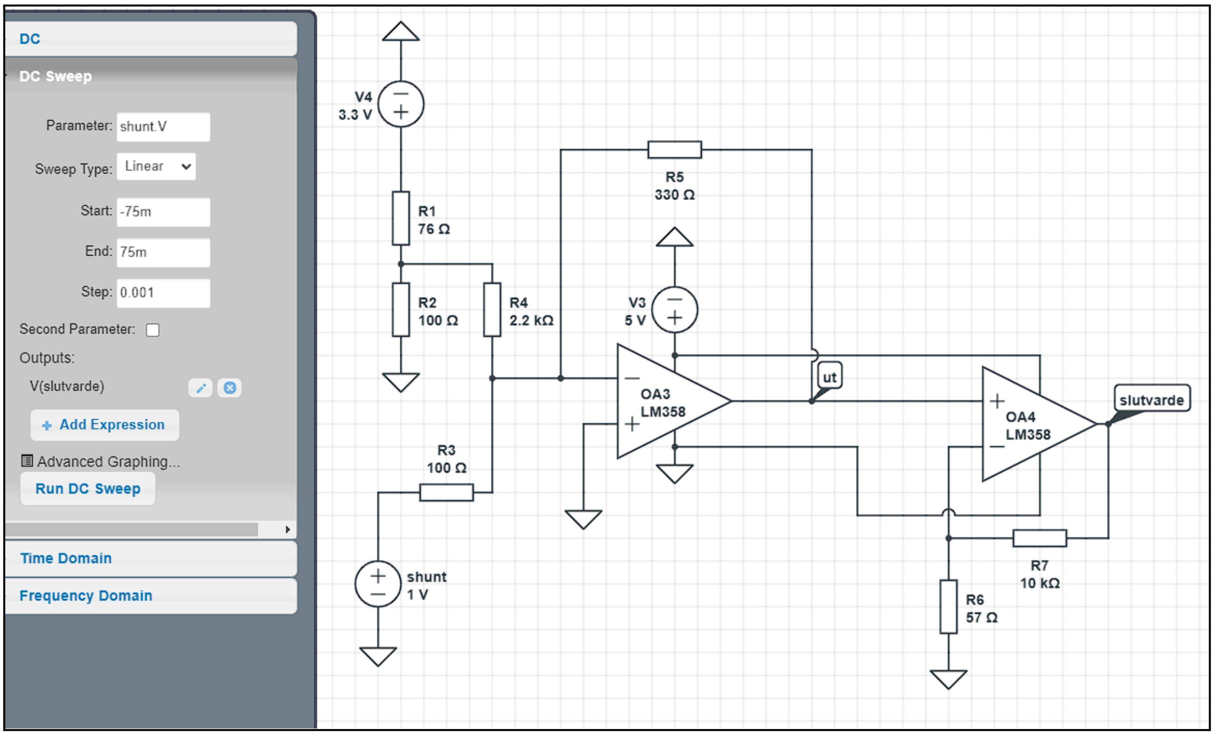

Figure 4, offset and amplifying circuit

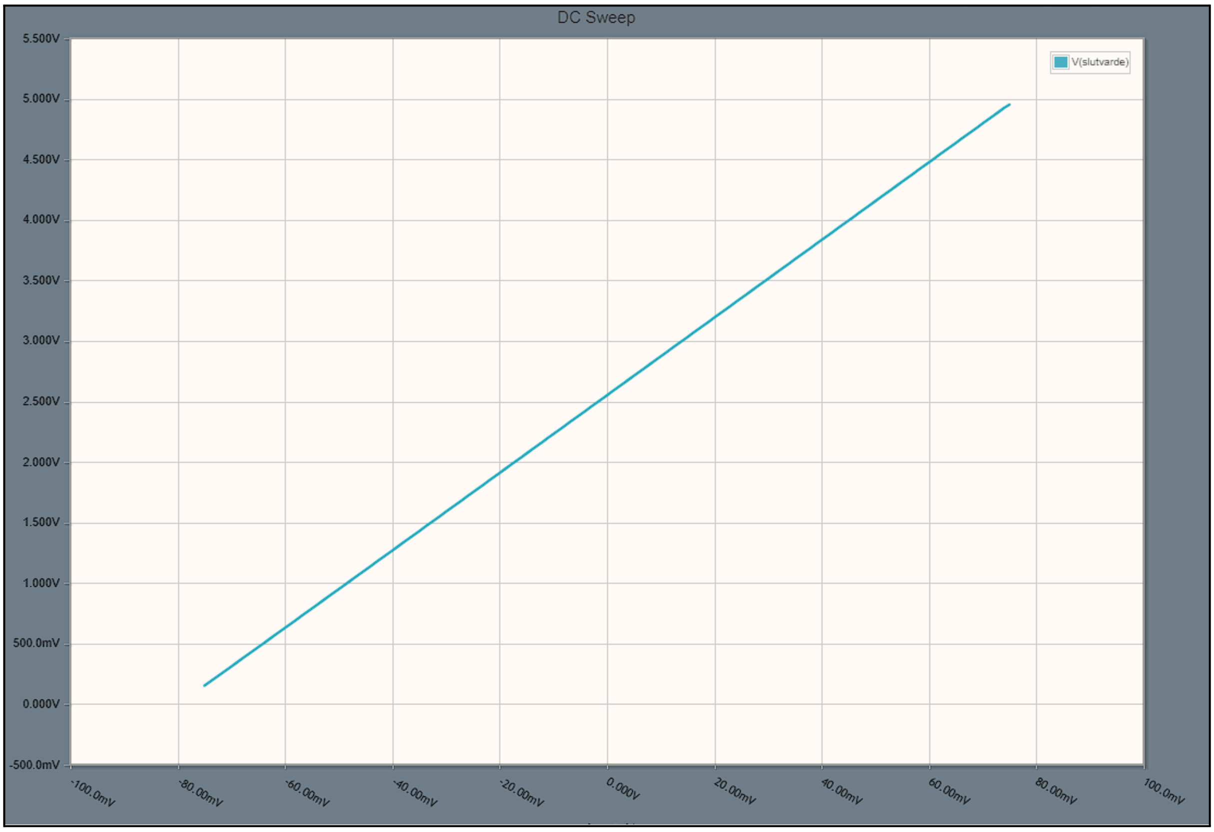

Figure 5, DC sweep to simulating the shunt's possible voltages after offset and amplification

4.3 Comparison with Other Diagnostics

When troubleshooting battery failures or anomalies, several techniques and devices can be used alongside or in comparison to a Battery Monitoring System (BMS). Here are some common ones and how they compare to the developed device describes in this report.

4.3.1 Onboard Diagnostics (OBD) Scanner

OBD scanners are commonly used to connect to a vehicle's onboard diagnostics system, offering insights into error codes, system parameters, and electronic issues. While OBD scanners excel in providing a broad overview of a vehicle's health and diagnosing common electronic problems, they may lack the specificity needed for detailed battery monitoring. Unlike the BMS developed in this report, OBD scanners may not directly measure battery current draw or provide real-time data on charge/discharge cycles.

4.3.2 Advanced Battery Management System (ABMS)

Advanced Battery Management Systems, akin to the BMS described in this report, offer comprehensive monitoring of battery health, temperature, and charge/discharge cycles. They are particularly effective in electric vehicles and renewable energy systems. ABMSs often integrate sophisticated algorithms and sensor arrays to optimize battery performance and prolong lifespan. Compared to traditional BMS setups, the developed BMS in this report focuses specifically on measuring current draw and providing detailed data visualization, which can be advantageous for pinpointing discharge anomalies. Since the troubled car described in this report does not have an ABMS installed, and considering the technical complexities and high costs associated with such a system, it is not deemed suitable for addressing the specific problem that our Battery Monitoring System (BMS) aims to solve.

4.3.3 Thermal Imaging

Thermal imaging utilizes infrared technology to detect hotspots or abnormal temperature distributions in batteries, signaling potential issues such as internal shorts or cell degradation. While thermal imaging is very suitable for identifying thermal irregularities, it primarily focuses on temperature variations and may not directly measure electrical parameters like current draw or voltage. Therefore, while complementary to the BMS, thermal imaging alone may not provide the complete picture necessary for diagnosing complex electrical issues related to battery discharge. Moreover, identifying the suspected fault in this situation as its highly likely a component error within the car's diverse set of components makes implementing thermal imaging challenging for pinpointing such faults on a large scale.

4.3.4 Circuit Analysis Tools

Circuit analysis tools, including oscilloscopes and power analyzers, are crucial for assessing electrical circuits' integrity and identifying faults related to battery operation. These tools offer detailed insights into voltage waveforms, current fluctuations, and circuit behavior. However, they typically require specialized expertise to interpret results effectively and may not offer the continuous monitoring capabilities of a dedicated BMS. The BMS developed in this report focuses on continuous, real-time monitoring of battery current draw and provides visual data representation for ease of analysis, which may be more user-friendly compared to complex circuit analysis tools.

4.4 Future Improvements and Recommendations

4.4.1 Enhanced Data Logging

Implementing more advanced data logging capabilities, such as storing additional parameters like temperature, voltage ripples and compensation for noise, can provide a more comprehensive understanding of the battery's behavior and health. This data can be instrumental in identifying patterns or correlations that may contribute to battery discharge issues.

4.4.2 Real-Time Alerts

Real-Time Alerts: Introducing real-time alerting mechanisms based on predefined thresholds for current draw or voltage levels can notify users of potential anomalies as they occur. This proactive approach can help in early detection and prompt action to prevent further battery damage or vehicle downtime.

4.4.3 Remote Monitoring and Control

Incorporating remote monitoring and control features can enable users to access BMS data and settings remotely. This can be particularly beneficial for service technicians, allowing them to monitor multiple vehicles' battery health and take proactive measures as needed. Initially, the project envisioned utilizing the ESP32's WiFi module to transmit data when the vehicle is parked at home, automatically connecting to the home WiFi network and identifying unsent data for remote monitoring. This functionality was planned to significantly enhance the BMS's usability but was ultimately postponed due to time constraints.

5. Conclusion

The Battery Monitoring System (BMS) developed for vehicle leakage current presents a promising solution to the challenges of unexpected battery discharging. Despite being a prototype, it effectively monitors the vehicles current from the starter battery and provides valuable insights through data visualization.

This project, born out of a personal need, has not only showcased practical application but also provided a valuable learning experience as an electrical engineering student at Chalmers University.

Moving forward, enhancements like advanced data logging, real-time alerts, and remote monitoring features can further improve the BMS's functionality. Collaborative efforts and continuous testing will be key to refining and validating the battery system.

Despite constraints faced during development, the BMS prototype represents a significant step towards leveraging technology for automotive diagnostics and highlights the potential for future advancements in this field.

6. Appendix

6.1 ESP32 Main Source Code

```
#include <Arduino.h>
#include <Wire.h>
#include <Adafruit_GFX.h>
#include <Adafruit_SSD1306.h>
#include "image_data.h"
#include <SPI.h>
#include <SdFat.h>
#include <RTClib.h>
#include <string.h>
#include <Adafruit_ADS1X15.h>
//#include <DS3231.h>

// change this value depending on the shunt used
#define SHUNT_SIZE 50 //amps / 75mV


#define SCREEN_WIDTH 128 // OLED display width, in pixels
#define SCREEN_HEIGHT 64 // OLED display height, in pixels
#define LEFT 1
#define RIGHT 0

#define INPUT_PIN 32
// Declaration for an SSD1306 display connected to I2C (SDA, SCL pins)
// The pins for I2C are defined by the Wire-library.
// On Wemos Lolin32 GPIO21(SDA), GPI022(SCL)

//setup and class for ADC
Adafruit_ADS1115 ads;  /* Use this for the 16-bit version */
//Adafruit_ADS1015 ads;     /* Use this for the 12-bit version */


// RTC DS3231 module
//class declraration
RTC_DS3231 rtc;
//DS3231  rtc(SDA, SCL);

// SD-card reader
SdFat sd;
SdFile file;
// Change this to the actual pin used for chip select
const int chipSelect = 5;


#define OLED_RESET     -1 // Reset pin # (or -1 if sharing Arduino reset pin)
#define SCREEN_ADDRESS 0x3C ///< See datasheet for Address; 0x3D for 128x64, 0x3C for 128x32
Adafruit_SSD1306 display(SCREEN_WIDTH, SCREEN_HEIGHT, &Wire, OLED_RESET);


/// @brief ////////FUNCTION DECLARATION /////////////
void write_file(float data, int count);
float get_adc_data();
void interface(float amps, int dir);
void move_arrow(int dir, float amps);
/// @brief //////////////////////////////////////////

void setup() {

 Serial.begin(115200);
 while (!Serial) {}  // wait for board
 // SSD1306_SWITCHCAPVCC = generate display voltage from 3.3V internally
 // Uncomment to use display
 if(!display.begin(SSD1306_SWITCHCAPVCC, SCREEN_ADDRESS)) {
```

```
    Serial.println(F("SSD1306 allocation failed"));
    for(;;); // Don't proceed, loop forever
  } else {
    Serial.println("SSD1306 allocation success");
  }

  // Clear the buffer
  display.clearDisplay();

  set analog pin GPIO4 to input to read
  pinMode(INPUT, INPUT);


// Ported to SdFat from the native Arduino SD library example by Bill Greiman
// On the Ethernet Shield, CS is pin 4. SdFat handles setting SS
/*
 SD card read/write

 This example shows how to read and write data to and from an SD card file
 The circuit:
 * SD card attached to SPI bus as follows:
 ** MOSI - pin 11
 ** MISO - pin 12
 ** CLK - pin 13
 ** CS - pin 4

 created   Nov 2010
 by David A. Mellis
 updated 2 Dec 2010
 by Tom Igoe
 modified by Bill Greiman 11 Apr 2011
 This example code is in the public domain.

 */

  // setup code for ADC
  // The ADC input range (or gain) can be changed via the following
  // functions, but be careful never to exceed VDD +0.3V max, or to
  // exceed the upper and lower limits if you adjust the input range!
  // Setting these values incorrectly may destroy your ADC!
  //                                                                ADS1015  ADS1115
  //                                                                -------  -------
  // ads.setGain(GAIN_TWOTHIRDS);  // 2/3x gain +/- 6.144V  1 bit = 3mV      0.1875mV (default)
  // ads.setGain(GAIN_ONE);        // 1x gain   +/- 4.096V  1 bit = 2mV      0.125mV
  // ads.setGain(GAIN_TWO);        // 2x gain   +/- 2.048V  1 bit = 1mV      0.0625mV
  // ads.setGain(GAIN_FOUR);       // 4x gain   +/- 1.024V  1 bit = 0.5mV    0.03125mV
  // ads.setGain(GAIN_EIGHT);      // 8x gain   +/- 0.512V  1 bit = 0.25mV   0.015625mV
  ads.setGain(GAIN_SIXTEEN);    // 16x gain  +/- 0.256V  1 bit = 0.125mV  0.0078125mV

  if (!ads.begin()) {
    Serial.println("Failed to initialize ADS.");
    while (1);
  }


  // RTC setup
  rtc.begin();
  // if (! rtc.begin()) {
  //   Serial.println("Couldn't find RTC");
  //   Serial.flush();
  //   while (1) delay(10);
  // }
  // //gets the time when compiled
  // rtc.adjust(DateTime(F(__DATE__), F(__TIME__)));



  // SD card setup

  // Initialize SdFat or print a detailed error message and halt
  // Use half speed like the native library.
  // change to SPI_FULL_SPEED for more performance.
```

```
 if (!sd.begin(chipSelect, SPI_HALF_SPEED)) sd.initErrorHalt();



}



//  // Code for SD card
//  // Initialize SD card
//   if (!SD.begin(5)) {
//     Serial.println("SD card initialization failed!");
//     return;
//   }
//   Serial.println("SD card initialized successfully!");

// >Variables
float current = 0.0;
int direction = RIGHT;
int direction_before = LEFT;
int direct_log = 0;
int count = 0;

void loop() {

 while(count < 60){
  float data = get_adc_data();
  Serial.println(data);
  // write to file
  write_file(data, count);

  delay(999.96); //compensating for runtime
  count++;
 }

 count = 0;

 // code for display")
 // Uncomment to use with display
  current = analogRead(INPUT_PIN);
  direct_log = analogRead(INPUT_PIN);
  Serial.println(direct_log);


  current = ((current * 200)/(1<<16)) - 12;
  // this means that the voltage is negative
  if(current < 0) direction = RIGHT;   // which means that the direction of current is charging the battery

  else if(current >= 0) direction = LEFT; // which means that the direction of current is discharging the battery


  //Serial.println(direction)
  //Serial.println(current);


  interface(current, direction);

  // Serial.println(current)
}


float get_adc_data(){
 int16_t data;
 float multiplier = 0.0078125F; /* ADS1115  @ +/- 6.144V gain (16-bit results) */
 int results = ads.readADC_Differential_0_1();
 return (results * multiplier * SHUNT_SIZE / 75 * 1000);
}


void write_file(float data, int count) {
 DateTime time = rtc.now();

 String date = time.timestamp(DateTime::TIMESTAMP_DATE);
 String txt = ".txt";
```

```
 String filename_merged = date + txt;
 // Convert filnamn to a char*
 char filename[filename_merged.length() + 1]; // +1 for null terminator
 filename_merged.toCharArray(filename, sizeof(filename));

 // Serial.println(date);
 // open the file for write at end like the Native SD library
 if (!file.open(filename, O_RDWR | O_CREAT | O_AT_END)) sd.errorHalt("opening file for write failed");


 if(count==0){
  file.println();
  file.print(time.timestamp(DateTime::TIMESTAMP_TIME));
  file.print(" --> ");
 }

 if(count < 59){
  file.print(data);
  file.print(", "); // Add a collon between elements
 }

 if(count == 59) file.println();
  // close the file:
 file.close();

}






void interface(float amps, int dir){
 //dir is either LEFT or RI
 display.clearDisplay();
 display.drawLine(41, 46, 45, 46, WHITE);
 display.drawLine(65, 46, 69, 46, WHITE);
 display.drawLine(71, 46, 75, 46, WHITE);
 display.drawLine(59, 46, 63, 46, WHITE);
 display.drawLine(53, 46, 57, 46, WHITE);
 display.drawLine(47, 46, 51, 46, WHITE);
 display.drawBitmap(79, 16, image_volvo_bits, 50, 50, WHITE);
 display.drawBitmap(78, 30, image_Layer_18_bits, 51, 17, WHITE);
 display.drawBitmap(0, 31, image_battery__2__bits, 40, 30, WHITE);
 move_arrow(dir, amps);
 display.display();
}


// function that moves the arrow either right or left showing the current direction
// this function below should be able to be run in the background of the code
void move_arrow(int dir, float amps){
 int x = 38;
 int x_slut = 69;

 if(dir == RIGHT){
 x = 38;
  display.setTextColor(WHITE);
  display.setTextSize(1);
  display.setCursor(3, 6);
  display.setTextWrap(false);
  display.print("Discharging:");
  display.print(amps);
  display.print(" A");
  for(int i = 0; i < 31; i++){
   display.drawBitmap(x, 37, image_Pin_arrow_right_9x7_bits, 9, 7, WHITE);
   display.display();
   delay(10);
   display.drawBitmap(x, 37, image_Pin_arrow_right_9x7_bits, 9, 7, BLACK);
   x++;
   if(x==x_slut) x = 38;
  }
 }
```

```
  else if(dir == LEFT){
  x = x_slut;
    display.setTextColor(WHITE);
    display.setTextSize(1);
    display.setCursor(3, 6);
    display.setTextWrap(false);
    display.print("Charging:");
    display.print(amps);
    display.print(" A");
    for(int i = 0; i < 31; i++){
      display.drawBitmap(x, 37, image_Pin_arrow_left_9x7_bits, 9, 7, WHITE);
      display.display();
      delay(10);
      display.drawBitmap(x, 37, image_Pin_arrow_left_9x7_bits, 9, 7, BLACK);
      x--;
      if(x==38) x = x_slut;
    }
  }
}
```

6.2 Python Code for Visualization

```
import matplotlib.pyplot as plt
import datetime

# Define a function to read the file and extract the data
def read_file(file_name):
    with open(file_name, 'r') as file:
        lines = file.readlines()
    data = []
    for line in lines:
        if '-->' in line:
            numbers_part = line.split('-->')[-1]
            numbers = [float(x) for x in numbers_part.strip().split(',') if x]
            data.extend(numbers)
    return data

# Define a function to plot the data
def plot_data(data):
    plt.plot(range(len(data)), data)
    plt.xlabel('Time (seconds)')
    plt.ylabel('Data')
    plt.title('Data over Time')
    plt.show()


# Get today's date
today = datetime.date.today()

# Format the date as a string in the format 'YYYY-MM-DD'
today_str = today.strftime('%Y-%m-%d')

# Define the file name
file_name = f'files/{today_str}.txt'



# Main code

# Read the file and extract the data
data = read_file(file_name)
print(data)

# Plot the data
plot_data(data)
```