Final Project Proposal

I. Evaluation

In the initial proposal, the energy meter culminated to a full PCB design and validation with a high voltage DC tractive system. Due to the accelerated timeline of 2-3 weeks and limited access to a high voltage circuit, the project will be adjusted to demonstrate a full working prototype and reliable data logging from each sensor. While there may not be enough time to fabricate, assemble, and test a PCB, part of the project will include a full design of the embedded energy meter PCB to fit the constraints and mechanical bounding box.

II. Project Overview

Proposed here is custom energy meter for automotive applications which processes various tractive system sensor signals and records the data to a storage device. Sensor signals to be processed include tractive system voltage, current, power, and battery temperature – pillars of energy indicators. To increase my exposure and knowledge of communication protocols, the selected sensors communicate with a variety of methods including 1-Wire and Controller Area Network (CAN) through a twisted wire automotive interface (TWAI) on the esp32.

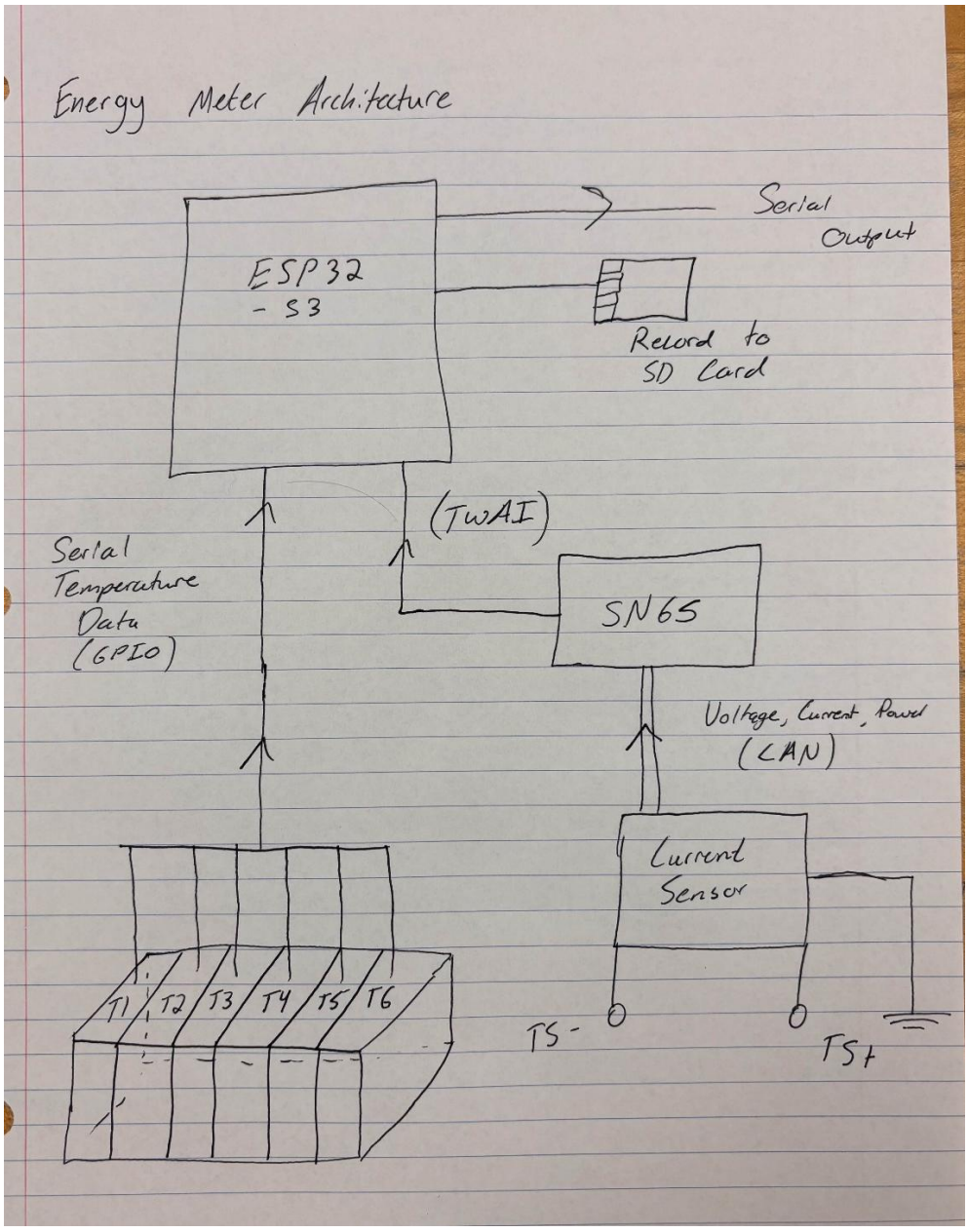
This energy meter proposal builds on a prior automotive PCB centered around converting analog signals to CAN data for the automotive bus. Known as a “CAN Expansion Board”, the PCB used an 8-bit Atmega328 central processor and MCP25265 for CAN control and transmission to convert analog automotive sensor signals to CAN data. While this design isn’t directly built on the CAN Expansion Board, similar chip architectures and protocols are used.

The following diagram and component list detail the computer architecture, dataflow, and required sensors for project execution. Initially, voltage and current will be collected from the IVT-S current sensor and power computed on board the sensor. Voltage, current, and power data will be transmitted over CAN from the IVT-S to a SN65HVD23 CAN transceiver, responsible for handling the differential pair hardware. As the central processing unit, an ESP-32-S3 will be used due to its processing speed and input/output capabilities. One of the benefits of using the ESP-32-S3 is its capacity for TWAI, eliminating additional hardware for a CAN controller. The TWAI will process and control the data coming from the CAN transceiver.

Simultaneously, multiple temperature sensors will be connected in series and communicated with via 1-Wire data transfer. 1-Wire is an efficient, low-speed, low-power serial communication protocol that only requires a single GPIO pin on the ESP-32. The ESP-32-S3 will compile all data from both sensors and send it to a serial port for real time monitoring along with recording all data to a microSD card.

Since this project is aimed at automotive and motorsport applications, circuit protection and energy indication is paramount. Not pictured in the central architecture diagram above, devices such as electrostatic discharge (ESD) diodes, transient voltage suppression (TVS) diodes,

and other protective devices will be used against noise and damage. Most importantly, addressable LEDs or comparator driven LEDs will indicate potential sensor faults, missed CAN messages, or energy thresholds.



Component List:

- MCU: ESP32-S3-WROOM-N8R8 ($5.50)
- Temperature sensor: DS18B20 ($3.00 x 5)
- CAN transceiver: SN65HVD230DR ($2.18)
- 8 GB microSD card ($8)
- Current Sensor: Isabellenhutte IVT-S (already purchased)

Total: $35.00

Project Justification

As discussed earlier, I have automotive design experience with the CAN Expansion Boards. While I did not design these circuits from scratch, I had iterated on the original design to equip them with ESD protection, signal filtering, and overall automotive improvement. The flaws with the CAN Expansion Boards lied in limits placed on the sample rate and bit depth of Atmega328. Therefore, this energy meter works with entirely new sensors, a more advanced 32-bit microprocessor and data pipeline, but familiar communication protocols.

Additionally, while the scope of this project may seem slightly large, my prior experience with intermediate to advanced PCB design should leave major difficulties with software and message timing. Therefore, before attempting the final PCB, the architecture will be prototyped with a MCP 2515 development module. This development module has a separate CAN transceiver and controller with familiar Arduino libraries, enabling for rapid prototyping of the CAN message exchange.

Lastly, I hope to pursue a potential career in advanced/smart manufacturing, where CAN protocol is used extensively. A robust PCB that explores different protocols for data collection and analysis would provide me with valuable knowledge of and challenges with CAN, 1-Wire, and SPI serial timing. Furthermore, this energy meter project will be further developed to be integrated with Panther Racing and validate the tractive system performance.

Schedule

Week 1: Prototype communication between esp32, CAN sensors, 1-Wire sensors, log to SD Card.
Week 2: Log energy calcs to SD Card; test with voltage divider add LED indicators, PCB design.

For the final demonstration, I plan to show the effective processing and logging of temperature, voltage, and current via the communication protocols described above. “Effective processing” will be defined by logging all signals sent from the sensors and correct LED signaling for when energy levels are exceeded. These acceptable energy levels will be set closer to the demonstration.

Roadblocks, comments, questions, alternatives, and concerns

The largest roadblock will be exploring the twisted wire automotive interface (TWAI) and the way it interacts with the CAN transceiver. There is plenty of documentation on the capability including some of the examples given below by Espressif:

Configuration:

```
#include "driver/gpio.h"
#include "driver/twai.h"

void app_main()
{
    //Initialize configuration structures using macro initializers
    twai_general_config_t g_config = TWAI_GENERAL_CONFIG_DEFAULT(GPIO_NUM_21, GPIO_NUM_22, TWAI_MODE_NOR
    twai_timing_config_t t_config = TWAI_TIMING_CONFIG_500KBITS();
    twai_filter_config_t f_config = TWAI_FILTER_CONFIG_ACCEPT_ALL();

    //Install TWAI driver
    if (twai_driver_install(&g_config, &t_config, &f_config) == ESP_OK) {
        printf("Driver installed\n");
    } else {
        printf("Failed to install driver\n");
        return;
    }

    //Start TWAI driver
    if (twai_start() == ESP_OK) {
        printf("Driver started\n");
    } else {
        printf("Failed to start driver\n");
        return;
    }

    ...

}
```

Transmission:

```c
#include "driver/twai.h"

...

//Configure message to transmit
twai_message_t message;
message.identifier = 0xAAAA;
message.extd = 1;
message.data_length_code = 4;
for (int i = 0; i < 4; i++) {
    message.data[i] = 0;
}

//Queue message for transmission
if (twai_transmit(&message, pdMS_TO_TICKS(1000)) == ESP_OK) {
    printf("Message queued for transmission\n");
} else {
    printf("Failed to queue message for transmission\n");
}
```

Reception:

```c
#include "driver/twai.h"

...

//Wait for message to be received
twai_message_t message;
if (twai_receive(&message, pdMS_TO_TICKS(10000)) == ESP_OK) {
    printf("Message received\n");
} else {
    printf("Failed to receive message\n");
    return;
}

//Process received message
if (message.extd) {
    printf("Message is in Extended Format\n");
} else {
    printf("Message is in Standard Format\n");
}
printf("ID is %d\n", message.identifier);
if (!(message.rtr)) {
    for (int i = 0; i < message.data_length_code; i++) {
        printf("Data byte %d = %d\n", i, message.data[i]);
    }
}
```